ARTIFACT EVALUATED USENIX AVAILABLE
ARTIFACT EVALUATED USENIX FUNCTIONAL
ARTIFACT EVALUATED USENIX REPRODUCED

Attack of the Bubbles: Straggler-Resilient Pipeline Parallelism for Large Model Training

Tianyuan Wu[†,*], *Lunxi Cao*[†,*], *Hanfeng Lu*[†], *Xiaoxiao Jiang*[†], *Yinghao Yu*[§], *Siran Yang*[§], *Guodong Yang*[§], *Jiamang Wang*[§], *Lin Qu*[§], *Liping Zhang*[§], *Wei Wang*[†]
[†]*Hong Kong University of Science and Technology*
[§]*Alibaba Group*

Abstract

Training large Deep Neural Network (DNN) models at scale often encounters straggler issues, mostly in communications due to network congestion, RNIC/switch defects, or topological asymmetry. Under advanced pipeline parallelism, even minor communication delays can induce significant training slowdowns. This occurs because (1) slow communication disrupts the pipeline schedule, creating cascading "bubbles" in a domino effect, and (2) current GPU kernel scheduling is susceptible to head-of-line blocking, where slow communication blocks subsequent computations, further adding to these bubbles. To address these challenges, we present PIPEMORPH, a straggler-resilient training system with two key optimizations. First, it optimally adapts the pipeline schedule in the presence of stragglers to absorb communication delays without inducing cascading bubbles, using a simple yet effective algorithm guided by an analytical model. Second, upon detecting slow communication, PIPEMORPH offloads communication operations from GPU to host memory and utilizes CPU-side RDMA for data transfer. This eliminates head-of-line blocking as subsequent computation kernels can be scheduled immediately on GPUs. Together, these optimizations effectively reduce pipeline stalls in the presence of communication stragglers, improving the training iteration time by 1.2-3.5× in our experiments under various settings.

1 Introduction

The rise of large Deep Neural Network (DNN) models has ushered in the golden age of Artificial Intelligence, leading to breakthroughs in applications that would have been considered science fiction even a few years ago [1, 15, 33, 43, 47, 51]. Training large DNN models typically requires combining tensor, data, and pipeline parallelism strategies across thousands of GPUs [22, 31, 41, 53], where providing high-throughput, low-latency communication is critical to enhancing the training performance [6, 11, 27].

However, at this scale, communication stragglers, manifested as slow links with extended pairwise transmission delays, are frequent [12, 35] and can have a significant performance impact [9, 36, 49]. Particularly in multi-tenant environments, jobs occupying a large number of GPUs frequently suffer from communication-induced slowdowns during life cycles [6, 49]. These stragglers originate predominantly from transient network congestion [6, 49] but can also persist due to hardware defects (e.g., RNIC or switch failures) or network topological asymmetry [50]. The presence of communication stragglers, even on a single link, slows down the entire training job due to frequent synchronizations necessitated by hybrid-parallelism strategies, causing up to 90% throughput degradation in production clusters [9, 36, 49].

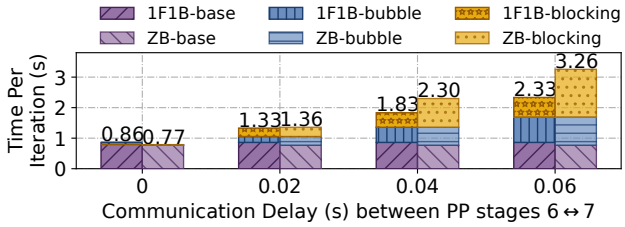
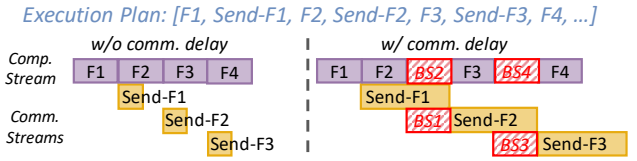


Figure 1: GPT-2 14B training performance on 8 nodes (one H800 GPU per node) with 8-stage PP, where minor communication delays between PP stages trigger dependency bubbles and blocking stalls, causing significant slowdowns in 1F1B and ZeroBubble (ZB) pipelines.

Production systems mainly focus on detecting communication stragglers in large-scale training [6, 9, 12, 49] and rely on traffic load balancing at flow or packet level to alleviate network congestion [4, 10, 14, 24]. However, these approaches are *agnostic* to the parallelism strategies of the training job and cannot effectively mitigate the straggler impacts on job performance. As illustrated in Figure 1, with pipeline parallelism (PP), even minor communication delays between two PP stages can result in significant training slowdowns, which grow rapidly as the delay increases. Our study identifies two issues that contribute to this large slowdown, with their impacts shown in Figure 1.

First, given sophisticated data dependencies in a PP schedule (e.g., Gpipe [18], 1F1B [30] and ZeroBubble [34]), a single communication straggler, when exceeding a certain threshold, can set off a domino effect, triggering *cascading bubbles* (i.e., GPU idle periods) that propagate across subsequent stages (Figure 5-bottom). These bubbles, which we call

*Equal contribution.



Blocking Stalls	Blocked Kernel	Root Cause
BS1&3	Send-F2&F3	Queuing: the previous comm. kernel not finished.
BS2&4	F3&F4	It is blocked by its predecessor comm. kernel.

Figure 2: Head-of-line blocking due to sequential kernel scheduling: slow comm. blocks subsequent comp.

dependency bubbles, cause severe misalignment within the pipeline, disrupting its entire schedule.

Second, existing GPU kernel scheduling is susceptible to *head-of-line blocking*, where slow communication can block subsequent computation. To illustrate this, we refer to Figure 2. Once a PP schedule is determined, a low-level kernel execution plan is generated accordingly, in which communication and computation operations are interleaved to overlap communication latency (e.g., `F1`, `Send-F1`, `F2`, `Send-F2`, ...). The GPU scheduler *sequentially schedules* operations following this plan. However, in the presence of a slow link, a communication operation cannot be scheduled immediately as previous communication operations are still queued up, pending for transmission. This blocks the scheduling of subsequent computation operations, introducing additional *blocking stalls* to the pipeline (Figure 2-right), which in turn triggers more bubbles, further aggravating training slowdowns.

Eliminating dependency bubbles and blocking stalls requires dynamically adapting the pipeline schedule with framework support, which remains lacking in existing systems. For instance, Falcon [49] simply reassigns slow links from communication-heavy DP groups to communication-light PP groups, without addressing subsequent pipeline bubbles. Recycle [13] and Oobleck [20] pre-compute a pipeline reconfiguration plan for handling GPU failures. However, communication stragglers cannot be addressed using a pre-computed plan as pipeline adaptation must be made dynamically based on the changing straggler magnitude.

In this paper, we present PIPEMORPH, a straggler-resilient system for efficient hybrid-parallel training. It addresses dependency bubbles and blocking stalls caused by slow communications with two key designs.

Straggler-resilient pipeline adaptation. We show analytically that a pipeline schedule (e.g., 1F1B [30] or ZeroBubble [34]) can tolerate slow communication up to a certain threshold without triggering cascading bubbles in a domino effect. This threshold is in proportion to the *slackness* between adjacent PP stages, defined as the *difference of the number of forward operations* scheduled in the two stages during the warm-up phase. Larger slackness enhances the pipeline's resilience to a longer communication delay. Based on this, PIPEMORPH initially generates a ZeroBubble schedule [34] that maximizes the minimum inter-stage slackness under memory and configuration constraints. It then monitors communication delays between PP stages. When the delay exceeds the tolerance threshold (given by our analytical model), it quickly reacts by adapting the pipeline schedule to increase the inter-stage slackness of the slow link, eliminating all or most straggler-induced dependency bubbles.

Decoupled data plane. PIPEMORPH further employs a *fully-decoupled data plane* to address head-of-line blocking of GPU kernel scheduling and the resulting blocking stalls. Upon detecting communication stragglers, the system transparently switches to a *delegation mode*, in which it offloads PP communications from GPU to host memory and uses dedicated delegate processes to perform data transfer via CPU-side RDMA. PIPEMORPH chooses to bypass the more efficient GPU-direct RDMA due to three design imperatives. First, it completely decouples PP communications from GPU execution, preventing slow communication from blocking subsequent GPU computations. Second, optimally adapting pipeline schedule in the presence of stragglers may require storing more activations than GPU memory can hold, where offloading to host memory becomes necessary. Third, given that PP schedule only requires light to moderate communications (compared to DP and TP), the performance overhead introduced by offloading can be minimized with system-level optimizations. This design additionally enables RNIC fault tolerance: upon detecting a RNIC failure, the system reroutes traffics through remaining healthy RNICs via the delegation path, obviating the need for checkpoint-and-restart failovers.

We implemented PIPEMORPH on top of the Megatron-LM [31,41] training framework, using ZeroBubble [34] as the base pipeline schedule to achieve the best performance. We evaluated PIPEMORPH using GPT-2 models of varying sizes, from 7B to 140B parameters, on H800 clusters. Compared to state-of-the-art straggler mitigation solutions, PIPEMORPH reduces the average training iteration time by 1.2-3.5× under various network latency conditions. In a large-scale deployment involving 128 H800 GPUs, PIPEMORPH consistently delivers high training throughput in real-world network environments with frequent communication stragglers, outperforming the ZeroBubble baseline by 1.36×.

2 Background and Motivation

2.1 Hybrid-Parallel DNN Training

The increasing scale of Deep Neural Networks (DNNs), driven by empirical scaling laws [23], has led to state-of-the-art models with hundreds of billions of parameters [1, 12, 15, 27, 39, 43, 51]. Training such large models requires high-performance computing (HPC) clusters comprising tens of thousands of GPUs [22, 31, 35], leveraging advanced parallelization strategies in three primary forms.

Data parallelism (DP) distributes identical model replicas

across GPU groups, with each replica processing a subset of the input data (mini-batches) concurrently [31, 37, 41]. Synchronization is performed at the end of each iteration via all-reduce operations, often spanning multiple GPU nodes interconnected through high-speed networks such as RDMA over Infiniband (IB) [14, 22, 35].

Tensor parallelism (TP) partitions individual tensors (e.g., weight matrices) within model layers across multiple GPUs [41,53]. While this technique parallelizes linear algebra operations within layers, it incurs significant communication overhead due to frequent reduce-scatter and all-reduce operations during forward and backward passes. Consequently, TP is typically restricted to single-node deployments with high-bandwidth GPU interconnects (e.g., NVLink).

Pipeline parallelism (PP) divides the model into sequential layer groups (*stages*) assigned to different GPUs [18, 31, 53]. Mini-batches are further split into micro-batches that flow through these stages in a pipelined manner, enabling parallel processing across stages. PP requires lower network bandwidth than DP/TP by communicating only activations and gradients at layer boundaries. However, it can suffer from reduced training throughput due to pipeline dependencies and bubbles (idle slots) as the number of stages increases. Modern PP implementations like 1F1B [30] and ZeroBubble (ZB) [34] address these issues, with ZB achieving bubble-free execution at the expense of increased memory footprint.

2.2 Reliability Issues

Training large DNN models over extended periods face reliability challenges, primarily manifested through *crash failures* and still-functioning but slow *stragglers* [9, 12, 20, 22, 48, 49, 51]. These issues stem from hardware failures, software errors, or resource contention, with even a single affected component disrupting the entire distributed training.

Crash failures. Crash failures have been extensively analyzed in recent studies [12, 13, 17, 20, 21, 29, 45, 48, 50]. Meta and ByteDance report that faulty GPUs and RNICs are the leading causes, accounting for 40% and 10% of failures, respectively [12, 17]. To address this, researchers have developed *fault-resilient solutions* for hybrid-parallel training, including (1) optimized checkpoint-and-restart mechanisms to minimize recovery overhead [29, 48] and (2) exploiting PP's functional redundancy to reduce checkpointing [13, 20, 45].

Stragglers. Stragglers—manifested as slow computation or communication—represent another major reliability concern in large-scale DNN training [9, 12, 49]. Prior studies [22, 49] indicate that *computation stragglers*, typically caused by GPU thermal throttling or hardware defects, are relatively rare ($\sim$0.5% occurrence) and short-living (usually recovering in 10 minutes). These incidents can be efficiently addressed by adjusting the parallelization of GPU devices [25, 49]. In contrast, *communication stragglers*, predominantly due to network congestion, are more frequent and persistent, often lasting for hours [6, 49]. In Alibaba's production multi-tenant clusters, over 36% of jobs using >50% GPUs experience slowdowns from communication issues [6]. In extreme cases, such stragglers can reduce training throughput by up to 90% [49].

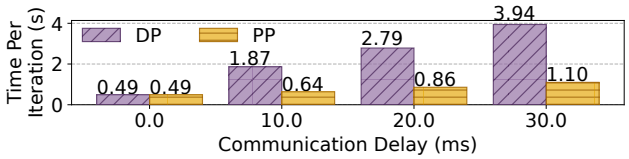


Figure 3: Iteration time growth under different per packet delays, where DP is more communication-sensitive to PP.

Communication stragglers may occur on cross-node links between DP groups (DP straggler) or between PP stages (PP straggler), but not in TP groups, as TP communications are confined to intra-node, high-bandwidth, and stable NVLinks [6, 49]. Compared to PP stragglers, DP stragglers can have a more pronounced impact on performance, as DP's all-reduce operations incur substantial communication costs and are bottlenecked by the slowest link in the all-reduce ring [7, 16, 25, 49, 55].

To illustrate this effect, we train a GPT2-7B model on 8 nodes (each with one H800 GPU) by configuring a (2 DP, 4 PP) ZeroBubble pipeline using Megatron-LM [31]. We manually inject a communication delay of 10/20/30 ms into a designated cross-node link (400 Gbps IB), which may affect DP *or* PP communication. As shown in Figure 3, when this link is part of the DP communication group, iteration time increases by up to 8.04$\times$. In contrast, when the same link is assigned for PP communication, the slowdown is limited to $\leq 2.24\times$. Motivated by these findings, recent work [49] proposes mitigating DP stragglers by reconfiguring the parallelization scheme to assign slow links to PP stages instead of DP groups. While this approach effectively converts a DP straggler to a PP straggler, it still results in significant slowdowns, which remain open to address.

3 Impact Analysis and Challenges

In this section, we systematically investigate how inter-stage communication stragglers lead to substantial pipeline stalls. Our investigation focuses on ZeroBubble (ZB) pipeline scheduling [34]—a generalized, fine-grained PP scheduling scheme that subsumes common approaches like 1F1B [30] as special cases. As illustrated in Figure 4, ZB eliminates pipeline bubbles by decomposing backward pass into two independent operators, backward input (B) and backward weight (W), then precisely orchestrating their execution. This generalized design encapsulates diverse pipeline behaviors, ensuring the broad applicability of our findings. In the following, we identify two critical delay propagation mechanisms: (1) domino effect of cascading bubbles due to PP's vulnerable dependency chains (§3.1) and (2) head-of-line blocking stalls

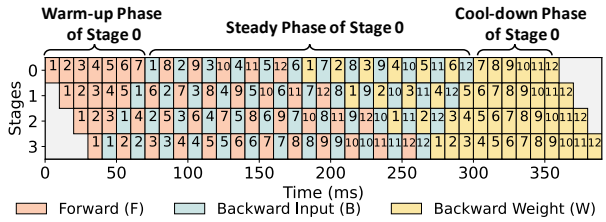


Figure 4: An ideal straggler-free ZeroBubble [34] pipeline with 4 stages and 12 microbatches, completing in 390 ms.

due to sequential GPU kernel scheduling (§3.2).

3.1 Domino Effect of Cascading Bubbles

Pipeline parallelism orchestrates stage execution with strict data dependencies. In ZB scheduling, these dependencies manifest through two key constraints: (1) a forward operator (F_j) in stage S_i requires completion of F_j in preceding stage S_{i-1}; (2) backward operators B_j and W_j in S_i must be scheduled after B_j completion in subsequent stage S_{i+1}. The presence of communication stragglers between stages S_i and S_{i+1} introduces additional latency c_i, with two immediate impacts: (1) forward operator F_j in stage S_{i+1} can only commence after the completion of F_j in S_i plus the communication delay c_i; (2) backward operators B_j and W_j in S_i must follow the completion of B_j in S_{i+1} plus the delay c_i.

We quantify the straggler impacts on pipeline schedules through simulations of a 4-stage pipeline processing 12 microbatches, with uniform operator execution time ($F = B = W =$ 10 ms). Figure 4 shows the ideal straggler-free ZB schedule completing in 390 ms with zero bubbles. Figure 5 depicts the resulting schedules in the presence of communication delays of 10 and 20 ms between stages 0 and 1. A 10 ms delay introduces a slight slowdown, extending the execution time from 390 to 400 ms. However, further increasing the delay to 20 ms results in a *non-linear growth* of pipeline stall to 440 ms. This occurs because the 20 ms delay pushes the first backward B_1 and subsequent F_8 in S_0 back to $t = 110$ ms, creating a bubble in S_1. This bubble propagates downstream, triggering cascading pipeline stalls in subsequent stages.

Extending our analysis, Figure 8 examines various ZB pipelines with different slackness parameters Δ_i (defined in §4.1). For each pipeline schedule, we gradually increase the communication delays between adjacent stages and depict in Figure 8 the resulting pipeline delays (left) and bubble rates (right). These results empirically demonstrate that localized communication delays exceeding a certain threshold create cascading dependency bubbles in a domino effect, leading to significant global pipeline stalls.

3.2 Head-of-Line Blocking Stalls

Communication stragglers further degrade pipeline performance through low-level GPU kernel scheduling anomalies,

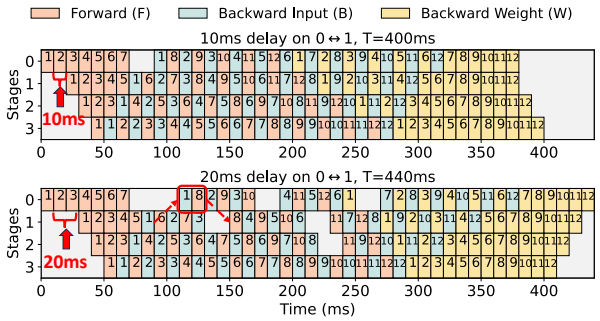


Figure 5: ZeroBubble schedule under $c_0 = 10/20$ ms delay between stage 0 and 1. Increasing c_0 from 0 to 10 ms only prolongs iteration time T by 10 ms, while an additional 10 ms delay introduces a 40 ms growth in T.

manifested as *head-of-line blocking stalls*. To demonstrate this problem, we conduct a GPT2-7B training experiment across four nodes (one H800 GPU per node), under a ZB schedule of(1 TP, 1 DP, 4 PP). We inject a 30 ms delay between PP stages 0 and 1 using NCCL network plugin. Figure 6 illustrates the profiled kernel execution result using NVIDIA Nsight Systems [32].

Blocking stalls. Given a pipeline schedule, a training framework (e.g., Megatron-LM [31], TorchTitan [26]) generates a *fixed execution plan* that interleaves computation (F, B, W) and communication operations, including send/recv-forward (SF/RF) and send/recv-backward (SB/RB). This execution plan maximizes computation-communication overlap through a carefully ordered operation scheduling sequence (e.g., $[F_1, SF_1, F_2, SF_2, \ldots]$ in Figure 6). The kernel scheduler sequentially launches these operations following this predetermined order. However, delayed communication operations stall subsequent computation operations, creating unexpected bubbles. In Figure 6, the delayed SF_2 blocks the launching of F_4 in stage 0, despite it being ready to execute after F_3.

Root cause analysis. Blocking stalls occur when the NCCL's transmission queue becomes saturated, which may disrupt CUDA's asynchronous execution. This is because each NCCL `send`/`recv` launches a GPU kernel to enqueue data transfers, which are handled asynchronously by a dedicated backend thread. Under normal circumstances without stragglers, the network transmission rate closely matches data generation. As a result, NCCL's internal queue does not fill up, allowing perfect overlap of computation and communication through the use of separate CUDA streams. This outcome aligns with the original design principle of frameworks such as Megatron-LM. However, when network links become slow, the queue builds up as pending transfers outpace actual data transmission. The launching of subsequent communication kernels (e.g., SF_3) hence blocks—they do not return control to the kernel scheduler until the queue space becomes available. This in turn prevents the CUDA scheduler from launching subsequent computation kernels (e.g., F_4 is blocked until SF_3

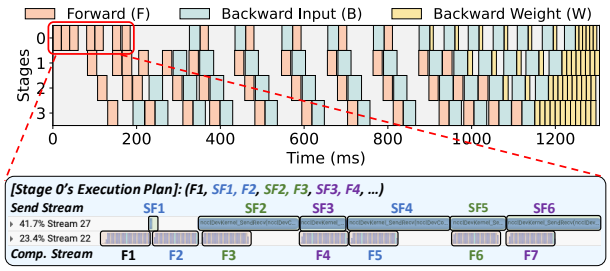


Figure 6: Slow communication (SF_2) induces HOL blocking stalls (F_4) due to sequential GPU kernel scheduling.

returns control, which itself waits on SF_2), thus creating head-of-line (HOL) blocking stalls in computation. As illustrated in Figure 6, such HOL blocking increases the iteration time from 710 ms to 1350 ms—a 1.9× slowdown. Notably, CUDA multi-streaming does not resolve this issue because all streams within a GPU context share the same NCCL communicator and transmission queue. As a result, once the queue becomes full, all streams are stalled and blocking is unavoidable.

3.3 Which Layer to Optimize?

Our analysis reveals that communication stragglers degrade pipeline performance through two mechanisms: dependency bubbles and HOL blocking stalls (see Figure 1 for quantitative contributions). Addressing these issues requires careful considerations of optimization layers within the system stack.

Network-level optimizations, such as ECMP [4, 14, 44] or packet spraying [10, 24], balance traffic at the flow or packet level. While effective for general congestion reduction, these approaches are *agnostic* to training semantics (e.g., parallelism strategies and communication patterns) and cannot prioritize critical communications over less sensitive transfers, nor can they address HOL blocking stalls. Also, even state-of-the-art load balancing techniques cannot eliminate network congestion, especially in multi-tenant clusters [2, 6].

Effective straggler mitigation requires **framework-level optimizations**, leveraging training semantics such as pipeline schedule, operator dependencies, and communication patterns. This semantic insight enables targeted mitigation strategies, such as pipeline adaptation and blocking-free kernel scheduling—none of these can be implemented in the network layer. However, existing reliability enhancement mechanisms for training frameworks are ineffective in addressing communication stragglers under pipeline parallelism. For instance, Malleus [25] exclusively targets computation stragglers without considering slow communications. XPUTimer [9] provides production-grade straggler detection yet lacks integrated mitigation mechanisms. Falcon [49] reassigns slow links from DP groups to PP stages, but fails to resolve resulting PP stragglers. Recycle [13] and Oobleck [20] rely on static pipeline reconfiguration plans to handle GPU failures, lacking adaptability to dynamic network conditions.

Notation	Explanation
S_i	The ith pipeline stage.
S	Total number of pipeline stages.
N	Total number of microbatches.
t	Execution time of a single operation $F/B/W$.
c_i	Communication latency between S_i and S_{i+1}.
T	The overall pipeline execution time.
x_i	Number of warm-up forwards in stage S_i.
Δ_i	Slackness between stages S_i and S_{i+1}.
δ	Simulator's time step size.

Table 1: Notations in the quantitative analysis and algorithms.

4 Straggler-Resilient Pipeline Adaptation

PIPEMORPH is a system that effectively mitigates communication stragglers for hybrid-parallel training with two key designs: (1) a straggler-resilient pipeline adaptation algorithm that dynamically adapts the pipeline schedule to minimize dependency bubbles (§3.1), and (2) a fully-decoupled data plane eliminating HOL blocking stalls (§3.2).

In this section, we describe the first design component, the pipeline adaptation algorithm, where *we assume no HOL blocking stalls*—which is guaranteed by our second design in §5. We first analytically quantify the accumulated pipeline delays caused by a slow link between PP stages (§4.1). Driven by this analytical result, we design the pipeline adaptation algorithm, including warm-up scheduling (§4.2) and full pipeline scheduling (§4.3). Key mathematical notations are summarized in Table 1 to guide subsequent analysis.

4.1 Quantitative Delay Analysis

Key insight. In §3.1, we empirically demonstrate that communication delays exceeding a certain threshold induce disproportionately significant pipeline stalls through cascading bubbles. We further develop an analytical model to quantify this effect. Our analysis identifies the *slackness* of a pipeline as the key structural resilience to communication delays, informally defined as *the difference of the warm-up forward counts in two adjacent stages*. Intuitively, the more warm-up forward operators the pipeline schedules in stage S_i than in S_{i+1}, the larger slackness it provides between the two stages, which can be utilized to "absorb" more dependency bubbles caused by inter-stage communication delays.

Analysis. To prove this result, we base our analysis on ZeroBubble (ZB) pipeline scheduling, as it is a more generalized design encapsulating common approaches like 1F1B as special cases without backward weight (W) costs. Our analytical findings are hence broadly applicable to 1F1B and other pipeline scheduling approaches.

In a ZB schedule, each pipeline stage operates through three phases (Figure 4): (1) the **warm-up phase** containing a configurable number of forward-only (F) operations, (2) the **steady phase** containing a mixture of forward (F), backward input (B), and backward weight (W) operations, and finally (3) the **cool-down phase** containing the remaining backward

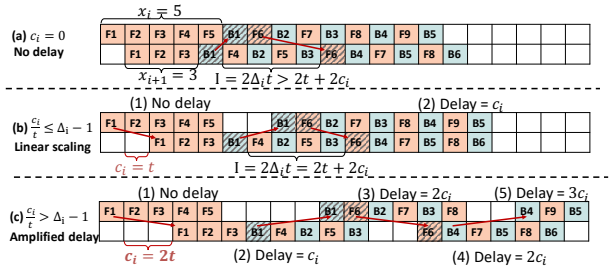


Figure 7: Analysis of accumulated pipeline delay with $\Delta_i = 2$.

weight (W) computations. For ease of presentation, we assume all three operations F, B and W have a uniform execution time ($t^F = t^B = t^W = t$). Nonetheless, our analysis extends to a more general heterogeneous setting. Let N be the total number microbatches in pipeline, S be the number of pipeline stages, and x_i be the number of forward operations scheduled in stage S_i during the warm-up phase, aka *warm-up forward count*. The following lemma shows that x_i is monotonically decreasing, with the proof given in the Appendix.

Lemma 1 (Monotonic warm-up) *For any pipeline schedule, the warm-up forward count is non-increasing over stages, i.e., $x_i \ge x_{i+1}$ for all $i = 0, 1, \ldots, S-1$.*

We formally define the slackness between stages S_i and S_{i+1} as $\Delta_i = x_i - x_{i+1}$, which is guaranteed non-negative by Lemma 1. The following theorem identifies the slackness as the key structural resilience to communication delays.

Theorem 1 (Delay resilience) *Let c_i be the communication delay between stages S_i and S_{i+1}. The accumulated pipeline delay caused by c_i is $\Theta(c_i)$ if $c_i \le (\Delta_i - 1)t$ but amplifies to $\Theta\left(\frac{Nc_i}{\Delta_i+1}\right)$ if $c_i > (\Delta_i - 1)t$, where t is the operation execution time (i.e., $t^F = t^B = t^W = t$).*

Proof: By the Lemma, Δ_i is non-negative, so we can prove this theorem by considering the following two cases.

Case 1: $c_i \le (\Delta_i - 1)t$. For any two *adjacent* B, F operations $B_{i,a}, F_{i,b}$ in S_i (e.g., B_1, F_6 in the Figure 7(b)), we can find its corresponding operations $B_{i+1,a}, F_{i+1,b}$ in S_{i+1}, and define the *feasible interval* I_i as the interval between the end of $B_{i+1,a}$ and the start of $F_{i+1,b}$. During the steady phase, this interval is inherently $2\Delta_i t$ in an ideal no-delay scenario (Figure 7 (a)). To absorb the communication delay, the total execution time of the following operations must not exceed I_i: (1) sending $B_{i,a}$ back to S_i, (2) calculation of $B_{i,a}$ and $F_{i,b}$ in S_i, and (3) sending $F_{i,b}$ to S_{i+1} (costs $2c_i + 2t$ in total).

Therefore, $c_i \le (\Delta_i - 1)t$ ensures that the $2\Delta_i t$ interval is larger than the $2c_i + 2t$ cost. Thus, the delay c_i is fully absorbed without propagating bubbles to subsequent operations (Figure 7 (b)). The accumulated delay is therefore bounded by $\Theta(c_i)$, as no cascading stalls occur.

Case 2: $c_i > (\Delta_i - 1)t$. For this case, the pipeline incurs cascading bubbles as illustrated in Figure 7 (c). This is due to each feasible interval I should be expanded from $2\Delta_i$ to $2c_i + 2t$ to fit the communication and computation operations.

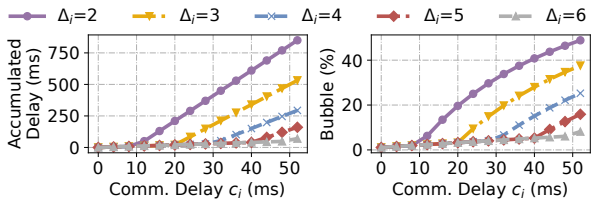


Figure 8: Simulated delay and bubble rate using different Δ_i for a pipeline with $N = 30$ microbatches and $t = 10$ ms.

This expansion postpones a group of $2\Delta_i + 2$ operations by $2c_i$, and the delay will accumulate to the subsequent group of operations. Therefore, for a pipeline with N operations, the overall delay will be $\Theta\left(\frac{Nc_i}{\Delta_i+1}\right)$, as each group of $\Delta_i + 1$ operations contributes c_i to the total. □

Theorem 1 essentially states that a pipeline with slackness Δ_i can tolerate a communication delay up to $(\Delta_i - 1)t$ without triggering cascading bubbles. This result can be extended to a more general setting where the execution times of F, B and W are non-uniform. In this case, communication delay c_i will not introduce cascading bubbles *if and only if*

$$t_i^F + t_i^B + 2c_i \le \Delta_i(t_{i+1}^F + t_{i+1}^B), \tag{1}$$

where t_i^F and t_i^B denote the execution time of forward F and backward input B operations in stage i, respectively. When there are multiple slow links, the accumulated delay is simply the summation of all stragglers' individual contributions.

Figure 8 empirically verifies our analytical findings in simulations: as the communication delay c_i grows beyond the threshold $(\Delta_i - 1)t$, the accumulated pipeline delay sharply increases, aligning with that predicted by Theorem 1.

4.2 Orchestrating Warm-up Forwards

Our previous analysis indicates that enhancing the pipeline's resilience to communication delays requires configuring a larger slackness between two stages. However, doing this comes at a cost of increased memory footprint, as more forward activations are maintained on device. We design pipeline scheduling algorithms that optimally orchestrate warm-up forwards in each stage (i.e., x_i), maximizing the straggler resilience under the memory constraint. Our algorithms include two strategies: (1) *initial planning* for pipeline initialization and (2) *dynamic adaptation* for reconfiguring the pipeline in response to straggler presence. We next explain how the two strategies orchestrate warm-up forwards in each stage, followed by constructing the full-pipeline schedule in §4.3.

Initial planning. During pipeline initialization, the system assumes no knowledge of stragglers. As they can occur on any link between two stages, the best strategy is to *maximize the minimum inter-stage slackness* within the pipeline, i.e., $\max \min_i \Delta_i$. This is equivalent to configuring a pipeline that *uniformly maximizes each* Δ_i under GPU memory constraints. Algorithm 1 shows how this can be achieved. It first computes

Algorithm 1 Initial Planning

Require: Number of pipeline stages S, available GPU memory capacity M, per-activation memory M^F.
Ensure: Initial warm-up forward counts $\{x_0, \ldots, x_{S-1}\}$.

1: **function** GETINITWARMUPFWDS(S, M, M^F)
2: $x_{\max} = \lfloor \frac{M}{M^F} \rfloor$ ▷ *Calculate max #activations on GPU.*
3: $x_0 \leftarrow x_{\max}$ ▷ *First stage holds $x_{\max}$ forwards.*
4: $\Delta_{\text{avg}} \leftarrow \lfloor \frac{x_{\max}-1}{S-1} \rfloor$
5: $r \leftarrow (x_{\max} - 1) \bmod (S-1)$
6: **for** $i \leftarrow 1$ **to** $S-1$ **do**
7: ▷ *Calculate Δ_{i-1} and x_i recursively.*
8: $\Delta_{i-1} \leftarrow \Delta_{\text{avg}} + 1$ **if** $i \leq r$ **else** Δ_{avg}
9: $x_i \leftarrow x_{i-1} - \Delta_{i-1}$
10: **return** $\{x_0, \ldots, x_{S-1}\}$

Algorithm 2 Dynamic Adaptation

Require: Number of pipeline stages S, per-stage forward/backward times $\{t_i^F\}, \{t_i^B\}$, inter-stage communication delays $\{c_i\}$.
Ensure: Adapted warm-up forward counts $\{x_0, \ldots, x_{S-1}\}$.

1: **function** GETADAPTEDWARMUPFWDS($S, \{t_i^F\}, \{t_i^B\}, \{c_i\}$)
2: ▷ *The last stage holds only one warm-up forward.*
3: $x_{S-1} \leftarrow 1$
4: ▷ *Calculate Δ_i and x_i backward-recursively.*
5: **for** $i \leftarrow S-2$ **to** 0 **do**
6: ▷ *Ensure bubble absorption by Equation 1.*
7: $\Delta_i \leftarrow \min\left(N - 2S, \max\left(\left\lceil \frac{t_i^F + t_i^B + 2c_i}{t_{i+1}^F + t_{i+1}^B} \right\rceil, 2\right)\right)$
8: $x_i \leftarrow x_{i+1} + \Delta_i$
9: **return** $\{x_0, \ldots, x_{S-1}\}$

the maximum number of forward activations that a GPU can maintain in memory, all of which are assigned to stage 0 (lines 2 and 3). With x_0 determined, it then computes the warm-up forward counts in subsequent stages to ensure that x_i's are monotonically decreasing (Lemma 1) with as balanced slackness Δ_i as possible (lines 4 to 9). The generated pipeline provides the maximum uniform delay resilience.

Dynamic adaption. Upon detecting a communication delay exceeding the tolerance threshold (given by Equation 1), the system reconfigures the pipeline to increase the slackness between the affected stages, aiming to "absorb" as many bubbles as possible. At this point, *memory constraints can be relaxed* as PIPEMORPH offloads activations to host memory—which provides significantly larger space than the device memory—and uses CPU-side RDMA for data transfer to eliminate HOL blocking stalls (details in §5). Therefore, the optimal strategy is to maximize Δ_i, under virtually no memory limit.

Algorithm 2 implements this strategy. Starting backward from the last stage requiring only one warm-up forward (line 3), it recursively computes the desired warm-up count x_i in the preceding stage (for-loop). Specifically, it uses profiled per-stage compute times and communication delays to determine the minimum required slackness Δ_i for all i using Equation 1, ensuring that Δ_i is large enough to absorb the observed delay while clipping it by $N - 2S$ to preserve enough forwards for other stages' warm-up (line 7). Once Δ_i is computed, x_i easily follows (line 8).

4.3 Full-Pipeline Orchestration

With the warm-up forward count determined in each stage, we now construct the complete pipeline execution schedule. Optimally orchestrating all operator executions across stages and microbatches formulates a mixed-integer linear programming (MILP) problem [15, 34] and is NP-hard [46]. We hence turn to an efficient heuristic that sequentially generates a pipeline schedule following its execution timeline, discretized into multiple time steps. At each time step, the algorithm does two things: (1) simulating pipeline execution and (2) making new scheduling decisions.

Simulation. First, it keeps track of the execution of previously scheduled operators ($F/B/W$) and updates their states in each stage—similar to running a discrete-time simulation. It also maintains a list of *schedulable operators* for each stage and updates it accordingly (e.g., an F becomes schedulable in stage S_i after its upstream F completes in S_{i-1}).

Operator selection. Second, for each idle stage, the algorithm makes new scheduling decisions based on the updated system state. It chooses an operator from the schedulable list following a *two-phase operator selection policy*. During the warm-up phase, each stage S_i executes only forward operators until reaching the assigned quota x_i (computed by Algorithm 1 or 2), ensuring the desired straggler resilience. After all warm-up forwards complete, the stage transitions into a steady phase, in which it selects operators from the schedulable list in a priority order of $B > F > W$. Specifically, backward input operators (B) are prioritized to immediately free activation memory and propagate dependencies upstream; forwards (F) are selected next, as they generate single downstream dependencies; backward weight (W) operators have the lowest priority, since they do not generate further dependencies and can be scheduled opportunistically.

Optimality and complexity. We will show in §7.4 that this simple heuristic generates pipeline schedules that closely approximate the optimum–obtained by solving an MILP problem–when the simulation is configured with a fine-grained step size δ. In terms of the complexity, let t_o be the longest operator execution time, i.e., $t_o = \max_i(t_i^F, t_i^B, t_i^W o)$. The algorithm completes in at most $3NS\lceil t_o/\delta \rceil$ steps, as each operator (of which there are $3N$ per stage over S stages) is scheduled at most once per δ interval. Each step involves S constant-time policy evaluations, resulting in an overall complexity of $O(NS^2\lceil t_o/\delta \rceil)$.

5 Decoupled Data Plane via Comm. Delegation

The effectiveness of the aforementioned pipeline adaptation algorithms (§4)—our first key design—is contingent on *eliminating head-of-line (HOL) blocking stalls*. In this section, we

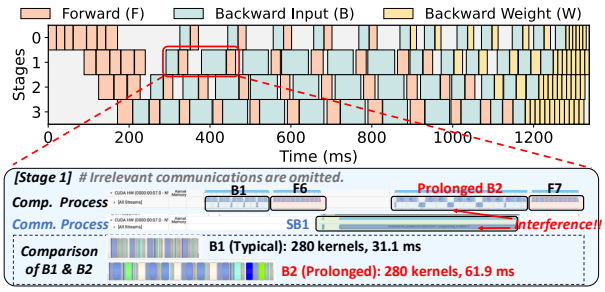


Figure 9: Naively adopting NCCL-based opportunistic communication [5, 42] solves the blocking issue, but introduces severe interference to computation.

start with a straw man solution and illustrate its ineffectiveness (§5.1). We then present our second key design to eliminate HOL blocking stalls (§5.2), which additionally provides fault tolerance to RNIC failures (§5.3).

5.1 Straw Man Solution

Recall in §3.2 that HOL blocking is caused by sequential launching of communication and subsequent computation operations (Figure 6). Therefore, the key to avoiding HOL blocking is to *decouple slow communication operations from the compute sequence*, thereby ensuring that all compute kernels can be launched without a delay.

A straw man solution is *opportunistic communication* [5, 42]. It delegates communication operations to some dedicated processes, allowing the main training process to concentrate on computation. These dedicated communication processes asynchronously retrieve data from shared buffers and transmit it to adjacent pipeline stages via GPU-direct RDMA using NCCL. However, this approach introduces significant interference to computation. As illustrated in Figure 9, although computation and communication operations are already in separated CUDA streams and the priority of communication process is carefully tuned to minimize interference, overlapping computation and communication results in substantial kernel execution slowdowns. For instance, stage-1's backward operation (B_2) increases from 31 ms to 61.9 ms when overlapped with SB_1. Our profiling reveals that, although this approach closes the kernel launch gaps, the runtime of individual kernels is greatly prolonged: a GEMM kernel that typically finishes in 110 μs is stretched to 2 ms under interference. As a result, despite being blocking-free, the pipeline's end-to-end performance degradation is still about 1.9×, no better than sequential execution.

5.2 CPU-based Communication Delegation

Key idea. While the NCCL-based straw man introduces severe interference, its delegation paradigm remains valid—provided we can avoid such interference. This inspires us to offload activation and gradient transfers from the GPU to

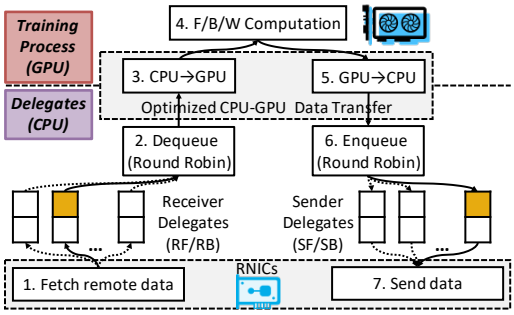


Figure 10: CPU-delegated data transmission path.

host memory upon straggler detection, and perform send/receive operations using dedicated CPU-side delegate processes to avoid interfering GPU computation. This design enables three key benefits. *First*, it fully decouples communication operations from GPU kernel scheduling, preventing slow communication from stalling GPU computation operations (blocking-free). *Second*, offloading activation and gradients to the host lifts the GPU memory pressure, enabling orchestrating a memory-intensive pipeline schedule with more warm-up forwards and larger slackness for enhanced straggler resilience (i.e., virtually no memory constraint in Algorithm 2). *Third*, it additionally provides RNIC fault tolerance: in case of GPU-side RNIC failures, the host RNICs serves as a backup.

Design. In our design, the delegated communication path is activated only upon the detection of communication delays. For each type of communication (e.g., send-forward), the framework launches multiple CPU communication processes, each with its own transmission queue. This multi-queue design ensures that the total data consumption rate keeps pace with the data production rate of computation. For example, if the GPU produces 8 activations per second while each communication delegate can consume only 2 per second, at least 4 sending queues are needed to avoid blocking and queue buildup.

Figure 10 illustrates the data transmission path. ① The receiver delegates eagerly fetch data from remote peers during training. The training process ② retrieves input data from a receiver queue in a *round-robin* manner and ③ copies it to GPU. The GPU ④ computes the results, which are ⑤ copied back to host and ⑥ enqueued into the corresponding sender queue. ⑦ Finally, the sender delegates send the results via CPU-side RDMA.

Optimizing data transfer. As our design bypasses GPU-direct RDMA (GDR) and utilizes a slower CPU-side RDMA, reducing the overhead of data movement between the host and the GPU becomes critical. To minimize this overhead, we design a fine-grained data pipeline with optimized CUDA kernels which move data asynchronously and only report to the training process when the data is ready.

Specifically, our optimization creates a pinned shared memory buffer for each delegate, which allows faster GPU data access than pageable memory through DMA and zero additional data copy between the delegate and the training pro-

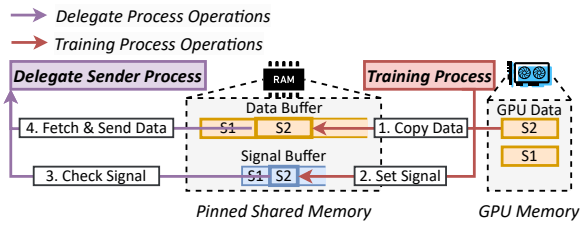


Figure 11: Optimized data transfer for sending data.

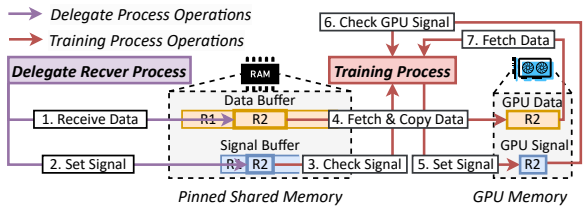


Figure 12: Optimized data transfer for receiving data.

cesses during IPC. For each replica of a communication type (e.g., send-forward), both its training process and itself can access a piece of pinned shared memory. When sending forward or backward, as shown in Figure 11, the training process initiates two sequential `cudaMemcpy()` operations in the same CUDA stream. It first ① copies data from GPU to host and then ② sets a copy completion signal to guarantee data integrity. After checking the signal, the delegate process ③ ensures copy completion and ④ sends data via RDMA.

When receiving forward or backward, as shown in Figure 12, once ① data is received from remote via RDMA, ② the delegate process sets a signal in shared memory indicating data ready. ③ Meanwhile, the training process checks this signal through busy waiting. After confirming, two sequential `cudaMemcpy()` operations are initiated to first ④ copy data from host to GPU, followed by ⑤ setting a signal in GPU memory to acknowledge data copy completion. Once the training process ⑥ sees this data ready signal, the subsequent compute operators can ⑦ consume this data safely.

5.3 Handling RNIC Failures

During NCCL initialization, each GPU is assigned a dedicated RNIC to avoid bandwidth contention within a node. Consequently, a single RNIC failure during NCCL communication results in a connection loss or a timeout error, even if the other RNICs are still well functioning. The GDR path is hence vulnerable to RNIC failures.

The CPU-based communication delegation provides an inherent tolerance to RNIC failures as it bypasses the faulty GDR path. To achieve this, PIPEMORPH wraps each communication operation in a `try-catch` block. Upon detecting an error indicative of an RNIC issue like `NCCLTimeout`, it automatically retries the failed operation using the CPU delegation path across all involved nodes. Therefore, each delegate process can flexibly choose an RNIC for data transfer using Gloo [19] during training. In the event of an RNIC failure, the delegate process reroutes the affected communication traffic from the faulty RNIC to a healthy one to continue data transfer, enabling uninterrupted training as opposed to conventional checkpoint-and-restart failover.

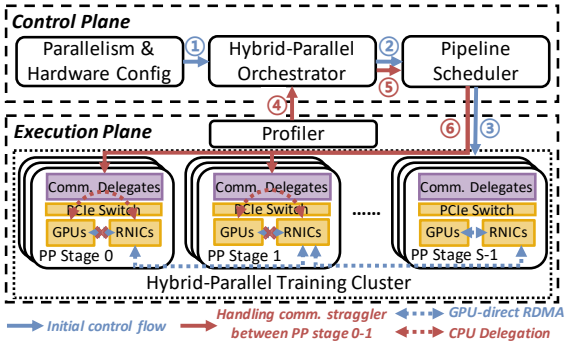


Figure 13: PIPEMORPH system design.

6 PIPEMORPH Design and Implementation

PIPEMORPH is an efficient parallel training system that integrates the two key designs described in §4 and §5 to deliver robust resilience against communication stragglers. As illustrated in Figure 13, it comprises three main components: (1) a *profiler* that continuously monitors each node's compute and communication performance, (2) a *hybrid-parallel orchestrator* that dynamically determines the communication topology and constructs TP, DP, and PP communication groups based on runtime performance, and (3) a *pipeline scheduler* that adaptively configures a resilient pipeline schedule against dynamic stragglers using the algorithms developed in §4.

Initialization. During system initialization, ① the orchestrator establishes the initial TP/DP/PP communication groups according to the parallelism strategy and hardware configuration (e.g., network topology). ② The scheduler then generates an initial pipeline schedule that provides uniform resilience to potential stragglers across all stages (§4). ③ This schedule is deployed on the cluster, and the training starts with all inter-node communication performed via GPU-direct RDMA.

Straggler mitigation. Throughout training, the profiler continuously tracks communication and computation performance. Upon detecting slow communication (using detection techniques in [9, 49]), it reports this straggler event to the orchestrator (④). If the affected link is between PP stages (e.g., stages 0 and 1), the orchestrator notifies the pipeline scheduler (⑤), which then adapts the pipeline schedule as described in §4. The updated schedule is then deployed on the cluster, and CPU communication delegates are activated at the affected nodes to eliminate HOL blocking stalls (⑥). If the slow link is part of a DP communication group, the orchestrator reconfigures the training topology to reassign this link for PP communication, effectively converting a DP

Model Size	7B	14B	30B	60B	140B
Parallelism (TP, DP, PP)	(1, 1, 4)	(1, 1, 8)	(2, 4, 8)	(4, 2, 8)	(8, 2, 8)
#GPUs	4	8	64	64	128

Table 2: Models and corresponding 3D-parallelism settings.

straggler into a less detrimental PP straggler [49], which is then addressed using the above mechanisms.

Implementation. PIPEMORPH is implemented on top of Megatron-LM [31] and ZeroBubble [34], comprising 5.3K lines of code (LoC), primarily in Python, with performance-critical data transfer kernels written in CUDA. The straggler detector and orchestrator are adapted from Falcon [49], while the profiler leverages CUDA Events and reports profiles via Redis [38]. For CPU-side communication, PIPEMORPH utilizes Gloo [19] to facilitate RDMA data transfers.

7 Evaluation

In this section, we evaluate PIPEMORPH to answer the following questions: (1) Does PIPEMORPH effectively address dependency bubbles and HOL blocking stalls caused by PP stragglers (§7.2)? (3) Does PIPEMORPH also effectively handle DP stragglers (§7.3)? (2) Can PIPEMORPH generate an optimal schedule and delegate communication with acceptable overhead (§7.4)? (4) How does PIPEMORPH perform in large-scale pretraining in the presence of frequent communication stragglers and RNIC failures (§7.5)?

7.1 Experimental Setup

Cluster setup. Our evaluation is conducted on a 128-GPU cluster, where each node is equipped with 8 NVIDIA H800 GPUs and 400 Gbps InfiniBand inter-node connections.

Baselines. We evaluate PIPEMORPH against four baselines.

1. 1F1B [30] is a classic pipeline schedule with low bubble rate and controllable memory footprint.
2. ZeroBubble (ZB) [34] is a SOTA pipeline schedule that eliminates bubbles via decoupled backward passes.
3. Falcon [49] migrates slow links to PP groups if stragglers occur on DP groups, but does not mitigate the stragglers' residual impact on pipeline execution.
4. PIPEMORPH-CPU only enables delegated communication (§5) without pipeline adaptation.

Models and Parallelism. We evaluate PIPEMORPH using GPT-2 models of varying sizes, ranging from 7B to 140B parameters, on up to 128 GPUs across 16 nodes. The models and corresponding parallelism settings are given in Table 2.

7.2 Mitigating PP Stragglers

Microbenchmark. Before introducing end-to-end performance, we first demonstrate the behavior of PIPEMORPH's

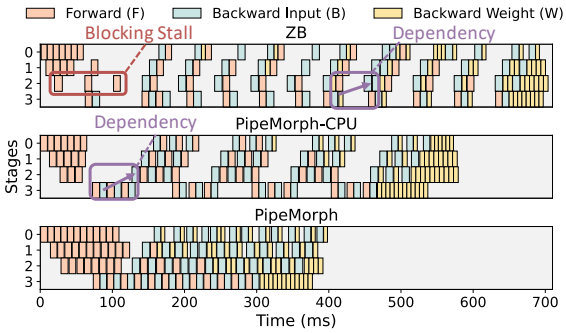


Figure 14: The actual execution of the schedule using PIPEMORPH and other two baselines on a 7B model under 30 ms delay on the link between last two PP stages.

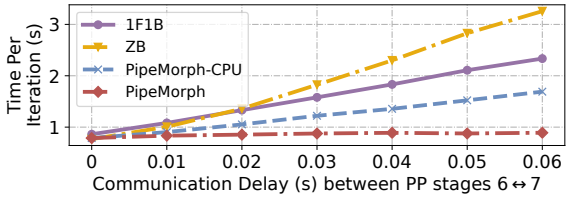


Figure 15: Sensitivity analysis of PIPEMORPH and baselines using a 14B model under various delay values.

two designs in addressing dependency bubbles and HOL blocking stalls using a GPT2-7B model. We inject a 30 ms delay between pipeline stages 2 and 3 and then profile the execution of each operator within an iteration using `cudaEvent`, with the execution timeline shown in Figure 14.

In the original ZB execution, we observe significant bubbles between warm-up forward operators in stage 2 caused by HOL blocking stalls and dependency-bubbles exacerbate the performance. This compound effect extends the iteration time to 703 ms with 57.4% bubble rate (1.9× longer than the normal execution). PIPEMORPH-CPU retains the same pipeline schedule as ZB but activates CPU-based delegation, eliminating HOL blocking stalls by redirecting the original GPU-direct RDMA to CPU-based RDMA operations. As a result, the iteration time improves to 579 ms, with the bubble rate reduced to 48.8% solely due to dependency issues. Further enabling pipeline adaptation (i.e., the complete PIPEMORPH) reduces the iteration time to 398 ms (only 30 ms slowdown) and the bubble rate to only 25.3%. This 1.76× improvement is achieved by increasing the slackness between stages 2 and 3 (i.e., Δ_2), substantially enhancing the pipeline's resilience to the injected communication delays.

Sensitivity analysis. We assess the impact of delay values on end-to-end iteration times by gradually increasing the latency between the last two stages of a 14B model from 0 to 60 ms. As shown in Figure 15, a 60 ms communication delay slows down 1F1B, ZB, and PIPEMORPH-CPU significantly by 2.70×, 4.24×, and 2.15×, respectively. Notably, ZB, though achieving better performance without stragglers, is more vulnerable to communication delays due to its tightly

coupled schedule. In contrast, PIPEMORPH consistently outperforms baseline systems with slightly increased iteration times. Specifically, under a 60 ms delay, PIPEMORPH measures a modest slowdown of 1.13× thanks to the two designs described in §4 and §5. Compared to ZB, switching to CPU-based communication delegation (PIPEMORPH-CPU) mitigates the straggler impact by 48.1%; this impact is further alleviated by 24.6% using pipeline adaptation.

Resilience to single-link degradation. We next evaluate PIPEMORPH's performance in the presence of a single-link straggler under various model and parallelism settings. As illustrated in Figure 16, we inject a delay of 30 ms (or 60 ms) into a single communication link between the first (or last) two PP stages. Across all model sizes from 7B to 60B, PIPEMORPH consistently outperforms baseline methods. In particular, it achieves up to 3.71× speedup over 1F1B (in the scenario of 7B, 60 ms, last two stages) and 3.49× speedup over ZB (14B, 60 ms, last). When the communication delay increases from 30 ms to 60 ms, the average iteration time measured across all models increases by only 1.07× using PIPEMORPH, compared to 1.49×, 1.69×, and 1.30× using 1F1B, ZB, and PIPEMORPH-CPU, respectively. These results suggest that dependency bubbles account for 39% of slowdown, while HOL blocking stalls contribute 23% on average. PIPEMORPH effectively mitigates both issues.

Note that the straggler location also matters. A delay occurring between the final pipeline stages has a more significant performance impact than one at the beginning, as there is less subsequent scheduling slackness (Δ_i) available to absorb it, leading to tighter computational dependencies. PIPEMORPH effectively addresses this issue through pipeline adaptation: it improves the average iteration time by 19.3% compared to PIPEMORPH-CPU under a 60 ms delay between the first two stages; this gain increases to 38.4% when the same delay occurs between the last two stages.

Multi-link degradation. To evaluate PIPEMORPH under more complex straggler conditions, we configure multiple stragglers in the 14B setting with 30 ms delays occurring on two adjacent links (0-1 and 1-2), two skip links (0-1 and 2-3), first-and-last links (0-1 and 6-7), and three skip links (0-1, 3-4, 6-7), respectively. Note that each degraded link incurs an additional 30 ms of latency, so the total delay scales with the number of affected links and results in a greater overall impact than a single link delay. As detailed in Figure 17, PIPEMORPH reduces the iteration time by 51.6% and 57.5% on average across the four settings, compared to 1F1B and ZB. Importantly, adaptive pipeline reconfiguration improves the average performance by 23.1% (comparing to PIPEMORPH-CPU), further validating the effectiveness of PIPEMORPH's scheduling algorithm under complicated delay conditions.

Dynamic stragglers. To evaluate PIPEMORPH against dynamic stragglers, we inject rapidly changing network delays into a GPT2-7B training workload. The latency of each inter-PP link is sampled from a normal distribution $\mathcal{N}(\mu = 30, \sigma = 10)$ms and updated at intervals drawn from three uniform distributions: A:$\mathcal{U}$(50, 100)ms, B: $\mathcal{U}$(100, 200)ms, and C: $\mathcal{U}$(200, 400)ms (Figure 18-upper, a shorter interval means a *higher changing frequency*). Despite delay changing is much faster than the iteration time and rescheduling only at iteration boundaries, PIPEMORPH consistently outperforms 1F1B and ZB, achieving up to 2.8× speedup (Figure 18-lower). Its robustness stems from the initial schedule's slackness absorbing minor jitters (§4.2), complemented by rescheduling at the next iteration to handle prolonged delay spikes. We leave finer-grained, intra-iteration rescheduling to future work.

7.3 Mitigating DP Stragglers

Communication stragglers can also occur between DP groups. We evaluate PIPEMORPH in this scenario using a GPT2-7B model on 8 nodes with ZB scheduling under a configuration of (1 TP, 2 DP, 4 PP). We inject 10/20/30 ms inter-node communication latency into a designated link. As shown in Figure 19, when this link is for DP communications, the baseline ZB degrades by 8.04×, far exceeding the impact of PP communication stragglers. In response, both PIPEMORPH and Falcon [49] mitigate this by reassigning the slow link to PP communication, leading to 3.6× improvement. PIPEMORPH further mitigates the residual PP straggler through pipeline adaptation and CPU-delegated communications, achieving 1.96× additional speedup over Falcon.

7.4 Optimality and Overhead

Performance and overhead of pipeline scheduling. We evaluate the performance of our pipeline scheduling algorithm described in §4 against the optimal schedule obtained by formulating an MILP problem. We consider four pipeline configurations (with 3-8 stages and 6-32 microbatches) with randomly generated profiles. As illustrated in Figure 20, configuring a smaller time step δ (higher $\lceil t_o/\delta \rceil$) for fine-grained simulation narrows the gap between the generated schedule and the optimum, at a cost of increased schedule generation time. In all settings, the gap, measured by the relative difference in execution time, is less than 1% when running at a fine granularity of $\lceil t_o/\delta \rceil = 30$. These near-optimal schedules are generated in less than 100 ms, as opposed to computing the optimal schedules that requires solving an MILP in hours.

Overhead of delegation. We stress-test the delegated path across models from 7B to 60B parameters. Under a *worst-case scenario* where all pipeline communications are forcibly routed through CPU delegates (All-CPU). Figure 21-left presents a breakdown analysis to quantify the components of introduced overhead, which is dominated by the necessary PCIe data transfers and slower CPU-side RDMA. Across the evaluated model sizes, device-to-host (D2H) copy times range from 0.6% to 1.2%, host-to-device (H2D) copies range

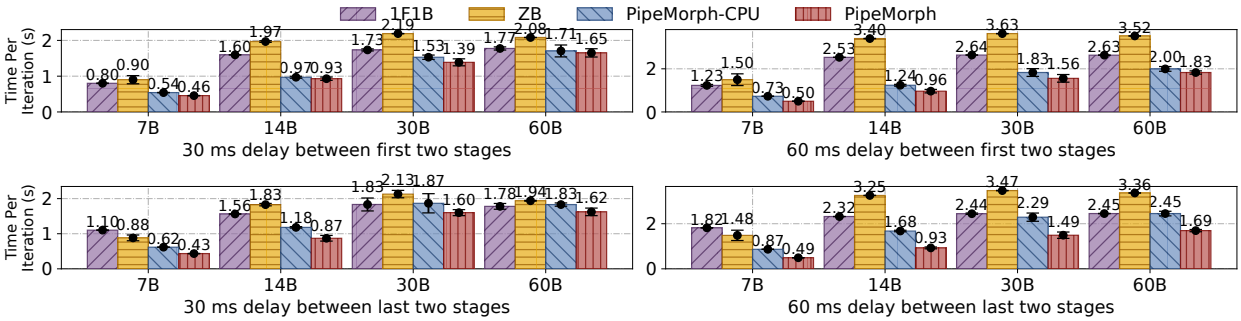


Figure 16: Evaluation on single inter-PP communication degradation under various model settings and delay locations.

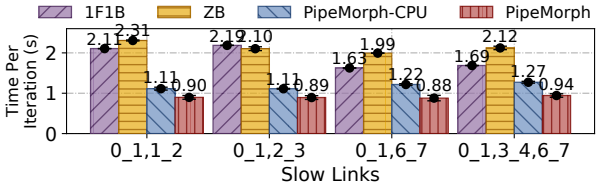


Figure 17: Iteration times of a 14B model under multiple simultaneous inter-PP communication stragglers.

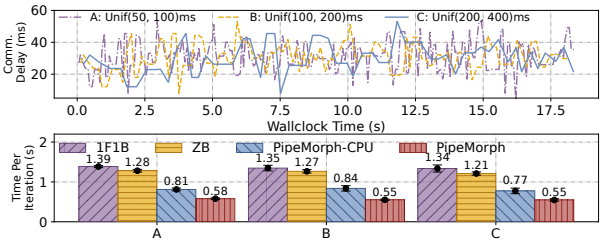


Figure 18: **Upper**: a 20s-part of communication delay traces A/B/C with different changing frequencies. **Lower**: corresponding performance of PIPEMORPH and baselines.

from 2.6% to 4.4%, and CPU-side RDMA ranges from 5.7% to 10.5%. PIPEMORPH effectively overlaps these operations via an optimized, asynchronous transfer mechanism (§ 5.2) to minimize end-to-end impact, and maintains comparable performance to the strongest GPU-direct RDMA baseline (i.e., ZB) for models with 7B or 14B parameters. At 30B or 60B scale, All-CPU introduces only up to 16.4% additional iteration time compared to GDR (Figure 21-right).

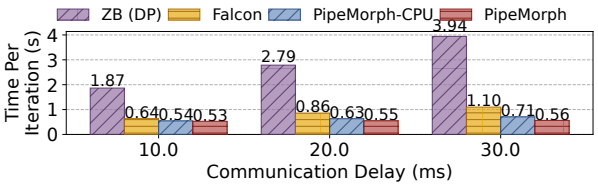


Figure 19: A communication straggler in DP group introduces significant baseline degradation. Both Falcon [49] and PIPEMORPH migrates this link to PP groups, while PIPEMORPH further optimizes the residual impacts.

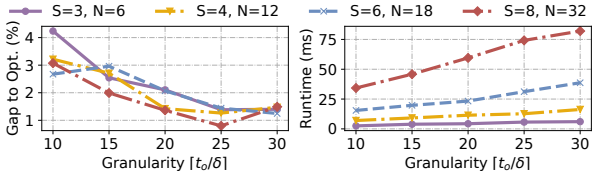


Figure 20: The relative error to optimal solution and solving time of PIPEMORPH's scheduler, where it achieves an near-optimal solution within 1% error in 0.1 seconds.

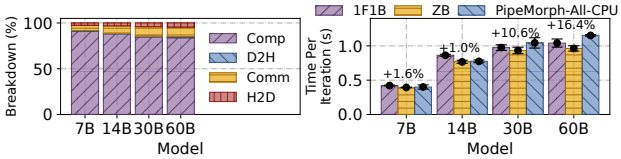


Figure 21: Worst-case overhead of PIPEMORPH's delegated communication w/o presence of communication stragglers. **Left**: breakdown analysis, **Right**: end-to-end performance.

7.5 PIPEMORPH in the Wild

We evaluate PIPEMORPH's robustness at scale by training a 140B model on 128 GPUs across 16 nodes under realistic network conditions. To do so, we first record a one-hour trace of inter-PP stage communication delays from a large-scale production training job. As shown in Figure 22-upper, the delay trace is highly bursty and fluctuating, with median/mean/99th-percentile of 13.0/22.3/79.3 ms.

For a fair comparison, we evaluated each method under exactly the same network conditions, achieved by replaying the recorded trace for each experiment run. Figure 22-lower shows that under these volatile network conditions, the throughput of ZB degrades significantly, at times performing worse than the simpler 1F1B. This vulnerability stems from ZB's fine-grained scheduling of backward pass operators (B, W), which creates tight data dependencies that are highly susceptible to delays. During periods of severe congestion (e.g., t=1700-2200s), the throughput of both baselines drops to below 30 iters/min. In contrast, PIPEMORPH consistently maintains its throughput above 40 iters/min by mitigating HOL blocking and dependency-induced bubbles.

Overall, PIPEMORPH achieves an average throughput of

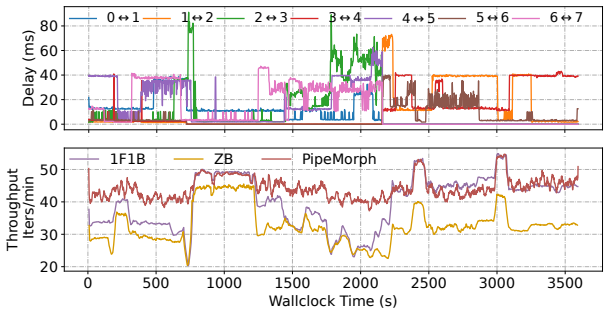


Figure 22: **Upper**: Inter-PP communication delays recorded from real-world network environments. **Lower**: Throughput of PIPEMORPH of pretraining a 140B model on 128 GPUs against baselines, where PIPEMORPH improves the end-to-end throughput by up to 1.36×.

45.9 iters/min, outperforming 1F1B (41.4 iters/min) and ZB (33.6 iters/min) by 1.11× and 1.36×, respectively. The performance gap becomes even more pronounced during the severely congested period (t=1300-2500s), where PIPEMORPH's throughput gains grow to 1.30/1.47× over 1F1B/ZB, demonstrating PIPEMORPH's ability to maintain high performance under realistic network circumstances.

8 Limitations and Future Works

CPU delegation overhead. While PIPEMORPH's CPU delegation effectively mitigates HOL blocking under communication stragglers, it introduces a residual overhead from PCIe data transfers. We note that recent advances in MoE training have addressed similar challenges with more GPU-native techniques. For instance, DeepEP [52] uses PTX-level programming to orchestrate communications on separate SMs, while FlashMoE [3] fuses them into a single CUDA kernel. A promising future direction is to adapt these optimizations for pipeline parallelism, which may offer a new path to minimize latency by removing host involvement entirely.

Intra-iteration re-scheduling. Another limitation of the current design is that pipeline rescheduling occurs only at iteration boundaries. While this policy is effective even for rapid network fluctuations, our analysis reveals a 5% performance gap to optimal under extremely high-frequency latency changes. This is because although we preserve slackness to handle minor jitters, it can be overwhelmed by large, mid-iteration delays, resulting in bubbles. Future work could therefore implement a dynamic, intra-iteration rescheduling mechanism that reacts to latency spikes as they occur, offering a more resilient solution for highly volatile environments.

9 Related Works

Reliability issues in training. Abundant studies address training crashes using checkpoints [29, 48], redundant computations [45], and elastic frameworks [20, 29, 45, 48]. However, seldom existing works address communication stragglers, while they have been identified in several reports [12, 22],. Malleus [25] solely mitigates compute stragglers without considering communications. Falcon [49] addresses slow communication by shifting the degraded links to PP groups without optimizing the residual impact. Crux [6]'s communication-aware scheduling tries to reduce the occurrence probability of stragglers, yet not mitigating their impacts.

Communication optimizations for training. Communication optimizations span three critical layers. Infrastructure-level efforts like Alibaba HPN [35] and Megascale [22] propose specialized network topologies for training clusters. At the library level, TACCL [40], ACCL [11], and MSCCL [8] develop optimized communication primitives for collective operations. Framework-level approaches including Megatron-LM [31], Varuna [5], and DeepEP [52] enhance computation-communication overlap in hybrid-parallel settings.

Pipeline parallelism optimizations. Pipeline scheduling remains challenging in distributed training. Classic approaches like Gpipe [18] and 1F1B [30] achieve comparable bubble rates, with 1F1B additionally reducing memory usage. Recent advances address distinct dimensions: Interleaved 1F1B [31] reduces bubbles via introducing virtual stages, while ZeroBubble [34] eliminates them through decoupled backward passes. For specialized architectures, DualPipe [15] optimizes MoE training pipelines and RLHFuse [54] tailors for RLHF workloads. In terms of improving reliability, Recycle [13] employs precomputed schedules with microbatch rerouting for fail-stop recovery, whereas SDPipe [28] trades training accuracy for computation straggler-resilience.

10 Conclusion

This paper presents PIPEMORPH, a system designed for efficient hybrid-parallel training in the presence of communication stragglers. PIPEMORPH employs dynamic pipeline adaption to minimize dependency bubbles and CPU-delegated communication to eliminate head-of-line blocking stalls. Experiments demonstrate that PIPEMORPH achieves 1.2-3.5× speedup over SOTA baselines under communication stragglers, and 1.36× higher throughput with zero restart overhead upon RNIC failures.

Acknowledgment

We thank our shepherd, Ram Ramjee, and the anonymous reviewers for their valuable comments that help improve the quality of this work. This work was supported in part by the Alibaba Innovative Research (AIR) Grant, RGC CRF Grant (Ref. #C6015-23G), RGC GRF Grant (Ref. #16217124), and NSFC/RGC CRS Grant (Ref. #CRS_HKUST601/24 and Ref. #CRS_PolyU501/23).

References

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

[2] Vamsi Addanki, Prateesh Goyal, and Ilias Marinos. Challenging the need for packet spraying in large-scale distributed training. *arXiv preprint arXiv:2407.00550*, 2024.

[3] Osayamen Jonathan Aimuyo, Byungsoo Oh, and Rachee Singh. Flashdmoe: Fast distributed moe in a single kernel. *arXiv preprint arXiv:2506.04667*, 2025.

[4] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.

[5] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[6] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. Crux: Gpu-efficient communication scheduling for deep learning training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 1–15, 2024.

[7] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[8] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. Mscclang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 502–514, 2023.

[9] Weihao Cui, Ji Zhang, Han Zhao, Chao Liu, Wenhao Zhang, Jian Sha, Quan Chen, Bingsheng He, and Minyi Guo. Xputimer: Anomaly diagnostics for divergent llm training in gpu clusters of thousand-plus scale. *arXiv preprint arXiv:2502.05413*, 2025.

[10] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *2013 Proceedings IEEE INFOCOM*, pages 2130–2138. IEEE, 2013.

[11] Jianbo Dong, Shaochuang Wang, Fei Feng, Zheng Cao, Heng Pan, Lingbo Tang, Pengcheng Li, Hao Li, Qianyuan Ran, Yiqun Guo, et al. ACCL: Architecting highly scalable distributed training systems with highly efficient collective communication library. *IEEE micro*, 41(5):85–92, 2021.

[12] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[13] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. Recycle: Resilient training of large dnns using pipeline adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 211–228, 2024.

[14] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 57–70, 2024.

[15] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[16] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Addressing the straggler problem for iterative convergent parallel ml. In *Proceedings of the seventh ACM symposium on cloud computing*, pages 98–111, 2016.

[17] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 709–729, 2024.

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

[19] Facebook Incubator. Gloo, 2025. Accessed: 2025-03-26.

[20] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 382–395, 2023.

[21] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[22] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. MegaScale: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[23] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[24] Yanfang Le, Rong Pan, Peter Newman, Jeremias Blendin, Abdul Kabbani, Vipin Jain, Raghava Sivaramu, and Francis Matus. Strack: A reliable multipath transport for ai/ml clusters. *arXiv preprint arXiv:2407.15266*, 2024.

[25] Haoyang Li, Fangcheng Fu, Hao Ge, Sheng Lin, Xuanyu Wang, Jiawen Niu, Yujie Wang, Hailin Zhang, Xiaonan Nie, and Bin Cui. Malleus: Straggler-resilient hybrid parallel training of large-scale models via malleable data and model parallelization. *arXiv preprint arXiv:2410.13333*, 2024.

[26] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, et al. Torchtitan: One-stop pytorch native solution for production ready llm pre-training. *arXiv preprint arXiv:2410.06511*, 2024.

[27] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[28] Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, and Zhihao Jia. Sdpipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training. *Proceedings of the VLDB Endowment*, 16(9):2354–2363, 2023.

[29] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. {CheckFreq}: Frequent,{Fine-Grained}{DNN} checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[30] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[31] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[32] NVIDIA Cooperation. Nvidia nsight systems, 2025. Accessed: 2025-03-24.

[33] OpenAI. Openai sora, 2024. Accessed: 2024-09-13.

[34] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism. *arXiv preprint arXiv:2401.10241*, 2023.

[35] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 691–706, 2024.

[36] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. {CASSINI}:{Network-Aware} job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1403–1420, 2024.

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

[38] Salvatore Sanfilippo. Redis - the real-time data platform, 2009. Accessed: 2024-09-08.

[39] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, et al. BLOOM: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2023.

[40] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz,

Jacob Nelson, Olli Saarikivi, and Rachee Singh. {TACCL}: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 593–612, 2023.

[41] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[42] Mohammed Sourouri, Tor Gillberg, Scott B Baden, and Xing Cai. Effective multi-gpu communication using multiple cuda streams and threads. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 981–986. IEEE, 2014.

[43] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[44] Dave Thaler and C Hopps. Multipath issues in unicast and multicast next-hop selection. Technical report, 2000.

[45] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, 2023.

[46] Jeffrey D. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[47] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. Machine learning model sizes and the parameter gap. *arXiv preprint arXiv:2207.02852*, 2022.

[48] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 364–381, New York, NY, USA, 2023. Association for Computing Machinery.

[49] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. FALCON: Pinpointing and mitigating stragglers for large-scale hybrid-parallel training. *arXiv preprint arXiv:2410.12588*, 2024.

[50] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, et al. SuperBench: Improving cloud AI infrastructure reliability with proactive validation. In *2024 USENIX Annual Technical Conference (ATC'24)*, pages 835–850, 2024.

[51] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[52] Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. Deepep: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025.

[53] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[54] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, et al. Rlhfuse: Efficient rlhf training for large language models with inter-and intra-stage fusion. *arXiv preprint arXiv:2409.13221*, 2024.

[55] Qihua Zhou, Song Guo, Haodong Lu, Li Li, Minyi Guo, Yanfei Sun, and Kun Wang. Falcon: Addressing stragglers in heterogeneous parameter server via multiple parallelism. *IEEE Transactions on Computers*, 70(1):139–155, 2020.

Appendix

Proof of the Lemma in § 4.1

Assume for contradiction that $x_i < x_{i+1}$ for some stage S_i. Since the warm-up phase of S_i ends after F_{i,x_i}, F_{i,x_i+1} must belong to the steady phase of S_i, preceded by $B_{i,1}$, i.e.,

$$F_{i,x_i} \prec B_{i,1} \prec F_{i,x_i+1}. \tag{2}$$

However, $B_{i+1,1}$ can only occur after S_{i+1}'s warm-up phase ends with $F_{i+1,x_{i+1}}$. Furthermore, due to pipeline dependencies, $F_{i+1,x_{i+1}}$ depends on $F_{i,x_{i+1}}$. Thus:

$$F_{i,x_{i+1}} \prec F_{i+1,x_{i+1}} \prec B_{i+1,1}. \tag{3}$$

Combining these inequalities, we have:

$$B_{i,1} \prec F_{i,x_{i+1}} \prec B_{i+1,1}. \tag{4}$$

This implies that $B_{i,1}$ starts before $B_{i+1,1}$ which violates the data dependency of backward between S_i and S_{i+1}. Hence, $x_i \geq x_{i+1}$ must hold for all $1 \leq i \leq S$. □

Scheduling Algorithm in § 4.3

We present the detailed algorithm of generating the full pipeline schedule, which employs a discrete-time simulator and a localized operation selection policy described in § 4.3. Algorithm 3 presents our two-stage operator selection policy, which guarantees the slackness on the communication stragglers and minimizes the bubble rate for subsequent steady and cool-down phases. Algorithm 4 then demonstrates the implementation of pipeline simulation, which asks the policy for operator selection and propagates scheduable operators to the upstream and downstream stages.

Algorithm 3 Operator Selection Policy

Require: Current stage i ($0 \leq i < S$), available operator sets $\mathcal{A}_i$ at current step of S_i, required warm-up forward counts x_i.
Ensure: An operator $o \in \{F, B, W\}$ or None if no operators should be executed.

```
function SELECTOP(i, A_i, x_i)
    if A_i = ∅ then
        return None
    ▷ Warm-up phase, do x_i forwards at first.
    if x_i > 0 then
        if F ∈ A_i then
            x_i ← x_i − 1
            return A_i.pop(F)
        else
            return None
    ▷ Execution priority after warm-up: B > F > W.
    Pri ← {B : 3, F : 2, W : 1}
    ▷ Find an available operator with the highest priority.
    return A_i.pop(max_{o∈A_i}(Pri[o.type]))
```

Algorithm 4 Full Pipeline Schedule Generation

Require: Number of stages S, number of microbatches N, per-stage compute times $\{t_i^F\}, \{t_i^B\}, \{t_i^W\}$, inter-PP communication times $\{c_i\}$, warm-up forward counts $\{x_i\}$, time step δ.
Ensure: A pipeline schedule $\mathcal{X}$.

```
function SCHEDULE(S, N, {t_i^F}, {t_i^B}, {t_i^W}, {x_i}, {c_i}, δ)
    X ← [∅] × S                          ▷ Initialize an empty schedule.
    A ← [∅] × S                          ▷ Current available operators.
    A_0 ← [F] × N                        ▷ Initially, F's are available for S_0.
    t ← 0
    while ∃ A ∈ A, A ≠ ∅ do
        for i ∈ [0 ... S − 1] do
            if not i.busy() then
                ▷ Decide the next operator to execute.
                o ← SELECTOP(i, A_i, x_i)
                i.execute(o, duration = t_i^{o.type})
                ▷ Add dependent operators after execution.
                if o.type = F and i ≠ S − 1 then
                    A_{i+1}.append(F(ready = t + c_i))
                else if o.type = B and s ≠ 0 then
                    A_{i−1}.append(B(ready = t + c_{i−1}))
                    A_{i−1}.append(W(ready = t + c_{i−1}))
                X_i ← X_i ∪ {o}                  ▷ Update schedule.
        t ← t + δ
    return X
```