PIPEMESH: Achieving Memory-Efficient Computation-Communication Overlap for Training Large Language Models

Fanxin Li, Shixiong Zhao, Yuhao Qing, Jianyu Jiang, Xusheng Chen, and Heming Cui, *Member, IEEE*

***Abstract*—Efficiently training large language models (LLMs) on commodity cloud resources remains challenging due to limitations in network bandwidth and accelerator memory capacity. Existing training systems can be categorized based on their pipeline schedules. Depth-first scheduling, employed by systems like Megatron, prioritizes memory efficiency but restricts the overlap between communication and computation, causing accelerators to remain idle for over 20% of the training time. Conversely, breadth-first scheduling maximizes communication overlap but generates excessive intermediate activations, exceeding memory capacity and slowing computation by more than 34%. To address these limitations, we propose a novel elastic pipeline schedule that enables fine-grained control over the trade-off between communication overlap and memory consumption. Our approach determines the number of micro-batches scheduled together according to the communication time and the memory available. Furthermore, we introduce a mixed sharding strategy and a pipeline-aware selective recomputation technique to reduce memory usage. Experimental results demonstrate that our system eliminates most of the 28% all-accelerator idle time caused by communication, with recomputation accounting for less than 1.9% of the training time. Compared to existing baselines, PIPEMESH improves training throughput on commodity clouds by 20.1% to 33.8%.**

***Index Terms*—Deep learning, distributed training, GPU, DNN, 3D parallelism, pipeline parallelism, machine learning.**

I. INTRODUCTION

LARGE language models (LLMs) [1], [2], [3], [4], [5], [6] based on transformers [7] have shown unprecedented capabilities. Training LLMs with up to hundreds of billions of parameters demands exceedingly vast computational resources (e.g., training Llama3 [8] 70B requires 6.4 million GPU hours of computation). Parallelization techniques that distribute the training workload across devices (i.e., accelerators) have been commonly adopted. Tensor parallelism [9] (TP) is an intra-host parallelization technique that splits the individual layer of the model across devices connected with high-speed interconnects such as NVLink. The input of an LLM is a sequence of tokens. Sequence parallelism [10] (SP) and context parallelism [11] (CP) are two techniques that are combined with TP and operate on the dimension of input sequences. Data parallelism (DP) and pipeline parallelism [12], [13] (PP) are two techniques for inter-host parallelization. DP splits the training data across devices and synchronizes the gradients and parameters between devices. PP splits the model layers into stages, with each stage assigned to a different device. The input batch is divided into multiple micro-batches, which are processed in a pipelined manner. State-of-the-art LLM training systems (i.e., 3D parallel training systems) combine TP, DP, and PP to scale training to thousands of devices.

Cloud platforms provide easy access to powerful computing resources like GPUs and NPUs. For research labs and small enterprises, training LLMs on clouds remains a more flexible and affordable option compared to supercomputers [14], [15] owned by giant companies. However, cloud-based resources typically face two limitations. First, the bandwidth of interconnects between hosts is limited, leading to increased communication time of inter-host parallelization techniques such as gradient synchronization in DP. For example, the bandwidth of a host with 8 A100 GPUs on AWS is only 400 Gbps, while the bandwidth of the 8×A100 host is 1.6 Tbps in supercomputing clusters [2], [16]. Second, the cloud contains many legacy but still powerful GPUs with limited memory (referring to HBM, high-bandwidth memory) capacity, such as V100 and A10 GPUs [17]. This limitation necessitates the application of memory-saving techniques like selective recomputation [10] and Zero Redundancy Optimizer [18] (ZeRO). However, these techniques introduce extra computation or communication overhead, further compounding the inefficiency of cloud-based LLM training. The major challenge we aim to address is how to maximize the utilization of the computing resources on clouds while navigating the limitations of interconnect bandwidth and HBM capacity.

The stringent problem we found is that existing training systems often struggle to achieve a balance between mitigating limited interconnect bandwidth and limited memory capacity. The pipeline schedule adopted by existing systems plays a crucial role in determining the extent of communication overlapped with computation and the amount of memory consumed. We introduce a new perspective that views the pipeline schedule

Received 10 June 2024; revised 10 June 2025; accepted 11 June 2025. Date of publication 27 June 2025; date of current version 17 July 2025. This work was supported in part by National Key R&D Program of China under Grant 2022ZD0160201, in part by HK RGC RIF under Grant R7030-22, in part by HK RGC GRF under Grant 17208223 and Grant 17204424, in part by Huawei flagship research grant in 2023, in part by SupernetAI, and in part by the HKU-CAS Joint Laboratory for Intelligent System Software. Recommended for acceptance by B. Nicolae. *(Corresponding authors: Shixiong Zhao; Heming Cui.)*

The authors are with the Department of Computer Science, University of Hong Kong, Hong Kong, SAR 999077, China (e-mail: fxli@cs.hku.hk; sxzhao@cs.hku.hk; yhqing@cs.hku.hk; jyjiang@cs.hku.hk; xschen@cs.hku.hk; heming@cs.hku.hk).

Digital Object Identifier 10.1109/TPDS.2025.3583983

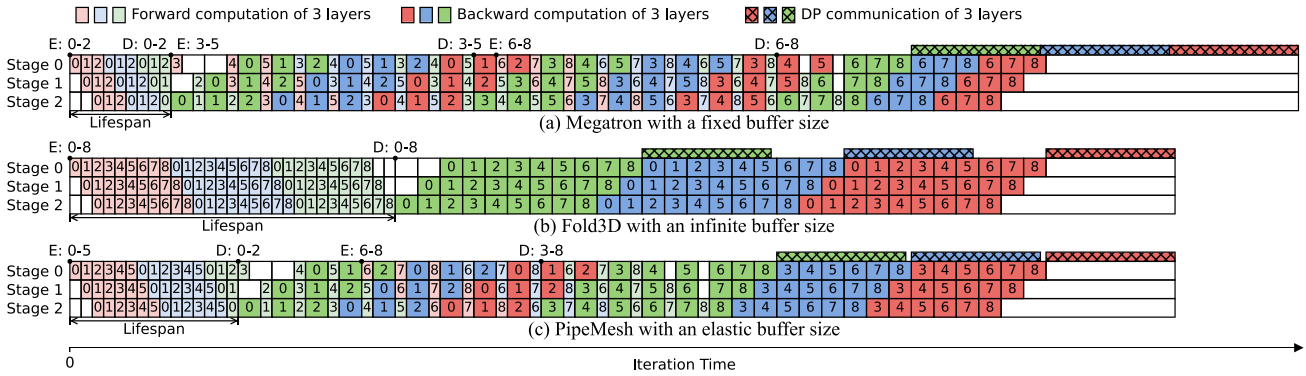


Fig. 1. Schedules adopted by Megatron, Fold3D, and PIPEMESH. Each pipeline stage contains 3 non-consecutive layers (corresponding to 3 model chunks defined in Section II-A). The numbers inside the boxes represent the micro-batch IDs. We depict the lifespan of the activations for micro-batch 0. The notation E: i-j represents the enqueuing of micro-batches with IDs ranging from i to j, while D: i-j represents the dequeuing of micro-batches with IDs from i to j. PIPEMESH overlaps most of the DP communication with computation, with the activation lifespan shorter than Fold3D.

as a process of enqueuing and dequeuing micro-batches to and from a buffer queue with an allocated size. Micro-batches are enqueued until the buffer size is reached, at which point the enqueue process runs concurrently with the dequeue process. The enqueue represents the forward pass of a micro-batch, while the dequeue represents the backward pass.

We summarize existing systems into two categories according to their pipeline schedules. The first category (e.g., PipeDream [13], Megatron [16], MegaScale [19], and the zero bubble pipeline [20]) adopts the depth-first schedule (DFS) tailored for memory capacity. As shown in Fig. 1(a), the buffer size is set to the number of pipeline stages, which is the minimum number of micro-batches required to fill the entire pipeline. DFS systems have the fastest micro-batch dequeue time, resulting in the shortest lifespan for the activations generated by each micro-batch. Theoretically, these systems have the minimum activation memory peak. However, the interleaving of enqueue and dequeue operations leads to a limited window for overlapping communication with computation when combined with DP. The communication for gradient synchronization at the end of each training iteration must wait for the computation for all micro-batches of a layer to complete before proceeding [21], [22] (see Fig. 7). Consequently, while DFS systems offer minimal activation memory consumption and are well-suited for increasingly long sequence lengths (the activation size grows with the sequence length; see Section II-B), they suffer from significant communication on the performance-critical path. Our evaluation of Megatron on the AWS V100 cluster shows that DP communication causes GPUs to be idle for over 20.2% of the training time.

The second category (e.g., Fold3D [21], breadth-first pipeline [22]) adopts the breadth-first schedule (BFS), which is tailored for limited interconnect bandwidth. These systems set an infinite buffer size and enqueue all micro-batches simultaneously, allowing the computation for all micro-batches of a layer to be scheduled together. This approach significantly advances the completion time of each layer, maximizing the window for overlapping communication and computation. Fold3D can hide over 80% of the DP communication when it is shorter than the computation. However, BFS systems substantially increase the activation lifespan, necessitating the use of various memory-saving techniques. To benefit from communication overlap while maintaining memory consumption comparable to Megatron, Fold3D needs to discard most of the activations for each layer generated during the forward propagation and fully recompute them during the backward propagation. Meanwhile, Fold3D requires CPU offloading, which transfers activations stored (i.e., the inputs of layers) from GPU memory to CPU memory during the forward propagation and moves the activations back to GPU memory before they are needed during the backward propagation.

The drawback of BFS systems is that recent works [10], [16] commonly adopt selective recomputation, which vastly reduces the recomputation cost with an acceptable increase in activation memory compared to full recomputation. Unfortunately, selective recomputation leads to a significant increase in memory consumption for BFS systems. The complexity of activation memory is $O(B \cdot a)$ for BFS systems and $O(P \cdot a)$ for DFS systems. a is the activation memory per micro-batch, P is the number of pipeline stages, and B is the micro-batch number ($B \gg P$). For systems like Fold3D, due to more activations stored, maintaining memory consumption on par with Megatron leads to increased overhead from offloading. Our evaluation of memory-constrained scenarios reveals that Fold3D's offloading introduces 34.1% overhead in computation time. Although Fold3D manages to hide communication equivalent to 23.8% of the computation time, the time of an iteration increases by 7.3% compared to Megatron.

Overall, systems from both categories fall into the same pitfall: their static pipeline schedules are optimized for either extreme memory saving or extreme communication overlap. However, in many cases, the memory is not used up for DFS systems, and the communication is not worthy of the excessive memory consumption. As illustrated in Section VI-A, Megatron does not fully utilize memory while communication is not hidden. Our evaluation (see Section VI-A) reveals that Fold3D provides windows for overlapping communication 208.4% to 233.4% larger.

We propose a new abstraction, named *elastic buffer queue*, which allows for more granular control over the trade-off

between communication overlap and memory consumption. By carefully managing the buffer queue size and the enqueue-dequeue process, we can optimize the pipeline schedule to maximize training throughput under the constraints of bandwidth and memory capacity. Specifically, PIPEMESH increases the buffer queue size to enqueue more micro-batches simultaneously, thereby scheduling the computation for more micro-batches of a layer together and providing a larger window for overlapping communication. The increase continues until either the available device memory capacity is fully utilized or the window exceeds the communication time. As shown in Fig. 1(c), a larger buffer size leads to less interleaving of micro-batches and earlier completion time of each layer.

However, two challenges exist when realizing PIPEMESH. The first challenge is that the buffer queue size is bounded by the memory available for activations, and the bounded buffer size allows hiding only limited communication in memory-constrained scenarios. To address this challenge, we propose a mixed sharding technique that applies ZeRO-2 and ZeRO-3 to the model parameters, reducing the memory consumed by gradients and parameters and allocating more memory to activations. PIPEMESH shards the gradients of ZeRO-2 and ZeRO-3 parameters across devices when the computation on micro-batches scheduled (dequeued) together finishes. The ZeRO-3 parameters are sharded, and PIPEMESH collects the full parameters before these parameters are needed for computation.

To support larger buffer sizes, we also devise a pipeline-aware selective recomputation technique to reduce the activation memory per micro-batch. PIPEMESH can have shorter recomputation times compared to BFS systems due to reduced activation lifespans. The second challenge is how to determine the optimal buffer size together with the recomputed activations. Either choice allows trading off memory for throughput improvement, which originates from reduced performance-critical communication or reduced computation. We took the first step in the literature to model the challenge as a joint optimization problem. Our observation to solve the problem is that we can divide it into a set of tractable sub-problems, where each sub-problem decides the solution for a set of consecutive operators in the model. We devised an algorithm based on dynamic programming to solve the sub-problems. Unlike existing activation recomputation methods [23], [24], [25] considering only the recomputation time saved when storing an activation, our pipeline-aware selective recomputation trades off between the saved recomputation time and the increased performance-critical DP communication.

PIPEMESH is designed for specific, yet common, training scenarios. It is built on the premise that GPU memory is a constrained resource, making techniques like full recomputation or extensive CPU offloading of activations, as seen in BFS systems, prohibitively expensive. Besides, PIPEMESH is particularly effective in large-scale DP where overlapping DP communication is crucial for performance. Moreover, PIPEMESH assumes the computation within a pipeline stage is sufficient to hide the communication between stages. It is not designed for scenarios where pipeline communication is the primary bottleneck. In such cases, a simpler configuration that minimizes pipeline communication would be more effective (see Section IV-A).

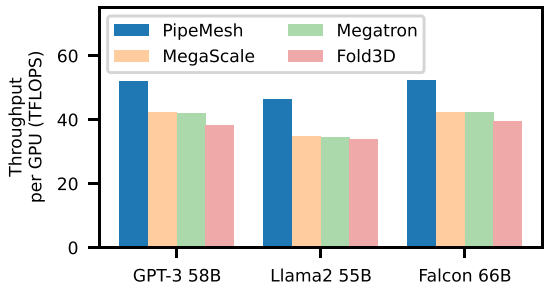


Fig. 2. Per-GPU throughputs of four systems. All models were trained on 96 V100 GPUs. PIPEMESH consistently achieved higher TFLOPS than the baselines.

We have implemented PIPEMESH based on Megatron. Our evaluation shows that:

- PIPEMESH greatly improves the training throughput on commodity clouds. PIPEMESH achieved 23.6%-33.8% higher throughput than Megatron on an AWS cluster with 96 V100 GPUs and 20.1% higher throughput on an AWS 72 A100 GPU cluster. PIPEMESH can reduce Megatron's cost of pre-training Llama2 55B on 2.4 T tokens from 302,463 GPU days to 226,508 GPU days.
- PIPEMESH is memory-efficient, unfailingly operating within the specified memory constraint and below the available GPU memory capacity.
- PIPEMESH's improvement is robust. PIPEMESH consistently showed improvements for models with different architectures and sizes (see Fig. 2).
- PIPEMESH is scalable. Our scalability evaluation on an A10 GPU cluster shows that PIPEMESH's throughput improvement over Megatron was always more than 27.1% from 48 to 192 GPUs.

Our major novelty is the elastic buffer queue abstraction, which enables the trade-off between communication overlap and memory consumption. We design an elastic pipeline schedule that can achieve state-of-the-art training throughput within the constraints of memory capacity and network bandwidth. To further reduce memory consumption, we propose mixed sharding and pipeline-aware selective recomputation techniques. We also introduce a cost model for the elastic pipeline schedule and the two aforementioned techniques, along with an algorithm to determine the optimal solution. PIPEMESH can significantly save the costs of training LLMs on commodity clouds for researchers and enterprises. Our code is released at github.com/hku-systems/pipemesh.

II. BACKGROUND AND MOTIVATION

A. Parallelism Dimensions

Data Parallelism: In data parallelism (DP), the training data is split across devices. Each device computes gradients of the full model on the allocated data, synchronizes the gradients with the other devices, and updates the model parameters using the

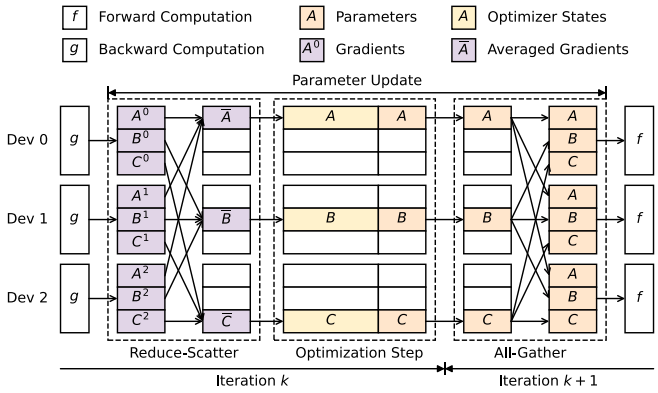


Fig. 3. Data parallel training with optimizer sharding across three devices (Dev 0, Dev 1, and Dev 2). During the parameter update phase, gradients from all devices are first aggregated using a reduce-scatter operation. Each device then updates its parameter shard using the reduced gradients. Finally, an all-gather operation reconstructs the full updated model on every device for the next iteration.

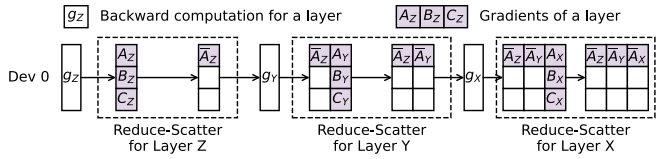


Fig. 4. Backward propagation with gradient sharding for a model with three layers (X, Y, and Z). We showcase the workflow on one of three data parallel devices. The backward pass for layer Z computes the full gradients, which are then partitioned into shards (A_Z, B_Z, and C_Z).

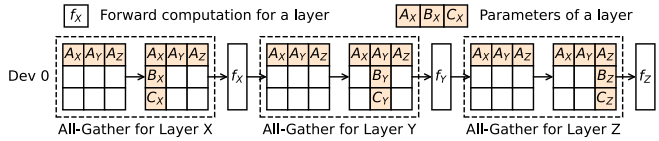


Fig. 5. Forward propagation with parameter sharding for a three-layer model (X, Y, and Z). The workflow on one of three data parallel devices is shown.

chosen optimizer (the update progress is defined as the optimization step). The basic form [26] of DP replicates the model and optimizer states (e.g., gradient momentum and variance of Adam [27] optimizer) across devices. The model size supported by the form of DP is limited since each device needs to hold a full copy of model parameters, gradients, and optimizer states.

ZeRO Optimization: ZeRO [18] proposes a 3-stage optimization for data parallelism to save memory usage by partitioning the model and optimizer states. ZeRO-1 (stage 1) groups optimizer states into D equal partitions for D data parallel devices, and each device only stores and updates $\frac{1}{D}$ of the optimizer states. At the end of each training iteration, gradients across all data parallel devices are averaged via a reduce-scatter operation. Specifically, gradients on each device are also divided into D partitions, such that the i^{th} data parallel device collects and averages the i^{th} gradient partitions from the other devices. During the optimization step, each device uses its gradient partition to update its optimizer state partition and corresponding parameters, which account for $\frac{1}{D}$ of the total parameters. At the start of the next iteration, the data parallel devices obtain all the updated parameters via an all-gather operation, where each device broadcasts its parameters to the other devices. In Fig. 3, we illustrate the optimizer sharding process and the reduce-scatter and all-gather operations with an example using three data parallel devices.

In Zero-1, each device still needs to store the parameters and gradients of the entire model during the computation. To save the memory allocated to gradients, ZeRO-2 (stage 2) further incorporates gradient sharding, which partitions gradients across devices during the backward propagation. Each time the size of gradients generated per device reaches a predefined bucket size, ZeRO-2 launches a reduce-scatter operation to reduce the bucket of gradients on each device to $\frac{1}{D}$ of its original size. As a result, the gradient size stored by each device is no larger than $\frac{1}{D}$ of the total gradient size plus the bucket size. Fig. 4 demonstrates the gradient sharding process when setting the bucket size to the gradient size of a layer.

Besides the gradients and optimizer states, ZeRO-3 (stage 3) also partitions the model parameters across data parallel devices. When a device requires parameters outside its partition for the computation, it receives these parameters from the other devices through all-gather communication. The received parameters are released immediately after being used during the forward or backward propagation. Fig. 5 shows how each device receives and releases parameters during the forward propagation with parameter sharding. Compared with ZeRO-2, ZeRO-3 reduces the memory allocated to parameters while incurring more all-gather communication.

Pipeline Parallelism: Pipeline parallelism [12], [13], [28], [29], [30] (PP) partitions the layers of a model across devices, and each device is a pipeline stage. The input's gradients and outputs of layers are propagated between the pipeline stages via point-to-point communication. The input batch for an iteration is split into micro-batches, and their computational tasks are pipelined. In bulk synchronous parallel (BSP) [31] training, a pipeline flush, which blocks the computation until all micro-batches in the pipeline finish, is inserted between two iterations. The blocking time is defined as the pipeline bubble.

Existing systems [16], [21] allow assigning noncontiguous layers to devices (i.e., pipeline stages) to overlap DP communication and achieve a smaller pipeline bubble. For LLMs with homogeneous layers, the model is evenly divided into multiple subsets of layers (each subset is called a model chunk). The model chunks are assigned to devices in a circular manner, and the devices have an equal number of model chunks. For example, when putting 12 layers on three devices, where each device holds two model chunks, the first device would have layers 1, 2, 7, and 8; the second device would have layers 3, 4, 9, and 10; and so on.

Depth-first schedule: Megatron [16] uses the depth-first schedule [13] to limit activation memory. We define the forward pass and the backward pass as the forward and backward computation of a model chunk on a micro-batch, respectively. Each pipeline stage first executes a few consecutive forward passes in the warm-up phase and then enters the steady 1F1B phase, where one forward pass is followed by one backward pass. In the

case where each stage in the P pipeline stages contains M model chunks, each stage first performs the forward computation of the first $M-1$ model chunks on the first P micro-batches. After that, the last stage directly enters the 1F1B phase, while each preceding stage must perform two additional forward passes. This is because the forward computation starts one forward pass earlier, and the backward computation starts one forward pass later (see Fig. 7(a)). Therefore, the first stage needs to perform $(M-1)P+2(P-1)$ consecutive forward passes. After completing all forward passes during the 1F1B phase, the pipeline stages execute the remaining backward passes during the cool-down phase.

B. Memory Usage of LLM Training

The memory consumption of LLM training can be classified into two parts: model states (i.e., parameters, gradients, and optimizer states) and activations. Even though advanced hardware (e.g., HBM3 [32]) enlarges the device's memory, it is still stringent compared with the growing memory requirement of LLM training. First, the scaling law [33] verifies that larger LLMs demonstrate better abilities and achieve higher accuracy in various tasks. With the growth of model size, both model states and activations consume more memory. Second, long-context inference [6] of LLMs requires training with longer sequences. The activation memory is proportional to the sequence length (defined as the number of tokens per training input) used.

Model State Memory: We analyze the memory consumed by optimizer states, gradients, and parameters for a typical training setup using mixed precision training [34] and Adam [27] optimizer. The optimizer states keep an fp32 copy of the parameters, momentum, and variance. Common implementations of mixed precision training (e.g., PyTorch AMP [35]) maintain only a single copy of fp32 parameters and dynamically downcast them to fp16 during computation without explicitly storing fp16 parameters. However, when using optimizer sharding, the fp16 parameters on a device are mostly collected from other devices via all-gather communication. Implementations like Megatron downcast the fp32 parameters all at once during the optimization step and maintain the fp16 parameters separately. Therefore, we account for both the memory allocated to fp32 and fp16 parameters in the following analysis.

Systems using the depth-first schedule (e.g., Megatron) only support ZeRO-1 due to the conflict between the gradient accumulation required for PP and the gradient sharding needed for ZeRO-2 and ZeRO-3. Specifically, these systems need to store the intact gradients throughout the iteration to accumulate the gradients generated by all micro-batches. When a device holds ψ parameters, the gradients and parameters on that device require 4ψ bytes of memory in total.

Systems using the breadth-first schedule (e.g., Fold3D) can incorporate ZeRO-3 (see Fig. 7(b)). For a model chunk, before its first forward or backward pass, Fold3D with ZeRO-3 (denoted Fold3D-Z3) executes an all-gather operation to restore the parameters of that model chunk. The parameters are released immediately after all the forward or backward passes of that model chunk finish. After the last backward pass of a model chunk, Fold3D-Z3 launches a reduce-scatter operation to shard the gradients of that model chunk. The communication introduced by ZeRO-3 overlaps with the computation. The gradient partition and parameter partition require $\frac{2}{D}\psi$ and $\frac{2}{D}\psi$ bytes of memory, respectively. Besides, the gradients, as well as the parameters, require $4\frac{\psi}{M}$ bytes of extra memory to accommodate the intact gradients and parameters of two model chunks used by overlapped computational and communication tasks. M is the number of model chunks per pipeline stage.

TABLE I
OPERATIONS IN THE ATTENTION BLOCKS OF DIFFERENT MODELS

Operation	FLOPs	S_{out}	Models
Query	$2sbh^2$	$2sbh$	GPT-3, Llama2, Falcon
Key.MHA	$2sbh^2$	$2sbh$	GPT-3
Key.GQA	$\frac{1}{4}sbh^2$	$\frac{1}{4}sbh$	Llama2, Falcon
Value.MHA	$2sbh^2$	$2sbh$	GPT-3
Value.GQA	$\frac{1}{4}sbh^2$	$\frac{1}{4}sbh$	Llama2, Falcon
Self-Attn	$4s^2bh$	$2sbh$	GPT-3, Llama2, Falcon
Linear.PA	$2sbh^2$	$2sbh$	GPT-3, Llama2, Falcon

All models use the same query transformation, selfattention (Self-Attn), and post-attention linear (Linear.PA) operations. Llama2 and Falcon use grouped-query attention (Key.GQA and Value.GQA), and GPT-3 uses multi-head attention (Key.MHA and Value.MHA). We report the forward pass FLOPs and the output size (S_{out}) for each operation. h is the hidden size of the transformer layer, s is the input sequence length, and b is the batch size.

TABLE II
OPERATIONS IN THE MLP BLOCKS OF DIFFERENT MODELS

Operator	FLOPs	S_{out}	Models
Linear.Up-$\frac{8}{3}h$	$\frac{16}{3}sbh^2$	$\frac{16}{3}sbh$	Llama2
Linear.Up-$4h$	$8sbh^2$	$8sbh$	GPT-3, Falcon
SwiGLU.Gate	$\frac{16}{3}sbh^2$	$\frac{16}{3}sbh$	Llama2
SwiGLU.SiLU	$\frac{40}{3}sbh$	$\frac{16}{3}sbh$	Llama2
SwiGLU.Mul	$\frac{8}{3}sbh$	$\frac{16}{3}sbh$	Llama2
GeLU	$56sbh$	$8sbh$	GPT-3, Falcon
Linear.Down-$\frac{8}{3}h$	$\frac{16}{3}sbh^2$	$\frac{16}{3}sbh$	Llama2
Linear.Down-$4h$	$8sbh^2$	$8sbh$	GPT-3, Falcon

Llama2 scales the hidden size to $\frac{8}{3}h$ (Linear.Up-$\frac{8}{3}h$) and then scales it down to h (Linear.Down-$\frac{8}{3}h$). GPT-3 and Falcon scale the hidden size to $4h$ (Linear.Up-$4h$) and then scale it down to h (Linear.Down-$4h$). SwiGLU.Gate, SwiGLU.SiLU, and SwiGLU.Mul are the gated linear unit, the SiLU function, and the element-wise multiplication in SwiGLU, respectively.

Activation Memory: We analyze the activation memory for three typical transformer-based model architectures (i.e., GPT-3 [1], Llama2 [2] and Falcon [5]). A transformer layer contains an attention block and an MLP block. The attention block consists of three parts: a set of query, key, and value transformations; a self-attention operation; and a post-linear operation. Table I summarizes these operations. The input undergoes the query, key, and value transformations before being processed by the self-attention operation. The query transformation maintains the input's hidden size, while the key and value transformations may either maintain or reduce it, depending on the attention mechanism used.

Table II lists the operations in the MLP block. The MLP block contains two linear operations, of which the first one scales up the hidden size and the second one scales down the hidden size to h. An activation function is applied to the output of the first linear operation. GPT-3 and Falcon use GeLU [36] as the activation function, and Llama2 uses SwiGLU [37], which

consists of a gated linear unit, a SiLU [38] function, and an element-wise multiplication. Besides the attention and MLP blocks, the normalization and dropout operations also contribute to activation memory.

Adding up the memory required for each operation, the activation memory per transformer layer for GPT-3 is $34sbh$, and that for Llama2 is $\frac{203}{6}sbh$. Since Falcon uses parallel attention and MLP blocks, only the sum of these two blocks' outputs is stored, and the activation memory per layer is $\frac{53}{2}sbh$. The operations in a layer vary greatly in computational costs. The GeLU and SiLU functions, normalization operations, and dropout operations are computationally much less expensive than the other operations. This variety allows PIPEMESH to recompute mostly the computationally lightweight operations to satisfy the memory requirement.

C. Challenges

DP communication is a significant bottleneck when training LLMs on commodity clouds. This is because existing systems (e.g., Megatron and MegaScale) typically employ the depth-first schedule, which provides limited windows for overlapping DP communication. Specifically, for P pipeline stages, DP communication can only overlap with the forward computation on the first P micro-batches and the backward computation on the last P micro-batches in an iteration. Our experiment training a 58B GPT-3 model using Megatron on the AWS 72 A100 GPU cluster demonstrates that only 31.3% of DP communication could be effectively overlapped with computation. The duration of the remaining non-overlapped DP communication is equal to 25.1% of the total computation time, representing a significant overhead.

Existing optimizations to mitigate the DP communication bottleneck primarily rely on either increasing batch size [3] or using asynchronous updates. The asynchronous update typically employs a one-step delay approach [39], where the optimization step in the $(i+1)^{th}$ iteration uses gradients from the i^{th} iteration, allowing the corresponding gradient synchronization communication to overlap with the $(i+1)^{th}$ iteration. However, larger batch sizes require training on more tokens to achieve the same training loss [33], [40], while the gradient staleness introduced by asynchronous updates slows down training convergence [41]. Consequently, despite higher training throughput, the total training time actually increases. Our evaluation of training a 58B GPT-3 model in Section VI-B demonstrates this trade-off.

III. PIPEMESH SYSTEM

A. Overview

Fig. 6 shows the architecture of PIPEMESH. To train an LLM, users need to input the LLM and the training configuration (i.e., the 3D parallel configuration) into PIPEMESH. The 3D parallel configuration can be generated by existing works like Piper [42], and we empirically find that the configuration generated by Piper is also optimal for PIPEMESH. On each device, PIPEMESH's runtime contains a *planner* and an *executor*. The planner divides the model into model chunks, derives the sharding strategy (ZeRO stage used for each parameter), and then selects the buffer queue size with the enqueue-dequeue process and the pipeline-aware selective recomputation strategy (see Section IV-B).

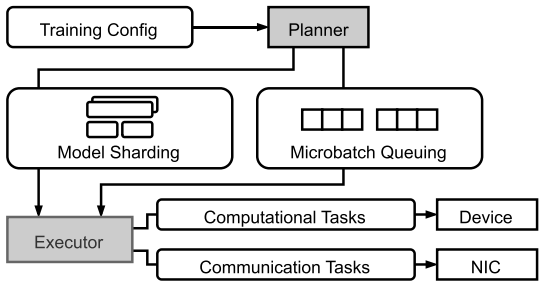


Fig. 6. PIPEMESH's Architecture. The gray boxes are PIPEMESH's components. The tasks are assigned to the device and its network interface card (NIC).

The sharding strategy and the enqueue-dequeue process are then fed into PIPEMESH's executor to generate the elastic pipeline schedule. The executor schedules tasks according to the pipeline schedule and assigns the computational tasks to the device and the communication tasks to the device's network interface card (NIC). The computational tasks employ the recomputation strategy selected by the planner. The DP communication tasks are generated based on the sharding strategy (see Section III-C) and overlap with the computational tasks. The PP and TP communication tasks are handled in the same way as existing works [16], [19].

B. Pipeline Schedule

The elastic pipeline schedule realizes the novel enqueue-dequeue mechanism introduced by PIPEMESH, which aims to maximize the overlap of DP communication while adhering to memory constraints. Unlike existing approaches that use fixed schedules, our approach uses two sequences (enqueue and dequeue) to allow for flexible micro-batch processing. We denote the enqueue sequence as $(e_1, e_2, \ldots)$, which means that the i^{th} enqueue operation processes e_i micro-batches together, and the dequeue sequence as $(d_1, d_2, \ldots)$, which means that the i^{th} dequeue operation processes d_i micro-batches together. The sum of elements in a sequence equals the number of micro-batches in an iteration.

By adjusting the elements in the enqueue and dequeue sequences, PIPEMESH creates variable-sized windows for overlapping communication in the data parallel dimension. This brings two benefits: first, PIPEMESH can adapt to different network bandwidth constraints and cluster configurations; second, it provides granular control over communication overlap for different types of DP communication. Specifically, the first element in the enqueue sequence and the last element in the dequeue sequence determine the overlapping window for optimizer sharding communication, while the other elements define the overlapping windows for gradient and parameter sharding communication.

PIPEMESH uniquely generates the pipeline schedule in two distinct steps: ① it orders the forward passes according to the given enqueue sequence and the backward passes according

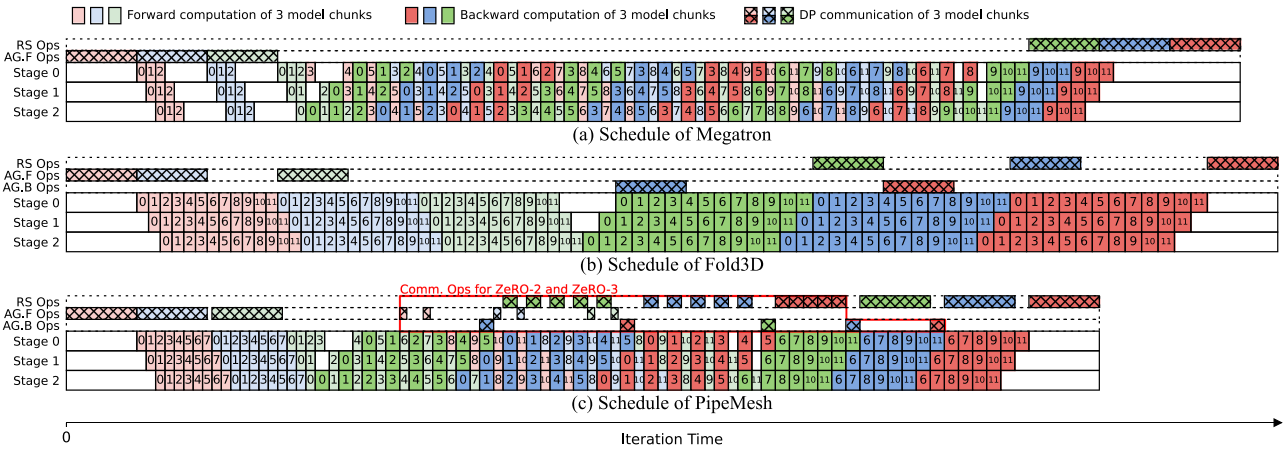


Fig. 7. Comparison of pipeline schedules. Fold3D incorporates ZeRO-3 and has a longer computation time due to activation offloading. We demonstrate the communication operations (comm. ops) on the first pipeline stage (stage 0). Among the operations for ZeRO-2 and ZeRO-3, reduce-scatter operations (RS ops) and all-gather operations for the backward computation (AG.B ops) overlap with the backward passes; all-gather operations for the forward computation (AG.F ops) overlap with the forward passes.

to the given dequeue sequence. PIPEMESH further overlaps DP communication with computation based on these ordered passes (see Section III-C); ② it determines the optimal interleaving of forward and backward passes, which minimizes the memory consumption for the given enqueue and dequeue sequences.

Step ②: All pipeline stages order the forward passes and backward passes in the same way. Forward passes on earlier enqueued micro-batches are executed before forward passes on later enqueued micro-batches, and similarly, backward passes on earlier dequeued micro-batches are executed before backward passes on later dequeued micro-batches. For a group of micro-batches enqueued or dequeued together, the forward or backward passes of each model chunk on these micro-batches are executed consecutively. The example given in Fig. 7(c) contains 12 micro-batches in an iteration. The enqueue process of PIPEMESH adopts sequence $(8, 4)$, which means PIPEMESH first enqueues 8 micro-batches and lets each pipeline stage complete the forward computation for micro-batches 0 to 7, and then enqueues the remaining 4 micro-batches and lets each pipeline stage complete the forward computation for micro-batches 8 to 11. The dequeue process of PIPEMESH adopts sequence $(6, 6)$, which means PIPEMESH first dequeues 6 micro-batches and lets each pipeline stage complete the backward computation for micro-batches 0 to 5, and then dequeues the remaining 6 micro-batches and lets each pipeline stage complete the backward computation for micro-batches 6 to 11.

Step ②: PIPEMESH interleaves forward and backward passes according to the task dependencies between pipeline stages. PIPEMESH runs the first backward pass right after its required forward pass finishes (for the last pipeline stage) or its required backward pass finishes (for other stages). Specifically, on the last stage, the backward pass starts after the last model chunk finishes its first forward pass. The backward passes on the other stages need to wait for inputs from the subsequent stages. After the first backward pass, PIPEMESH performs one forward pass followed by one backward pass (1F1B) until all forward passes are finished. PIPEMESH then finishes the remaining backward passes.

On each stage, the activation memory usage reaches the peak before the first backward pass starts. Afterward, each backward pass releases activation memory equal to that consumed by the previous forward pass, and the memory usage becomes steady. In Section IV-A, we formulate the memory consumption at each stage for the elastic pipeline schedule.

A requirement for the enqueue and dequeue sequences is that each element in both sequences should be no smaller than the number of pipeline stages to fill the pipeline. Specifically, due to the circular assignment of model chunks, the forward passes on the first stage depend on the forward passes on the last stage, and the backward passes on the last stage depend on the backward passes on the first stage. If the element is smaller than the number of pipeline stages, there would not be enough computational tasks between the forward or backward passes of different model chunks on the same stage, and an additional bubble would be introduced.

C. Communication Schedule

PIPEMESH builds on top of ZeRO-3. It utilizes the parameter and gradient sharding concepts from ZeRO-3 to reduce memory consumption on each device. PIPEMESH further introduces a novel communication scheduling mechanism specifically designed to optimize performance with pipeline parallelism, which is our primary contribution.

In network-constrained scenarios, applying ZeRO-3 to all parameters is problematic because, except for the breadth-first schedule, communication cannot be fully overlapped, resulting in significant training slowdowns (see Section VI-E). We propose a novel mixed sharding technique that applies different ZeRO stages to different parameters to address this limitation.

Like Megatron, PIPEMESH shards all the optimizer states. Once a model chunk has completed the backward computation for all micro-batches in an iteration, PIPEMESH shards gradients of that model chunk across D data parallel devices through a reduce-scatter operation, and each device retains $\frac{1}{D}$ of the gradients. The optimization step starts after the reduce-scatter

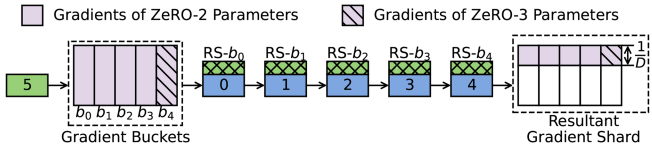


Fig. 8. A fraction of the backward passes from the pipeline schedule in Fig. 7(c). The gradients of the third model chunk are sharded during the backward computation of the second model chunk. The gradients of ZeRO-2 and ZeRO-3 parameters in a model chunk are divided into 5 buckets. RS-b_i represents the reduce-scatter operation for bucket b_i.

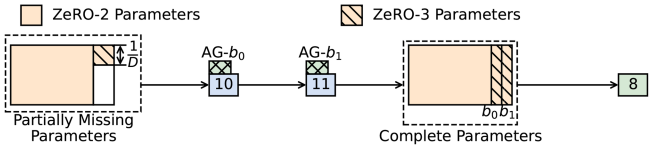


Fig. 9. A fraction of the forward passes from the pipeline schedule in Fig. 7(c). The ZeRO-3 parameters of the third model chunk are divided into two buckets, and corresponding all-gather operations overlap with the two forward passes of the second model chunk. AG-b_i represents the all-gather operation for bucket b_i.

operations for all model chunks finish. At the next iteration, PIPEMESH sequentially launches an all-gather operation for each model chunk to collect updated parameters. Starting from the second model chunk, the all-gather operation of a model chunk is initiated with the first forward pass of the model chunk preceding it. Compared with Megatron, PIPEMESH overlaps more communication for optimizer sharding with computation, benefiting from more micro-batches enqueued and dequeued together.

PIPEMESH shards the gradients of ZeRO-2 and ZeRO-3 parameters. The communication for gradient sharding of a model chunk is triggered whenever the backward passes switch from the computation of that model chunk to the computation of another. PIPEMESH overlaps the triggered reduce-scatter communication with subsequent backward passes. Specifically, PIPEMESH divides the gradients to be reduced into a few buckets (determined by the planner in Section IV-B) and overlaps the communication on a bucket with one backward pass. For example, when executing the pipeline schedule in Fig. 7(c), on each pipeline stage, the backward computation of the third model chunk on micro-batches 0 to 5 is followed by the backward computation of the second model chunk on micro-batches 0 to 5. PIPEMESH launches reduce-scatter operations for the third model chunk after its backward computation on micro-batch 5 ends. Fig. 8 further depicts how these reduce-scatter operations run in parallel with the backward computation of the second model chunk.

Similar to systems like DeepSpeed, PIPEMESH accumulates the gradients produced by the computation on different micro-batches in an iteration. The reduce-scatter communication results in each device retaining a shard of the gradients. PIPEMESH then accumulates the resultant gradient shard of a model chunk with the previous gradient shard of the same model chunk (if it exists). For example, in Fig. 7(c), the gradient shard generated from the reduce-scatter communication on gradients for micro-batches 0 to 5 is accumulated with the gradient shard generated from the reduce-scatter communication on gradients for micro-batches 6 to 11.

PIPEMESH shards ZeRO-3 parameters and replicates the remaining parameters across data parallel devices. PIPEMESH separately manages the ZeRO-3 parameters used for the forward and backward computation and ensures that the devices receive all parameters of a model chunk before the computation of that model chunk begins by launching all-gather communication in advance. To be specific, the ZeRO-3 parameters in each model chunk are divided into B_{Z3} buckets. Before the forward or backward passes switch to a model chunk, PIPEMESH performs B_{Z3} all-gather operations, each of which collects a bucket of parameters in that model chunk. These operations overlap with the B_{Z3} forward or backward passes that occur before the model chunk switch. Since the backward passes take longer than the forward passes, we use different B_{Z3} values for the forward and backward passes (see Section IV-B). Fig. 9 illustrates how a device collects the parameters of the third model chunk before its forward computation on micro-batch 8.

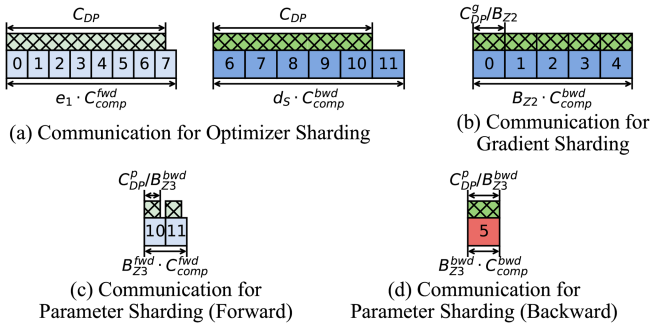
The collected ZeRO-3 parameters of a model chunk are released each time its forward passes on micro-batches enqueued together or its backward passes on micro-batches dequeued together finish.

Our mixed sharding technique reduces memory consumption while maintaining training performance by leveraging the strengths of different ZeRO stages and carefully scheduling communication tasks.

IV. PIPEMESH PLANNER

We use the following notation in this section:

- P: Pipeline parallel size (number of pipeline stages).
- D: Data parallel size (number of data parallel devices).
- M: Number of model chunks per pipeline stage.
- B: Number of micro-batches per pipeline.
- C_{comp}: Computation time of a model chunk on a micro-batch. We use C_{comp}^{fwd} for the time of a forward pass and C_{comp}^{bwd} for the time of a backward pass.
- C_{DP}: Time spent on reduce-scatter communication to reduce all gradients or all-gather communication to collect all parameters for a model chunk.
- C_{DP}^{g}: Time spent on reduce-scatter communication to reduce gradients of ZeRO-2 and ZeRO-3 parameters for a model chunk.
- C_{DP}^{p}: Time spent on all-gather communication to collect ZeRO-3 parameters for a model chunk.
- R: Length of the enqueue sequence $(e_1, e_2, \ldots, e_R)$.
- S: Length of the dequeue sequence $(d_1, d_2, \ldots, d_S)$.
- B_{Z2}: Number of gradient buckets for ZeRO-2 and ZeRO-3 parameters in a model chunk.
- $B_{Z3}^{fwd}, B_{Z3}^{bwd}$: Numbers of parameter buckets for ZeRO-3 parameters in a model chunk during the forward and backward passes, respectively.
- ψ, ψ_g, ψ_p: Numbers of parameters, ZeRO-2 and ZeRO-3 parameters, and ZeRO-3 parameters on a device, respectively.



(a) Communication for Optimizer Sharding

(b) Communication for Gradient Sharding

(c) Communication for Parameter Sharding (Forward)

(d) Communication for Parameter Sharding (Backward)

Fig. 10. DP communication of the third model chunk for the pipeline schedule in Fig. 7(c).

A. Cost Model

The objective of PIPEMESH's planner is to maximize throughput under the given memory constraint. Maximizing throughput equals minimizing the time of an iteration. In this subsection, we model the iteration time and the memory usage on each device for PIPEMESH and its elastic pipeline schedule. The iteration time T_{iter} is written as:

$$T_{iter} = T_{comp} + T_{bubble} + T^{crit}_{PP} + T^{crit}_{DP} \tag{1}$$

where T_{comp} is the total computation time of the forward and backward passes in an iteration, T_{bubble} is the pipeline bubble time, and T^{crit}_{PP} and T^{crit}_{DP} are the PP and DP communication times on the performance-critical path, respectively.

The computation time T_{comp} is calculated as $M \cdot B \cdot C_{comp}$. The recomputation time is included in C_{comp}, and all model chunks use the same recomputation strategy (see Section IV-B). The pipeline bubble time T_{bubble} is calculated as $(P-1)C_{comp}$, equal to the sum of the first forward passes and the last backward passes on all pipeline stages, excluding the first stage.

PIPEMESH prioritizes scenarios where communication-computation overlap is feasible. Specifically, the planner intentionally constrains its search space to configurations where C^{fwd}_{comp} remains larger than C_{PP} (defined as the time spent on receiving the input of a forward pass or backward pass) by limiting the number of model chunks per pipeline stage. Thus, only the PP communication of the first forward pass and the last backward pass does not overlap with computation. The critical-path PP communication T^{crit}_{PP} is calculated as $2(P-1)C_{PP}$. PIPEMESH may not be optimal for cases where pipeline communication dominates ($C_{PP} > C^{fwd}_{comp}$), such as ZeRO-1 without tensor parallelism. In such cases, the ideal strategy would be to minimize pipeline communication by placing only one model chunk per pipeline stage. However, this conflicts with PIPEMESH's core mechanism of using multiple model chunks to enable DP communication overlap.

To formulate the critical-path DP communication T^{crit}_{DP}, we divide the DP communication into the communication for optimizer sharding, gradient sharding, and parameter sharding, respectively. On each pipeline stage, the communication for optimizer sharding of all model chunks, except for the first model chunk, overlaps with computation (see Fig. 10(a)). The reduce-scatter communication of a model chunk can overlap with at most e_1 forward passes, and the all-gather communication can overlap with at most d_S backward passes. e_1 is the first element in the enqueue sequence, and d_S is the last element in the dequeue sequence. We formulate the critical-path communication $\mathcal{D}_o$ for optimizer sharding as:

$$\mathcal{D}_o = 2C_{DP} + (M-1)\left(\mathcal{H}\left(C_{DP}, e_1 \cdot C^{fwd}_{comp}\right) + \mathcal{H}\left(C_{DP}, d_S \cdot C^{bwd}_{comp}\right)\right) \tag{2}$$

where $\mathcal{H}(C, C') = \max(0, C - C')$ checks whether communication C fully overlaps with computation C'. If not, the computation time is deducted from the communication time.

The communication for gradient sharding of all model chunks takes the same amount of time. For a model chunk, each time it finishes the backward computation on micro-batches dequeued together, PIPEMESH triggers the corresponding gradient sharding communication, which overlaps with the subsequent B_{Z2} backward passes (see Fig. 10(b)). The gradient sharding communication of each model chunk is triggered $S-1$ times for S dequeue operations in an iteration since the last dequeue operation triggers the optimizer sharding communication. The critical-path communication $\mathcal{D}_g$ for gradient sharding is formulated as:

$$\mathcal{D}_g = (S-1)M \cdot \mathcal{H}\left(C^g_{DP}, B_{Z2} \cdot C^{bwd}_{comp}\right) \tag{3}$$

We separately consider the communication incurred by parameter sharding for forward and backward computation. For each model chunk, PIPEMESH performs all-gather communication each time before the model chunk starts forward computation on micro-batches that are enqueued together or backward computation on micro-batches that are dequeued together. The communication overlaps with B^{fwd}_{Z3} forward passes (see Fig. 10(c)), or B^{bwd}_{Z3} backward passes (see Fig. 10(d)). The all-gather communication for forward computation is triggered for every enqueue operation except the first one. The critical-path communication $\mathcal{D}^{fwd}_p$ for parameter sharding during the forward computation is written as:

$$\mathcal{D}^{fwd}_p = (R-1)M \cdot \mathcal{H}\left(C^p_{DP}, B^{fwd}_{Z3} \cdot C^{fwd}_{comp}\right) \tag{4}$$

The all-gather communication for backward computation is triggered for all dequeue operations. The last model chunk directly uses the parameters for forward computation and does not require all-gather communication for the first dequeue operation. The critical-path communication $\mathcal{D}^{bwd}_p$ for parameter sharding during the backward computation is written as:

$$\mathcal{D}^{bwd}_p = (S \cdot M - 1) \cdot \mathcal{H}\left(C^p_{DP}, B^{bwd}_{Z3} \cdot C^{bwd}_{comp}\right) \tag{5}$$

The critical-path DP communication T^{crit}_{DP} equals the sum of $\mathcal{D}_o$, $\mathcal{D}_g$, $\mathcal{D}^{fwd}_p$, and $\mathcal{D}^{bwd}_p$.

Memory Usage: We model the memory consumed by the optimizer, gradients, and parameters for the mixed precision training and Adam optimizer setup. The optimizer states on a device require $\mathcal{M}_o = \frac{12}{D}\psi$ bytes of memory.

PIPEMESH allocates intermediate buffers to stage the communication for buckets, which introduces a manageable memory overhead. For gradient sharding, the buffer size is equivalent to the total size of the ZeRO-2 and ZeRO-3 parameters' gradients

within a single model chunk. For parameter sharding, the total buffer size is twice the number of ZeRO-3 parameters in a model chunk since we allocate separate buffers for the forward and backward passes.

PIPEMESH allocates up to $\mathcal{M}_g = 2(\psi - \psi_g) + (\frac{2}{D} + \frac{4}{M})\psi_g$ bytes of memory for gradients. The gradients of ZeRO-1 parameters need $2(\psi - \psi_g)$ bytes of memory, and the gradient partition of ZeRO-2 and ZeRO-3 parameters needs $\frac{2}{D}\psi_g$ bytes of memory. Besides, the backward computational task requires $2\frac{\psi_g}{M}$ bytes of memory for a model chunk's full ZeRO-2/ZeRO-3 gradients, and the intermediate buffer for gradient sharding uses another $2\frac{\psi_g}{M}$ bytes.

PIPEMESH allocates up to $\mathcal{M}_p = 2(\psi - \psi_p) + (\frac{2}{D} + \frac{8}{M})\psi_p$ bytes of memory for parameters. The ZeRO-1 and ZeRO-2 parameters need $2(\psi - \psi_p)$ bytes of memory, and the partition of ZeRO-3 parameters needs $\frac{2}{D}\psi_p$ bytes of memory. The additional $\frac{8}{M}\psi_p$ bytes of memory are used as follows: forward and backward computational tasks each require $\frac{2}{M}\psi_p$ bytes to store full ZeRO-3 parameters of a model chunk, while the intermediate buffer for parameter sharding occupies $\frac{4}{M}\psi_p$ bytes. Since $\mathcal{M}_p$ is no smaller than 2ψ when $M \leq 4$, PIPEMESH only enables ZeRO-3 when $M > 4$.

The activation memory is determined by the number of consecutive forward passes before the first backward pass. According to the pipeline schedule introduced in Section III-B, the i^{th} pipeline stage executes up to $e_1(M-1) + 2(P-i) + 1$ consecutive forward passes. Each pipeline stage first performs the forward computation of the first $M-1$ model chunks on the e_1 micro-batches. Afterwards, the last stage executes one extra forward pass followed by the first backward pass, while each preceding stage performs two additional forward passes. The required activation memory $\mathcal{M}_a$ is calculated as the size of the activations stored by these consecutive forward passes. We define $\mathcal{M}_{sum}$ as the sum of $\mathcal{M}_o$, $\mathcal{M}_g$, $\mathcal{M}_p$, and $\mathcal{M}_a$.

The theoretical lower bound for the iteration time T_{iter} in PIPEMESH is $(M \cdot B + P - 1)C^*_{comp} + T^{crit}_{PP} + 2C_{DP}$. C^*_{comp} is the minimum value of C_{comp} and represents the computation time excluding recomputation. $2C_{DP}$ corresponds to the optimizer sharding communication time for the first model chunk, which necessarily remains on the critical path. In our experimental evaluation, PIPEMESH achieves iteration times that are 5.1% to 7.2% larger than this theoretical lower bound. The gap between theoretical and achieved performance can be attributed to two main factors (see Section VI-A): the practical slowdown in computation when overlapping with communication and the additional computation overhead introduced by PIPEMESH's selective recomputation strategy.

B. Planning Algorithm

PIPEMESH's planner determines the optimal buffer queue size q of the elastic pipeline schedule and identifies the set of operations O to be recomputed in order to minimize the iteration time under the memory budget $\mathcal{M}_b$:

$$\arg\min_{q,O} T_{iter} \quad \text{subject to } \mathcal{M}_{sum} < \mathcal{M}_b \tag{6}$$

The planner mainly consists of two parts: the first part determines the enqueue and dequeue sequences and the sharding strategy for all possible buffer sizes; the second part determines the buffer size and the pipeline-aware selective recomputation strategy of PIPEMESH jointly.

Determine the model chunk number: The planner first determines the number of model chunks. As the number of model chunks increases, the DP communication of the first model chunk, which is always on the performance-critical path, decreases. Thus, the planner increases the number of model chunks M as long as the PP communication time C_{PP} does not exceed the forward pass time C^{fwd}_{comp}.

Determine the enqueue sequence: The enqueue sequence is derived from the buffer queue size q. The planner sets the first element in the enqueue sequence e_1 to q to maximize the window for overlapping the communication incurred by optimizer sharding. The remaining $B-q$ micro-batches are evenly distributed across $\lceil \frac{B-q}{q} \rceil$ enqueue operations to minimize the sequence length and the number of model chunk switches. The planner uses a dequeue sequence of the same length as the enqueue sequence ($R = S$), allowing the enqueue and dequeue processes to be interleaved.

Determine the dequeue sequence and sharding strategy: The planner ensures that the communication for gradient sharding and parameter sharding can fully overlap with computation by selecting proper ψ_g and ψ_p. Before that, the planner determines the dequeue sequence based on the comparison between the total backward computation time $B \cdot C^{bwd}_{comp}$ and potential DP communication time $S \cdot C_{DP}$. If $B \cdot C^{bwd}_{comp} \geq S \cdot C_{DP}$, it indicates that even if gradient sharding is enabled for all parameters, the corresponding communication can fully overlap with computation. Therefore, the planner sets all parameters to either ZeRO-2 or ZeRO-3 (i.e., $\psi_g = \psi$). The micro-batches are evenly divided across the dequeue operations, and all elements in the dequeue sequence are set to $\frac{B}{S}$. B_{Z2} is set to $\lceil \frac{C_{DP}}{C^{bwd}_{comp}} \rceil$, which corresponds to the minimum number of backward passes required to hide communication for gradient sharding of a model chunk. We use the computation time of a backward pass without recomputation because the recomputation strategy has not been determined at this stage. PIPEMESH then uses the remaining backward passes for parameter sharding and sets B^{bwd}_{Z3} to $\frac{B}{S} - B_{Z2}$ and the number of ZeRO-3 parameters ψ_p to the largest value that makes $\mathcal{D}^{bwd}_p = 0$. Similarly, B^{fwd}_{Z3} is set to $\lceil \frac{C^p_{DP}}{C^{fwd}_{comp}} \rceil$.

If $B \cdot C^{bwd}_{comp} < S \cdot C_{DP}$, the planner sets a portion of parameters to ZeRO-2 and the remaining parameters to ZeRO-1 (i.e., $\psi_p = 0$) to avoid critical-path communication for gradient sharding. d_S, the last element in the dequeue sequence, is set to the minimum value that satisfies $d_S \cdot C^{bwd}_{comp} \geq C_{DP}$. The remaining $B - d_S$ micro-batches are then evenly assigned to $S-1$ dequeue operations, and B^{bwd}_{Z3} equals $\frac{B-d_S}{S-1}$. The number of ZeRO-2 parameters ψ_g is increased as long as the corresponding communication for gradient sharding can be fully overlapped ($D_g = 0$).

Overall, for each buffer queue size q, the planner determines the enqueue and dequeue sequences with the sharding strategy

and then calculates the critical-path DP communication time $T_{DP}^{crit}(q)$ and the allocated memory $\mathcal{M}_{sum}(q)$.

Determine buffer queue size and pipeline-aware selective recomputation strategy: Selective recomputation allows PIPEMESH to enable a larger buffer size with a larger window for overlapping communication with computation. The planner selects the buffer size and determines which activations to recompute in order to minimize the iteration time, taking into account both the critical-path DP communication time and the time required for recomputation. The same recomputation strategy is used for all model chunks, and the computation times of different model chunks for a micro-batch are the same. PIPEMESH collects the sizes of activations and the computation times for all operations in a model chunk. After performing a topological sort for the DAG of these operations, we denote the activation size of the k-th operation as o_k, and its computation time as u_k. We present an algorithm based on dynamic programming to find the optimal set of operations to recompute.

We define a dynamic programming table $L[k][o]$ as the minimum execution time when the memory consumed by activations of the first k operations is o. The execution time sums the recomputation time and the critical-path DP communication time. We initialize the table as:

$$L[0][0] = T_{DP}^{crit}(B) \tag{7}$$

which is the critical-path DP communication time when the buffer queue size equals the micro-batch number B. We recursively construct the dynamic programming table based on the optimal substructure of the subproblems:

$$\begin{aligned} L[k][o] = \min (& L[k-1][o] + j \cdot u_k, L[k-1][o-o_k] \\ & + I(o-o_k, o)) \end{aligned} \tag{8}$$

where $L[k-1][o]$ and $L[k-1][o-o_k]$ are the subproblems for which we have planned the recomputation strategy for the first $k-1$ operations. The first term in the $\min$ function refers to the case where the k-th operation is recomputed. j equals $M \cdot B + (P-1)$ and $j \cdot u_k$ corresponds to increased computation time and pipeline bubble time due to recomputing the operation. The second item corresponds to the case where the activation of the operation is stored.

We define $I(o', o)$ as the increased critical-path DP communication when the activation memory is changed from o' to o, and we formulate it as:

$$I(o', o) = T_{DP}^{crit}(F(o)) - T_{DP}^{crit}(F(o')) \tag{9}$$

Function $F(o)$ looks for the largest buffer queue size q that satisfies $\mathcal{M}_{sum}(q) < M_b$ when the size of activations stored per forward pass is o.

After constructing the dynamic programming table, we choose $o^* = \arg\min_o L[K][o]$, where K is the number of operations in a model chunk. The buffer queue size is set to $F(o^*)$. We can reconstruct the optimal set O by tracing back the operations recomputed when forming $L[K][o^*]$.

The search space size is $B \cdot 2^K$, where B represents the number of possible buffer queue size choices and 2^K is the number of possible recomputation strategy choices, as each operation can either be recomputed or not. The time complexity of our algorithm is $O(K\frac{A^{\max}}{A^{\min}} + B)$, where $A^{\max}$ represents the sum of activation sizes across all operations, and $A^{\min}$ is the greatest common divisor of all operation activation sizes. $O(B)$ is the time used to construct function $F(\cdot)$, which maps activation memory to PIPEMESH's buffer queue size.

Algorithm 1: PIPEMESH Executor.

Input: enqueue sequence $s_e = (e_1, e_2, \cdots)$, dequeue sequence $s_d = (d_1, d_2, \cdots)$, pipeline parallel size P, model chunk number M and micro-batch number B

```
1  Initialize model chunk lists chunk_fwd, chunk_bwd;
   // number of consecutive forward passes
      at the start of an iteration
2  n_fwd ← (M − 1)e_1 + 2(P − i);
   // number of forward passes followed by
      backward passes
3  n_fwdbwd ← M · B − n_fwd;
4  for j = 1 to s_e.length do
5      for k = 1 to M do
           // add the model chunk ID to
              forward list
6          for n = 1 to e_j do
7              chunk_fwd.append(k);
8  for j = 1 to s_d.length do
9      for k = M to 1 do
           // add the model chunk ID to
              backward list
10         for n = 1 to d_j do
11             chunk_bwd.append(k);
12 for n = 1 to n_fwd do
13     forwardPass();
14 for n = 1 to n_fwdbwd do
15     forwardPass();
16     backwardPass();
17 for n = 1 to n_fwd do
18     backwardPass();
19 Function forwardPass():
20     chunkID ← chunk_fwd.popleft();
21     allGather((chunkID + 1) mod M);
22     forward(chunkID);
23 Function backwardPass():
24     chunkID ← chunk_bwd.popleft();
25     if gradBucket((chunkID + 1) mod M) then
26         reduceScatter((chunkID + 1) mod M);
27     else
28         allGather((chunkID − 1) mod M);
29     backward(chunkID);
```

V. PIPEMESH EXECUTOR

The executor realizes the elastic pipeline schedule given the enqueue and dequeue sequences. Algorithm 1 describes the executor's logic: it invokes all computational and communication tasks based on the pipeline stage rank i. The executor first orders the forward and backward passes, respectively (lines 4– 11). It then sequentially executes a few forward passes (line 13), interleaved forward and backward passes (lines 15 and 16), and a few backward passes (line 18).

During the forward passes, the executor launches all-gather communication (line 21) for the ZeRO-3 parameters of the next model chunk. During the backward passes, the executor launches either reduce-scatter communication for gradients of the previous model chunk's ZeRO-2 and ZeRO-3 parameters (line 26) or all-gather communication (line 28) for the ZeRO-3 parameters of the next model chunk.

TABLE III
MODELS USED FOR END-TO-END EVALUATION

Model	l	n_q	n_{kv}	h	s
GPT-3 58B	72	64	64	8192	3072
Llama2 55B	80	64	8	8192	4096
Falcon 66B	96	64	8	8192	3072

l is the number of layers. n_q and n_{kv} are the numbers of query heads and key/value heads in the attention block, respectively. h is the hidden size of a transformer layer. s is the length of an input sequence.

VI. EVALUATION

Testbeds: We conducted most of the experiments on an AWS cluster with 12 p3dn.24xlarge nodes. Each node has 8 NVIDIA V100 GPUs (32 GB memory and 125 fp16 TFLOPS per GPU) interconnected by NVLink, and the nodes are connected through a 100 Gbps network. We further evaluated on an AWS cluster with 12 p4d.24xlarge nodes. Each node has 8 NVIDIA A100 GPUs (40 GB memory and 312 fp16 TFLOPs per GPU) interconnected by NVLink, and the nodes are connected through a 400 Gbps network. We also used 48 pnv4.28xlarge nodes on Tencent Cloud with a total of 192 NVIDIA A10 GPUs to evaluate the scalability of PIPEMESH. Each node has 4 NVIDIA A10 GPUs (24 GB memory and 125 fp16 TFLOPS per GPU). The network bandwidth of a node is 50 Gbps. Unless otherwise specified, we used the AWS cluster with 96 V100 GPUs as our default testbed.

Baselines: We took Megatron [16], MegaScale [19], and Fold3D [21] as our baselines. Megatron and MegaScale are the state-of-the-art 3D parallel training systems for LLMs. Fold3D is designed for 3D parallel training on commodity clouds and can overlap most of the DP communication. The Megatron release (Core 0.4.0) we used provides the functionality of overlapping DP and PP communication with computation. We implemented MegaScale's communication component based on Megatron's codebase according to the description in MegaScale's paper. Specifically, we modified Megatron's data loader and performed a series of data loading operations before computation. By doing so, we overlapped the first all-gather operation with the advanced data loading operations. We also decoupled the point-to-point send and receive operations following MegaScale. We did not compare with pure data parallelism approaches such as ZeRO stages 1-3 or PyTorch FSDP. This is because pure data parallelism faces excessive DP communication costs for synchronizing full parameters and gradients of a model across all devices. In contrast, 3D parallelism leverages the complementary strengths of three parallelization dimensions to enable superior throughput [16].

ZeRO stages: We used ZeRO-3 for Fold3D and ZeRO-1 for Megatron and MegaScale. PIPEMESH applied different ZeRO stages to different parameters.

Data, pipeline, tensor parallelism degrees: We list the parallelization configurations, including the number of model chunks per pipeline stage, in Table IV. The parallelization configuration is derived by Piper [42], an algorithm that automatically determines the optimal configuration for the given model and cluster. For all systems, we enabled sequence parallelism [10] in the tensor parallel dimension.

CPU offloading: Fold3D offloaded activations that couldn't fit into GPU memory to CPU memory, while other systems kept the model states and activations in GPU memory.

Checkpoint frequency: We saved a model checkpoint to CPU memory every 500 iterations. We used synchronous checkpointing, and the overhead was small in practice.

Gradient accumulation: We set the micro-batch size to 1, which is large enough to saturate GPU ALUs. We accumulated gradients of $\frac{B}{P}$ micro-batches per device for batch size B and pipeline parallel degree P.

Models: We trained 3 notable LLMs (i.e., GPT-3 [1], Llama2 [2], and Falcon [5]). GPT-3 is natively supported by Megatron, and we used the operations (e.g., RMS-norm [43], SwiGLU [37] and grouped-query attention [44]) provided by Megatron to implement Falcon and Llama2. We selectively recomputed the self-attention operations following common practices [10], [45]. We chose the model sizes that can be supported by Megatron without incurring out-of-memory errors. When Megatron enables DP, it can support smaller model sizes compared to those supported by ZeRO-3. This is because Megatron replicates the model parameters and gradients across data parallel devices, which significantly increases memory consumption per device compared to ZeRO's parameter sharding approach.

The official Llama2 suite includes a limited number of model size options. To test larger models that our cluster could support, we constructed additional model size variants by experimenting with different numbers of layers and hidden sizes while maintaining the core architecture of Llama2. The model sizes selected favor the baselines, which use pipeline schedules designed for memory-constrained cases. Detailed model architectures are listed in Table III. We trained these models on the OpenWebText [46] dataset. To demonstrate the effectiveness of PIPEMESH, we set the memory budget for each model to match the memory usage of Megatron, allowing for a fair comparison of performance under equivalent memory constraints.

Training configurations: We trained each model on 5B tokens. We used the cosine learning rate scheduler and warmed up the learning rate on 0.25B tokens. We determined the learning rate by testing a range of learning rates and selecting the one that achieves the minimum loss after training on 0.5B tokens. For each model, we selected the per-iteration batch size that allowed Megatron to make that model converge in the shortest amount of time.

One-step delayed update: We implemented the one-step delay mechanism [39] in Megatron by delaying the reduce-scatter and all-gather communication by one step. Specifically, in the

TABLE IV
BREAKDOWN OF THE PERFORMANCE CRITICAL PATH WHEN TRAINING THREE LLMS

Model	B	T,P,D	System	M	q	T_{comp}^{fwd}	T_{comp}^{bwd}	T_{bubble}	T_{DP}^{crit}	T_{PP}^{crit}	$\mathcal{M}_{CPU}^{extra}$	$\mathcal{M}_{GPU}$	$\mathcal{M}_a$	$\mathcal{M}_g$	$\mathcal{M}_p$	Thrp.	MFU
GPT-3 58B 72×A100	36	(8, 3, 3)	PIPEMESH	6	6	1554.1	2567.2	114.5	245.1	140.3	n/a	29.9	11.5	3.0	3.9	120.6	38.7%
			Megatron	2	n/a	1492.2	2485.4	331.5	993.6	262.8	n/a	30.1	9.6	4.5	4.5	100.4	32.2%
			MegaScale	2	n/a	1501.3	2497.0	333.2	986.4	254.0	n/a	30.1	9.6	4.5	4.5	100.3	32.2%
			Fold3D	6	n/a	2312.1	3431.6	159.5	247.7	134.9	73.5	37.0	28.7	3.0	3.0	89.3	28.6%
GPT-3 58B 96×V100	72	(8, 4, 3)	PIPEMESH	6	12	5012.0	9217.1	296.4	757.9	505.9	n/a	25.3	11.4	2.3	2.7	52.1	41.7%
			Megatron	2	n/a	4914.5	9109.6	876.5	3923.1	922.7	n/a	25.6	9.9	3.4	3.4	42.1	33.7%
			MegaScale	2	n/a	4917.9	9104.5	876.4	3863.8	919.0	n/a	25.6	9.9	3.4	3.4	42.3	33.8%
			Fold3D	6	n/a	7997.2	12225.0	421.3	760.4	509.7	216.9	29.0	43.0	2.3	2.3	38.1	30.5%
Llama2 55B 96×V100	48	(8, 4, 3)	PIPEMESH	5	8	4883.6	8841.1	514.7	859.2	635.6	n/a	25.6	11.8	2.4	2.8	46.2	37.0%
			Megatron	1	n/a	4789.0	8664.1	2522.5	4263.8	1155.1	n/a	25.7	10.6	3.2	3.2	34.8	27.7%
			MegaScale	1	n/a	4769.7	8651.5	2516.5	4223.1	1167.5	n/a	25.7	10.6	3.2	3.2	34.5	27.6%
			Fold3D	5	n/a	7842.8	11725.7	733.8	854.7	617.6	213.4	29.2	42.5	2.4	2.4	34.0	27.2%
Falcon 66B 96×V100	72	(8, 4, 3)	PIPEMESH	8	12	6035.9	10805.4	263.1	638.8	548.7	n/a	26.5	11.6	2.2	2.7	52.2	41.8%
			Megatron	3	n/a	5911.3	10676.7	691.1	4595.8	967.5	n/a	26.9	9.3	3.8	3.8	42.3	33.8%
			MegaScale	3	n/a	5917.4	10687.9	691.9	4566.4	955.1	n/a	26.9	9.3	3.8	3.8	42.3	33.8%
			Fold3D	8	n/a	9117.8	13900.9	359.7	632.1	536.4	240.0	29.1	44.7	2.2	2.2	39.4	31.6%

We list each model with the cluster used. We report the batch size per iteration (B) and the 3D parallel configuration (tensor parallel size T, pipeline parallel size P, and data parallel size D). M is the number of model chunks on a pipeline stage, and q is the buffer queue size used by the elastic pipeline schedule of PipeMesh. We also report the forward computation time (T_{comp}^{fwd}, in microseconds), the backward computation time (T_{comp}^{bwd}), the pipeline bubble time (T_{bubble}), the critical-path DP communication time (T_{dp}^{crit}), and the critical-path PP communication time (T_{dp}^{crit}) in an iteration. Specifically for Fold3D, we report its extra CPU memory used for offloading ($\mathcal{M}_{CPU}^{extra}$) per host.We list the peak GPU memory usage ($\mathcal{M}_{GPU}$, in GB), peak activation memory usage ($\mathcal{M}_a$), peak gradient memory usage ($\mathcal{M}_g$) and peak parameter memory usage ($\mathcal{M}p$) for the first pipeline stage. We provide the device throughput (Thrp., in TFLOPS) and model FLOPs utilization (MFU). n/a means the column is not applicable to the system as explained in Section 6.1.

$(i+1)^{th}$ iteration, we first perform reduce-scatter communication on gradients from the i^{th} iteration, then update the optimizer states, and finally broadcast the updated parameters to all data parallel devices through all-gather communication. Given sufficient GPU memory in our setup, we maintained these gradients and parameters for communication in GPU memory.

We focus on addressing the following questions. Section VI-A: Does PIPEMESH enable faster training for LLMs? Section VI-B: How do batch sizes and asynchronous updates affect training convergence? Section VI-C: How does PIPEMESH perform in large-scale training? Section VI-D: How does PIPEMESH perform under different training hyper-parameters (e.g., the number of model chunks)? Section VI-E: How effective are PIPEMESH's components?

A. End-to-End Performance

Table IV lists the performance breakdown of PIPEMESH and the baselines when training three LLMs. The iteration time is broken down into forward computation time, backward computation time, pipeline bubble time, and critical PP and DP communication times. The memory usage is categorized into three parts: memory consumed by activations, gradients, and parameters. The elastic buffer queue size is specific to the elastic pipeline schedule of PIPEMESH. For all three LLMs, the planner of PIPEMESH found dequeue sequences identical to the enqueue sequences, which contain two elements. We evaluate the performance of these systems using two key metrics: throughput per GPU and model FLOPs utilization (MFU). MFU is calculated as the ratio of model FLOPs per second to the GPU's theoretical peak throughput. Model FLOPs represent the floating point operations required for the forward and backward computation, irrespective of implementations such as recomputation.

On the A100 cluster, PIPEMESH achieved a throughput of 120.6 TFLOPS, which is 20.1% higher than Megatron. This performance improvement stems from PIPEMESH's ability to reduce critical-path DP communication by 75.3%. Megatron only managed to overlap 31.3% of DP communication due to limited overlapping windows. In contrast, PIPEMESH's elastic pipeline schedule provided larger overlapping windows, enabling it to overlap 82.5% of DP communication. On the V100 cluster, PIPEMESH demonstrated the highest throughput, ranging from 46.2 to 52.2 TFLOPS, for all the LLMs evaluated. This represents a 1.24× to 1.34× speedup over both Megatron and MegaScale.

Despite the significant portion of iteration time attributed to DP communication (optimizer sharding communication in Megatron), the 3D parallelism configuration derived by Piper remains optimal. Increasing the pipeline parallel size further leads to a reduction in DP communication; however, this is offset by increased PP communication and a larger pipeline bubble, resulting in an overall increase in iteration time.

MegaScale achieved similar performance to Megatron due to the use of the depth-first pipeline schedule, which limits the overlap between DP communication and computation. However, MegaScale's critical DP communication is marginally smaller than Megatron's because MegaScale also overlaps the DP communication with data loading operations. In our evaluation, the data loading operations per iteration ranged from 40.3 to 78.1 microseconds.

PIPEMESH also reduced the critical PP communication compared to Megatron. Breath-first systems [21], [22] reveal that Megatron leaves the PP communication of forward passes on the first micro-batch and backward passes on the last micro-batch on the performance-critical path. By running micro-batches more than the pipeline parallel size during the warm-up phase, the critical PP communication, except for that of the first forward pass and the last backward pass, can be hidden.

PIPEMESH divided the model into more model chunks compared with Megatron. The critical PP communication in Megatron, which is proportional to the number of forward and backward passes on a micro-batch, increases with the number

of model chunks. Although Megatron can reduce the pipeline bubble by increasing the number of model chunks, the benefits are overshadowed by the increased critical PP communication. In contrast, PIPEMESH is designed to hide most of the PP communication, allowing it to benefit from a larger number of model chunks without incurring the same communication overhead.

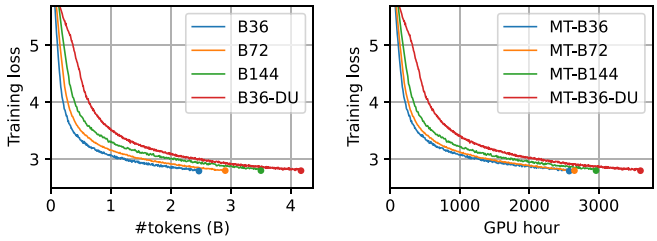
When training the Llama2 55B model with Megatron, we encountered out-of-memory errors when attempting to place multiple model chunks on a pipeline stage. While Falcon 66B has more parameters than Llama2 55B, the per-layer activation size of Llama2 is actually larger than Falcon's when processing the same number of tokens with the same hidden size (as detailed in Section II-B). Additionally, we used a longer sequence length for training Llama2 (see Table III). In our experiments, Falcon 66B generated 1.86 GB of activations per device for a micro-batch, while Llama2 55B generated 2.66 GB of activations per device. According to the formula for calculating warm-up forward passes defined in Section II-A, Llama2 would require an additional 4.2 GB of memory when placing two model chunks per pipeline stage, which exceeded the GPU memory constraint.

Compared with Megatron, the computation time in PIPEMESH is slightly larger due to two factors. The first factor is that the communication overlapped with the computation would contend for GPU resources and slow down the computation [47]. The computation in PIPEMESH, which overlapped with more communication, became slower. However, the contention only slowed down the computation by 2.4% to 3.1%.

The second factor is that PIPEMESH recomputed more operations, increasing the backward computation time. When training GPT-3 58B on the V100 cluster, PIPEMESH recomputed all GeLU, layer-norm and dropout operations in a model chunk. Besides, PIPEMESH also recomputed one query transformation operation among three transformer layers in a model chunk. For Llama2 55B, PIPEMESH recomputed all SiLU, RMS-norm and element-wise multiplication operations and three key-value transformation operations among four transformer layers in a model chunk. For Falcon 66B, PIPEMESH recomputed all GeLU and layer-norm operations in a model chunk. PIPEMESH only incurred small recomputation overhead. The additional recomputation time introduced by PIPEMESH for GPT-3 58B is less than 2.6% of the forward computation time. The ratio is smaller for the other two LLMs.

Memory usage breakdown: Due to PIPEMESH's mixed sharding strategy, it requires less memory for model states compared to Megatron. This allows PIPEMESH to allocate more memory for activations when operating under the same total memory budget as Megatron. When further incorporating selective recomputation, PIPEMESH achieved slightly lower total memory usage compared to Megatron in practice. For instance, when training the 58B GPT-3 model on the V100 cluster, per device, PIPEMESH allocated 1.5 GB more memory for activations while using 1.8 GB less memory for model states compared to Megatron.

PIPEMESH's mixed sharding strategy set a portion of parameters to ZeRO-3 and the rest to ZeRO-2 for all the models evaluated. The percentages of parameters set to ZeRO-3 on the V100 cluster were 84.4%, 90.9%, and 86.9% for GPT-3 58B, Llama2 55B, and Falcon 66B, respectively. PIPEMESH fully overlapped the communication for gradient sharding and parameter sharding with computation.



(a) Loss vs. Training tokens (b) Loss vs. GPU hours

Fig. 11. Training of GPT-3 58B using Megatron continued until the training loss reached 2.8 on the A100 cluster (B: batch size, MT: Megatron, DU: one-step delayed update). When using larger batch sizes or one-step delayed update, more training tokens were required.

TABLE V
TRAINING TOKENS REQUIRED FOR GPT-3 58B TO REACH A LOSS OF 2.8

Batch Size	36	72	144	36-DU
#tokens (B)	2.47 (1x)	2.90 (1.17x)	3.56 (1.44x)	4.17 (1.69x)
Thrp. (A100)	100.4 (1x)	114.9 (1.14x)	123.9 (1.23x)	121.3 (1.21x)
Thrp. (V100)	32.8 (1x)	42.1 (1.28x)	49 (1.49x)	46.6 (1.42x)

We list Megatron's throughput (Thrp., in TFLOPS) when using these batch sizes. DU represents onestep delayed update. The throughput of batch size 36 with DU is lower than batch size 144 due to a higher pipeline bubble ratio.

PIPEMESH effectively reduced the activation memory footprint through recomputation. For example, the per-layer activation size of Falcon 66B was reduced from 79.5 MB in Megatron to 43.5 MB in PIPEMESH.

B. Large Batch Sizes and Asynchronous Update

Our experiments with GPT-3 58B using different batch sizes and the one-step delay revealed trade-offs between computational efficiency and convergence, as illustrated in Fig. 11 and Table V. While larger batch sizes and the one-step delay reduced the proportion of critical-path DP communication in an iteration and improved throughput, they also increased the number of tokens required to reach a target loss. This observation aligns with existing research [33], [40]. Compared to our original batch size settings for Megatron in Table IV, these alternative configurations actually resulted in longer overall training times. On the A100 cluster, increasing the batch size from 36 to 72 and 144 led to 2.6% and 16.8% longer training times, respectively, because they required 17.4% and 44.1% more tokens. Even with the one-step delay effectively hiding nearly all DP communication, training time increased by 39.7% compared to Megatron with synchronous updates.

We also observed that optimal batch sizes varied across different cluster configurations, even when training the same model. For instance, with a batch size of 36, critical-path DP communication constituted 29.5% of the iteration time on the V100 cluster but only 17.2% on the A100 cluster. When increasing the batch size to 72 to reduce the ratio of critical-path DP communication, the V100 cluster showed a 28.4% throughput improvement that outpaced the increased token requirement. In contrast, the A100 cluster only achieved a 14.4% throughput increase, indicating

TABLE VI
MODELS USED IN THE WEAK SCALING SETUP AND CORRESPONDING TRAINING CONFIGURATIONS

#GPUs	Model	B	l	n_{qkv}	h	TP	PP	DP	M
48	GPT-3 15B	48	24	56	7168	4	6	2	4
96	GPT-3 26B	96	32	64	8192	4	8	3	4
192	GPT-3 49B	192	48	72	9216	4	12	4	4

We report the number of layers (l), the number of query, key and value heads (n_{qkv}), and the hidden size (h) of each model. The training configuration includes the number of GPUs (#GPUs), the 3D parallelism configuration (TP, PP and DP), the batch size per iteration (B), and the number of model chunks (M).

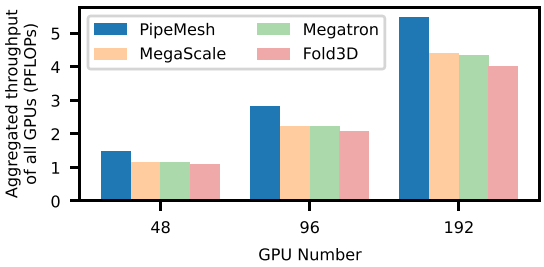


Fig. 12. Weak scaling evaluation on different numbers of GPUs. PIPEMESH consistently achieved higher throughput for all the scales evaluated.

that increasing the batch size to 72 would not reduce overall training time.

C. Scalability

We evaluated the scalability of PIPEMESH using a weak scaling setup. Following the baseline [16], the setup scales the number of GPUs and the model size simultaneously. We varied the number of GPUs from 48 to 192 and proportionally increased the model size by increasing the number of layers and the hidden size. Table VI lists the models used and corresponding training configurations of PIPEMESH. With the scaling of GPU number, both the pipeline parallel size and data parallel size increased accordingly, while the tensor parallel size was bounded by the number of GPUs in a host.

Fig. 12 shows that PIPEMESH's throughput was 25.0% to 27.2% higher than Megatron and MegaScale. The critical DP communication of the baselines always accounted for a large portion (from 32.0% to 38.6%) of the iteration time for all the scales we evaluated. The key reason is that the DP communication kept being much greater than the computation during the warm-up and cool-down phases of the baselines. In contrast, PIPEMESH was able to overlap most of the DP communication, since both the computation and DP communication grew when scaling the model size and the batch size. For all three scales, PIPEMESH enqueued and dequeued half of the micro-batches each time, and the forward and backward passes scheduled together contained enough computation to hide the DP communication.

D. Ablation Study

We evaluated the impact of the number of model chunks and the buffer queue size on PIPEMESH's performance. We used the Falcon 66B model and the 3D parallelism configuration reported in Section VI-A. We first kept the buffer queue size at 12 and varied the number of model chunks. Fig. 13 shows the throughput of

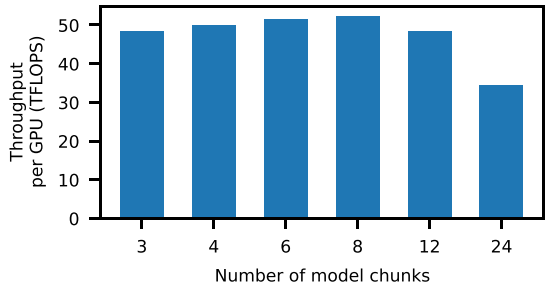


Fig. 13. The throughput when training Falcon 66B with a buffer queue size of 12 and various model chunk numbers. The model chunk number selected by PIPEMESH's algorithm achieved the highest throughput.

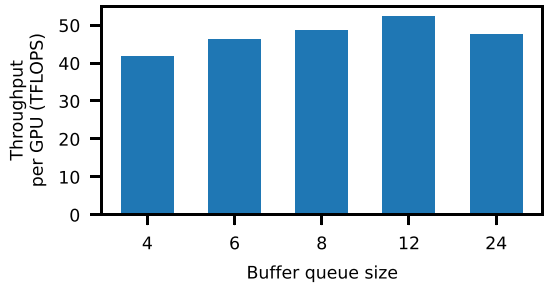


Fig. 14. The throughput for Falcon 66B when setting the number of model chunks to 8 and varying the buffer queue size. The buffer queue size selected by PIPEMESH's algorithm achieved the highest throughput.

PIPEMESH when using 3, 4, 6, 8, 12, and 24 model chunks. When the number of model chunks was 8, PIPEMESH balanced the critical PP communication and the sum of the pipeline bubble and the critical DP communication. For the model chunk numbers smaller than 8, less DP communication was overlapped and more pipeline bubble existed. For the model chunk numbers larger than 8, the PP communication exceeded the computation and offset the benefits of decreased pipeline bubble and critical DP communication.

Fig. 14 demonstrates how PIPEMESH performed when setting the buffer queue size to 4, 6, 8, 12, and 24. For each size, we selected the recomputation strategy and mixed sharding strategy based on PIPEMESH's algorithm. The buffer queue size of 12, which was automatically selected by PIPEMESH, made PIPEMESH achieve the highest throughput. With that size, PIPEMESH overlapped optimizer sharding communication except that of the first model chunk. Less optimizer sharding communication was overlapped for buffer queue sizes less than 12, leading to increased critical DP communication time and iteration time. For the buffer queue size greater than 12, PIPEMESH had to additionally recompute all query transformation operations and up-scaling linear operations in each model chunk, increasing the computation time by 12.0%.

E. Effectiveness of Components

We evaluated the effectiveness of the elastic pipeline schedule and mixed sharding strategy of PIPEMESH. We trained Falcon 66B, setting all parameters to ZeRO-1 (PIPEMESH-Z1), ZeRO-2 (PIPEMESH-Z2), and ZeRO-3 (PIPEMESH-Z3), respectively. To

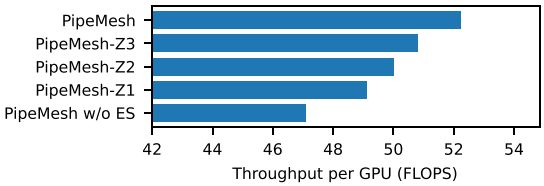


Fig. 15. The throughput for Falcon 66B when using PIPEMESH with different ZeRO stages (PIPEMESH-Z1, PIPEMESH-Z2, and PIPEMESH-Z3) and PIPEMESH without the elastic pipeline schedule (PIPEMESH w/o ES). The elastic pipeline schedule and mixed sharding strategy are crucial for achieving high throughput.

ensure a fair comparison, we used the same buffer queue size for all systems. Additionally, we evaluated a variant of our system without the elastic pipeline schedule (PIPEMESH w/o ES), which employed the depth-first pipeline schedule and enabled ZeRO-3. For these systems, we set the number of model chunks to 8 and allocated a memory budget equal to the memory consumption of Megatron. The recomputation strategy for each system was selected using a simplified version of PIPEMESH's algorithm. Fig. 15 presents the throughput achieved by each system. Compared to PIPEMESH, the throughput of PIPEMESH w/o ES, PIPEMESH-Z1, PIPEMESH-Z2, and PIPEMESH-Z3 was 9.7%, 6.0%, 4.2%, and 2.7% lower, respectively. The throughput degradation observed in PIPEMESH-Z3 can be attributed to increased DP communication resulting from parameter sharding. In contrast, the reduced throughput in the other systems was primarily caused by increased recomputation time. PIPEMESH-Z2 had to additionally recompute two query transformation operations among the three transformer layers of each model chunk; PIPEMESH-Z1 had to recompute one query transformation operation and one up-scaling linear operation in each model chunk; PIPEMESH w/o ES had to recompute all three query transformation operations and two up-scaling linear operations in each model chunk.

F. Lessons Learned

PIPEMESH has two limitations. First, the recomputation used by PIPEMESH increases the computation time compared with Megatron. PIPEMESH leverages selective recomputation to overlap communication with more micro-batches within the memory capacity. Our evaluation shows that the gain of the reduced performance-critical communication achieved by PIPEMESH significantly outperforms the cost of the increased computation, and PIPEMESH can effectively improve the overall training throughput. Second, same as Megatron, PIPEMESH is designed mainly for LLM pre-training and is also applicable to fine-tuning methods that update all the model parameters (i.e., instruction fine-tuning [48], [49]). Recent works [50], [51], [52], [53] propose to fine-tune open-source LLMs by freezing most of the model parameters and updating only limited parameters. These workloads have far less computation requirement and memory consumption and can be efficiently supported by ZeRO or FSDP [54].

VII. RELATED WORK

Pipeline parallel training: Pipeline parallelism [12], [13], [28], [30], [55], [56], [57], [58] is commonly used for training large DNN models. BPipe [59] transfers activations between GPUs to balance memory usage across pipeline stages. Hippie [60] is designed to automatically partition models and reduce the memory overhead for large DNN models. AdaPipe [58] adaptively configures the recomputation and pipeline stage partitioning strategies to reduce the training cost. These systems are orthogonal to PIPEMESH, making a trade-off between memory usage and communication overlap for pipeline parallel training combined with data parallelism.

Parallelization algorithms: Alpa [61] and Unity [62] automatically partition a model across multiple devices by solving a cost minimization problem. However, these approaches focus on finding the optimal parallelization strategy for existing pipeline schedules (e.g., the 1F1B schedule [13]), while PIPEMESH proposes a new elastic pipeline schedule. We believe that PIPEMESH is orthogonal to Alpa and Unity.

Activation recomputation: Activation recomputation reduces memory usage during DNN training by selectively recomputing activations instead of storing them all. For an n-layer DNN, a heuristic [23] that checkpoints layers in equal intervals achieves $O(n)$ memory cost. Checkmate [24] is a system that searches for optimal recomputation strategies in reasonable times (under an hour) using ILP solvers. Online algorithms [25], [63] can be applied to DNNs with dynamic graphs. Current recomputation approaches [10], [45], [64] for LLMs discard the activations generated by self-attention operations during forward passes. Different from the approaches that minimize the recomputation time within the memory budget, PIPEMESH leverages recomputation to enable larger windows for overlapping DP communication.

Overlapping communication in distributed training: Overlapping communication with computation is an effective approach to improve the throughput of distributed training. Systems like P3 [65], TicTac [66], and ByteScheduler [67] are designed for training without pipeline parallelism and overlap DP communication based on priority scheduling. Eager-1F1B schedule [68] shifts the forward computational tasks of 1F1B schedule and launches more warm-up micro-batches to overlap the PP communication. Centauri [69] attempts to overlap DP communication by increasing warm-up micro-batches, but its effectiveness is limited by the available memory for activations of the increased warm-up micro-batches. In contrast, PIPEMESH achieves memory usage comparable to Megatron while enabling the overlapping of DP communication in memory-constrained scenarios.

VIII. CONCLUSION

We present PIPEMESH, a 3D parallel training system that overlaps communication with computation in a memory-efficient way. We design the elastic pipeline schedule that allows PIPEMESH to enable ZeRO-2 and ZeRO-3 to save the memory consumed by model states. PIPEMESH also incorporates selective recomputation to reduce the memory consumed by activations.

PIPEMESH can significantly accelerate LLM training under constraints of memory capacity and network bandwidth.

REFERENCES

[1] T. B. Brown et al., "Language models are few-shot learners," 2020, *arXiv: 2005.14165*.

[2] H. Touvron et al., "Llama 2: Open foundation and fine-tuned chat models," 2023, *arXiv:2307.09288*.

[3] B. Workshop et al., "BLOOM: A 176B-parameter open-access multilingual language model," 2022, *arXiv:2211.05100*.

[4] A. Q. Jiang et al., "Mistral 7B," 2023, *arXiv:2310.06825*.

[5] E. Almazrouei et al., "The falcon series of open language models," 2023, *arXiv:2311.16867*.

[6] M. N. Team, "Introducing MPT-30B: Raising the bar for open-source foundation models," 2023. Accessed: Jun. 22, 2023. [Online]. Available: www.mosaicml.com/blog/mpt-30b

[7] A. Vaswani et al., "Attention is all you need," 2017, *arXiv: 1706.03762*.

[8] Meta Llama 3, 2024. [Online]. Available: https://llama.meta.com/llama3/

[9] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training multi-billion parameter language models using model parallelism," 2019, *arXiv: 1909.08053*. [Online]. Available: http://arxiv.org/abs/1909.08053

[10] V. Korthikanti et al., "Reducing activation recomputation in large transformer models," 2022, *arXiv:2205.05198*.

[11] H. Liu, M. Zaharia, and P. Abbeel, "Ring attention with blockwise transformers for near-infinite context," 2023, *arXiv:2310.01889*.

[12] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," 2018, *arXiv: 1811.06965*.

[13] D. Narayanan et al., "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 1–15.

[14] NVIDIA DGX GH200, 2023. [Online]. Available: https://www.nvidia.com/en-gb/data-center/dgx-gh200/

[15] NVIDIA DGX BG200, 2024. [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-b200/

[16] D. Narayanan et al., "Efficient large-scale language model training on GPU clusters using megatron-LM," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, Art. no. 58. [Online]. Available: https://doi.org/10.1145/3458817.3476209

[17] Q. Weng et al., "MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters," in *Proc. 19th USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 945–960.

[18] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory optimizations toward training trillion parameter models," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–16.

[19] Z. Jiang et al., "MegaScale: Scaling large language model training to more than 10,000 GPUs," 2024, *arXiv:2402.15627*.

[20] P. Qi, X. Wan, G. Huang, and M. Lin, "Zero bubble pipeline parallelism," 2023, *arXiv:2401.10241*.

[21] F. Li et al., "Fold3D: Rethinking and parallelizing computational and communicational tasks in the training of large DNN models," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 5, pp. 1432–1449, May 2023.

[22] J. Lamy-Poirier, "Breadth-first pipeline parallelism," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2023, pp. 48–67.

[23] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," 2016, *arXiv:1604.06174*.

[24] P. Jain et al., "Checkmate: Breaking the memory wall with optimal tensor rematerialization," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2020, pp. 497–511.

[25] M. Kirisame et al., "Dynamic tensor rematerialization," 2020, *arXiv: 2006.09616*.

[26] A. Sergeev and M. Del Balso, "Horovod: Fast and easy distributed deep learning in tensorflow," 2018, *arXiv: 1802.05799*.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.

[28] S. Zhao et al., "vPipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel DNN training," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 3, pp. 489–506, Mar. 2022.

[29] B. Yang, J. Zhang, J. Li, C. Ré, C. Aberger, and C. De Sa, "PipeMare: Asynchronous pipeline parallel DNN training," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2021, pp. 269–296.

[30] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel DNN training," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 7937–7947.

[31] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[32] J. Choquette, "NVIDIA hopper GPU: Scaling performance," in *Proc. 2022 IEEE Hot Chips 34 Symp.*, 2022, pp. 1–46.

[33] J. Kaplan et al., "Scaling laws for neural language models," 2020, *arXiv: 2001.08361*.

[34] P. Micikevicius et al., "Mixed precision training," 2017, *arXiv: 1710.03740*.

[35] Automatic mixed precision package - torch.amp — pytorch.org,, 2020. Accessed: Feb. 01, 2025. [Online]. Available: https://pytorch.org/docs/stable/amp.html

[36] D. Hendrycks and K. Gimpel, "Gaussian error linear units (GELUs)," 2016, *arXiv:1606.08415*.

[37] N. Shazeer, "GLU variants improve transformer," 2020, *arXiv: 2002.05202*.

[38] S. Elfwing, E. Uchibe, and K. Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," *Neural Netw.*, vol. 107, pp. 3–11, 2018.

[39] J. Ren et al., "ZeRO-Offload: Democratizing billion-scale model training," in *Proc. 2021 USENIX Annu. Tech. Conf.*, 2021, pp. 551–564. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/ren-jie

[40] S. McCandlish, J. Kaplan, D. Amodei, and O. D. Team, "An empirical model of large-batch training," 2018, *arXiv: 1812.06162*.

[41] S. Stich, A. Mohtashami, and M. Jaggi, "Critical parameters for scalable distributed learning with large batches and asynchronous updates," in *Proc. Int. Conf. Artif. Intell. Statist.*, 2021, pp. 4042–4050.

[42] J. M. Tarnawski, D. Narayanan, and A. Phanishayee, "Piper: Multidimensional planner for DNN parallelization," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2021, Art. no. 1902.

[43] B. Zhang and R. Sennrich, "Root mean square layer normalization," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 12360–12371.

[44] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "GQA: Training generalized multi-query transformer models from multi-head checkpoints," 2023, *arXiv:2305.13245*.

[45] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2022, pp. 16344–16359.

[46] jcpeterson/openwebtext., 2019. [Online]. Available: https://github.com/jcpeterson/openwebtext

[47] S. Rashidi et al., "Enabling compute-communication overlap in distributed deep learning training platforms," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit.*, 2021, pp. 540–553.

[48] J. Wei et al., "Finetuned language models are zero-shot learners," 2021, *arXiv:2109.01652*.

[49] V. Sanh et al., "Multitask prompted training enables zero-shot task generalization," 2021, *arXiv:2110.08207*.

[50] J. He, C. Zhou, X. Ma, T. Berg-Kirkpatrick, and G. Neubig, "Towards a unified view of parameter-efficient transfer learning," 2021, *arXiv:2110.04366*.

[51] N. Houlsby et al., "Parameter-efficient transfer learning for NLP," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 2790–2799.

[52] E. J. Hu et al., "LoRA: Low-rank adaptation of large language models," 2021, *arXiv:2106.09685*.

[53] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLORA: Efficient finetuning of quantized LLMs," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2024, Art. no. 441.

[54] Y. Zhao et al., "PyTorch FSDP: Experiences on scaling fully sharded data parallel," 2023, *arXiv:2304.11277*.

[55] H. Oh, J. Lee, H. Kim, and J. Seo, "Out-of-order backprop: An effective scheduling technique for deep learning," in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 435–452.

[56] S. Fan et al., "DAPPLE: A pipelined data parallel approach for training large models," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 431–445.

[57] S. Li and T. Hoefler, "Chimera: Efficiently training large-scale neural networks with bidirectional pipelines," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, pp. 1–14.

[58] Z. Sun et al., "AdaPipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning," in *Proc. 29th ACM Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2024, pp. 86–100.

[59] T. Kim, H. Kim, G.-I. Yu, and B.-G. Chun, "BPIPE: Memory-balanced pipeline parallelism for training large language models," in *Proc. Int. Conf. Mach. Learn.*, 2023, pp. 16639–16653.

[60] D. Li et al., "A memory-efficient hybrid parallel framework for deep neural network training," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 4, pp. 577–591, Apr. 2024.
[61] L. Zheng et al., "Alpa: Automating inter-and intra-operator parallelism for distributed deep learning," 2022, *arXiv:2201.12023*.
[62] C. Unger et al., "Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 267–284.
[63] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, "Memory-efficient backpropagation through time," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 4125–4133.
[64] T. Dao, "FlashAttention-2: Faster attention with better parallelism and work partitioning," 2023, *arXiv:2307.08691*.
[65] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed DNN training," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2019, pp. 132–145.
[66] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "TicTac: Accelerating distributed deep learning with communication scheduling," in *Proc. 2nd Conf. Mach. Learn. Syst.*, 2019, pp. 418–430.
[67] Y. Peng et al., "A generic communication scheduler for distributed DNN training acceleration," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 16–29.
[68] Y. Zhuang et al., "On optimizing the communication of model parallelism," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2023, pp. 526–540.
[69] C. Chen et al., "Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning," in *Proc. 29th ACM Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2024, pp. 178–191.

Fanxin Li received the BE degree from Xi'an Jiaotong University, in 2019. He is currently working toward the PhD degree with the University of Hong Kong. His research interests include distributed machine learning and cloud computing.

Shixiong Zhao received the bachelor's degree from the University of Hong Kong (HKU), the master's degree from HKUST, and the PhD degree in computer science from HKU. He was under the supervision of Prof. Heming Cui. His research interests include large-scale distributed systems for machine learning, distributed systems, and system security.

Yuhao Qing received the bachelor's degree from the City University of Hong Kong. He is currently working toward the PhD degree in computer science with the University of Hong Kong (HKU), under the supervision of Prof. Heming Cui. His research interests includes machine learning systems and cloud computing.

Jianyu Jiang received the bachelor's degree from Xi'an Jiaotong University, in 2016, and the PhD degree from the University of Hong Kong, in 2023. He is currently a researcher, with a broad research interest in large language model (LLM) training and inference systems, heterogeneous computing, and system security. He published more than 10 research paper, and served as the artifact evaluation committee co-chairs of OSDI/ATC in 2023 and 2024. He won the distinguished paper award in ACSAC 2017.

Xusheng Chen received the bachelor's degree from the University of Hong Kong (HKU), and the PhD degree from the University of Hong Kong, supervised by Prof. Heming Cui. He is a research scientist with Huawei Cloud. His research focuses on large-scale distributed systems, including distributed storage systems, distributed databases, and serverless systems.

Heming Cui (Member, IEEE) is an associate professor in computer science of HKU. His research interests include operating systems, programming languages, distributed systems, and cloud computing, with a particular focus on building software infrastructures and tools to improve reliability and security of real-world software.