USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

CrossPipe: Towards Optimal Pipeline Schedules for Cross-Datacenter Training

Tiancheng Chen, Ales Kubicek, Langwen Huang, and Torsten Hoefler, *ETH Zurich*

https://www.usenix.org/conference/atc25/presentation/chen-tiancheng

This paper is included in the Proceedings of the
2025 USENIX Annual Technical Conference.

July 7–9, 2025 • Boston, MA, USA

ISBN 978-1-939133-48-9

CrossPipe: Towards Optimal Pipeline Schedules for Cross-Datacenter Training

Tiancheng Chen[1], Ales Kubicek[1], Langwen Huang[1] and Torsten Hoefler[1]

[1]Department of Computer Science, ETH Zürich

Abstract

Training large language models (LLMs) now requires resources that exceed a single datacenter, making cross-datacenter strategies increasingly crucial. We present CrossPipe, a framework designed to optimize model training across geographically distributed datacenters by explicitly modeling and mitigating the impact of network latency and limited bandwidth. It enables unified analysis and optimization incorporating both pipeline parallelism (PP) and opportunities for overlapping data parallelism (DP) communication. CrossPipe generates optimized pipeline schedules using either solver-based optimal or fast near-optimal greedy algorithms, built upon a flexible execution engine that separates scheduling logic from communication details. Our evaluation shows that CrossPipe reduces training time by up to 33.6% compared to traditional pipeline schedules under identical memory constraints. When memory constraints are relaxed, CrossPipe maintains strong performance despite communication delays, approaching the efficiency of idealized schedules without delays. CrossPipe offers improved scalability and resource utilization, particularly in environments with high network latency or limited bandwidth.

1 Introduction

Large language models (LLMs) have revolutionized natural language processing, demonstrating remarkable capabilities in tasks such as text generation, translation, and question answering. These models, trained on massive datasets, exhibit sophisticated context understanding and generate human-like responses. Their applications span scientific research [59], content generation [29], and personal assistants [31]. LLM training consists of three stages: pre-training, fine-tuning, and alignment, with pre-training consuming the majority of computational resources [11, 15, 66].

As LLM performance scales with model size and data volume, computational demands have increased exponentially. Recent studies estimate that training compute for state-of-the-art models quadruples annually [48], necessitating both vertical scaling (faster accelerators) and horizontal scaling (distributed computation). While GPU performance and energy efficiency continue to improve steadily [16], the power and infrastructure required to support LLM training is growing even faster. If current trends hold, GPU counts must nearly triple each year, with power consumption rising accordingly.

The escalating computational demands of LLMs are straining existing infrastructure, particularly power supply systems. To address these energy requirements, companies like Microsoft, Google, and Amazon are turning to nuclear energy sources to power their new AI datacenters [7, 36, 46], emphasizing the need for reliable and high-capacity power sources. Scaling a single datacenter introduces challenges including local power limitations and increased vulnerability to outages. Report suggests that deploying multiple smaller facilities is more practical than scaling a single massive one [5].

As LLM scale grows, multi-datacenter training is becoming essential [1], distributing both compute and energy loads. However, geographic distribution introduces significant communication inefficiencies that must be addressed to support this shift. In the context of cloud-based training, allocating large blocks of GPUs in one region is often infeasible [54], making cross-regional GPU acquisition a practical alternative. The high cross-region communication cost poses challenges to the efficiency of existing training methods. This work attempts to assess the impact of network inefficiencies to synchronous training tasks and improve the performance to reduce the cost and energy consumption.

This paper introduces CrossPipe[1], a framework that improves the efficiency of cross-datacenter (cross-DC) LLM training through the following contributions:

- **Analysis:** We present a comprehensive analysis of cross-DC training methodologies and show that pipeline parallelism is the most feasible approach in this setting.
- **Performance Model and Algorithm:** We present a latency

[1] The code is available at https://github.com/spcl/crosspipe.

and bandwidth-aware performance model specifically designed for the cross-DC environment. This model enables the co-optimization of pipeline schedules with potential data parallelism (DP) communication overlap, unifying the modeling of cross-DC PP and cross-DC DP. Next, we introduce a system-aware pipeline schedule generation algorithm: CrossPipe. The algorithm leverages either constraint optimization techniques to generate *optimal* cross-DC pipeline schedules (Section 4.1) or fast greedy algorithm to generate efficient and *near-optimal* schedules (Section 4.2).

- **Framework:** Finally, we propose and implement a flexible and easily extensible pipeline execution engine featuring a two-layer abstraction that decouples block scheduling from communication arrangement (detailed in Section 6.3). This design enables efficient deployment of different pipeline schedules, including those generated by CrossPipe.

2 Cross-DC Training

2.1 Parallelism Strategies

Distributed LLM pre-training [4] employs a combination of different parallelism strategies (termed *hybrid parallelism*) to partition the workload across GPU clusters. Table 1 lists symbols and notations used in this paper.

Tensor Parallelism (TP) [2]: Splits each model layer across multiple GPUs [27,53], requiring extensive collective communication (e.g., Reduce-Scatter and Allgather [37]) during both forward and backward passes. Due to limited opportunities for overlap [64] and high communication costs, TP is typically restricted to high-bandwidth domains (e.g., NVLink [40]), making it unsuitable for spanning geo-distributed DCs.

Pipeline Parallelism (PP): Divides the model layers into n_{PP} *stages*, with each stage assigned to a different GPU. Communication occurs only at stage boundaries via point-to-point send/receive of activations and gradients.

Data Parallelism (DP): Replicates the full model on each GPU, where distinct batches are processed independently and gradients are synchronized across replicas. DP is usually applied with ZeRO [45] to reduce memory redundancy. This work assumes DP with ZeRO stage 1, partitioning optimizer states without increasing communication overhead compared to vanilla DP. Higher ZeRO stages introduce extra communication with diminishing memory savings.

Sequence Parallelism (SP) [3]: Scales sequence dimension [21,33] and is typically applied at the end of pre-training to increase model context window [12].

Expert Parallelism (EP): Distributes the expert MLPs in Mixture of Experts (MoE) [20, 51] models. The per-layer, high-volume, and dynamic Alltoall communication in EP makes it challenging to deploy on cross-DC links.

Notation	Description
n_{DC}	# of datacenters
$n_{\{TP,PP,DP\}}$	TP, PP, DP size
$T_{\{F,B,W\}}$	Runtime of F, B, W block
$M_{\{F,B,W,L\}}$	Net memory change in F, B, W block and memory budget
M_L	Memory limit per device
α, β	Communication latency and inverse of bandwidth
T_α, T_β	Communication cost matrix
b, $\hat{B}$	Microbatch size and global batch size
n_{mb}, ε	Number of microbatches per DP rank, and ratio n_{mb}/n_{PP}
s, d	Model sequence length and hidden dimension
n_{sub}	Number of parts in a sub-block schedule (Section 4.2)

Table 1: List of symbols and notations.

Key Insight: TP, SP, and EP introduce layer-wise communication with high frequency and/or volume, making them highly sensitive to the latency and limited bandwidth typical of cross-DC links. Therefore, PP and DP emerge as the primary candidates for cross-DC traffic, due to their less frequent (PP, DP) or point-to-point (PP) communication patterns.

2.2 Communication Model

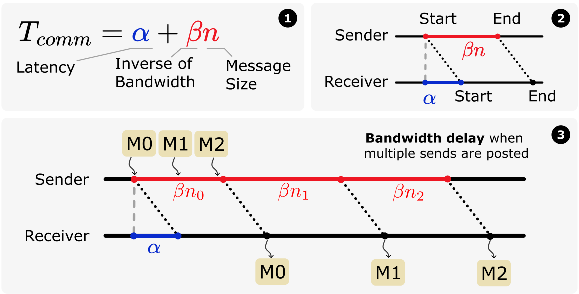


Figure 1: Alpha-Beta communication model ❶ sending a single message ❷. When multiple pending messages are to be sent, the latter messages need to wait for the previous ones to be placed on the transmission link first. This results in an extra bandwidth delay ❸.

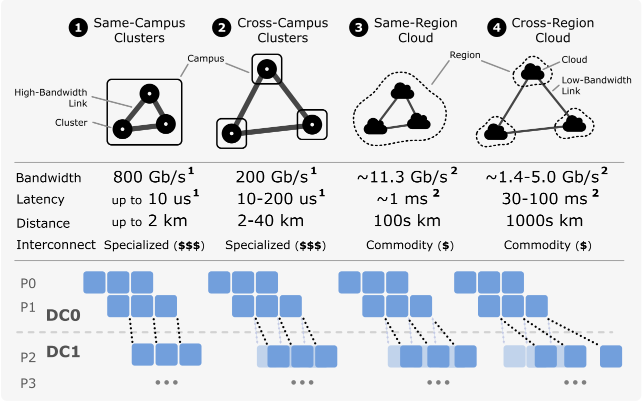
In this work, we assume a small number of DCs (e.g., $n_{DC} \leq 4$). For modeling the communication time T_{comm}, we adopt the Alpha-Beta model, accounting for both latency (α) and bandwidth limitations (β). Multiple concurrent messages incur additional queuing delays, illustrated in Figure 1.

2.3 Distributed Training Infrastructure

Cross-DC infrastructure setups can be categorized into four primary types, as shown in Figure 2. We differentiate between high-performance clusters and public cloud environments, further classified by geographic proximity: either nearby (same-campus, same-region) or distant (cross-campus, cross-region).

[2] Also known as Operator Parallelism.
[3] Also known as Context Parallelism.

- **Same-Campus Clusters** ❶ setup represents tightly interconnected DCs on the same campus, typically connected via frontend network or additional switch layer. This setup features low latency (up to 10 us [4]) and high bandwidth (800 Gb/s [4] per port), making communication overhead negligible, but still slightly higher than a single cluster.
- **Cross-Campus Clusters** ❷ setup represents geographically distributed DCs (typically up to 40 km apart, using public products like NVIDIA MetroX) interconnected with high-bandwidth links (200 Gb/s [4] per port). Network latency is bounded by the physical distance (10-200 us [4]).
- **Same-Region Cloud** ❸ setup represents closely allocated instances within the same cloud region. This setup features low bandwidth (around 11.3 Gb/s [5]) [55] and higher latency (around 1 ms [5]) [55] compared to the setups discussed above due to the usage of less specialized networking hardware.
- **Cross-Region Cloud** ❹ setup represents instances allocated across cloud regions or even continents. This setup inherits the same properties as a same-region cloud but with significantly higher latency (30-100 ms [5]) [55] and even lower bandwidth (1.4-5.0 Gb/s [5]) [55] due to distance.



	❶ Same-Campus Clusters	❷ Cross-Campus Clusters	❸ Same-Region Cloud	❹ Cross-Region Cloud
Bandwidth	800 Gb/s ^{1}	200 Gb/s ^{1}	~11.3 Gb/s ^{2}	~1.4-5.0 Gb/s ^{2}
Latency	up to 10 us ^{1}	10-200 us ^{1}	~1 ms ^{2}	30-100 ms ^{2}
Distance	up to 2 km	2-40 km	100s km	1000s km
Interconnect	Specialized ($$$)	Specialized ($$$)	Commodity ($)	Commodity ($)

Figure 2: Cross-DC infrastructure setup types and their impact on the PP communication at DC boundaries.

Table 2: The hidden dimension d, number of parameters N of LLMs. D and E refer to dense and MoE models respectively.

Model	d	N (10^9)	N/d (10^6)
Mistral 7B (D) [23]	4096	7.24	1.77
Mixtral 8x7B (E) [24]	4096	46.7	11.4
Qwen2.5 32B (D) [44]	5120	32.8	6.41
DeepSeek V3 (E) [9]	7168	685	95.6
Llama 3 405B (D) [12]	16384	406	24.8

These infrastructure variations drastically affect communication characteristics. To analyze their impact independent of specific hardware or model configurations, we normalize the communication time components (T_{lat} and T_{bw}) by the max per-microbatch forward computation time per stage (T_F), see Section 3.3). This yields dimensionless ratios, T_{lat}/T_F and T_{bw}/T_F, which capture the relative cost of communication.

2.4 Cross-DC Parallel Dimension Selection

Hybrid parallelism exhibits structured communication patterns, with over 99% of GPU pairs having no direct traffic [65]. The choice of parallelism strategy across DCs significantly influences training efficiency. Figure 3 compares two viable options identified in Section 2.1: cross-DC PP and cross-DC DP. Cross-DC PP communication volume is characterized by $sd * n_{DP}$ while cross-DC DP communication volume is characterized by model parameters N. The key hyperparameters of some LLMs are shown in Table 2. We analyzed both cross-DC PP and cross-DC DP for Llama 3 405B in Section 5.2, demonstrating that cross-DC PP is generally the better choice. The increased popularity of MoE models further shifts the preference towards cross-DC PP [6], since experts in MoE models introduce extra DP communication volume compared to dense models with similar width d.

3 Pipeline Model

3.1 Computation Blocks

PP partitions model chunks across n_{PP} stages, processing input microbatches in sequence. During the backward pass, gradients propagate in reverse, from the last model chunk back to the initial one. Periods when devices remain idle while awaiting required data are termed pipeline bubbles. To evaluate pipeline efficiency, we define the bubble ratio as the fraction of idle time over total time per device.

We denote the Forward computation for each chunk as the F block, and the Backward computation as the B block. Each B block can be further split into an input data gradient computation block (DGrad, or D) and a weight gradient computation block (WGrad, or W) [43]. An illustration of this decomposition is provided in Appendix B. This finer granularity facilitates the construction of more efficient pipeline schedules with reduced bubble ratios.

3.2 System Parameters

Although our focus is on training Transformer-based [61] LLMs in cross-DC environments, our approach applies to large models structured as a sequence of layers. Relevant

[4] Calculation based on the best commercially available hardware (NVIDIA LinkX / MetroX), assuming 5 ns/m.

[5] Measured.

[6] Llama 3 405B is trained with $s = 8192$, $n_{DP} = 128$, yielding $N/sdn_{DP} \approx 23.6$. DeepSeek-V3 is trained with $s = 4096$, $n_{DP} = 128$, yielding $N/sdn_{DP} \approx 182$. Notice that Llama 3 405B is wider in d and therefore can be viewed as the basis of a MoE model much larger than DeepSeek-V3.

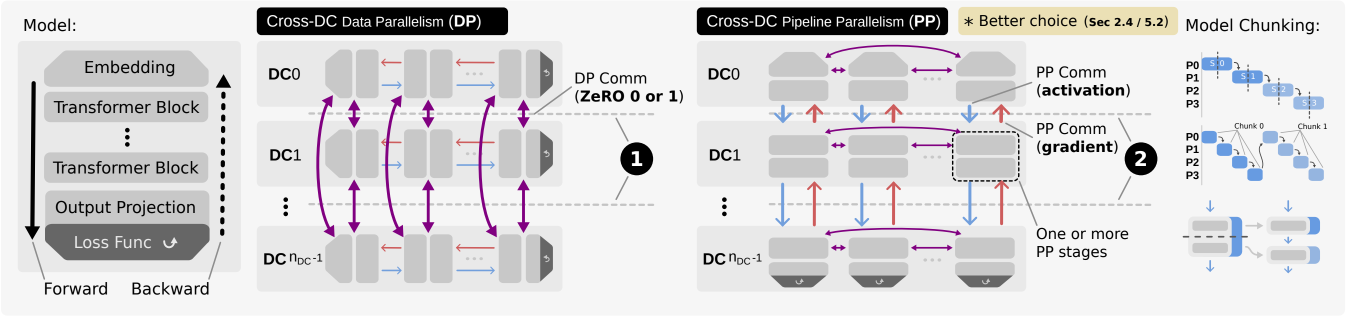


Figure 3: Typical LLM architecture (left). **Cross-DC DP:** Each DC maintains independent model copies. ❶ Collective operations (Allgather and Reduce-Scatter, or Allreduce) synchronize gradients and update parameters (ZeRO stage 0 or 1). **Cross-DC PP:** The model is partitioned among DCs at layer boundaries. Each DC holds one or more pipeline *stages*. DP communication happens internally within each DC to synchronize gradients of stages each holds. ❷ Inter-DC communication employs point-to-point send/receive operations for exchanging activations and gradients. Stages can further split into finer *chunks* to enhance scheduling efficiency (Section 3.3).

system parameters include memory usage, per-chunk computation time, and inter-stage communication latency and bandwidth delays. Memory consumption encompasses static elements (parameters, gradients, optimizer states) and dynamic allocations (activations cached during F blocks and released after D and W).

3.3 Pipeline Schedules

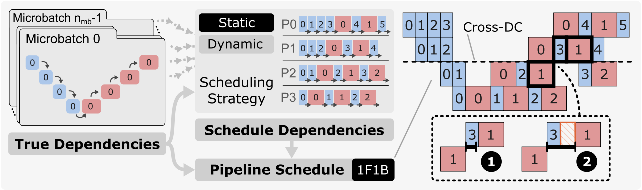


Figure 4: Construction of a 1F1B static schedule. True dependencies guide the creation of schedule dependencies. The resulting acyclic dependency graph governs execution and is used for runtime estimation (Section 3.5). Highlighted parts (right) show the timing of activation/gradient arrival either ❶ enables immediate scheduling or ❷ causes delays (bubbles).

Pipeline schedules are represented as acyclic dependency graphs, with vertices as pipeline blocks and edges representing two types of dependencies: true dependencies (data dependencies within each microbatch) and schedule dependencies (execution order within each pipeline stage). The construction of a 1F1B schedule is illustrated in Figure 4.

True dependencies reflect actual data flow across blocks (forward for activations, backward for gradients). Figure 5 shows key traversal (data flow) patterns: Unidirectional (UD), Bidirectional (BD), Loop, and Wave.

Schedule dependencies define the execution order of pipeline blocks within each stage. These dependencies are determined using either a static or dynamic scheduling strategy. Static strategies predetermine block placements (usually hand-optimized), while dynamic strategies adapt based on system parameters. CrossPipe schedules (Section 4) fall under dynamic scheduling.

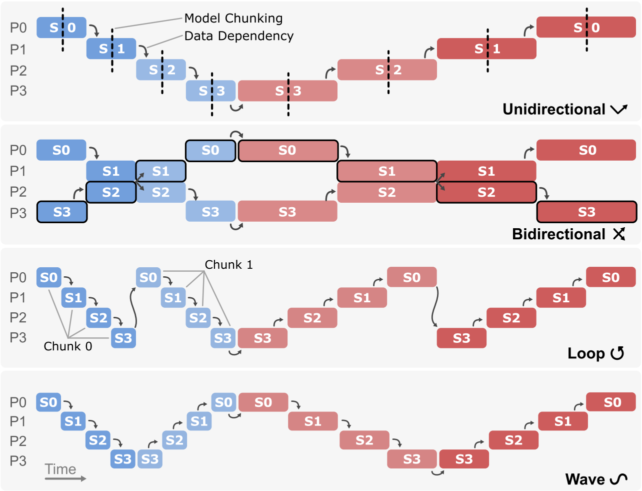


Figure 5: Traversal patterns for a single microbatch (two microbatches shown in the Bidirectional pattern). Loop and Wave patterns leverage model chunking to refine granularity.

3.4 Problems of Static Scheduling

In the presence of communication delays, we identify two major limitations that reduce the efficiency of static schedules in cross-DC PP: static execution order (scheduling-level) and static communication arrangement (implementation-level).

Static Execution Order: Static schedules are optimized under the assumption of negligible communication cost, as in single-DC settings. When directly applied to cross-DC training, they fail to adapt to higher communication delays, resulting in pipeline inefficiencies visualized as *bubble strides* (illustrated in ❷ Figure 6 and Appendix C). **Schedule** ❷ in

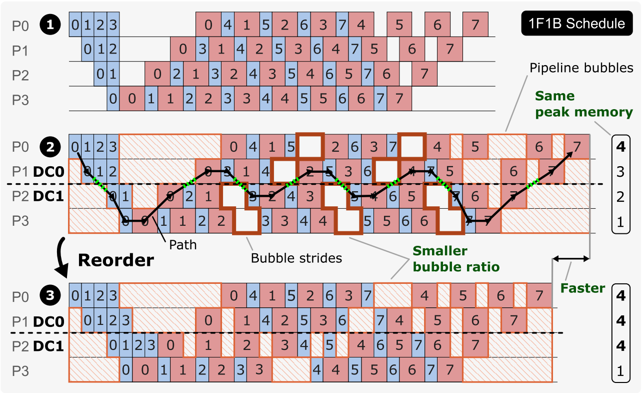


Figure 6: ❶ Original 1F1B schedule. ❷ 1F1B schedule with Cross-DC PP communication which leads to *bubble strides*. A *Path* is depicted (→) including cross-DC boundary crossings (•••). ❸ The schedule after reordering is more efficient while maintaining the same peak memory. More microbatches or memory budget can help to further reduce runtime.

Figure 6 depicts a critical path in a 1F1B schedule across 2 DCs involving 8 cross-DC PP communications. Since the path consists solely of true and schedule dependencies (Section 3.3), its length imposes a lower bound on overall runtime. For a 1F1B schedule of n_{mb} microbatches, there exists a path containing $O(n_{mb})$ cross-DC communications. As a result, communication delays are amplified proportionally, significantly degrading training throughput. A detailed analysis of this amplification effect is presented in Section 5. CrossPipe addresses this limitation via dynamic scheduling strategies, detailed in Section 4. **Schedule ❸** in Figure 6 illustrates that reordering pipeline blocks can improve efficiency while maintaining the same peak activation memory if needed. **Static Communication Arrangement**: Existing frameworks such as Megatron-LM often group pipeline communication operations (e.g., GPU 0 sending to GPU 1 while receiving from GPU 1) for simplicity and hardware efficiency, which introduces implicit synchronization. Moreover, even if this grouping is avoided, the two-sided communication pattern introduces synchronization between the sender and receiver in each send/recv operation. Due to variations in stage execution time, the receiver may fail to post the corresponding receive in time, causing the sender to wait. These delays disrupt stage alignment, which many hand-optimized schedules assume, and propagate bubbles across the pipeline. The interleaved 1F1B schedule [39] overlaps communication with one computation block to mitigate this synchronization cost. However, its static design is only effective under small delays. To address this, CrossPipe decouples scheduling logic from communication orchestration, allowing more fine-grained and adaptive execution, as elaborated in Section 6.3.

3.5 Pipeline Performance Model

We develop a performance model to estimate the runtime of a pipeline by leveraging the topological ordering of its dependency graph (Section 3.3). This model assumes that scheduling and communication orchestration are decoupled, thereby excluding delays caused by synchronization overhead (see Section 3.4). The start time of each block is determined by the maximum of two values: (1) the completion time of the preceding block on the same stage (❶ in Figure 4), and (2) the completion time of the dependent block plus the communication delay (❷ in Figure 4). The communication delay consists of a fixed latency component and a bandwidth-related component, which depends on both link bandwidth and current occupancy (Section 2.2).

4 CrossPipe Schedules

4.1 Optimal Schedule

The optimal pipeline schedule depends on the system parameters and can be framed as a job scheduling problem. Prior work [43] formulates this using mixed integer linear programming. Building on this, we elevate communication operations to first-class citizens alongside computation, incorporating both latency and bandwidth delays into the formulation, and generalize this to traversal patterns (Figure 5). This leads us to define the problem as a constraint optimization (CO) task. In addition to yielding start and end times for all operations, the solution inherently determines the execution order of communication operations that share the same cross-DC link, thereby handling link contention and scheduling order of cross-DC communications implicitly.

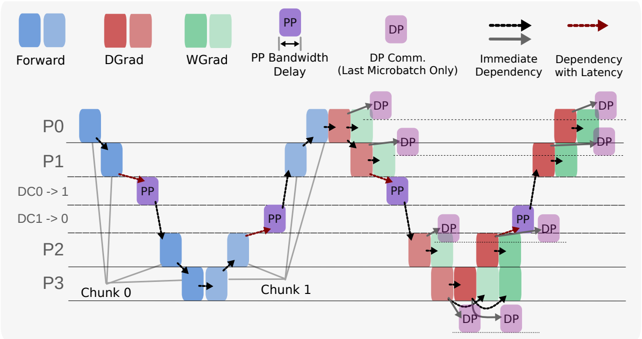


Figure 7: The data dependency in a Wave schedule with modeling of both computation and communication. For each model chunk, the DP communication only depends on the W block of last microbatch. The modeling of vanilla DP is shown in the figure. For DP with ZeRO stage 1, an Allgather block should precede the F block of the first microbatch in each model chunk.

Sets and Indices Every *compute* operation is uniquely identified by the triple $(s,k,t,m) \in \mathcal{S} \times \mathcal{K} \times \mathcal{T} \times \mathcal{M}$. Each communication operation $c \in \mathcal{C}$ transfers the data produced by a compute operation on stage $\mathrm{src}(c)$ to stage $\mathrm{dst}(c)$.

- $\mathcal{M}$: Microbatches, indexed by $mb = 1, \ldots, n_{mb}$
- $\mathcal{S}$: Devices (pipeline stages), indexed by $s = 1, \ldots, n_{PP}$
- $\mathcal{T}$: Operation types, $\mathcal{T} = \{F, D, W\}$
- $\mathcal{K}$: Model chunks
- $\mathcal{P}$: Compute operations (indexed by (s,k,t,m))
- $\mathcal{C}$: Communication operations
- $\mathcal{O} = \mathcal{P} \cup \mathcal{C}$: All operations

Inputs The input variables are closely related to system parameters in Section 3.2.

- d_o: Duration of $o \in \mathcal{O}$ (bandwidth time for $c \in \mathcal{C}$, computation time otherwise)
- ℓ_c: Latency delay for $c \in \mathcal{C}$
- m_p: Net memory change after compute $p \in \mathcal{P}$ completes
- $M_s^{\max}$: Memory limit of device s
- $\mathrm{Pred}(o)$: Immediate predecessor of o in data dependency

Decision Variables

- $t_o \in \mathbb{R}_{\geq 0}$: Start time of operation $o \in \mathcal{O}$
- $x_{o,o'} \in \{0,1\}$: Order for operations sharing a device/link

Constraints

- **Data Dependencies** An example of data dependency of Wave schedules is shown in Figure 7.

$$\forall o \in \mathcal{O},\ \forall p \in \mathrm{Pred}(o): \quad t_o \geq t_p + d_p + \begin{cases} \ell_p & \text{if } p \in \mathcal{C} \\ 0 & \text{if } p \in \mathcal{P} \end{cases}$$

- **Resource Non-overlap** For any $o, o' \in \mathcal{O}$ sharing a device or link, overlapping is not allowed:

$$t_o + d_o \leq t_{o'} + H(1 - x_{o,o'})$$
$$t_{o'} + d_{o'} \leq t_o + H x_{o,o'}$$

  H is a large constant bounding the scheduling horizon.

- **On-device Memory Capacity** Let $u_{p,q} = 1$ iff compute p completes before q starts. Then for all $s \in \mathcal{S}$ and $q \in \mathcal{P}$ assigned to s:

$$\sum_{\substack{p \in \mathcal{P} \\ \mathrm{device}(p) = s}} m_p\, u_{p,q} \leq M_s^{\max}$$

- **Microbatch Order within Stage and Type** For any $o, o' \in \mathcal{P}$ on same device s and type $t \in \mathcal{T}$, if microbatch index $mb(o) < mb(o')$, then:

$$t_o + d_o \leq t_{o'}$$

  This constraint reduces the search space.

Objective We minimize the makespan, defined as the time from the earliest start to the latest finish on the first device:

$$\min\left(t_{\mathrm{last}(0)} + d_{\mathrm{last}(0)} - t_{\mathrm{first}(0)}\right)$$

DP Overlap Modeling DP communication can be modeled as distinct operations triggered after the completion of the W block for the final microbatch of the current model chunk. In the case of ZeRO stage 1, the corresponding weight All-gather operations are scheduled before the first F block of each model chunk. An illustrative example is shown in Figure 7. The objective is then extended to account for the added communication.

Solver Scalability We evaluate the runtime performance of both MILP and CO solvers on identical pipeline scheduling problems, as detailed in Appendix D.1. It demonstrates the feasibility of using solver-based methods in production environments. However, the search space expands rapidly with the number of pipeline stages, which can lead to prohibitive computation time for extremely large-scale scenarios or dynamically changing system parameters. In such cases, solver-based scheduling becomes less practical, motivating the need for alternative approaches (Section 4.2).

4.2 Greedy Schedule

To address the scalability limitations of solver-based approaches, we introduce a greedy schedule generation algorithm. This method is designed to rapidly generate near-optimal pipeline schedules while adapting to potentially dynamic system conditions.

4.2.1 Greedy Sub-block Scheduling

The greedy algorithm operates using *local* information, only considering scheduled blocks and those ready for scheduling. To counter the suboptimal decisions typical of greedy methods, we employ block-splitting: each computation block is divided into n_{sub} sub-blocks. This finer granularity enhances the ability to reduce pipeline bubbles with negligible scheduling overhead compared to training iteration time.

Algorithm Inputs The inputs to the greedy algorithm include: Per-stage runtimes of F, D, and W blocks: T_F, T_D, T_W; corresponding memory usage: M_F, M_D, and M_W; per-stage memory limit: M_L; communication delay matrices: α and β, where $\alpha[i,j]$ and $\beta[i,j]$ represent latency and bandwidth inverse of communication from device i to device j. The scheduling procedure for the *CrossUDSub* schedule is outlined in Algorithm 1.

4.2.2 Scheduling Loop

The scheduling loop consists of three core steps:

Stage Selection The `next_stage_to_schedule` method identifies the stage with the earliest schedulable time, which is defined as the maximum between the end time of the last scheduled operation and the earliest available time of schedulable operations.

Algorithm 1 Greedy Generation for *CrossUDSub* Schedule

```
Output: Per-stage schedule $S_d, \forall d \in [n_{PP}]$
for $i$ in $[n_{mb}]$ do
    Add $F_i$ operation for stage 0 to $S_0$'s schedulable operations.
end for
while True do                                        Scheduling Loop (Sec 4.2.2)
    $cur \leftarrow$ next_stage_to_schedule()
    if no stage is schedulable then
        break
    end if
    $p_{cur} \leftarrow$ schedulable operation of highest priority on stage $cur$
    Schedule next sub-block of $p_{cur}$
    if $p_{cur}.type$ = D and no remaining sub-blocks of $p_{cur}$ then
        Add W operation to schedulable operations of current stage
    end if
    Let $p_{next}$ be the operation dependent on $p_{cur}$
    if $p_{next}$ exists and no remaining sub-blocks of $p_{cur}$ then
        $T_{lat} \leftarrow \alpha[cur, next]$
        $T_{bw} \leftarrow \beta[cur, next] * \text{Msg\_Size}$
        $E_{bw} \leftarrow$ bw_model($p_{cur}.T_{end}, cur, next, T_{bw}$)
        $p_{next}.T_{avail} \leftarrow E_{bw} + T_{lat}$
        Add $p_{next}$ to $S_{next}$'s schedulable operations.
    end if
end while
```

Operation Selection The scheduler selects operations available at or after the end of the last scheduled operation on the chosen stage. When multiple options exist, it applies a heuristic priority across three phases:

- **Warm-up phase**: prioritizes F blocks
- **Steady phase**: interleaves F and D full blocks
- **Tear-down phase**: prioritizes D over W blocks

When memory constraints prevent scheduling F or D blocks, a W sub-block is scheduled.

Operation Scheduling The selected operation is scheduled on its stage. If it is the last sub-block of a D block, the corresponding W is added to the schedulable operations of the stage. The dependent blocks are then made schedulable on the receiving stage with the earliest start time calculated using the communication model.

4.2.3 Bandwidth Occupancy Model

To model bandwidth contention, we use a simple range-based bandwidth occupancy model. Communication is assumed to begin immediately upon completion of the relevant computation block. The BW_model $(T_{ready}, src, dst, T_{bw})$ function determines the earliest available transmission window of length T_{bw}, starting at or after T_{ready}, and returns its end time.

4.2.4 Performance Characteristics

While greedy algorithms do not guarantee global optimality, our approach demonstrates strong empirical performance. As detailed in Section 5&7, the CrossUDSub schedule achieves:

- Equivalent performance to ZB-H1 [43] under negligible communication delays and same memory constraints.
- Faster than static schedules under non-negligible communication delays by filling sub-block size bubbles.
- Further improvements when memory constraints are relaxed, allowing greater scheduling flexibility.

4.2.5 Time Complexity

The main loop executes $3n_{mb}n_{sub}n_{PP}$ iterations, with each iteration scheduling one sub-block. Identifying the next stage and highest priority operation incurs $O(n_{mb}n_{sub}\log(n_{mb}n_{sub}))$ cost. Hence, the overall complexity is $O(n_{mb}^2 n_{sub}^2 n_{PP}\log(n_{mb}n_{sub}))$. In practice, since the number of schedulable operations per stage remains small, the runtime approximates $O(c \cdot n_{mb}n_{sub}n_{PP})$ for a small constant c.

5 Analysis

In this section, we use simulation experiments based on the performance model described in Section 3.5 to investigate two key questions: (1) How do different pipeline schedules respond to latency and bandwidth delays? (2) Between cross-DC PP and cross-DC DP, which is more efficient for training in cross-DC settings?

5.1 Schedule Efficiency

Schedule	Type	WGrad	Bubble Ratio	Memory	DP Overlap
1F1B [38]	UD	Combined	High	Medium	Medium
IV1F1B [39]	Loop	Combined	Medium	Medium+	Medium+
ZBH1 [43]	UD	Split	Medium	Medium	Low
ZBV [43]	Wave	Split	Low	Medium	Low

Table 3: Static pipeline schedules used in the analysis. UD and BD stands for unidirectional and bidirectional. IV1F1B is the abbreviation for the interleaved 1F1B schedule.

We compare various pipeline schedules under increasing communication delay in a cross-DC PP setting. Bidirectional (BD) schedules are excluded as they involve both PP and DP cross-DC communication. The main static schedules that we focus on are summarized in Table 8. We use $n_{PP} = 4, n_{mb} = 8$, and simulate 2 DCs with 2 stages each. Dynamic schedules are generated with the same memory limits as their static counterparts (e.g., CrossUD mirrors 1F1B). Delay sensitivity is measured as slowdown relative to the ZBV schedule under no communication delay. Delay is varied using T_{lat}/T_F (latency delay) and T_{bw}/T_F (bandwidth delay), where T_F is the per-stage forward computation time. Key observations from Figure 8 include:

- WGrad-split schedules consistently outperform unified-backward ones due to finer scheduling granularity.
- Wave schedules are more efficient in low-delay settings, while UD schedules become superior as delays grow.
- Loop schedules show the highest sensitivity to delays, due to more frequent cross-DC communication (6 per micro-batch, compared to 4 for Wave and 2 for UD).

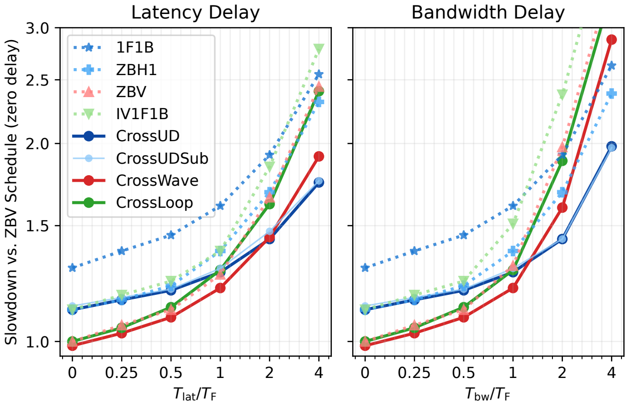


Figure 8: Impact of latency and bandwidth delay on runtime across different pipeline schedules. Static (····) and dynamic (—) schedules are compared. Setup: 4 stages, 8 microbatches (2 DCs, 2 stages per DC). *Cross*-prefixed schedules are generated by the CO solver (Section 4.1). Slowdown is measured relative to the ZBV schedule at zero delay.

- The greedy CrossUDSub schedule matches the solver-based CrossUD in most delay regimes, highlighting its efficacy as a lightweight alternative.
- When delays are small, latency and bandwidth contribute equally to runtime. However, once bandwidth delay exceeds the forward time per chunk, it induces additional pipeline bubbles from queuing (Section 2.2).

5.2 Cross-DC PP vs. Cross-DC DP

We simulate iteration times for cross-DC PP and cross-DC DP approaches using the Llama 3 405B model [12] under various latency and bandwidth conditions (detailed in Appendix E).

Results in Figure 9 show that latency (ranging from 4–128 ms) has little impact on runtime in this scenario, as the per-stage forward time ($T_F \approx 109$ms) keeps the delay ratio low. However, bandwidth significantly affects performance. Cross-DC PP outperforms cross-DC DP by up to 3.05x when the cross-DC link bandwidth is limited to 4 GB/s. This gap narrows as bandwidth increases, becoming negligible beyond 1024 GB/s. Compared to the ideal single-DC case, cross-DC PP sees only a 1.3× slowdown at 64 GB/s. These results suggest that for large models with long per-stage computation time (T_F), bandwidth is the primary bottleneck in cross-DC communication. Under such conditions, cross-DC PP offers superior efficiency relative to DP, particularly when network resources are constrained.

6 CrossPipe Implementation

Schedules generated by CrossPipe can adapt to configuration changes, including PP size, hybrid parallelism setups, and system parameters. In contrast, static PP modules in existing frameworks support only a limited, hard-coded range of schedules, making them difficult to adapt to and extend. Our implementation addresses these limitations through the CrossPipe module, which integrates seamlessly with the existing training framework. We use Megatron-LM as our base framework. The CrossPipe module is primarily implemented in Python, with components in C++ to enable latency and bandwidth injection (Section 6.4) for emulating cross-DC network conditions on a homogeneous cluster.

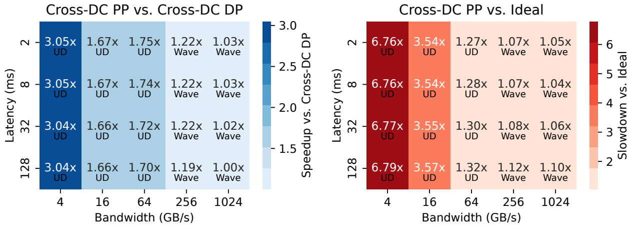


Figure 9: Simulation results comparing cross-DC PP and DP for Llama 3 405B training across two DCs. Left: Speedup of cross-DC PP over cross-DC DP. Right: Slowdown of cross-DC PP compared to an ideal single-DC setup. Labels indicate exact values and optimal schedule types per configuration.

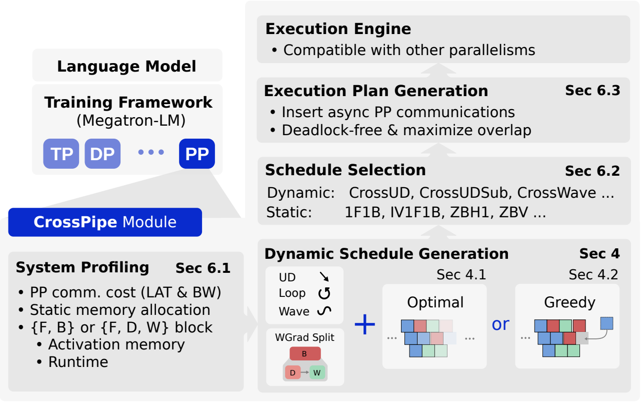

Language Model
Training Framework
(Megatron-LM)
TP
DP
•••
PP
CrossPipe Module
System Profiling
Sec 6.1
• PP comm. cost (LAT & BW)
• Static memory allocation
• {F, B} or {F, D, W} block
• Activation memory
• Runtime
Execution Engine
• Compatible with other parallelisms
Execution Plan Generation
Sec 6.3
• Insert async PP communications
• Deadlock-free & maximize overlap
Schedule Selection
Sec 6.2
Dynamic: CrossUD, CrossUDSub, CrossWave ...
Static: 1F1B, IV1F1B, ZBH1, ZBV ...
Dynamic Schedule Generation
Sec 4
Sec 4.1
Sec 4.2
UD
Loop
Wave
WGrad Split
B
D
W
+
Optimal
or
Greedy

Figure 10: Components of the CrossPipe module.

An overview of our implementation is shown in Figure 10 . The module begins by collecting system parameters via lightweight benchmarks (Section 6.1). It then generates dynamic pipeline schedules (defining the order and timing of computation blocks) using either the constraint optimization solver (Section 4.1) or the greedy algorithm (Section 4.2). A schedule with the best simulation performance is selected (Section 6.2). Next, CrossPipe lowers it to a concrete execution plan by inserting and optimizing communication operations (Section 6.3). This decouples high-level scheduling logic from low-level execution, enabling dynamic, fine-grained control. The selected schedule can also be hot-swapped during training if better options are found.

6.1 System Profiling

CrossPipe collects critical metrics in a single iteration using lightweight profiling. These include runtime and memory usage of F, D, and W blocks, as well as communication delay parameters (α, β). We follow the model partitioning strategy of Llama 3 [12], treating embedding and output layers as transformer layers to ensure load balance across stages.

6.2 Schedule Selection

CrossPipe selects the schedule with the best estimated performance and supports hot-switching to adapt to changes during training.

- **Static schedules** are well-suited for single-DC training.
- **Dynamic schedules** are more suitable in cross-DC settings with high or varying communication costs. They also adapt better to available memory. Under rich memory budgets, dynamic schedules may increase the number of in-flight F blocks to improve efficiency.

6.3 Execution Plan

In this step, CrossPipe converts the selected pipeline schedule into an execution plan by inserting non-blocking communication operations. This plan is executed by the CrossPipe engine integrated into the training framework.

Communication Orchestration We use NCCL as the communication backend to leverage high-bandwidth intra-node interconnects and reduce inter-node data movement overheads. To decouple point-to-point communications in PP, we dedicate four GPU streams for each direction and role ({Send, Recv} × {Next, Prev}), avoiding interference and deadlocks. In both directions, Recv operations are reordered to align with the corresponding Sends, avoiding NCCL deadlocks. NCCL implements a rendezvous protocol for point-to-point communication, requiring both the sender and the receiver to synchronize before the transfer begins. To maximize communication overlap, we post Recv operations ahead of their corresponding Sends based on profiling estimates. This delay-aware arrangement improves overlap and is applied to both static and dynamic schedules (evaluated in Section 7).

6.4 Latency and Bandwidth Injection

We extend the PyTorch `ProcessGroupNCCL` C++ backend to inject latency and bandwidth delays in specific Send/Recv operations. This allows us to emulate various cross-DC network conditions (as described in Section 2.3) within a single cluster. Latency is injected on the receiver side of cross-DC communication, while bandwidth is throttled by running spinning kernels on the communication streams of both sender and receiver. Implementation details are provided in Appendix F.1, and validation results in Appendix F.2.

Name	Hidden Dim.	Int. Dim.	Att. Heads	KV Heads	Layers
M8	4096	14336	32	8	30+2
M70	8192	28672	64	8	62+2

Table 4: Hyperparameters of models used in the evaluation. Example: M70 is a model with Transformers layers of the same size as the ones in the Llama 3 70B model. The number of layers is reported as number of transformer layers + embedding & output layers.

7 Evaluation

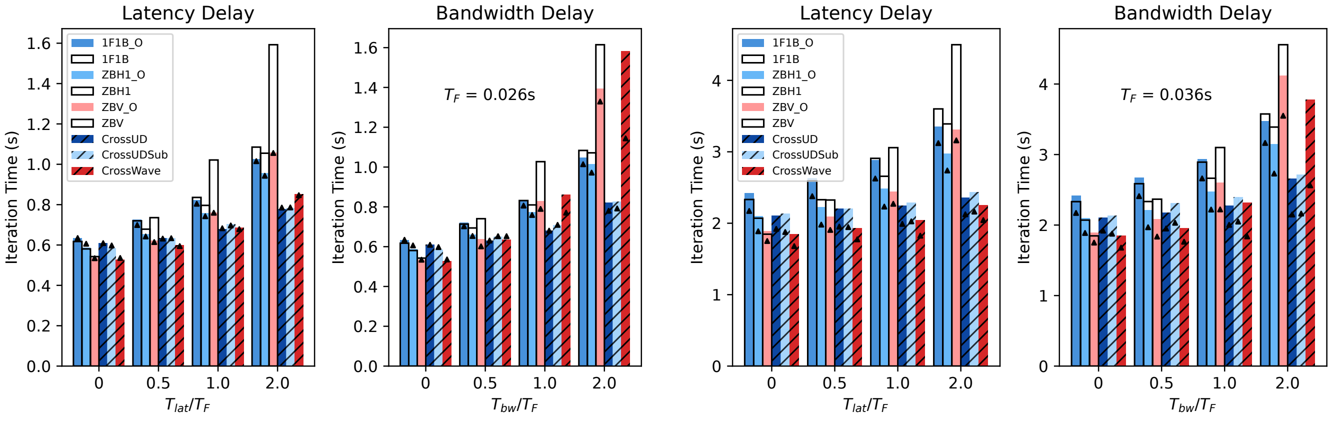
We conducted comprehensive evaluations on the Alps supercomputer to validate CrossPipe's performance and scalability. Each compute node is equipped with four GH200 Grace Hopper Superchips [41]. Each GH200 features 96 GB HBM3 memory integrated with the Hopper GPU die and 120 GB LPDDRX5 memory connected to the Grace CPU. The chips utilize a fully-connected topology with six NVLink 4.0 links between each GH200 pair, providing 200 Gb/s bandwidth per link per direction. Network connectivity is provided by HPE Cray Cassini-1 200 Gb/s NICs in a Dragonfly [25] topology using HPE Cray Slingshot-11 [8, 49] interconnect.

We use LLMs built up from the Llama-style Transformer layers, the hyperparameters of which are listed in Table 4. Each model configuration follows the naming convention `M{model_size}`, where the model_size indicates the size of Transformer layers it contains. For example, M70 uses the Transformer block of the same size as the one in the Llama 3 70B model. The models are then constructed by replicating and stacking identical Transformer layers, along with the vocabulary embedding layer and the output layer. We evaluate the following schedules: 1F1B, IV1F1B, ZBH1, ZBV, CrossUD, CrossUDSub, and CrossWave. The reason to exclude bidirectional (BD) and Loop schedules is explained in Section 5. By default, we set the microbatch size $b = 1$ and sequence length $s = 4096$. In these configurations, the message size for pipeline communication (activations/gradients) is approximately $32n_{DP}$ MB for M8 and $64n_{DP}$ MB for M70. FlashAttention [50] is enabled for higher throughput and less peak memory consumption. For each schedule, we measure 64 iterations and report the minimum value to minimize the network noise effects [17].

7.1 Impact of Latency and Bandwidth Delay

We evaluate the performance of schedules on the cluster with various emulated latency delay T_{lat} or bandwidth delay T_{bw} for each PP communication crossing the DC boundary, using the injection mechanism from Section 6.4. We conduct the experiments on both M8 and M70 models in a two-DC setting. The parallelism configurations are:

- **M8:** $n_{TP} = 2$, $n_{PP} = 4$, $n_{DP} = 1$, $GBS = 2n_{PP}n_{DP} = 8$.
- **M70:** $n_{TP} = 4$, $n_{PP} = 8$, $n_{DP} = 1$, $GBS = 2n_{PP}n_{DP} = 16$.



(a) Model M8

(b) Model M70

Figure 11: Evaluation of static and CrossPipe schedules under various emulated latency and bandwidth delay ratios. The runs of static schedules with communication arrangement optimization (Section 6.3) is marked with the suffix _O. The runtime prediction from the performance model (Section 3.5) is represented as (▲). Schedules are compared using the iteration time in seconds (lower is better). Dynamic schedules (Cross*) use the same peak memory budget as their corresponding static base schedule (e.g., CrossUD matches 1F1B).

We vary delay ratios $T_{lat}/T_F, T_{bw}/T_F \in \{0.0, 0.5, 1.0, 2.0\}$. T_F is defined as the maximum runtime of the per-microbatch forward computation among stages. The results are shown in Figure 11. For the static schedules, we vary two settings: with or without the delay-aware communication orchestration (Section 6.3). Runs with this optimization are marked with the suffix _O. This optimization reduces the impact of delayed receivers, aligning closely with the assumptions of our performance model (Section 3.5). This model accurately predicts the runtime of tested schedules in most configurations. Overall, the CrossPipe schedules show superior performance compared to static schedules, with a reduction in runtime of up to 33.6% (vs. original) or 21.9% (vs. optimized), achieved by the M70 model at $T_{bw}/T_F = 2$.

To ground these delay ratios in realistic scenarios, we take the M70 model as an example. We assume a practical DP size of $n_{DP} = 16$, with intra-DC DP communication fully overlapped. Under this setting, the message size per PP communication is calculated by $bsdn_{DP} * 2 = 1$ GB. Given a forward time of $T_F = 0.038$ s (Figure 11.b), the resulting injected latency delays range from 19 ms to 76 ms, and simulated bandwidth from 105 Gbps to 421 Gbps.

7.2 Further Bubble Reduction

Section 7.1 shows that block reordering helps to reduce PP runtime when the memory budget and global batch size (GBS) are strictly constrained. However, substantial bubble ratios persist under high communication costs. This observation necessitates examination of trade-offs among runtime, GBS and memory budget. Also, we take layer-wise activation recomputation [6] into consideration. Since the GBS varies among the settings, we use runtime per microbatch to compare schedules. We conduct the experiments on the M70 model in a 2-DC setting, with $n_{TP} = 4, n_{PP} = 8, n_{DP} = 1$, under delay combinations: $(T_{lat}/T_F, T_{bw}/T_F) \in \{(0,0), (0.25,2), (2,0.25), (2,2)\}$, across three configurations:

- **Case 1:** $GBS = 2n_{PP}n_{DP} = 16$, activation memory budget 1.0× (same as 1F1B), no recomputation.
- **Case 2:** $GBS = 32$, activation memory budget 1.0×, layer-wise recomputation.
- **Case 3:** $GBS = 32$, activation memory budget 2.0×, no recomputation.

The result is shown in Table 5. Static schedules cannot leverage extra memory budget to further reduce pipeline bubbles ('-' in case 3). Without delay, where $(T_{lat}/T_F, T_{bw}/T_F) = (0,0)$, the efficiency of the CrossPipe schedules is comparable to manually optimized static schedules. Increasing GBS amortizes bubbles in the warm-up and tear-down phases by extending the length of the steady phase which contains fewer bubbles. Under high latency (e.g.,$(2,0.25)$), increasing GBS and memory budget helps the CrossWave schedule to achieve pipeline efficiency (0.115 s per microbatch), matching the no delay case (0.118 s per microbatch). When the bandwidth delay dominates (e.g., $(2,0.25)$ or $(2,2)$), increasing both GBS and memory budget improves schedule efficiency by up to 1.33× (0.196 s to 0.147 s per microbatch for CrossUD). In general, the bandwidth delay is harder to mitigate than the latency delay under the same settings of GBS and memory budget. Layer-wise recomputation generally does not improve the runtime of dynamic schedules, as the recomputation during the backward pass negates its low memory benefits.

$\frac{T_{lat}}{T_F}$	$\frac{T_{bw}}{T_F}$	Case	Static			Dynamic (This Work)		
			1F1B	ZBH1	ZBV	UDSub	UD	Wave
0	0	1	0.151	0.133	**0.118**	0.137	0.137	0.119
		2	0.174	0.168	0.161	-	0.165	**0.157**
		3	-	-	-	0.121	0.118	**0.108**
0.25	0.25	1	0.168	0.15	0.148	0.149	0.142	**0.127**
		2	0.193	0.187	0.177	-	0.17	**0.159**
		3	-	-	-	0.123	0.123	**0.112**
0.25	2	1	0.241	0.23	0.315	0.181	**0.177**	0.25
		2	0.262	0.259	0.33	-	**0.185**	0.274
		3	-	-	-	**0.144**	0.15	0.235
2	0.25	1	0.242	0.229	0.314	0.16	0.153	**0.148**
		2	0.262	0.258	0.329	-	0.173	**0.163**
		3	-	-	-	0.127	0.124	**0.115**
2	2	1	0.321	0.309	0.473	0.198	**0.196**	0.256
		2	0.333	0.331	0.476	-	**0.196**	0.291
		3	-	-	-	**0.145**	0.147	0.245

Table 5: M70 model, 2-DC training. Runtime per microbatch of cross-DC PP solutions under various configurations (case 1-3) and communication delay. The best result of each configuration is shown in bold (lower is better). CrossUDSub, CrossUD and CrossWave schedules are abbreviated as UDSub, UD and Wave, respectively.

7.3 Scale to More DCs

We extend our analysis to 4 homogeneous interconnected DCs with uniform cross-DC link characteristics, using the same setup as the previous section (M70, $n_{TP} = 4$, $n_{PP} = 8$, $n_{DP} = 1$, now with 2 stages per DC). Table 6 confirms previous findings: CrossPipe is competitive without delay and outperforms static schedules in cross-DC scenarios. CrossWave excels CrossUD(Sub) at low delays but suffers under higher delays, especially under high bandwidth delays. With extra memory budget and GBS (Case 3), CrossUD schedule achieves 0.178 s per microbatch, only 22.8% slower than the corresponding 2-DC scenario at $(T_{lat}/T_F, T_{bw}/T_F) = (2,2)$. The larger bubble size in 4-DC training makes recomputation more effective here. Layer-wise recomputation with increased GBS (Case 2) is comparable to or outperforms the baseline without recomputation (Case 1) in most delay settings.

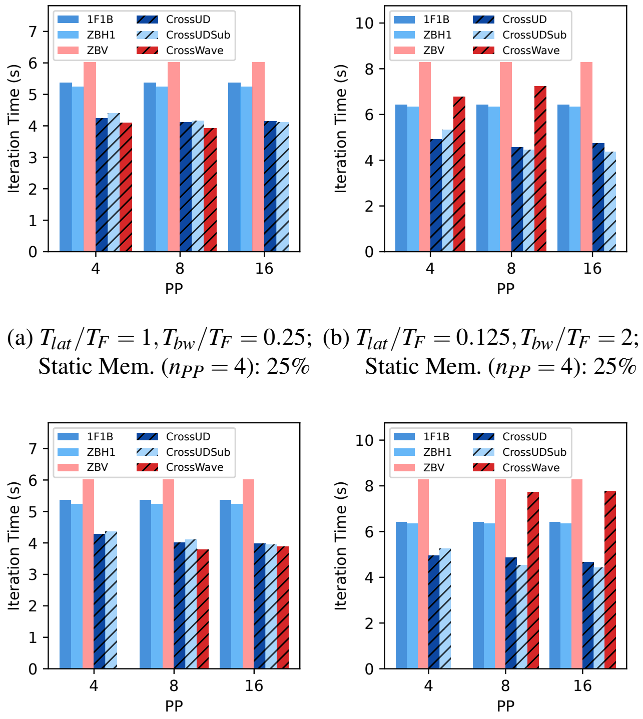
7.4 Trade-off of PP and DP

We analyze the choice of large PP size vs. large DP size in a 2-DC homogeneous training scenario. With a fixed number of compute nodes per DC and setting n_{TP} to the number of GPUs within each compute node, the product $n_{PP} \times n_{DP}$ equals the total node count of 2 DCs. Given the number of microbatches $n_{mb} = \varepsilon n_{PP}$, the global batch size $GBS = \varepsilon n_{PP} n_{DP}$ depends solely on the ratio ε. Then the GBS remains the same with various combinations of PP and DP.

We evaluated the trade-off on the M70 model across three PP configurations ($n_{PP} \in \{4, 8, 16\}$) with four combinations

$\frac{T_{lat}}{T_F}$	$\frac{T_{bw}}{T_F}$	Case	Static			Dynamic (This Work)		
			1F1B	ZBH1	ZBV	UDSub	UD	Wave
0	0	1	0.149	0.133	**0.119**	0.138	0.138	0.122
		2	0.173	0.168	0.16	-	0.167	**0.157**
		3	-	-	-	0.123	0.119	**0.115**
0.25	0.25	1	0.177	0.158	0.161	0.155	0.148	**0.141**
		2	0.198	0.19	0.181	-	0.173	**0.163**
		3	-	-	-	0.126	0.123	**0.115**
0.25	2	1	0.269	0.249	0.339	**0.216**	0.217	0.286
		2	0.274	0.269	0.331	-	**0.198**	0.29
		3	-	-	-	**0.158**	0.162	0.262
2	0.25	1	0.268	0.248	0.337	0.2	**0.197**	0.214
		2	0.274	0.269	0.33	-	0.184	**0.177**
		3	-	-	-	**0.138**	**0.138**	0.139
2	2	1	0.359	0.338	0.512	**0.268**	0.271	0.339
		2	0.349	0.346	0.479	-	**0.213**	0.295
		3	-	-	-	**0.178**	**0.178**	0.264

Table 6: M70 model, 4-DC training. Runtime per microbatch of each PP schedule is listed. The rest of the configurations remain the same as Table 5.



(a) $T_{lat}/T_F = 1, T_{bw}/T_F = 0.25$; Static Mem. ($n_{PP} = 4$): 25%

(b) $T_{lat}/T_F = 0.125, T_{bw}/T_F = 2$; Static Mem. ($n_{PP} = 4$): 25%

(c) $T_{lat}/T_F = 1, T_{bw}/T_F = 0.25$; Static Mem. ($n_{PP} = 4$): 50%

(d) $T_{lat}/T_F = 0.125, T_{bw}/T_F = 2$; Static Mem. ($n_{PP} = 4$): 50%

Figure 12: Trade-off between PP and DP ($n_{PP} \times n_{DP}$ is fixed) in a 2-DC training. Using M70 with 16 nodes ($n_{TP} = 4$ fixed) and a fixed GBS. Each subplot shows iteration time vs. n_{PP}. Labels indicate delay ratios (T_{lat}/T_F, T_{bw}/T_F) and static memory usage. Memory in percentage shows the consumption of static memory at $n_{PP} = 4$ w.r.t. device limit.

of communication delay and memory settings. For communication delay $(T_{lat}/T_F, T_{bw}/T_F)$, we test high latency, low bandwidth $(1, 0.25)$ and low latency, high bandwidth $(0.125, 2)$ settings, with a static memory budget of 25% and 50%. This percentage is defined as the allocation of static memory (parameters, gradients, and optimizer states) at $n_{PP} = 4$ w.r.t. the device memory limit. Larger n_{PP} reduces the static memory per stage and the activation memory size per microbatch, so that it increases the activation memory budget and allows for more in-flight forward blocks. Also with larger n_{PP}, the communication volume is lower due to smaller DP, but at a higher frequency. While the absolute latency remains constant when increasing n_{PP}, the latency delay ratio T_{lat}/T_F increases due to reduced T_F as n_{PP} grows (less work per stage). $\varepsilon = 4$ to enable more in-flight blocks in each schedule to benefit from available memory. Since the GBS (total workload) and the number of compute nodes (total GPUs) are fixed, we use runtime to compare different configurations. As shown in Figure 12, these factors balance out. The schedule efficiency remains largely invariant to PP/DP configurations across scenarios. This balance likely stems from the flexibility of CrossPipe to fully utilize the available memory with sufficient GBS.

8 Discussion

8.1 Heterogeneous DCs

In this work, we primarily evaluated homogeneous GPUs and nodes across DCs due to cluster constraints. However, our approach can be extended to heterogeneous environments. If compute resources (e.g., GPU, network) differ significantly between DCs, our generated schedules may be suboptimal because faster nodes will finish computation earlier and create bubbles in the pipeline, and slower nodes may run out of memory since their device memory is usually more limited. A practical solution is to maintain homogeneity within each pipeline stage, which aligns with current DC practices where nodes within the same cabinet tend to be identical. The compute nodes with faster GPUs can be assigned more layers to balance the computation time across stages. Our formulation (Section 4.1) and greedy algorithm (Section 4.2) already support stage-specific parameters, naturally supporting this adjustment.

8.2 Network Dynamics and Fault Tolerance

Real-world cross-DC networks exhibit variability and are prone to failures. CrossPipe can employ several strategies to enhance robustness:

- Short-Term Variations (seconds or less): Section 7.2 shows that CrossPipe schedules can trade system resources (e.g., device memory, by allowing more in-flight microbatches) for efficiency. Conservative (higher) latency and/or (lower) bandwidth estimations can be used to generate schedules that tolerate small spikes in communication delay.
- Longer-Term Variations (minutes or more): Network conditions can shift due to traffic or routing. CrossPipe's flexible execution engine (Section 6) supports hot-switching of pipeline schedules. The system with CrossPipe enabled can periodically re-profile network conditions (Section 6.1) and generate new, tailored greedy schedules (Section 4.2) to adapt without interrupting training.
- Packet Loss/Link Errors: Transient network errors such as packet drops can be handled by mechanisms like Forward Error Correction (FEC) [14] or Selective Repeat [2], leading to spikes in communication delay as mentioned above.
- Node Failures: The failure of a compute node requires higher-level mechanisms beyond pipeline scheduling. Efficient checkpointing, such as asynchronous methods [35] and in-memory approaches [67], is necessary to save training state efficiently and recover from the last checkpoint with minimal progress loss.

9 Related Works

Pipeline Parallelism (PP): 1F1B in PipeDream [38], GPipe [19], DAPPLE [13], interleaved 1F1B [39], bidirectional pipeline from Chimera [30], BPipe [26] and MPress [69] for memory balancing, BFSPP for more DP communication overlap [28], Hanayo [34] for wave-like schedules, ooo backprop [42] and zero-bubble PP [43] for weight gradient splitting, AdaPipe [57] for co-optimizing layer distribution and recomputation, DHelix [62] for microbatch co-execution to overlap communication, DistMM [18] for multimodal model training. Sequence-level pipeline parallelism [32, 56].

Training on restricted networks with PP: Varuna [3] explores training on spot VMs with commodity networking. Bamboo [60] studies resilient training on preemptible instances. Oobleck [22] improves training resilience via pipeline templates. SWARM [47] proposes reliable training via temporary randomized pipelines. [68] studies device assignment in hybrid parallel training with geo-distributed nodes. CocktailSGD [63] combines multiple compression techniques to train models efficiently on low-bandwidth networks. FusionLLM [58] accelerates decentralized training via activation and gradient compression. DiLoCo [10] explores robust asynchronous training on poorly connected machines.

10 Conclusion

In this work, we first introduced a validated pipeline performance model that explicitly accounts for latency and bandwidth delays in cross-datacenter links. Using this model, we demonstrated that pipeline parallelism is often the superior approach for the parallelism dimension spanning across datacenters, especially under constrained network conditions. Next, we leveraged the model to develop optimal and near-optimal algorithms for generating pipeline schedules that minimize cross-datacenter communication delays while adhering to memory constraints. Finally, we integrated these methods into a flexible execution engine featuring a two-layer abstraction (block scheduling and communication arrangement) that works seamlessly with existing training systems, such as Megatron-LM.

Our evaluation shows that CrossPipe effectively overcomes the challenges of cross-datacenter training. It reduces the training time by up to 33.6% compared to traditional pipeline schedules in a cross-DC setup, all while maintaining the same memory constraints. When memory constraints are relaxed, CrossPipe in a cross-DC setup is able to achieve a similar training time as a static ZBV schedule in a single-DC setup where there is almost no communication delay. CrossPipe thus offers improved scalability and resource utilization, making large-scale distributed training more feasible and efficient.

Acknowledgments

We are grateful to our shepherd and the anonymous reviewers for their insightful comments and constructive feedback. We thank the CSCS team for providing access to the Ault and Alps machines, as well as for their outstanding technical support. We are grateful to Siyuan Shen and Mikhail Khalilov for their valuable advice, and to Timo Schneider for his assistance with infrastructure at SPCL. We also acknowledge the Polish high-performance computing infrastructure PLGrid (HPC Center: ACK Cyfronet AGH) for providing computational resources and support.

References

[1] Reed Albergotti. Microsoft Azure CTO: US data centers will soon hit size limits. *Semafor*, October 2024. Technology.

[2] Miltiades Anagnostou and Emmanuel Protonotarios. Performance analysis of the selective repeat arq protocol. *IEEE Transactions on Communications*, 34(2):127–135, 2003.

[3] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: Scalable, low-cost training of massive deep learning models, 2021.

[4] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, 2018.

[5] Bloomberg. Tech firms are asking energy giant nextera for enough electricity to power miami, 2024.

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.

[7] João da Silva. Google turns to nuclear to power AI data centres. *BBC News*, October 2024. Business.

[8] Daniele De Sensi, Salvatore Di Girolamo, Kim H McMahon, Duncan Roweth, and Torsten Hoefler. An in-depth analysis of the slingshot interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[9] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025.

[10] Arthur Douillard, Qixuan Feng, Andrei A. Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc'Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. Diloco: Distributed low-communication training of language models, 2024.

[11] Jiangfei Duan, Shuo Zhang, Zerui Wang, Lijuan Jiang, Wenwen Qu, Qinghao Hu, Guoteng Wang, Qizhen Weng, Hang Yan, Xingcheng Zhang, et al. Efficient training of large language models on distributed infrastructures: A survey. *arXiv preprint arXiv:2407.20018*, 2024.

[12] Abhimanyu Dubey et al. The llama 3 herd of models, 2024.

[13] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: A pipelined data parallel approach for training large models, 2020.

[14] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

[15] Zeyu Han, Chao Gao, Jinyang Liu, Sai Qian Zhang, et al. Parameter-efficient fine-tuning for large models: A com-

prehensive survey. *arXiv preprint arXiv:2403.14608*, 2024.

[16] Marius Hobbhahn, Lennart Heim, and Gökçe Aydos. Trends in machine learning hardware, 2023. Accessed: 2024-10-15.

[17] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. The impact of network noise at large-scale communication performance. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, 2009.

[18] Jun Huang, Zhen Zhang, Shuai Zheng, Feng Qin, and Yida Wang. DISTMM: Accelerating distributed multimodal model training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1157–1171, Santa Clara, CA, April 2024. USENIX Association.

[19] 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.

[20] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

[21] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models, 2023.

[22] 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*, SOSP '23, page 382–395. ACM, October 2023.

[23] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.

[24] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

[25] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Computer Architecture News*, 36(3):77–88, 2008.

[26] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. BPipe: Memory-balanced pipeline parallelism for training large language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 16639–16653. PMLR, 23–29 Jul 2023.

[27] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models, 2022.

[28] Joel Lamy-Poirier. Breadth-first pipeline parallelism, 2023.

[29] Junyi Li, Tianyi Tang, Wayne Xin Zhao, Jian-Yun Nie, and Ji-Rong Wen. Pre-trained language models for text generation: A survey. *ACM Computing Surveys*, 56(9):1–39, 2024.

[30] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[31] Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, et al. Personal llm agents: Insights and survey about the capability, efficiency and security. *arXiv preprint arXiv:2401.05459*, 2024.

[32] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 6543–6552. PMLR, 18–24 Jul 2021.

[33] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context, 2023.

[34] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2023.

[35] 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.

[36] Andrew Moseman. Amazon vies for nuclear-powered data center: The deal has become a flash point over energy fairness. *IEEE Spectrum*, August 2024.

[37] MPI Forum. *MPI: A Message-Passing Interface Standard Version 3.1*, 2015.

[38] 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.

[39] 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.

[40] NVIDIA. Nvidia dgx-1 with tesla v100 system architecture, 2017.

[41] NVIDIA. Nvidia grace hopper superchip architecture. Whitepaper, NVIDIA Corporation, 2024.

[42] Hyungjun Oh, Junyeol Lee, Hyeongju Kim, and Jiwon Seo. Out-of-order backprop: An effective scheduling technique for deep learning. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 435–452, 2022.

[43] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism, 2023.

[44] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

[45] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, November 2020.

[46] Reuters. Microsoft deal signals booming demand from data centers to power AI. *Reuters*, September 2024. Energy, Grid & Infrastructure, Nuclear.

[47] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. Swarm parallelism: Training large models can be surprisingly communication-efficient, 2023.

[48] Jaime Sevilla and Edu Roldán. Training compute of frontier ai models grows by 4-5x per year, 2024. Accessed: 2024-10-15.

[49] Kawthar Shafie Khorassani, Chen Chun Chen, Bharath Ramesh, Aamir Shafi, Hari Subramoni, and Dhabaleswar Panda. High performance mpi over the slingshot interconnect: Early experiences. In *Practice and Experience in Advanced Research Computing*, pages 1–7. 2022.

[50] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024.

[51] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017.

[52] Siyuan Shen, Langwen Huang, Marcin Chrapek, Timo Schneider, Jai Dayal, Manisha Gajbe, Robert Wisniewski, and Torsten Hoefler. Llamp: Assessing network latency tolerance of hpc applications with linear programming, 2024.

[53] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[54] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. Ml training with cloud gpu shortages: Is cross-region the answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, EuroMLSys '24, page 107–116, New York, NY, USA, 2024. Association for Computing Machinery.

[55] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. Ml training with cloud gpu shortages: Is cross-region the answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, pages 107–116, 2024.

[56] Ao Sun, Weilin Zhao, Xu Han, Cheng Yang, Xinrong Zhang, Zhiyuan Liu, Chuan Shi, and Maosong Sun. Seq1f1b: Efficient sequence-level pipeline parallelism for large language model training, 2024.

[57] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. Adapipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 86–100, New York, NY, USA, 2024. Association for Computing Machinery.

[58] Zhenheng Tang, Xueze Kang, Yiming Yin, Xinglin Pan, Yuxin Wang, Xin He, Qiang Wang, Rongfei Zeng, Kaiyong Zhao, Shaohuai Shi, Amelie Chi Zhou, Bo Li, Bingsheng He, and Xiaowen Chu. Fusionllm: A decentralized llm training system on geo-distributed gpus with adaptive compression, 2024.

[59] Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. Large language models in medicine. *Nature medicine*, 29(8):1930–1940, 2023.

[60] 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, 2022.

[61] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[62] Haiquan Wang, Chaoyi Ruan, Jia He, Jiaqi Ruan, Chengjie Tang, Xiaosong Ma, and Cheng Li. Hiding communication cost in distributed llm training via microbatch co-execution, 2024.

[63] Jue Wang, Yucheng Lu, Binhang Yuan, Beidi Chen, Percy Liang, Christopher De Sa, Christopher Re, and Ce Zhang. CocktailSGD: Fine-tuning foundation models over 500Mbps networks. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 36058–36076. PMLR, 23–29 Jul 2023.

[64] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 93–106, New York, NY, USA, 2022. Association for Computing Machinery.

[65] Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani. Rail-only: A low-cost high-performance network for training llms with trillion parameters, 2024.

[66] Yufei Wang, Wanjun Zhong, Liangyou Li, Fei Mi, Xingshan Zeng, Wenyong Huang, Lifeng Shang, Xin Jiang, and Qun Liu. Aligning large language models with human: A survey. *arXiv preprint arXiv:2307.12966*, 2023.

[67] 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.

[68] Binhang Yuan, Yongjun He, Jared Quincy Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christopher Re, and Ce Zhang. Decentralized training of foundation models in heterogeneous environments, 2023.

[69] Quan Zhou, Haiquan Wang, Xiaoyan Yu, Cheng Li, Youhui Bai, Feng Yan, and Yinlong Xu. Mpress: Democratizing billion-scale model training on multi-gpu servers via memory-saving inter-operator parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 556–569, 2023.

A Runtime Definition

In cross-DC training, due to the relatively high communication cost, the stage 0 usually finishes last among all the stages. So we adopt the most used definition of pipeline runtime as the following: the duration from the start of the first forward block on PP stage 0 till the completion of the last block on any PP rank (or the final DP communication on any rank). This definition aligns with frameworks that apply additional synchronization for global gradient norm computation (for clipping) and numerical anomaly detection (NANs/INFs) in mixed-precision training.

B Weight Gradient Separation

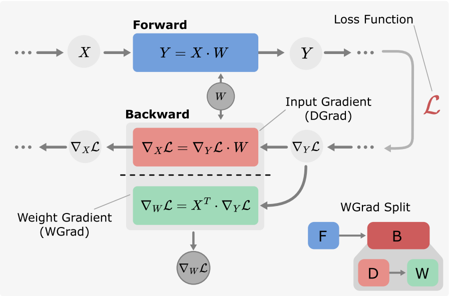


Figure 13: Separating input gradient computation (DGrad, or D) and weight gradient computation (WGrad, or W) in a linear layer.

Figure 13 illustrates how each B block can be further divided into two parts: input data gradient computation (DGrad, or D) and weight gradient computation (WGrad, or W) [43].

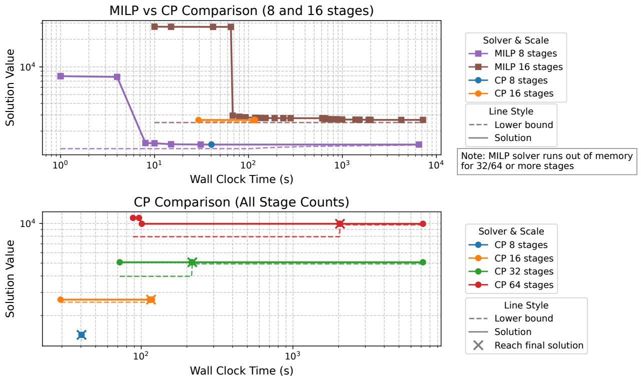


Figure 14: Scalability of MILP and CP solvers finding optimal pipeline schedules. Results show time-to-solution for varying pipeline stage counts (n_{PP}) with a fixed microbatch factor ($n_{mb} = 2n_{PP}$). Markers indicate the moments when the solver discovers an improved solution.

C Bubble Strides in Schedules

Figure 15 further illustrates the existence of *bubble strides*. We show schedules with 16 stages and 2 DCs for the static schedules that we compared in our work (1F1B, IV1F1B, ZBH1, and ZBV). The latency delay on the cross-DC link is set to $1.5 \cdot T_F$. Despite this relatively small latency (compared to the size of the pipeline blocks), the delay accumulates throughout the schedule rather than being absorbed or mitigated. This accumulation underscores the need for adaptive scheduling.

D Optimal Scheduling

D.1 Scalability of MILP and CO Solvers

We compare the scalability of both MILP and CO solvers, Gurobi (12.0.0) and CPLEX (22.1.1) respectively, both among the best solvers in their fields. The MILP formulation is from [43] and the CO formulation from Section 4.1. We conduct the experiments on a machine with AMD EPYC 7742 @ 2.25GHz CPU (128 physical cores), 256GB RAM. Each solver is configured to 256 worker threads, with a time limit of 7200 seconds. We use Gurobi Optimizer with the `NodefileStart=0.5` parameter to handle potential memory limitations.The solver is early-terminated if the relative gap between the objective value of the best integer solution found so far (ObjVal) and the best objective bound (ObjBound) is below 1%, defined as $\frac{|ObjBound - ObjVal|}{|ObjVal|} \leq 0.01$. The time to solution of both solvers are shown in Figure 14. The CO solver scales better than the MILP solver for pipeline schedule generation, ideally because of its specialized optimization w.r.t. job scheduling problems. The MILP solver runs out of memory even for runs of 32/64 stages and converges slower than CO solver. On the other hand, CO solver shows tractable performance when problem size scales up, and is able to find a good feasible solution in a reasonable amount of time while spending most of the time searching for a tighter bound.

E Comparing Cross-DC PP and Cross-DC DP

We use a 2 DC setup for the Llama 3 405B model training. The training configuration is taken from Llama 3 [12]: $n_{TP} = 8$, $n_{PP} = 16$, $n_{DP} = 64$, $seq\_len = 8192$, $GBS = 2n_{PP}n_{DP}$.

We estimate the computation time of each layer by the following equation:

$$T_{layer} = \frac{C_{layer}}{P_{GPU} \times n_{TP}}$$

where C_{layer} is the FLOPs count of a transformer layer in the 405B model, P_{GPU} is the practical BF16 performance of GPU (500 TFLOPs per second is used in the analysis, considering the TP communication overhead). Assuming the GPUs

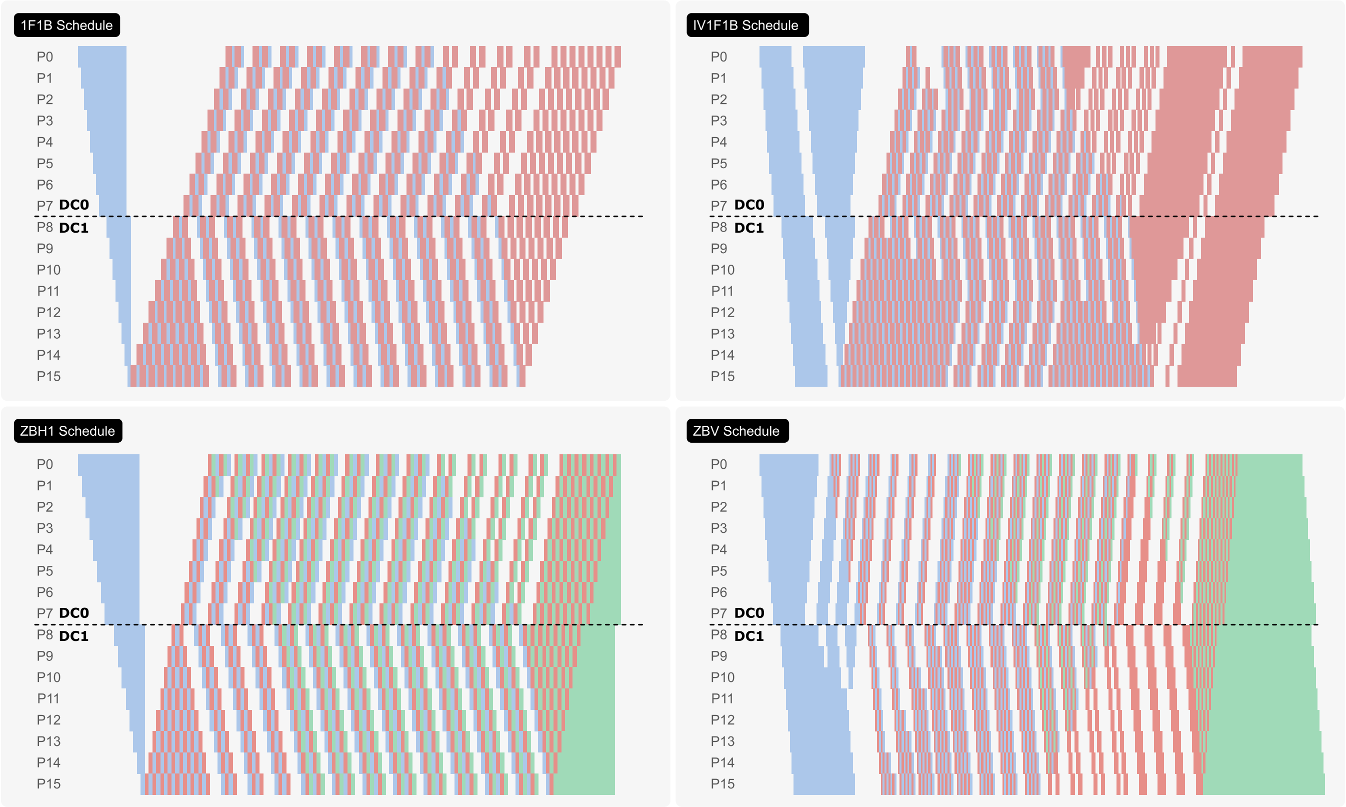


Figure 15: Illustration of bubble strides in various static pipeline schedules (1F1B, IV1F1B, ZBH1, and ZBV). Setup: 16 stages, 2DCs (8 stages per DC), $n_{mb} = 32$.

are equally distributed in 2 DCs. In cross-DC PP, we apply solver-based schedules (CrossUD or CrossWave, whichever is better for the given delay). In cross-DC DP, we use the ZBV schedule and assume that the extra cross-DC communication is not overlapped. For Allreduce (or Reduce-Scatter + All-gather) DP communications, we apply the bandwidth-optimal Ring-algorithm. The cross-DC DP communication cost can be estimated as $2 \times (\alpha + 2 \times \frac{N}{2} \times \beta) = 2\alpha + 2N\beta$ which accounts for two rounds of communication between two DCs and half of the parameters/gradients are sent/received. N is the number of model parameters, the extra factor 2 in the bandwidth term comes from the size of the BF16 datatype (2 bytes/param).

F Latency and Bandwidth Delay Injection

F.1 Implementation Details

Figure 16 illustrates our mechanism for injecting latency and bandwidth delay, used to emulate cross-DC network behavior in a controlled setting, without modifying core NCCL behavior or requiring network hardware manipulation. GPU kernel execution, including that of NCCL communication kernels, is asynchronous to the host CPU, making precise delay injection non-trivial.

- **Bandwidth Delay**: To simulate limited bandwidth, we inject additional delay into the communication path by occupying the communication stream after each NCCL send/receive. Specifically, a spin kernel is posted to the same stream immediately after the NCCL call, both on the sender and receiver sides. The spin kernel duration is computed to match the target delay. This effectively stalls further communication or computation that shares the stream, emulating a fully utilized link.
- **Latency Delay**: We extend PyTorch NCCL backend by adding a new method, `handle.wait_with_lat_delay()`, which is invoked on the receiver side during each blocking wait on NCCL communication. It is similar to the original `wait()` method but adds a controlled amount of host-side spinning to delay the launch of subsequent computation kernels.

The injection process contains the following steps: ❶ Sender and receiver post matching asynchronous point-to-point operations.❷ Optional compute kernels may be posted to overlap with communication. ❸ The receiver calls `wait_with_lat_delay` to synchronize the computation stream with the communication stream. ❹ A CUDA event measures the elapsed time from the completion of the communication kernel to the current time on the computation stream. ❺ The host then synchronizes with the computation stream to retrieve the elapsed time and calculates the remaining delay (if any) to inject. ❻ A spinning kernel is posted to the computation stream for the remaining delay. This mechanism ensures that delay is not introduced when communication has already completed, and only partially injected when communication is partially complete (e.g., due to overlap with computation), thereby preserving the correct performance behavior of latency-hiding schedules.

This injection method is specifically designed for the CrossPipe implementation, where communication in four directions is split across four concurrent streams. Injecting delays into collective communications is more complex. While latency delays can be introduced within communication libraries [52], simulating bandwidth delays may require additional configurations at the network switch level.

In our communication model, we account only for latency and bandwidth delays along the critical path. We assume that the transmission of control messages, such as "ready to send" signals in a rendezvous protocol, is removed from the critical path. This assumption is crucial for maintaining performance, especially under high-latency conditions.

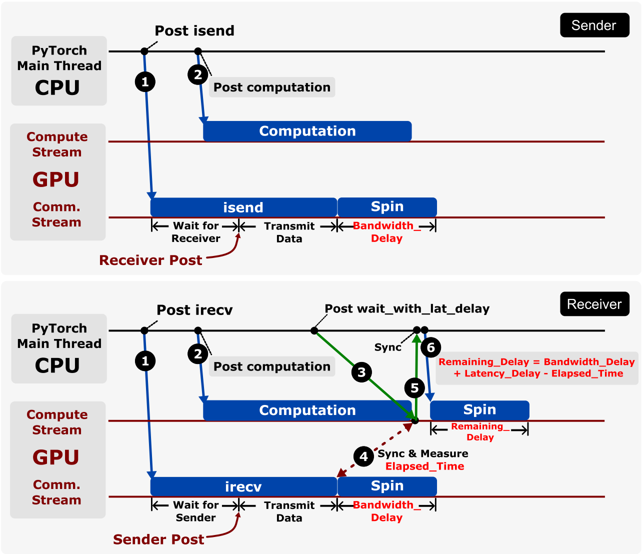


Figure 16: Mechanism for latency and bandwidth delay injection. Top: sender side. Bottom: receiver side.

F.2 Validation

To validate the accuracy of our delay injection methods (Section 6.4), we conduct experiments on a single GH200 node using 4 GB messages.

For latency tests, we use ping-pong communication between two processes: each round consists of a sender transmitting a message, waiting for a reply, and measuring the round-trip time. We divide the round-trip latency by two and compare it against the baseline (no injection). For bandwidth tests, the sender transmits multiple large messages back-to-back, while the receiver posts matching receives. We compute the average time per message and compare it to the baseline.

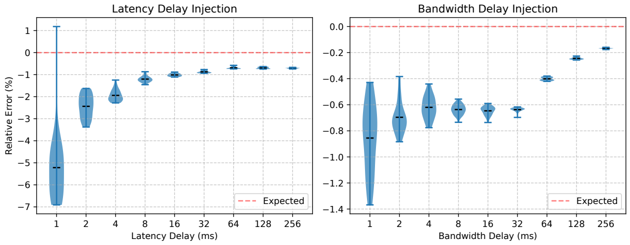


Figure 17: Validation of latency and bandwidth delay injection methods described in Section 6.4.

For each delay setting, we compute the relative error between the observed and expected delays. Specifically, we perform multiple iterations per setting, discard outliers, and report the percentage deviation from the expected delay. These measurements are then visualized using violin plots.

Results in Figure 17 demonstrate that our injection methods accurately reflect the communication model described in Section 2.2. Minor deviations primarily stem from host CPU synchronization and slight underestimation of GPU clock rate by `cudaGetDeviceProperties` in the spinning kernel. The validation shows the injected delay closely matches the target delay, confirming the mechanism's suitability for emulating cross-DC network conditions.