DynaPipe: Optimizing Multi-task Training through Dynamic Pipelines

Chenyu Jiang[*]
jchenyu@connect.hku.hk
The University of Hong Kong

Zhen Jia
zhej@amazon.com
Amazon Web Services

Shuai Zheng[†]
shuai@boson.ai
Boson AI

Yida Wang
wangyida@amazon.com
Amazon Web Services

Chuan Wu
cwu@cs.hku.hk
The University of Hong Kong

Abstract

Multi-task model training has been adopted to enable a single deep neural network model (often a large language model) to handle multiple tasks (e.g., question answering and text summarization). Multi-task training commonly receives input sequences of highly different lengths due to the diverse contexts of different tasks. Padding (to the same sequence length) or packing (short examples into long sequences of the same length) is usually adopted to prepare input samples for model training, which is nonetheless not space or computation efficient. This paper proposes a dynamic micro-batching approach to tackle sequence length variation and enable efficient multi-task model training. We advocate pipeline-parallel training of the large model with variable-length micro-batches, each of which potentially comprises a different number of samples. We optimize micro-batch construction using a dynamic programming-based approach, and handle micro-batch execution time variation through dynamic pipeline and communication scheduling, enabling highly efficient pipeline training. Extensive evaluation on the FLANv2 dataset demonstrates up to 4.39x higher training throughput when training T5, and 3.25x when training GPT, as compared with packing-based baselines. DynaPipe's source code is publicly available at https://github.com/awslabs/optimizing-multitask-training-through-dynamic-pipelines.

Packing 是怎么做的?

如何处理句长不同的?

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; **Machine learning**.

Keywords: distributed systems, multi-task learning, pipeline parallelism

ACM Reference Format:
Chenyu Jiang, Zhen Jia, Shuai Zheng, Yida Wang, and Chuan Wu. 2024. DynaPipe: Optimizing Multi-task Training through Dynamic Pipelines . In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3627703.3629585

1 Introduction

Recent studies have shown that a single deep neural network (DNN), e.g., a large language model (LLM), can be trained/fine-tuned on a mixture of datasets to perform multiple tasks effectively [6, 24, 32, 35]. For example, T0 [32] is fine-tuned on 62 different NLP datasets and can perform a wide-range of tasks including question answering, sentiment analysis, summarization and sentence completion. Flan-T5 and Flan-PaLM [35] are fine-tuned on 473 datasets from 146 categories of tasks.

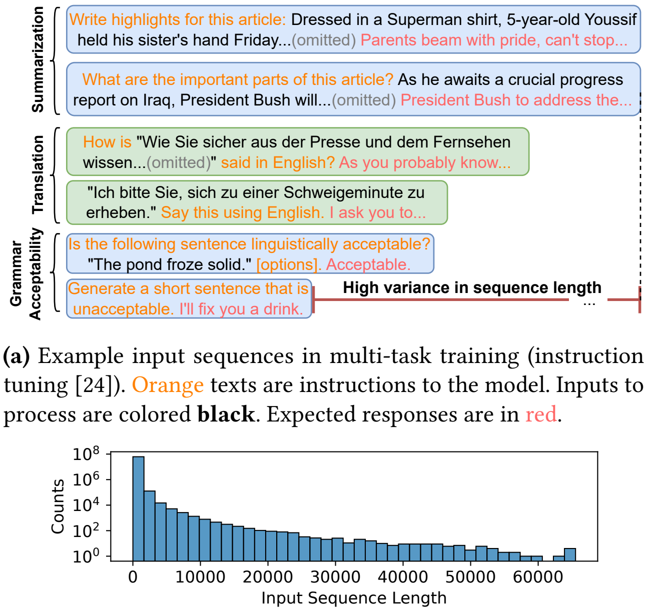
A crucial aspect of multi-task training is the accommodation of diverse text sequence lengths across various tasks or datasets. Tasks like summarization or information extraction usually involve a long context text as input, while only one sentence is usually used as input to simple question answering (e.g., checking the grammatical acceptability of the sentence) (Fig. 1a). The average input sequence length is 977.73 tokens in the CNN/Daily Mail [14] dataset for the text summarization task, while the MNLI dataset [36] for textual entailment only has an average input sequence length of 51.59. This results in high sequence length variations in multi-task dataset mixtures (e.g., FLANv2 [20]), as shown in Fig. 1b. During training, batches of uniform-length samples are usually needed to be fed into the accelerator devices (e.g., GPUs) for efficient processing. The input sequences need to be uniformly padded to at least the largest sequence length in each mini-batch (which can be very long due to the presence of long-context tasks), resulting in excessive padding, increased memory consumption and wasted computation.

The *packing* [30] approach has been advocated to alleviate this issue, which concatenates multiple short samples

[*] Work done during Chenyu's internship at AWS.
[†] Work done while at AWS.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629585



(a) Example input sequences in multi-task training (instruction tuning [24]). Orange texts are instructions to the model. Inputs to process are colored **black**. Expected responses are in red.

(b) The sequence length distribution in FLANv2 [20] zero shot dataset (truncated at 65536). Y-axis is in log scale.

Figure 1. Model inputs exhibit high sequence length variance in multi-task training.

to form a long sample whose length matches the largest input sequence length. Packing can be effective in reducing padding. Since almost all current language models use the Transformer [34] architecture, attention is computed among tokens in each long sequence during training and such attention computation is wasted among unrelated samples packed into the same sequence. Such computation waste grows quadratically with sequence length, leading to extensive overhead in case of large sequence lengths. Attention computation among unrelated samples can also have negative impact on model performance [18]. Additional attention masks [35] and adjustments of the positional embeddings [18] are needed to exclude this cross-contamination effect, which complicates model implementation.

将多个短seqs pack成一个长seq 会影响模型性能（必须添加额外mask）且会造成计算开销的显著增长

For better multi-task training efficiency, we propose a **dynamic micro-batching** approach to address the variable-length input challenges. Pipeline parallelism is commonly adopted in LLM training [25]: the large model is partitioned into stages deployed over multiple devices; the input mini-batch of training samples in each training iteration is partitioned into micro-batches, and the micro-batches are processed across the devices in a pipelining manner. Our key idea is that we only need to ensure similar sequence lengths among samples within each micro-batch, but not across micro-batches, such that we can group samples accordingly, minimize padding and eliminate any unnecessary attention computation or masking among unrelated samples.

Current pipeline training systems adopt uniform micro-batch sizes, and do not efficiently support processing of micro-batches with different sequence lengths, memory consumption and execution time. We design DynaPipe, a dynamic micro-batching pipeline training framework that enables efficient multi-task model training with different input sequence lengths. DynaPipe automatically optimizes micro-batching, pipeline and communication scheduling in each training iteration. We make the following contributions in designing DynaPipe.

▹ **We devise an efficient dynamic programming-based method to optimize micro-batch construction**, that balances the trade-off between padding reduction, computation efficiency and memory consumption of the micro-batches. Upon input of each mini-batch, we first sort the samples to minimize the sequence length difference between adjacent samples. We then use dynamic programming to decide optimal splits of the sorted sample list into micro-batches, exploiting our cost model on per-iteration LLM training time under pipeline parallelism.

▹ **We propose pipeline schedules that are robust to execution time variations of the micro-batches.** We identify that the commonly adopted 1F1B pipeline schedule [25] is prone to blocking (device idling during pipeline execution) under vacillating micro-batch execution time, based on the concept of safety stocks [5]. To mitigate the issue, we advocate adaptive scheduling which controls the injection time of micro-batches into the pipeline. We also make it memory-aware, maximizing training throughput while observing device memory limits.

▹ **We design effective communication schedule to allow irregular communication patterns in our dynamic pipeline.** Naïve communication schedule (i.e., sending tensors to the next pipeline stage immediately after production, and receiving them just before use) causes deadlocks in our dynamic pipelines since different processing stages of a micro-batch are no longer scheduled tightly one after another. We perform ahead-of-time planning, scheduling both send and receive operations at the production time of each tensor, which is guaranteed to be deadlock-free.

We implement DynaPipe on PyTorch [29]. Extensive evaluation on the FLANv2 dataset [20] reveals up to 4.39x throughput improvement when training T5 [30], and 3.25x when training GPT [6], compared with packing-based baselines.

2 Background and Motivation

2.1 Multi-task model training

Modern LLMs (e.g., T5 [30]) can perform different tasks without the need for task-specific model structures. To perform multi-task training, we only need to produce a mixture of data from different datasets. The mini-batches are then randomly sampled from the dataset. The exact way to mix data

一个 mini-batch 同时由来自多个 datasets 的 data 组成

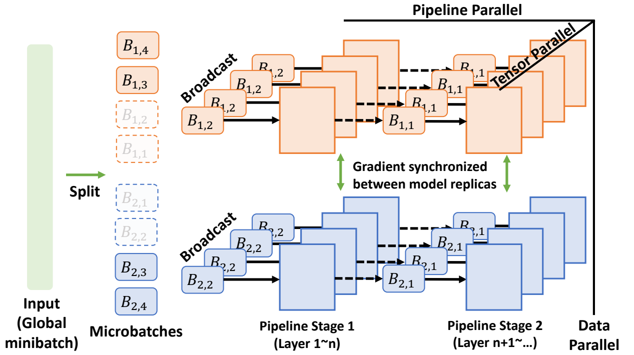


Figure 2. Illustration of 3D parallelism and input data partitioning. Each square block indicates (partitioned) model layers (operators) on a GPU. Different data parallel model replicas are denoted with different color (within each replica, model is partitioned through tensor and pipeline parallelism). The input global mini-batch is split into micro-batches (each containing different data). $B_{i,j}$: the jth micro-batch for model replica i.

(i.e., proportion of each dataset in the final mixture) can be method-dependent [30].

Multi-task LLMs are commonly trained with a combination of data, tensor and pipeline parallelism (i.e., 3D parallelism) to address memory pressure induced by their large model size. With pipeline parallelism, the model layers are partitioned among devices (stages), and a mini-batch is split into smaller micro-batches in the batch dimension. In each training iteration, the micro-batches are sequentially executed with gradient accumulated across the micro-batches. The forward and backward order for each device is determined by the pipeline schedule (e.g., 1F1B [25], where each stage executes one forward pass and one backward pass, alternatively). Tensor parallelism shards computation within individual operators (e.g., matrix multiplication) to different devices, and is agnostic to micro-batching. In data-parallel training, the model is replicated on each device and a different portion of the mini-batch is processed on each replica. Fig. 2 illustrates the input partitioning with 3D parallelism.

The input samples to a multi-task model often have vastly different sequence lengths. For efficient processing of hardware accelerator such as GPUs, samples are usually batched, forming a single input tensor. To accommodate longer sequences, shorter samples in a input batch need to be padded to (at least) the length of the longest sequence in the batch. Under extreme sequence length variations, the amount of padding can be substantial. For example, naïve padding (padding every sample to the length of the longest sequence in a mini-batch) of samples from the FLANv2 [20] dataset leads to more than 80% of padding tokens in a mini-batch. Memory and computation resources are wasted processing the unused padding tokens. While sorting (bucketing) samples by sequence length before batching can alleviate padding [28], it destroys randomness of batch sampling and may degrade model performance: for example, after sorting, the mini-batches with long sequence lengths may only consist of samples from a small number of tasks (such as summarization); such homogeneous batches may harm the performance of the multi-task model since it harms the model's generalizability [1, 12].

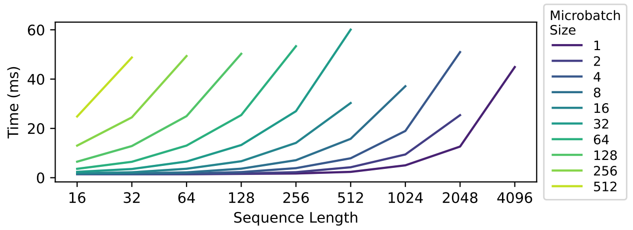


Figure 3. Computation time of a single Transformer encoder layer in T5-11B on an A100 GPU. The execution time exhibits super-linear growth with sequence length.

在数据集粒度按句长排序会影响模型性能

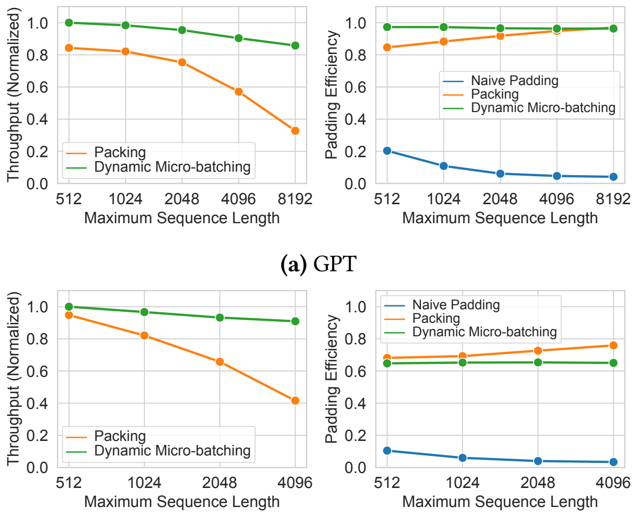
2.2 The current packing solution

To alleviate the padding problem, packing is a common solution, that concatenates multiple short sequences at the sequence dimension to form a single sequence that matches a (predefined) maximum sequence length [30]. Individual sequences that are longer are usually truncated. Cross contamination between samples that are packed into the same sequence may happen, since attention is calculated across unrelated samples, affecting model prediction. Special masks are used during self-attention [18] to prevent it, which zero out the attention scores between unrelated samples. Since the packed sequences have similar lengths, padding is greatly reduced.

However, computation overhead of packed sequences may increase substantially with the maximum sequence length. The computation complexity of self-attention increases quadratically with sequence length [34]. Fig. 3 shows super-linear increase in computation time of a Transformer layer (an LLM is often a stack of Transformer layers) with the sequence length. Training throughput of the whole model is given in Fig. 4, when training GPT [6] and T5 [30] on the FLANv2 dataset. We observe more than 50% throughput decrease when the maximum sequence length increases from 512 to 8192, while the total number of non-padding tokens only increases by 13.2% (due to less truncation).

Padding 导致巨长显著增长，进而影响计算性能

We advocate dynamic micro-batching that adapts the number of micro-batches and micro-batch sizes in each training iteration according to the input data, to efficiently tackle the variable sequence length problem. By grouping samples with similar sequence lengths into the same micro-batch, we reduce the amount of padding needed without introducing unnecessary attention computation as in packing. Fig. 4



(a) GPT

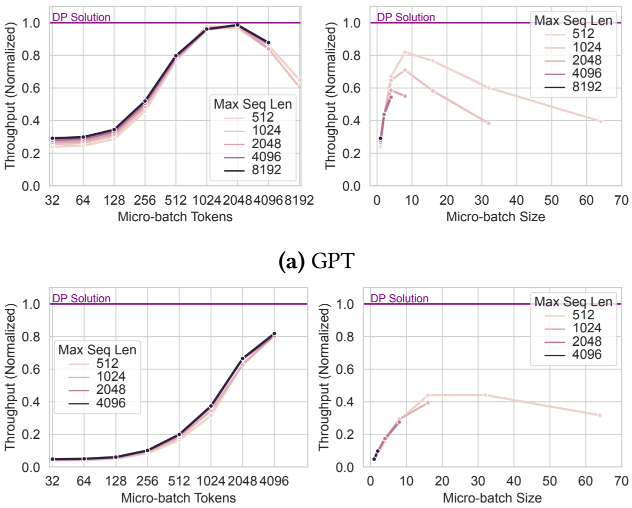
(b) T5

Figure 4. GPT and T5's training performance under packing and dynamic micro-batching.

gives preliminary comparison of the dynamic batching approach (with micro-batches split using our dynamic programming method) with packing under the same settings. The padding efficiency is computed by dividing the non-padding tokens by the total number of tokens processed (padding and non-padding). Dynamic micro-batching achieves comparable padding efficiency as packing and better training throughput, which only slightly drops when the maximum sequence length increases.

2.3 Challenges of dynamic micro-batching

▹ **No principled way to split training mini-batches into micro-batches of different sequence lengths.** Most current pipeline training systems use micro-batches of exactly the same shape: the same number of samples per micro-batch (i.e., the same micro-batch size) and the same sequence length among samples in the micro-batches (padded or packed sequences in case of different sequence lengths). Other possible methods include generating micro-batches of the same token count, so that there are fewer samples in micro-batches of larger sequence lengths. Fig. 5 shows training throughput under the two micro-batching methods. Using a uniform micro-batch size leads to out-of-memory errors (OOMs) when the micro-batch size increases and the maximum sequence length is large; when the maximum sequence length is small, the performance first improves due to the increase in computation efficiency, and then drops because of more padding at larger micro-batch sizes. Equal token count-based micro-batching achieves much better training throughput, while still experiencing OOM before reaching the highest throughput during T5 training.



(a) GPT

(b) T5

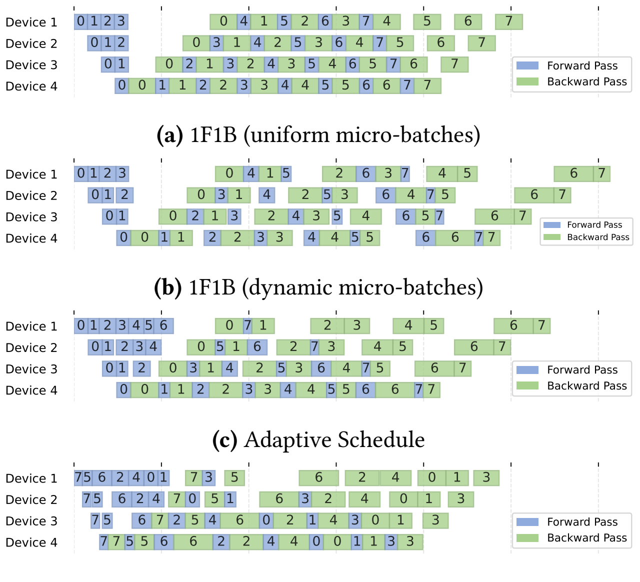
Figure 5. Training performance of GPT and T5 under different micro-batching methods. Left figures show results for token-based micro-batching, and right figures are for micro-batching using fixed micro-batch size. All throughput values are normalized over that achieved by our dynamic programming (DP)-based micro-batching method.

We observe that the choice of micro-batch size or the token number in the two methods greatly affects the training throughput. We design an efficient dynamic programming-based algorithm to decide optimal micro-batching in each training iteration, that strikes a good trade-off between padding efficiency, computation efficiency and memory consumption.

▹ **No efficient pipeline schedules for micro-batches of diverse execution times.** Most existing pipeline schedules (e.g., 1F1B [25]) assume identical execution time of micro-batches, and schedule micro-batch processing over consecutive stages tightly one after another (Fig. 6a). In this way, any variation in micro-batches' execution time may cause blocking (computation/communication waiting for communication/computation), creating more "bubbles" in the training pipeline (Fig. 6b).

To further quantify the effect of variable micro-batch execution time on 1F1B schedule, we randomly disturb the execution time of the micro-batches (assumed to be uniform originally) by noises from a zero-mean Gaussian distribution. Fig. 7 shows that the per-iteration training time with 1F1B schedule grows rapidly as the variation level increases, especially when there are more pipeline stages.

We present a scheduling algorithm that is more robust to dynamic micro-batches, as shown in Fig. 7. However, to achieve a higher throughput, it consumes more memory than 1F1B. We make the algorithm memory-aware, so it achieves higher throughput than 1F1B when memory is abundant,


Device 1 0 1 2 3 0 4 1 5 2 6 3 7 4 5 6 7
Device 2 0 1 2 0 3 1 4 2 5 3 6 4 7 5 6 7
Device 3 0 1 0 2 1 3 2 4 3 5 4 6 5 7 6 7
Device 4 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7
Forward Pass
Backward Pass
Device 1 0 1 2 3 0 4 1 5 2 6 3 7 4 5 6 7
Device 2 0 1 2 0 3 1 4 2 5 3 6 4 7 5 6 7
Device 3 0 1 0 2 1 3 2 4 3 5 4 6 5 7 6 7
Device 4 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7
Forward Pass
Backward Pass
Device 1 0 1 2 3 4 5 6 0 7 1 2 3 4 5 6 7
Device 2 0 1 2 3 4 0 5 1 6 2 7 3 4 5 6 7
Device 3 0 1 2 0 3 1 4 2 5 3 6 4 7 5 6 7
Device 4 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7
Forward Pass
Backward Pass
Device 1 7 5 6 2 4 0 1 7 3 5 6 2 4 0 1 3
Device 2 7 5 6 2 4 7 0 5 1 6 3 2 4 0 1 3
Device 3 7 5 6 7 2 5 4 6 0 2 1 4 3 0 1 3
Device 4 7 7 5 5 6 6 2 2 4 4 0 0 1 1 3 3
Forward Pass
Backward Pass

(a) 1F1B (uniform micro-batches)

(b) 1F1B (dynamic micro-batches)

(c) Adaptive Schedule

(d) Adaptive Schedule & Micro-batch Reordering

Figure 6. Pipeline training under dynamic micro-batching (except 6a) with different pipeline schedules.

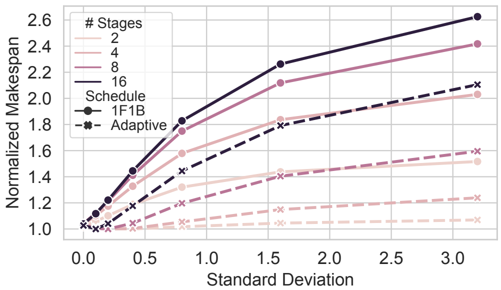
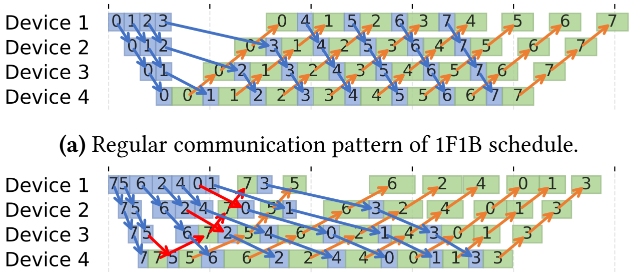


Figure 7. Per-iteration training time (makespan) of different pipeline schedules under different variation levels of micro-batch execution time. X axis is the standard deviation of the introduced variation (zero-mean Gaussian). The makespan is normalized over the no variation case.


Device 1 0 1 2 3 0 4 1 5 2 6 3 7 4 5 6 7
Device 2 0 1 2 0 3 1 4 2 5 3 6 4 7 5 6 7
Device 3 0 1 0 2 1 3 2 4 3 5 4 6 5 7 6 7
Device 4 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7
Device 1 7 5 6 2 4 0 1 7 3 5 6 2 4 0 1 3
Device 2 7 5 6 2 4 7 0 5 1 6 3 2 4 0 1 3
Device 3 7 5 6 7 2 5 4 6 0 2 1 4 3 0 1 3
Device 4 7 7 5 5 6 6 2 2 4 4 0 0 1 1 3 3

(a) Regular communication pattern of 1F1B schedule.

(b) Irregular and dynamic communication pattern of dynamic pipelines. Red arrows indicate communications which can trigger deadlock if not scheduled correctly (only one shown for each pair of devices).

Figure 8. Difference in communication pattern between 1F1B (with uniform micro-batches) and DynaPipe schedules. Green (blue) blocks denote backward (forward) computation. Blue and orange arrows indicate the communication of activations during forward pass and gradients during the backward pass, respectively.

while automatically limiting memory consumption under high memory pressure.

▹ **Improper communication order between pipeline stages may lead to deadlocks in dynamic pipelines.** Current pipeline systems send an intermediate tensor to the next stage right after its production, and launch the receive operation of the tensor just before it is used, when applying 1F1B schedule. Since there is no gap between consecutive execution stages of a micro-batch in 1F1B schedule, the send and receive ops naturally align in time. As shown in Fig. 8a, each crossing of arrows (a pair of sends in reverse direction) can be implemented by a fused communication operator. However, our pipeline schedule is different for each iteration and can produce irregular communication patterns where execution of consecutive stages of the same micro-batch are scheduled far apart (Fig. 8b), causing deadlocks for naïve communication schedule (i.e., start sending whenever the result of a stage is ready and start receiving whenever previous stage's result is needed). For example, the uppermost red arrows in Fig. 8b shows device 1 sending the activation of micro-batch 0 to device 2, while at the same time, device 2 is trying to send the gradient of micro-batch 7 to device 1. However, under naïve schedule, device 1 will continue to send the activation of micro-batch 1 to device 2 before launching a corresponding receive for micro-batch 7. Since only one communication operation can happen between each pairs of devices (required by libraries like NCCL [27]), this creates a communication order mismatch thus can result in deadlocks (fusing of communication ops like in 1F1B scheduling is also infeasible due to the extra sending of the activation of micro-batch 1).

这里的说法是不对的，1f1b 就会存在所谓的deadlock情况

We reorder the send and receive operations to resolve deadlocking. For example, we can make Device 1 receive micro-batch 0's activation before it sends micro-batch 7's gradient, and make Device 1 receive micro-batch 7's gradient before it sends micro-batch 1's activation. Given a pipeline schedule, we schedule both send and receive operators together when an intermediate tensor is produced, ensuring the communication order is consistent across different stages.

将同一个 tensor 的 send/recv schedule 在一起，实际 torch 有这种面向 deadlock 情况的通信原语

3 DynaPipe Overview

We propose DynaPipe to enable efficient pipeline training of multi-task models with dynamic micro-batching. DynaPipe comprises of two main modules: (1) *Planners* that run on CPUs, perform optimization and generate *execution plans* for each training iteration; (2) *Executors* that retrieve and

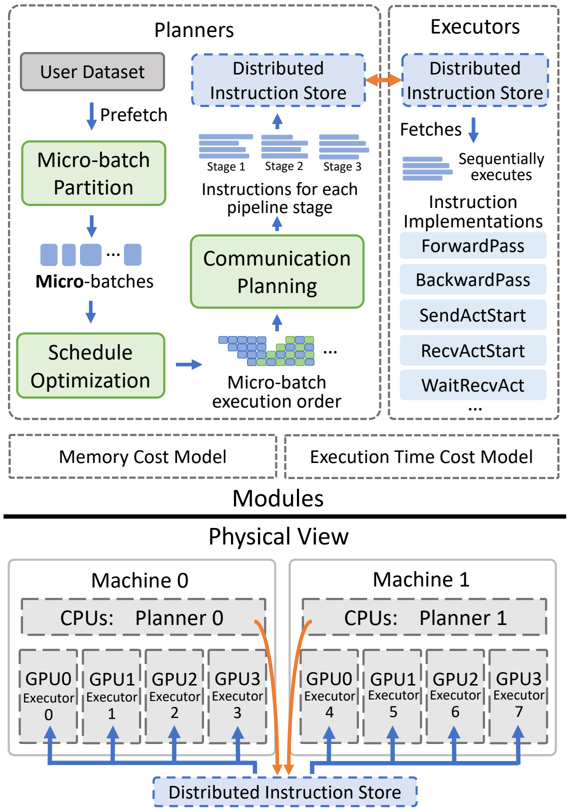


Figure 9. System architecture of DynaPipe.

execute the assigned execution plans on the GPUs. A system overview of DynaPipe is given in Fig. 9. We detail the components and terminologies used in our design as follows.

Execution plans specify micro-batch splitting, pipeline execution schedule, the communication order and the shape of all communicated tensors on each executor (GPU). They are represented as sequences of *pipeline instructions*, following the design principle of DeepSpeed [22]. The *pipeline instructions* include *ForwardPass, BackwardPass*, which executes the forward/backward computation for a micro-batch, *SendAct, RecvAct, SendGrad, RecvGrad*, which sends/receives model activation/gradients. These *pipeline instructions* abstract key operations (ops) in pipeline-parallel training. We further split every type of communication into two conjugate pipeline instructions: a *Start* op (e.g., *RecvActStart*), which launches the communication into an asynchronous GPU communication stream, and a *Wait* op (e.g., *WaitRecvAct*), which adds a dependency between the communication stream and the computation stream, allowing computation to wait for the result of communication. The instruction abstraction allows flexible pipeline scheduling and communication planning.

Planners pre-fetch training data from user-provided dataset. For each mini-batch of training samples, a planner splits the mini-batch into micro-batches using our dynamic programming algorithm (§4). It then generates an optimized pipeline execution schedule (§5) and decides the appropriate execution order of communication between pipeline stages (§6), along with the shapes of the communicated tensors. All the above decisions are compiled into an *execution plan* and pushed to a distributed instruction store in the host memory of one of the machines (e.g., machine 0), ready to be fetched by corresponding executors.

Executors retrieve execution plans from the instruction store and executes the pipeline instructions in the order specified in the execution plans using the underlying deep learning framework (e.g., Megatron-LM [26] and PyTorch [29]).

Cost models estimate the execution time and memory consumption of a single layer of the model executing a micro-batch under different micro-batch sizes and sequence lengths on a single GPU. They are used to guide all decisions in the planners. To construct these cost models, we run memory consumption and execution time profiling for both forward and backward passes under different combinations of micro-batch size and sequence length at power-of-two intervals (e.g., micro-batch size of 1, 2, 4, etc, and sequence length of 32, 64, 128, etc). For training with only data and pipeline parallelism, single-GPU profiling is sufficient since the communication cost is constant (under different micro-batch size and sequence lengths) for data parallelism and very small for pipeline parallelism. Tensor parallelism profiling runs multi-GPUs to capture the significant communication cost. Linear interpolation is used to bridge the gaps between sampled data points. We show that this simple cost modeling suffices to provide good estimation of execution time and peak memory consumption for training iterations in Sec. 8.6.

估计每个layer 不同输入时的执行时间和memory

To hide the plan generation overhead, we overlap model execution and the execution plan generation of future iterations in DynaPipe, as planners and executors run on different hardware resources. We exploit the abundance of CPU cores to parallelize plan generation on a machine. When the training is run on multiple machines, DynaPipe distributes execution plan generation of distinct training iterations to different machines.

4 Micro-batch Construction

In each training iteration, micro-batching samples of different sequence lengths should attend to the trade-offs among padding, computation time, memory consumption and pipeline bubble size, for throughput maximization without OOM. We present an algorithm to optimize micro-batch partitioning.

Model the iteration time under pipeline parallelism. We group a set of N input sequences (samples), $\mathbf{S}$, in the current training iteration into a set of micro-batches, $\boldsymbol{\pi} = \{\mathbf{M}_1, \mathbf{M}_2, \cdots, \mathbf{M}_m\}$, where $\mathbf{M}_i \subseteq \mathbf{S}$ represents a micro-batch and $\mathbf{M}_i$'s are disjoint. Let $t_f(\mathbf{M}_i)$ ($t_b(\mathbf{M}_i)$) be the forward (backward) pass execution time of micro-batch i obtained from the cost model, and let $t(\mathbf{M}_i) = t_f(\mathbf{M}_i) + t_b(\mathbf{M}_i)$. Let c be the number of pipeline stages. The execution time of

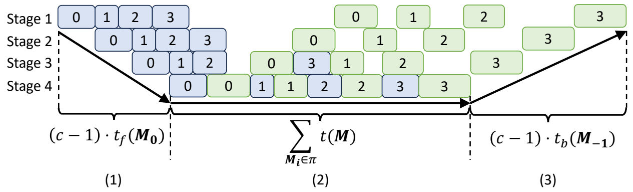

Stage 1
Stage 2
Stage 3
Stage 4
0 1 2 3
$(c-1)\cdot t_f(\boldsymbol{M}_0)$
$\sum_{M_i\in\pi} t(\boldsymbol{M})$
$(c-1)\cdot t_b(\boldsymbol{M}_{-1})$
(1)
(2)
(3)

Figure 10. Approximation of pipeline execution time. c: the number of pipeline stages, $c = 4$ in this example. $\mathbf{M}_0$, $\mathbf{M}_{-1}$: the first and the last micro-batch. $t_f(\mathbf{M})$, $t_b(\mathbf{M})$: the forward and backward execution time of micro-batch $\mathbf{M}$. $t_f(\mathbf{M}) + t_b(\mathbf{M}) = t(\mathbf{M})$.

1F1B pipeline 的端到端时间建模

the entire pipeline can be modelled in three parts (Fig. 10): (1) the time for the forward pass of the first micro-batch ($\mathbf{M}_0$) to reach the last stage, $(c-1)\cdot t_f(\mathbf{M}_0)$; (2) the execution time of all micro-batches on the last stage, $\sum_{\mathbf{M}_i\in\pi} t(\mathbf{M})$; (3) the time for the backward pass of the last micro-batch ($\mathbf{M}_{-1}$) to reach the first stage, $(c-1)\cdot t_f(\mathbf{M}_{-1})$. However, under dynamic micro-batching where the execution time of micro-batches differs, $t_f(\mathbf{M}_0)$ and $t_b(\mathbf{M}_{-1})$ is determined by the exact pipeline schedule which we do not know in advance. Therefore we model $t_f(\mathbf{M}_0)$ and $t_b(\mathbf{M}_{-1})$ using the execution time of longest micro-batch, so the total execution time of part (1) and (3) can be approximated as $(c-1)\cdot \max\{t(\mathbf{M_i})|\mathbf{M_i}\in\boldsymbol{\pi}\}$. The iteration time is thus $t_{iter} = (c-1)\cdot\max\{t(\mathbf{M_i})|\mathbf{M_i}\in\boldsymbol{\pi}\} + \sum_{\mathbf{M_i}\in\boldsymbol{\pi}} t(\mathbf{M_i})$. We seek to derive the best micro-batch assignment $\boldsymbol{\pi}$ that minimizes the iteration time, i.e., maximizes the training throughput:

$$\min_{\boldsymbol{\pi}}\left\{(c-1)\cdot\max\{t(\mathbf{M_i})|\mathbf{M_i}\in\boldsymbol{\pi}\} + \sum_{\mathbf{M_i}\in\boldsymbol{\pi}} t(\mathbf{M_i})\right\} \quad \text{(Eq1)}$$

Determine the order of samples. The problem of assigning samples into disjoint sub-sets (micro-batches) while optimizing an objective (throughput) belongs to the family of set partitioning problems (SPP), which is NP-hard [17]. We simplify the problem by ordering the samples first and then grouping consecutive samples into micro-batches using a dynamic programming (DP) approach.

For sample ordering, a natural intuition is that *to minimize padding, micro-batches should contain samples with similar sequence lengths*. For decoder-only models (e.g. GPT [6]), sorting the samples according to their sequence lengths suffices. For encoder-decoder models like T5 [30] with multiple input sequences (i.e. a input sequence processed by encoder, and a target sequence fed into the decoder), we can sort the samples first by the length of the input sequence and then by the target sequence. Alternatively, we can take the pair of input and target sequence lengths as a 2D point, and find a visiting order that minimizes the sum (or maximum) of distances between adjacent points. This can be solved by an off-the-shelf Travelling Salesmen Problem solver. We compare the two methods to order samples in Sec. 8.4.

Partition ordered samples with dynamic programming. Now we have an ordered list of samples $\mathrm{S} = [\mathbf{s_1}, \mathbf{s_2}, \cdots, \mathbf{s_N}]$. We construct a DP algorithm to optimally partition the list. Let $\boldsymbol{\pi}^*_{\mathrm{S[:n]}}$ represent the optimal partition of the first n samples in S, which minimizes the total execution time of the resulting micro-batches. Let $f(n; t_{max}) = \sum_{\mathbf{M_i}\in\boldsymbol{\pi}^*_{\mathrm{S[:n]}}} t(\mathbf{M_i})$, where t_{max} denotes the maximum micro-batch execution time. We have

$$f(n; t_{max}) = \min_{1\le i\le n-1}\{f(i; t_{max}) + t(\mathbf{M}_{\mathrm{S[i+1:n]}})|t(\mathbf{M}_{\mathrm{S[i+1:n]}}) \le t_{max}\} \quad \text{(Eq2)}$$

where $\mathbf{M}_{\mathrm{S[i+1:n]}}$ denotes the micro-batch that is constructed from samples $\mathbf{s_{i+1}}, \mathbf{s_{i+2}}, \cdots, \mathbf{s_n}$. To find the best micro-batch partitions minimizing (Eq1), we only need to find t_{max} that minimizes $(c-1)\cdot t_{max} + f(N; t_{max})$. There are $O(N^2)$ unique possible t_{max} values, since there are at most $\frac{N(N+1)}{2}$ ways to construct a single micro-batch by consecutively partitioning $\mathbf{S}$. For each t_{max}, finding $f(N; t_{max})$ takes $O(N^2)$ steps. Therefore, the computation complexity of this DP approach is $O(N^4)$. With input sequences in multi-task model training, many t_{max} values are very similar to each other. We may greatly speed up the DP algorithm by sampling t_{max} at fixed intervals (in our evaluation, we only consider possible t_{max} values $5us$ apart from each other).

Balance data parallel model replicas. The above algorithm splits an input mini-batch into micro-batches for execution in a single pipeline. When the pipeline training is combined with data parallelism, the micro-batches should also be distributed among different data-parallel model replicas, balancing the execution time between all pipelines. We extend our micro-batching algorithm to handle hybrid data and pipeline parallel training. Let $\boldsymbol{\pi_d} \subseteq \boldsymbol{\pi}$ be the collection of micro-batches for model replica d. The iteration time under hybrid data and pipeline parallelism becomes

$$t^{dpp}_{iter} = \max_d\left\{(c-1)\cdot\max\{t(\mathbf{M_i})|\mathbf{M_i}\in\boldsymbol{\pi_d}\} + \sum_{\mathbf{M_i}\in\boldsymbol{\pi_d}} t(\mathbf{M_i})\right\},$$

which denotes the maximum execution time across all data parallel model replicas. We minimize its upper bound

$$(c-1)\cdot\max\{t(\mathbf{M_i})|\mathbf{M_i}\in\boldsymbol{\pi}\} + \max_d\left\{\sum_{\mathbf{M_i}\in\boldsymbol{\pi_d}} t(\mathbf{M_i})\right\}$$

in our micro-batching. The first term is the same as that in (Eq1), and the second term is the maximum total micro-batch execution time among model replicas. Minimizing the second term is not easy, since it requires solving another subset partition problem, which is NP-hard [17]. We approximate the second term using the tight lower bound $\frac{1}{|\mathrm{D}|}\sum_{\mathbf{M_i}\in\boldsymbol{\pi}} t(\mathbf{M_i})$, which is achieved when the total micro-batch execution time is equal across all model replicas. Then, the approximated objective to minimize becomes

$$\min_{\boldsymbol{\pi}} \left\{ (c-1)\cdot \max\{t(\mathbf{M_i})|\mathbf{M_i}\in\boldsymbol{\pi}\} + \frac{1}{|\mathbf{D}|}\sum_{\mathbf{M_i}\in\boldsymbol{\pi}} t(\mathbf{M_i}) \right\}.$$

Comparing it with (Eq1), the only difference lies in the constant $\frac{1}{|\mathbf{D}|}$ in the second term. Therefore, we can optimize this objective using the same DP algorithm: we first solve for micro-batch partitioning to minimize the above objective, and then partition resulting micro-batches among data parallel replicas to minimize $\max_d \left\{\sum_{\mathbf{M_i}\in\boldsymbol{\pi}_d} t(\mathbf{M_i})\right\}$, by (approximately) solving the subset partition problem using the Karmarkar–Karp algorithm [16].

不同 stages 的 memory 存储需求不一样，导致一个 micro-batch 在不同 stages 的计算上 memory limit 不同，这会导致 DP 的最优子结构失效

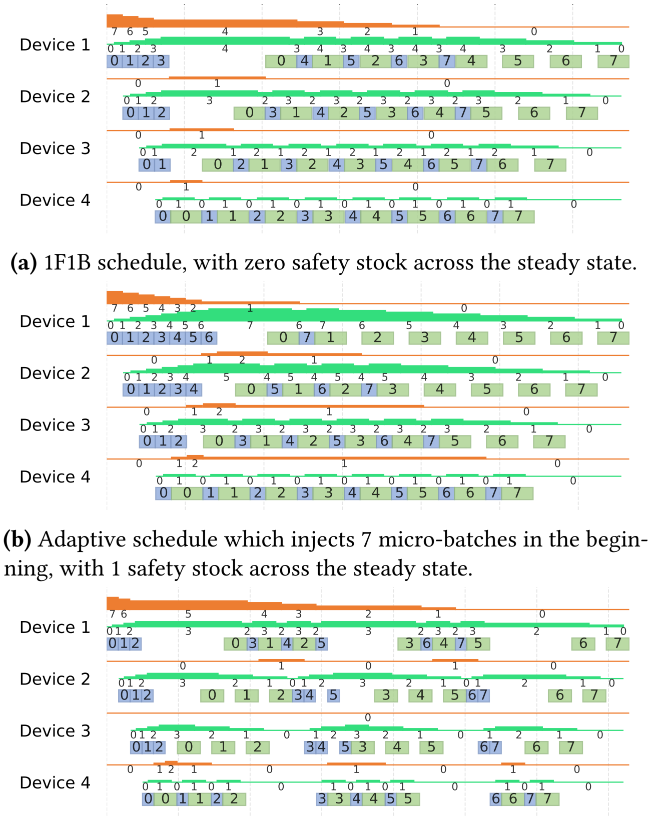
Limit memory consumption. In pipeline parallel training, the peak memory consumption (in 1F1B scheduling) is determined by the maximum accumulated activation during a sliding window of c micro-batches, since on the nth device, it executes $c+1-n$ forward passes, followed by regular backward-forward cycles. Such a sliding window introduces dependency between micro-batching decisions, which destroys the optimal substructure property of DP. Therefore, we resort to limiting the memory consumption of every micro-batch $\mathbf{M}_{\mathbf{S}[\mathbf{i+1:n}]}$ during DP, i.e., only considering $\mathbf{M}_{\mathbf{S}[\mathbf{i+1:n}]}$s that do not violate memory limit in (Eq2). The per-micro-batch memory limit is related to the pipeline schedule. In 1F1B schedule, the per-micro-batch memory limit is set to $\frac{1}{c}$ of the device memory limit. In the next section, we show different schedules with different such factors ranging from 1 to $\frac{1}{m}$ (m is the number of micro-batches).

5 Pipeline Execution Schedule

We analyze pipeline execution performance under dynamic micro-batching and propose pipeline schedules for better throughput under non-uniform micro-batch execution time.

Analysis of pipeline execution performance. Let op $f_{i,j}$ ($b_{i,j}$) denote the forward (backward) computation of micro-batch i on the jth device. We associate a virtual buffer with each device, containing the ops ready for execution on it (i.e., whose proceeding forward/backward stages have been completed). Ops are removed from this buffer when its execution on the device starts. Whenever device j has executed an op $f_{i,j}$ or $b_{i,j}$, the op of the next stage, $f_{i,j+1}$ ($b_{i,j}$ for the final forward stage) or $b_{i,j-1}$ is added to the buffer of the corresponding device ($j+1$ or $j-1$). Ops in this buffer are called safety stocks of that device, in the scheduling literature [5]. To prevent device idling, it is essential to maintain non-empty safety stocks when the device has executed an op and is ready for the next. In 1F1B scheduling shown in Fig. 11a, only the first stage has 7 safety stocks initially since all micro-batches are ready for execution (no dependencies on prior stages); its safety stock gradually depletes as more micro-batches are executed. For all other stages, the number of safety stocks is zero throughout the steady states (the period where forwards and backwards are closely packed and follows strict one-forward-one-backward order). This is because 1F1B schedules consecutive stages closely together without any gap time in between. Whenever the previous stage finishes executing an micro-batch (and thus the added to the safety stock of the current stage), the current stage will immediately start the computation of this micro-batch, resulting in a net zero change in safety stock level. With zero safety stocks, any deviation in micro-batch execution time will result in device idling.

只在 steady phase 时满足



(a) 1F1B schedule, with zero safety stock across the steady state.

(b) Adaptive schedule which injects 7 micro-batches in the beginning, with 1 safety stock across the steady state.

(c) Adaptive schedule with peak memory limited to 3 micro-batch activations. Micro-batch 3 and 6 are delayed due to memory limit.

Figure 11. The trade-off between safety stock size (orange row for each device) and memory consumption (green row) under different pipeline schedules.

Schedules robust for dynamic micro-batching. We seek a pipeline execution schedule that maintains more safety stocks at each device. The problem of micro-batch execution scheduling can be viewed as a special type of the re-entrant flow shop problem [13] (i.e., scheduling jobs onto machines where each job follow the same process route through the machines; a machine can be used more than once by a job), since all our micro-batches passes through the devices following the same routine (perform forward computation once on each stage, then follow the reverse route backward).

Cyclic scheduling is an algorithm that has demonstrated commendable performance in solving re-entrant flow shop problems [5]. Under cyclic scheduling, execution on each device is divided into cycles; in each cycle, each device tries to execute exactly one forward pass and one backward pass of any micro-batch. Each device maintains two buffers of ready ops (forward and backward), and fetches an op to execute from each buffer in each cycle. If no ops are available, the corresponding forward or backward pass is skipped. Like 1F1B schedule, cyclic schedule will interleave forward and backward pass of micro-batches during the course of scheduling. However, unlike 1F1B schedule which fixes the execution order of all micro-batches regardless of their execution time and memory consumption, cyclic schedule provides us with a systematic way of controlling when micro-batches should be injected into the pipeline during the scheduling process (We can mark all micro-batches as not "ready" on the first stage during initialization. To inject a micro-batch, we insert it into the buffer of ready forward ops on the first device). We refer to such micro-batch-injection-regulated cyclic schedules as adaptive scheduling, since the injection time can be adjusted adaptively for different input micro-batches.

Adaptive scheduling 就是可以根据 input 构造出来的 micro batches 来动态调整 micro batch 顺序

Micro-batch injection time in turn affects the level of safety stocks. For example in Fig. 11b, if we inject more micro-batches into the pipeline at the beginning, we raise the number of safety stock to one at each device during the steady state. This means at least one micro-batch is ready for execution for each device, therefore, a device will not idle even if the previous stage does not produce the activations in time (e.g., when executing a large micro-batch). The increase in safety stock level leaves room for variations in execution time of the micro-batches.

Optimizing trade-off between time and memory. Injecting more micro-batches also increases the memory consumption, since devices need to accumulate more activations in memory. In Fig. 11b, since 7 micro-batches are injected at the beginning of the schedule (compared to 4 in 1F1B), the activation memory of maximum 7 micro-batches needs to be accumulated. Conversely, we can reduce memory consumption by delaying micro-batch injection until previously accumulated activations are freed up by the backward pass. In Fig. 11c, we delay the injection of memory-consuming micro-batches 3 and 6 until backward pass of micro-batches 0 to 2 and 3 to 5 have been executed, freeing up more activation memory. This reduces the peak accumulated activation to 3 micro-batches.

Memory-aware adaptive scheduling algorithm. To maximize throughput while limiting peak memory consumption, we dynamically decide the injection or delayed execution of micro-batches in the pipeline. We give our memory-aware adaptive scheduling in Alg. 1. During execution scheduling, each device keeps track of the current memory consumption (lines 9, 15). On scheduling a forward pass of a micro-batch, if memory consumption exceeds the device memory limit, the device will skip forward passes until backward passes have freed up enough memory to avoid OOM (line 14). In this way, the training can continue without OOM as long as the activation of *one* single micro-batch fits into device memory.

Algorithm 1: Memory-aware Adaptive Scheduling

```
Inputs  : $C$ - the number of devices (stages), $M$ - the
          number of micro-batches, $a_{i,j}$ - activation
          memory of micro-batch $i$ on the $j$th device;
          $l_j$ - memory limit of device $j$
Outputs : $O_j$ - micro-batch execution order on device
          $j$
1  $O_j \leftarrow [], \forall j \in [1, C]$
   /* Op buffers $S_j^f, S_j^b$, current memory $m_j$ for device $j$ */
2  $S_j^f, S_j^b \leftarrow [], m_j \leftarrow 0, \forall j \in [1, C]$
   /* Initialize forward buffer on device 1 */
3  $S_1^f \leftarrow [a_{1,1}, a_{2,1}, \ldots, a_{M,1}]$
4  while $\exists j$ s.t. $S_j^f \neq []$ or $S_j^b \neq []$ do
       /* New ops unlocked this cycle */
5      $N_j^f, N_j^b \leftarrow [], \forall j \in [1, C]$
6      foreach device $j$ do
7          if $S_j^b \neq []$ then                 // Schedule a backward op
8              $a_{i,j} = S_j^b$.pop(0)           // get first op in buffer
9              $m_j$ -= $a_{i,j}$                          // update memory
10             $O_j$ += $(i, \text{'B'})$                  // record op order
11             append next stage op to corresponding
                 $N_{j^*}^b$, if exists
12         if $S_j^f \neq []$ then                  // Schedule a forward op
13             $a_{i,j} = S_j^f$.pop(0)           // get first op in buffer
14             if $m_j + a_{i,j} < l_j$ then
15                 $m_j$ += $a_{i,j}$                      // update memory
16                 $O_j$ += $(i, \text{'F'})$              // record op order
17                 append next stage op to the
                     corresponding $N_{j^*}^f$ or $N_{j^*}^b$, if exists
18             else
19                 $S_j^f$.prepend($a_{i,j}$)
20     $S_j^f \leftarrow S_j^f + N_j^f, S_j^b \leftarrow S_j^b + N_j^b$
```

Micro-batch ordering. Our memory-aware adaptive schedule algorithm assumes an ordered list of micro-batches as input. The injection order of micro-batches could also impact throughput due to variations in micro-batch execution time. Owing to the complexity of the scheduling problem, modeling this effect, let alone optimizing it, is considerably challenging. We address this issue by clustering the micro-batches by predicted execution time (using our cost model), assuming that micro-batches with similar execution time should be scheduled in proximity, and permute execution

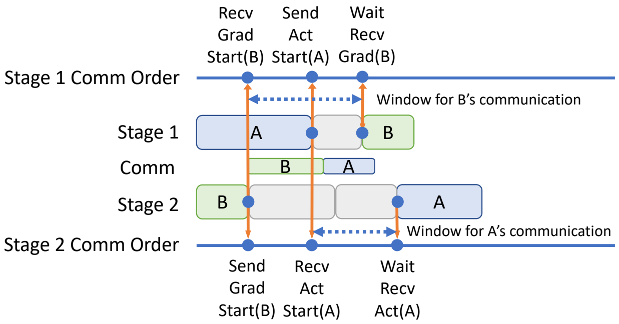


Figure 12. Planning the order of send and receive operations using simulated execution timeline.

order of the resulting clusters to find the best order attaining highest throughput. We find that merely 3 or 4 clusters are adequate to attain satisfactory performance (i.e., throughput increase is insignificant when further increasing the number of clusters) through empirical study.

6 Communication Planning

Given the micro-batches and their execution order in a training iteration, we further generate a communication plan specifying the order of all communication between pipeline stages (i.e., sending activation to the next stage and receiving gradient from the previous stage for each micro-batch at each stage). To avoid deadlocking, we need to make sure that all pairs of sends and receives are executed in the same order on adjacent stages. We guarantee this by scheduling *both* send and receive operations together at the time where the tensor to be sent is generated.

Specifically, we simulate a device computation timeline using the pipeline schedule generated in Sec. 5 and the micro-batch execution time cost model, as shown in Fig. 12. Then, we iterate through operations (ops) in the timeline (forward and backward of micro-batches) in ascending order of their end time, while maintaining a queue for each pipeline stage to record the communication order. Upon processing each op, we push the corresponding send *Start* op (*SendActStart* for forward passes, and *SendGradStart* for backward passes) into the queue of the executing stage. At the same time, we also insert the matching receive *Start* op into the receiver's queue. The leftmost two vertical orange lines in Fig. 12, demonstrates this process.

Wait 是用
来让 comp
等待 comm
完成

To minimize blocking, we schedule the corresponding *Wait* ops as late as possible, i.e. we only add *Wait* ops before computations that consume the received tensor (the last two orange lines in Fig. 12). This maximizes the time window in which communication can execute without blocking computation.

The pipeline execution and communication schedule is specified as a sequence of instructions for each device. We then calculate the shapes of the communicated tensors based on the shape of input micro-batches and the model architecture, and include the shapes in the generated instructions to avoid exchanging them at runtime.

7 Implementation and Other Optimizations

DynaPipe is implemented using 10K LoC in Python, with additional 500 LoC in C++ for accelerating the DP algorithm. We use Redis [21] as our distributed *instruction store*. Communication in pipeline training is implemented based on PyTorch's distributed communication package with NCCL [27] backend. We implement the set of *instructions* in around 400 LoC in Megatron-LM [26] with PyTorch nightly version 2.1.0.dev20230322+cu117. We further enable ZeRO [31] optimizer by integrating Megatron-LM with DeepSpeed [22] version 0.9.1 since it's often used together with data parallelism. We adopt the Megatron-LM's implementation for data and tensor parallelism while apply DynaPipe to replace its pipeline modules, enabling training with mixed 3D parallelism.

To leverage DynaPipe with Megatron-LM, users can directly reuse existing training scripts with additional arguments specifying configurations for DynaPipe (e.g., the device memory limit and the number of CPU cores to use during planning). To extend DynaPipe to frameworks other than Megatron-LM, users only need to implement the *pipeline instructions* in the framework.

Dynamic recomputation. Activation checkpointing (recomputation) [7] is a widely-used technique to reduce memory consumption during DNN training, by recomputing the activations during backward pass instead of storing them. However, they come with extra computation cost. There are also multiple ways to apply recomputation (e.g., [26]), resulting in different trade-offs between training throughput and memory consumption. Under dynamic micro-batching, the peak memory consumption in different training iterations varies; thus we dynamically decide the best recomputation scheme for each iteration (i.e., the one with the least computation overhead without triggering OOM). This is achieved by repeating scheduling and micro-batch partitioning under different assumptions for recomputation method (using different cost-models).

Reducing memory fragmentation. Dynamic tensor shapes exacerbate the pressures to caching memory allocators (e.g., in PyTorch) since their dynamic memory requirement causes frequent cache misses. Under memory pressure, we sometimes observe blocking *cudaMalloc*'s and *cudaFree*'s during training, which are caused by the allocator failing to find a usable memory block and PyTorch's effort to defragment GPU memory. To reduce training slow-down caused by these blocking operations, we instruct PyTorch to use a single unified memory pool to manage all CUDA memory, and pre-allocate all GPU memory into the pool before training starts.

This eliminates the need for allocating (and freeing) CUDA memory during runtime.

8 Evaluation

Testbed Set-up We conduct our experiments in a cluster of 4 Amazon EC2 p4d.24xlarge instances (32 GPUs in total). Each p4d node is equipped with 8 NVIDIA A100 (40GB) GPUs and 96 vCPU cores. NVSwitch connects the GPUs within each node, and the nodes are connected by a 400Gbps network with EFA [3] enabled.

DNNs We evaluate DynaPipe by training two popular LLM models: GPT [6] (decoder-only architecture) and T5 [30] (encoder-decoder architecture). We evaluate each model on four different cluster sizes, with the model size scaling accordingly. For GPT, we scale the model parameters following the configuration in the GPT-3 paper [6]. For T5, since T5-11B is already the largest model specified in its paper and its hidden size in feed forward layers is huge, we simply scale the number of layers. We list the detailed model specifications in Table 1.

Dataset We use the zero-shot version of the FLANv2 [20] dataset in our experiments, which consists of 1836 different tasks and is one of the largest public multi-task training data collections. The full dataset contains 15M training samples. To reduce evaluation costs, we randomly down-sample it to 100K samples. All our metrics reported are collected during one epoch of training on the down-sampled dataset.

Baselines We use Megatron-LM integrated with DeepSpeed (*MLM+DS*) as the training system baseline, which implements packing (i.e., pack multiple sample into the same sequence so the resulting sequence length matches the specified maximum sequence length). For each experiment, we grid search through common 3D parallelism combinations (power of twos in each of the data, tensor and pipeline parallel dimensions, with tensor parallelism limited to intra-node only) for both baselines and DynaPipe to use on the given cluster configuration. DynaPipe implements 3D parallelism by reusing the data and tensor parallel implementation of Megatron-LM, but replacing its pipeline modules with our *executor*. For baselines, we grid search common combinations of micro-batch size and activation checkpointing strategy, and report the best results.

Metrics We report the system throughput in terms of actual tokens processed, which does not count the padding tokens. Specifically, throughput is calculated by dividing the total number of tokens in the training dataset by the amount of time needed for one epoch of training.

8.1 Throughput under sequence length scaling

We fix the global batch size (i.e., size of the input mini-batch in each training iteration, to be divided among devices if data parallelism is used) to 65536 tokens and vary the maximum sequence length allowed in the dataset (sequences that are longer are truncated). Since the baseline (*MLM+DS*) and DynaPipe may achieve maximum throughput under different grid-searched parallelisms, we also evaluate the throughput of the baseline when it uses the same parallelism configuration as DynaPipe (*MLM+DS (c)*).

In Fig. 13, we observe that in most cases, the throughput of *MLM+DS* decreases rapidly as maximum sequence length scales up, due to the super-linear relationship between computation time and maximum sequence length (Fig. 3). DynaPipe dynamically decides the sequence lengths in micro-batches, and its performance is determined more by the average sequence length than the maximum. As a result, while we still see throughput decrease in DynaPipe as maximum sequence length increases (since we allow longer sequence to exist without truncation), the decrease is much less than that of *MLM+DS*.

On T5 (16 and 32 GPUS), DynaPipe scales to higher sequence lengths than baselines. This is because our memory-aware adaptive schedule can dynamically adjust the schedule to accommodate memory limit (Fig. 11c), achieving a lower peak memory consumption than 1F1B which is used by baselines. In many cases (e.g. T5 on 8 GPUs), we also find the MLM+DS under-performs using the optimal parallelism for DynaPipe. This indicates that the performance characteristic is very different for DynaPipe and packing-based baselines.

8.2 Throughput under global batch size scaling

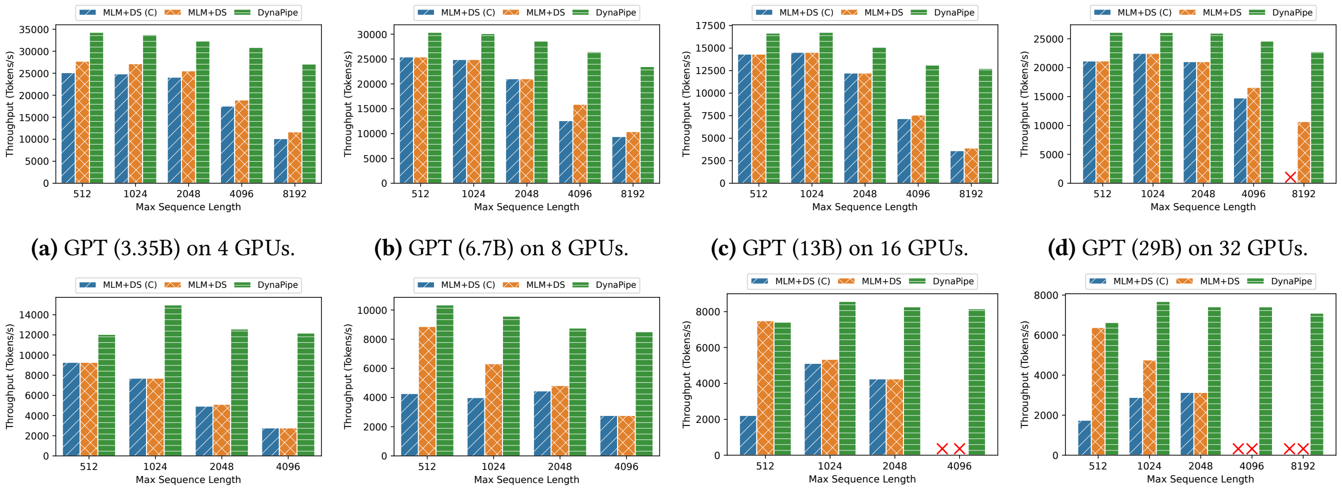
In Fig. 14, we set the maximum sequence length to 2048 and adjust the global batch size. For the GPT model, both MLM+DS and DynaPipe's performance increases with global batch size, since larger global batch sizes reduce the synchronization frequency of data parallelism and the size of the pipeline bubble in pipeline parallelism. We also observe the performance of DynaPipe increase faster than MLM+DS, since DynaPipe also benefits from the increased opportunity to optimize micro-batch splitting when global batch size is large. For T5 model, the baselines make and extensive use of tensor parallelism whose performance is less affected by the global batch size. While DynaPipe uses higher degree of pipeline parallelism, therefore still see the performance improvement when global batch size scales up.

8.3 Padding efficiency

For GPT models, both packing and our dynamic micro-batching can achieve a high padding efficiency (>0.8, Fig. 15a), with ours slightly higher. However, as we can see from the throughput results in Fig. 13a, high padding efficiency does not directly translate to better throughput. In Fig. 15a, we also see the padding efficiency of packing increases with maximum sequence length. For T5 models, packing has a very high padding efficiency in terms of input to the encoder, while padding efficiency to the decoder is much lower. Our padding efficiency is more balanced between encoder and decoder,

Model	# GPUs	# layers	Model Dim	# Heads	# KV Channels	FFN Dim	# Param (B)
GPT	4,8,16,32	16,32,40,16	4096,4096,5140,12288	32,32,40,96	128	16384,16384,20560,49152	3.35,6.7,13,29
T5	4,8,16,32	12,24,48,96	1024	128	128	65536	5.5,11,22,44

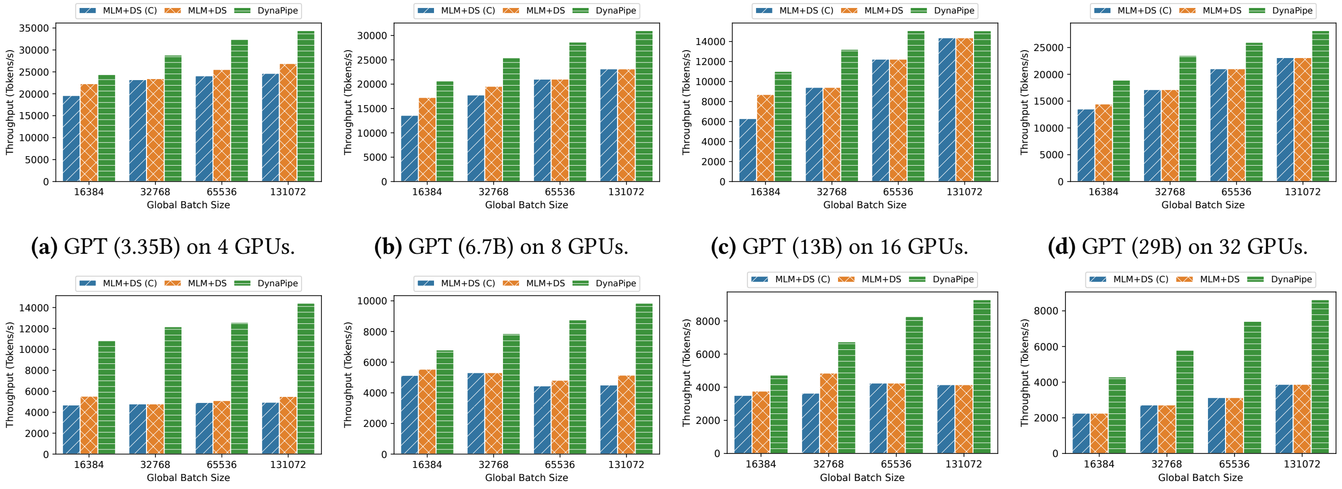
Table 1. DNN model configurations. For T5, "# layers" refers to layers present in both the encoder and the decoder.



(a) GPT (3.35B) on 4 GPUs. **(b)** GPT (6.7B) on 8 GPUs. **(c)** GPT (13B) on 16 GPUs. **(d)** GPT (29B) on 32 GPUs.

(e) T5 (5.5B) on 4 GPUs. **(f)** T5 (11B) on 8 GPUs. **(g)** T5 (22B) on 16 GPUs. **(h)** T5 (44B) on 32 GPUs.

Figure 13. Training throughput under different maximum sequence lengths.



(a) GPT (3.35B) on 4 GPUs. **(b)** GPT (6.7B) on 8 GPUs. **(c)** GPT (13B) on 16 GPUs. **(d)** GPT (29B) on 32 GPUs.

(e) T5 (5.5B) on 4 GPUs. **(f)** T5 (11B) on 8 GPUs. **(g)** T5 (22B) on 16 GPUs. **(h)** T5 (44B) on 32 GPUs.

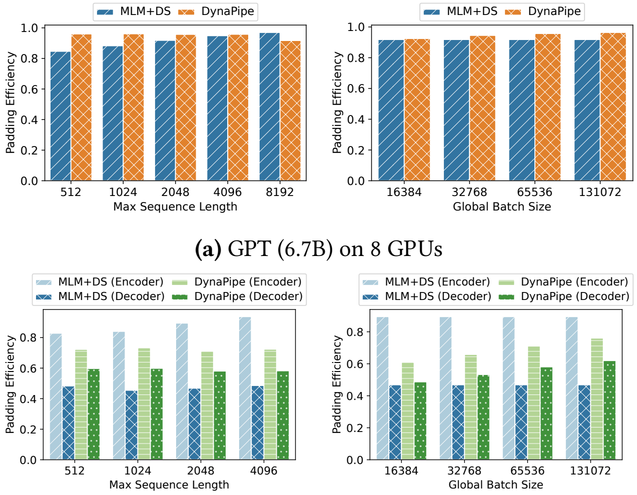
Figure 14. Training throughput under different global batch sizes.

since we consider both input sequence lengths during our DP algorithm.

8.4 Ablation study

We assess the design components within DynaPipe to analyze and decompose its performance improvement.

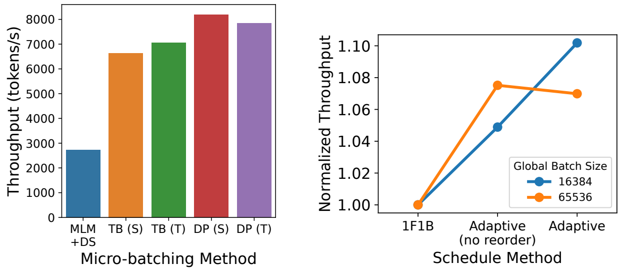
We first compare our dynamic programming algorithm against packing in *MLM+DS* and token-based (*TB*) micro-batching (which splits micro-batches so that each micro-batch contains roughly the same number of tokens), when training T5 with maximum sequence length 4096 and global batch size 65536 on 8 GPUs in Fig. 16a. In this setting, the optimal parallelism configuration does not use pipelining, isolating the impact of micro-batching. After searching for the best number of tokens per micro-batch, we find that *TB* already achieves significantly higher throughput than *MLM+DS*, indicating the inefficiency of packing-based solutions. Our dynamic programming algorithm further outperforms *TB* (without the need for parameter searching) by



(a) GPT (6.7B) on 8 GPUs

(b) T5 (11B) on 8 GPUs

Figure 15. Padding efficiency case study.

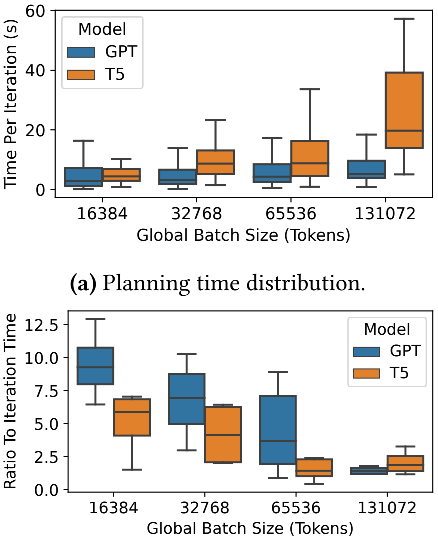


(a) Micro-batching methods. **(b)** Pipeline schedule methods.

Figure 16. Ablation study.

striking better trade-offs among padding, computation efficiency and memory consumption. We also evaluate different ways of determining the order of samples in our dynamic micro-batching and the TB baseline, i.e., sorting (S) versus solving a traveling salesman problem using a solver (T), and find that they do not impact the performance much.

We next compare our pipeline schedule (adaptive scheduling with and without micro-batch ordering) with 1F1B schedule, when training GPT with the same maximum sequence length and GPU settings as above. The grid-searched best parallelism uses 4 pipeline stages. As shown in Fig. 16b, our pipeline schedule achieves 10.1% and 7.4% throughput improvement over 1F1B, under global batch size 16384 and 65536, respectively. The performance gain is less than that from our simulations (Fig. 7), since now our dynamic micro-batching includes the longest micro-batch execution time in the cost function (Eq1), which, when minimized, produces more uniform micro-batches. The effect of micro-batch ordering is less prominent under a large global batch size, since sequence lengths in micro-batches can be made more similar by our DP algorithm when the global batch size is large (due to more opportunities for better micro-batch splitting).



(a) Planning time distribution.

(b) Comparison between average planning time and average iteration time (planning time / iteration time).

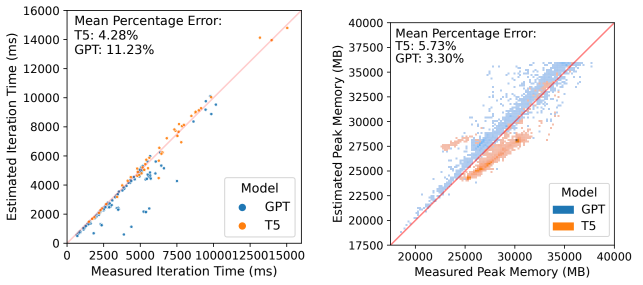
Figure 17. Execution planning time.

8.5 Execution planning time

We present the single-thread execution plan generation time during all our experiments in Fig. 17a. Execution planning for most training iterations takes less than 20 seconds for both GPT and T5 when the global batch size is small. Under larger global batch sizes, the time needed for our dynamic micro-batch and scheduling algorithms increases. The execution planning can be parallelized among the large number of CPU cores or even multiple nodes, allowing it to be completely overlapped with GPU computation. Fig. 17b compares our planning time to actual execution time per iteration. Across all our experiments, the average planning time to iteration time ratio peaks at 12.9x, which means full overlapping between training and planning is feasible with only 13 CPU cores, much less than available cores in typical LLM training instances (Amazon EC2 p4d [2], Microsoft Azure ND A100 v4 [23] and Google Could a2-highgpu-1g [11], each possessing 96 vCPU cores). In our experiments, we parallelize plan generation on 64 CPU cores in each machine that participates in training, and observe no slow-down caused by execution plan generation.

8.6 Accuracy of cost models

Fig. 18 illustrates the prediction accuracy of our iteration time and memory cost models, where data points are collected from all our experiments. For T5, the estimated iteration time matches well the actual value. The iteration time prediction error for GPT is slightly larger, and the out-liners



(a) Iteration time.

(b) Peak Memory Consumption.

Figure 18. Prediction accuracy of our iteration time and peak memory cost models.

are due to the all-reduce operation in data parallelism it uses, which we do not model. For memory estimation, we achieve lower than 6% mean prediction error for both T5 and GPT. These confirm that our cost models can provide accurate signals for guiding our optimizations.

9 Related Works

3D parallel training frameworks. Megatron-LM [26], DeepSpeed [22] are two popular frameworks supporting 3D parallel LLM training. Alpa [40] further automates the parallelization of the model, considering both intra- (including but not limited to data and tensor parallelism) and inter-operator (i.e., pipeline) parallelism. These frameworks assume fixed micro-batch sizes, fixed number of micro-batches and use a static pipeline schedule (e.g., 1F1B). They also do not consider construction of micro-batches, leaving the decision to users. DynaPipe optimizes micro-batch construction, supports dynamic micro-batches, and adopts adaptive pipeline schedule to accelerate multi-task training. Approaches that shard the model and optimizer states (e.g., ZeRO [31], FSDP [39]), or optimize communication in data parallelism (e.g., MiCS [38]) are also often used together with 3D parallelism. These approaches are orthogonal to DynaPipe and may be used in conjunction.

Sort dataset before batching. Some libraries (e.g., fairseq [28] and tensor2tensor [33]) offer an option to sort the dataset before constructing the mini-batches, so each mini-batch will contain samples with similar sequence lengths (also referred to as bucketing). Such bucketing destroys the randomness in batch construction thus may affect model performance. DynaPipe fully respects users' mini-batch construction method and only reorders samples within each mini-batch, preserving mathematical equivalence with models trained using traditional methods (padding or packing).

Custom attention kernels that ignore padding. ByteTransformer [37] implements special CUDA kernels to skip padding during self-attention. FlashAttention [9] also include attention kernels allowing variable sequence lengths. However, to ignore padding in other parts of the models, the model code needs to be adapted accordingly. Both libraries also only offer padding-free implementation for the BERT [10] model. Through dynamic micro-batching, DynaPipe directly minimizes padding in the inputs without needing to modify any model components. Recent updates (since the submission of DynaPipe) in FlashAttention also includes a method to "unpack" samples from packed inputs, enabling the variable length attention kernels to be used in conjunction with packing. It would be interesting future work to benchmark the performance of packing while enabling such kernel optimizations. None of the above works discussed the performance implications when jointly used with data or pipeline parallelism. DynaPipe optimizes for hybrid 3D parallel training by balancing the data parallel model replicas during micro-batch construction and using efficient pipeline schedules for dynamic micro-batches.

Training LLMs with extremely long sequences. Algorithmic approaches like sparse attention [8] and Longformer [4] tries to lower the quadratic complexity of self-attention in sequence length. Systematic approaches like DeepSpeed-Ulysses [15] and LightSeq [19] partition the model inputs at sequence dimension and distribute the calculation of self-attention to multiple machines. DynaPipe pursue different goals from these works since we do not aim to improve the computation speed or memory consumption for extremely long sequence lengths. Instead, we tackle the problem of input sequence length variation and avoid packing short input samples to long sequences using dynamic micro-batching. It would be interesting future work to study how to concurrently utilize these methods and DynaPipe to achieve both goals.

10 Conclusion

In this paper, we propose DynaPipe, which uses dynamic micro-batching as a solution for the highly variable sequence length in multi-task training, circumventing packing which is computationally inefficient. We propose a dynamic programming algorithm to optimize micro-batch construction, present a pipeline scheduling algorithm that is robust to micro-batch execution time variations, and design an effective pipeline communication planning mechanism for efficient dynamic pipeline training. Extensive evaluation demonstrates up to 4.39x speed up when training T5, and 3.25x when training GPT.

11 Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Jongsoo Park for their valuable feedback. This work was supported by an Amazon Research Award (ARA) on AWS AI and grants from Hong Kong RGC under the contracts HKU 17208920 and C7004-22G (CRF).

References

[1] Armen Aghajanyan, Anchit Gupta, Akshat Shrivastava, Xilun Chen, Luke Zettlemoyer, and Sonal Gupta. 2021. Muppet: Massive Multi-task Representations with Pre-Finetuning. arXiv:2101.11038 [cs.CL]

[2] Amazon Web Services, Inc. 2023. Amazon EC2 P4d Instances. https://aws.amazon.com/ec2/instance-types/p4/.

[3] Amazon Web Services, Inc. 2023. Elastic Fabric Adapter. https://aws.amazon.com/hpc/efa/.

[4] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. (2020). arXiv:2004.05150 [cs.CL]

[5] Tami Boudoukh, Michal Penn, and Gideon Weiss. 2001. Scheduling jobshops with some identical or similar jobs. *Journal of Scheduling* 4, 4 (2001), 177–199.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris 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 *Proc. of NeurIPS*, Vol. 33. 1877–1901.

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. (2016). arXiv:1604.06174 [cs.LG]

[8] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. (2019). arXiv:1904.10509 [cs.LG]

[9] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Proc. of NeurIPS*.

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

[11] Google. 2023. GPU platforms. https://cloud.google.com/compute/docs/gpus#a100-40gb.

[12] Ananth Gottumukkala, Dheeru Dua, Sameer Singh, and Matt Gardner. 2020. Dynamic sampling strategies for multi-task reading comprehension. In *Proc. of ACL*. ACL, 920–924.

[13] Stephen C Graves, Harlan C Meal, Daniel Stefek, and Abdel Hamid Zeghmi. 1983. Scheduling of Re-Entrant Flow Shops. *Journal of operations management* 3, 4 (1983), 197–207.

[14] Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching Machines to Read and Comprehend. In *Proc. of NeurIPS*. 1693–1701.

[15] 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. (2023). arXiv:2309.14509 [cs.LG]

[16] Narendra Karmarkar and Richard M Karp. 1982. *The differencing method of set partitioning*. Computer Science Division (EECS), University of California Berkeley.

[17] Richard M Karp. 2010. *Reducibility among combinatorial problems*. Springer.

[18] Mario Michael Krell, Matej Kosec, Sergio P. Perez, and Andrew Fitzgibbon. 2022. Efficient Sequence Packing without Cross-contamination: Accelerating Large Language Models without Impacting Performance. arXiv:2107.02027 [cs.CL]

[19] Dacheng Li, Rulin Shao, Anze Xie, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. 2023. LightSeq: Sequence Level Parallelism for Distributed Training of Long Context Transformers. (2023). arXiv:2310.03294 [cs.LG]

[20] Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V. Le, Barret Zoph, Jason Wei, and Adam Roberts. 2023. The Flan Collection: Designing Data and Methods for Effective Instruction Tuning. arXiv:2301.13688 [cs.AI]

[21] Redis Ltd. 2023. Redis. https://redis.io/

[22] Microsoft. 2023. DeepSpeed. https://github.com/microsoft/DeepSpeed.

[23] Microsoft. 2023. ND A100 v4-series. https://learn.microsoft.com/en-us/azure/virtual-machines/nda100-v4-series, Last accessed on 2023-10-31.

[24] Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. 2022. Cross-Task Generalization via Natural Language Crowdsourcing Instructions. In *Proc. of ACL*. ACL, 3470–3487.

[25] 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 *Proc. of SOSP*. ACM, 1–15.

[26] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proc. of SC*. ACM, 1–15.

[27] NVIDIA. 2023. NCCL. https://developer.nvidia.com/nccl

[28] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. Fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proc. of NAACL*. ACL.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proc. of NeurIPS*. 8024–8035.

[30] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[31] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory optimizations toward training trillion parameter models. In *Proc. of SC*. IEEE, 1–16.

[32] Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. 2022. Multitask Prompted Training Enables Zero-Shot Task Generalization. In *Proc. of ICLR*. OpenReview.net.

[33] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. 2018. Tensor2Tensor for Neural Machine Translation. (2018). arXiv:1803.07416 [cs.LG]

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Proc. of NeurIPS* 30 (2017).

[35] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. 2022. Finetuned Language Models are Zero-Shot Learners. In *Proc. of ICLR*. OpenReview.net.

[36] Adina Williams, Nikita Nangia, and Samuel Bowman. 2018. A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference. In *Proc. of NAACL*. ACL, 1112–1122.

[37] Yujia Zhai, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. 2023. ByteTransformer: A high-performance transformer boosted for variable-length inputs. In *Proc. of IPDPS*. IEEE, 344–355.

[38] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. 2022. MiCS: Near-Linear Scaling for Training Gigantic Model on Public Cloud. *Proc. VLDB Endow.* 16, 1 (2022), 37–50.

[39] 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. (2023). arXiv:2304.11277 [cs.DC]

[40] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *Proc. of OSDI*. USENIX, 559–578.

A Artifact Appendix

A.1 Abstract

This artifact appendix documents the steps to reproduce Fig. 13 through 18 in the paper *DynaPipe: Optimizing Multi-task Training through Dynamic Pipelines*. The experiments are expected to be run on a single AWS EC2 p4d instance. The evaluation is expected to take around 26 hours to complete.

A.2 Description & Requirements

A.2.1 How to access.

The code used to generate results in the paper is mainly implemented in two repositories:

- DynaPipe: contains the main implementation of DynaPipe. Can be accessed through https://github.com/awslabs/optimizing-multitask-training-through-dynamic-pipelines
- A modified version of Megatron-LM: Main modifications include adding support for packing in the dataloader, implementing the pipeline instructions for DynaPipe, and adding the scripts for running the experiments. Can be accessed through https://github.com/chenyu-jiang/Megatron-LM

A permanent copy of the artifact can be found at https://zenodo.org/records/8413925 (DOI: 10.5281/zenodo.8413925), which contains a copy of this document, a pre-built docker image containing all codes and dependency required to run the artifact, and pre-processed datasets used in the experiments.

A.2.2 Hardware dependencies.

The evaluation is expected to be performed on a single Amazon EC2 p4d instance. Evaluation on other platforms with multiple GPUs (which supports PyTorch and Megatron-LM) is also possible when generating everything from scratch (see later sections for details).

A.2.3 Software dependencies.

- PyTorch (>= 2.1.0)
- DynaPipe (please see *requirements.txt* in the repository for details)
- Modified Megatron-LM (dependencies of original Megatron-LM still apply)
- Modified DeepSpeed: https://github.com/chenyu-jiang/DeepSpeed. We removed a timer that introduce unnecessary synchronization which disrupts our schedule and disabled overflow checking for more consistent throughput measurement.

A.2.4 Benchmarks.

We used the FLANv2 dataset in our experiments. For artifact evaluation, we provide a pre-downloaded and pre-processed dataset in the provided machine (also accessible in the Zenodo repository). To copy the pre-downloaded dataset into the container, run the following command outside of the container:

```
cd ~/preprocessed_datasets
docker cp datasets dynapipe:/root/Megatron-LM
```

Otherwise, the dataset can be downloaded using the following steps:

1. Clone the repository for the dataset (a fork of the original repository with some version mismatch fixed. Also added a downloading script) and install dependencies:

   ```
   git clone https://github.com/chenyu-jiang/
   text-to-text-transfer-transformer.git
   cd text-to-text-transfer-transformer
   pip3 install -r requirements.txt
   ```

2. Download the raw dataset
   (generates *supervised_proportional.jsonl*):

   ```
   python3 prepare_dataset.py
   ```

3. Perform some initial cleaning
   (generates *cleaned_supervised_proportional.jsonl*):

   ```
   python3 clean_dataset.py
   ```

4. Preprocess the dataset with Megatron-LM's data loader script (generates *.bin* and *.idx* files)

   ```
   cd <path_to_modified_MegatronLM>
   ./experiment_scripts/run_preprocess_flan.sh
   <path_to_cleaned_jsonl>
   ```

A.3 Set-up

We provide a Dockerfile to setup a container image for evaluation. To generate the image, run:

```
git clone
 https://github.com/chenyu-jiang/Megatron-LM.git
cd Megatron-LM/docker
./build_image.sh
```

For artifact evaluation, a pre-built image will be installed on the provided machine (also available in the Zenodo repository). To create a container from the image, run (inside the docker directory):

```
./run.sh
```

You will find DynaPipe and the modified Megatron-LM at */root* in the container.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): DynaPipe significantly outperforms the state-of-the-art packing-based systems for multi-task training. This is proven by the experiment (E1) described in Section 8 whose results are illustrated in Figure 13 and 14.
- (C2): DynaPipe achieves comparable batching efficiency as packing-based systems. The batching efficiency statistics are illustrated in Figure 15.

- (C3) The dynamic-programming-based micro-batch generation and memory-aware adaptive scheduling algorithms proposed in DynaPipe out-performs naive alternatives. This is proven by experiment (E2) in Section 8, results illustrated in Figure 16.
- (C4) The planning overhead of DynaPipe is low (when parallelized onto multiple CPU cores). The planning time distribution is illustrated in Figure 17.
- (C5) DynaPipe's cost models can achieve a high prediction accuracy. Cost model accuracy statistics is illustrated in Figure 18.

A.4.2 Experiments.

Note: we provide some pre-computed results for some very time-consuming steps like grid searching for the best parallelism. To produce everything from scratch, see the README in https://github.com/chenyu-jiang/Megatron-LM. We also provide a single script for running all needed experiments at */root/Megatron-LM/experiment_scripts/run_all.sh*.

Experiment (E1): [Throughput Benchmark] [18 compute-hours]: Uses pre-generated configs (obtained by the grid search) to run full benchmarks for throughput comparison. Note for artifact evaluation, only Fig.13 (a)(b)(e)(f) and Fig.14 (a)(b)(e)(f) can be generated on a single p4d node. The other sub-figures of Fig.13 and 14 require multiple p4d nodes. For Fig.17, only Fig.17 (a) will be generated. Verifies (C1),(C2),(C4),(C5).

[How to] Run the following command in the docker container:

```
cd /root/Megatron-LM
# performs benchmark
# (generates raw results in ./experiments directory)
./experiment_scripts/run_benchmark.sh
# generate figures
./experiment_scripts/generate_figure_13_14.sh
./experiment_scripts/generate_figure_15.sh
./experiment_scripts/generate_figure_17.sh
./experiment_scripts/generate_figure_18.sh
```

[Results] Figure 13, 14, 15, 17, 18 will be generated in */root/Megatron-LM/reproduced_figures*.

Experiment (E2): [Ablation] [8 compute-hours]: Performs ablation study to verify (C3).

[How to] Run the following command in the docker container:

```
cd /root/Megatron-LM
# performs ablation experiments
# (generates raw results in ./experiments directory)
./experiment_scripts/run_ablation.sh
# generate figures
./experiment_scripts/generate_figure_16.sh
```

[Results] Figure 16 will be generated in */root/Megatron-LM/reproduced_figures*.