# TrainMover: Efficient ML Training Live Migration with No Memory Overhead

*ChonLam Lao*[‡§*], *Minlan Yu*[‡], *Aditya Akella*[◇], *Jiamin Cao*[§], *Yu Guan*[§], *Pengcheng Zhang*[§], *Zhilong Zheng*[§], *Yichi Xu*[§], *Ennan Zhai*[§], *Dennis Cai*[§] *Jiaqi Gao*[§]

[‡]*Harvard University*, [§]*Alibaba Cloud*, [◇]*University of Texas at Austin*

## Abstract

Machine learning training has emerged as one of the most prominent workloads in modern data centers. These training jobs are large-scale, long-lasting, and tightly coupled, and are often disrupted by various events in the cluster such as failures, maintenance, and job scheduling. To handle these events, we rely on cold migration, where we first checkpoint the entire cluster, replace the related machines, and then restart the training. This approach leads to disruptions to the training jobs, resulting in significant downtime. In this paper, we present TrainMover, a live migration system that enables machine replacement during machine learning training. TrainMover minimizes downtime by leveraging member replacement of collective communication groups and sandbox lazy initialization. Our evaluation demonstrates that TrainMover achieves 16× less downtime compared to all baselines, effectively handling data center events like straggler rebalancing, maintenance, and unexpected failures.

## 1 Introduction

Large Language Models (LLMs) have gained significant attention in the past few years [5, 7, 26, 28, 29, 35, 36]. The scaling law guides people in designing and training larger LLM models to improve performance. LLM training jobs are commonly deployed on decentralized parallel training frameworks (*e.g.,* Megatron-LM [33] and NeMo [20]), span thousands of GPUs, and last for weeks to months. For example, training GPT-3 with 175 billion parameters on 1024 GPUs required approximately 34 days [25], while Llama 3 with 405 billion parameters was trained over 54 days using up to 16,000 H100 GPUs [13]. LLM training commonly adopts model parallelism and data parallelism and requires coordination from all GPUs.

This large-scale, long-lasting, tightly coupled distributed training is often disrupted by various events in the cluster such as failures, maintenance, and job (re)scheduling. First, as the scale increases, there are frequent failures and anomalies of individual components, which can slowdown or halt the entire cluster. Alibaba's report [40] shows that 60% of large-scale training jobs experience slowness due to various reasons including hardware anomalies, software contentions, etc., resulting in a 35% increase in job completion time (JCT) due to cluster slowness. Llama 3 training job experienced a mean-time-to-failure (MTTF) of 2.7 hours [13]. Second, as training takes weeks to months, we cannot simply defer maintenance jobs (repairs, upgrades, etc.) until after the training because delayed maintenance increases the safety and security risks of the cluster [13]. Maintainance often requires rebooting the server or switch and interrupts the training job [1, 19]. Third, in a shared cluster, we need to reschedule jobs and rebalance resources, when a new high-priority job joins the system [8, 41], or when an entire pod is now available for a training job running on fragmented resources across pods [37].

To handle these events, the current best practice is **cold migration** which includes three steps: checkpoint, replace, and restart [9, 14, 18, 38]. When a cluster event happens, the training framework either waits for the next scheduled checkpoint or triggers one immediately. The scheduler stops the training job, replaces the abnormal servers with backups in the same cluster, and restarts the job based on the last checkpoints. This results in minutes-level downtime caused by schedulers, checkpoint storage, and training framework initialization (see Section §2), which, when amplified across thousands of GPUs, can lead to extremely high costs.

Recent works such as ReCycle [10], Oobleck [17], and Parcae [8] propose to handle failure events with **live reconfiguration**. When failure happens, these works keep the training job with the remaining servers by changing the hyperparameters (such as batch size) or parallelism schemes. However, as the cluster size changes, optimization operators designed for the original cluster size and topology [18] become ineffective, leading to degraded training throughput. For example, after reconfiguration, some intensive communication parallelism groups may span across different racks, disrupting machine training locality and increasing communication overhead among machines. Furthermore, reducing the number of machines can increase memory pressure on the remaining GPUs, potentially resulting in out-of-memory issues.

In this paper, we introduce TrainMover, which enables **live migration** for LLM training frameworks. TrainMover leverages standby servers that are generally available in the cluster and shifts the workload from source GPUs (*migration leavers*) to standby GPUs (*migration joiners*) without restarting the entire training job or changing its parallelization scheme. TrainMover carefully manage the migration overlap period to prevent interference with ongoing training, minimize downtime, and avoid GPU memory overhead during migration. Train-

[*] Work done during ChonLam Lao's internship at Alibaba Cloud.

Mover leverages two key techniques, partial replacement of collective communication group members and sandbox lazy initialization, to reduce the migration downtime introduced by the joiners' communication and computation initialization. TrainMover removes unnecessary interactions between the joiners and the rest of the participants so that the joiners can prepare in the background without interrupting the training job.

**Member replacement of collective communication groups:** The current collective communication library does not support partially replacing members in the collective communication group (CCG) after initialization. The only way to migrate the communication channels from leavers to joiners is to destroy the CCGs on the leavers and re-initiate them on the joiners, which introduces a long downtime. TrainMover proposes the CCG member replacement algorithm to solve this problem. The algorithm guarantees the replaced CCG's communication graph is equivalent to the re-initiated one and therefore frees from performance degradation. TrainMover confines the communication graph generation and intra-server connection initialization within the joiners so that they can be performed in the background and only leave inter-server connection establishment on the critical path.

**Sandbox lazy initilization:** Lazy initialization is widely used in current LLM training frameworks to reduce the preparation time and perform runtime optimizations. After the joiners replace the leavers and load the model, the entire training job is delayed because the joiners rerun the lazy initializations. TrainMover introduces sandbox lazy initialization to warm up the joiners before replacing the leavers. After each joiner finishes the preparation, the sandbox triggers one emulated iteration and intercepts every inter-machine collective communication. The sandbox replaces each inter-machine collective communication with pre-recorded tensors so that joiners can initialize without other participants and the emulated iteration can align with a normal one.

TrainMover leverages the two key techniques to handle various types of data center events, such as maintenance, straggler handling, and rebalancing, while proposing an additional workflow to address unexpected failure events. For data center events, TrainMover prepares the joiners in a sandbox environment, ensuring the training job continues running untouched on the original cluster. This approach minimizes disruption and maintains training performance during migrations. When the joiners are ready, TrainMover freezes the training job, replaces the CCG member, and synchronizes the latest training model from the leavers to the joiners. Once finished, TrainMover resumes the training job in the new cluster with the original configuration and training throughput. For unexpected failure events, TrainMover directly freezes all machines and initializes the joiners' communication and computation at the same time. TrainMover loads the model from other servers if the framework provides redundant copies of the model. Otherwise, the entire cluster falls back to the last saved checkpoint. Our experiments demonstrate that TrainMover achieves 16× less downtime compared to baselines under frequent live migration. During unexpected failures, TrainMover enables recovery 2.35x faster than the baselines.

## 2 Background and Motivation

### 2.1 Distributed LLM Training

State-of-the-art LLM models consist of hundreds of billions of parameters and are trained atop datasets with trillions of tokens. Training such a model requires a cluster with exaflops of computing power, an efficient distributed training algorithm, a failure-resilient training framework, and a responsive cluster scheduler.

**Training cluster.** The training cluster contains thousands of GPU servers with tens of thousands of GPUs. For example, Alibaba's HPN cluster [27] supports 15K GPUs per pod, while Meta deploys two clusters [13], each with 24K Nvidia H100 GPUs, to train the Llama 3 model. The training clusters commonly deploy two independent networks: the frontend network connects GPU servers with services such as storage, logging, *etc.*, and carries training samples, checkpoints, and logs, while the backend network provides high-speed interconnect for all GPUs, with 4–8× more bandwidth than the frontend network [11,27].

**Distributed training framework.** The training framework, such as Megatron-LM [33], initializes computation and communication in the GPU cluster, loads training data from storage, executes the distributed training algorithm, and monitors cluster health during runtime. It periodically snapshots model parameters and optimizer states, saving them to checkpoint storage. To fit large models into limited per-GPU memory and accelerate training [16,24,25,33], frameworks use model and data parallelism. Model parallelism (e.g., tensor parallelism (TP), pipeline parallelism (PP), and sequential parallelism (SP)) partitions the model across GPUs in different dimensions—TP splits at the tensor level, while PP splits at the layer level. Data parallelism (DP) duplicates model and optimizer states, with each replica processing a subset of the training data. Optimizer states often consume more GPU memory than model parameters [30]. To mitigate this, recent frameworks [2,18,30,38] use distributed optimizers (DO), which evenly distribute optimizer states across DP groups, eliminating redundancy and enabling larger models to fit within the cluster. Collective communication libraries (CCLs) like NCCL facilitate GPU communication and exchange intermediate results, ensuring efficient iteration completion.

**Cluster scheduler.** A central cluster scheduler (*e.g.*, Slurm) manages the entire cluster. Upon receiving a training job, it gathers available servers, performs health checks, deploys the job, and starts the training framework. On a shared environment, the scheduler also prepares a isolated virtualized environment and creates a private channel between the GPU servers and peripherals such as storage and logging service.

When a job finishes or fails, the scheduler waits for the training framework to exit or terminates it after a timeout, cleans up the GPU servers, destroys the virtualized environment if needed, and marks the servers as available. Leveraging operator expertise, the scheduler can detect and isolate anomaly machines and reschedule jobs automatically.

## 2.2 Call for Live Migration Primitive

Sustaining high training throughput is challenging. A running job's performance can change between different runs and different iterations in the same run due to various reasons. Overheating or power loss lowers the GPU's clock frequency. Switch, optical module, or NIC failures create network bottlenecks and delay CCL completion. Contention between the training process and other processes running in the background reduces the CPU time for training. Job scheduler's suboptimal job assignment creates unnecessary traffic collisions, which incur longer CCL completions. FALCON [40] measured a 35% JCT increase due to various slowdown events.

Job interruption is another major reason. The training job can be interrupted by disruptive events such as hardware failures, software bugs, scheduler preemption, regular maintenance, *etc*. Meta experienced 466 job interruptions during their 54-day Llama 3 training [13].

When resolving anomalies, maintaining the same parallelization strategy (*i.e.*, migration) is critical since the training framework is specially tuned for the specific cluster size and the topology. For example, Meta [13] developed a memory consumption estimator and a performance-projection tool to find the best parallelism configuration to achieve the highest training throughput with minimum communication overhead and avoid GPU memory overflow. They also adjusted the parallelism strategy and workload allocation so that no imbalance exists between the machines and the cluster's overall throughput is maximized. Readjusting the configuration can lower the efficiency and cause resource wastage.

Furthermore, data center architects are fully prepared for migration and usually provide enough redundancy in the cluster. For example, the Llama 3 training cluster contains 24K GPUs while only 16K are used for training the model [11], HPN7.0 provides 8 redundant machines in the first-tier network to tolerate hardware failures [27].

Currently, checkpoint-replace-restart (cold migration) [4, 14, 15, 18, 38] is the best practice for resolving anomalies. When slowness is detected or maintenance is scheduled, to avoid losing unsaved progress, the operator can either trigger one checkpoint proactively or wait for the next scheduled checkpoint. Operators also schedule periodic checkpoints to handle unexpected interruptions. After the job exits and the root cause is located, the scheduler then isolates anomaly machines, replaces them with healthy ones, and restarts the job.

However, cold migration reduces the cluster availability because it introduces many scheduler operations including job cleanup, reschedule, and re-initialization. Under unexpected failures, it also discards the unsaved progresses. To quantify the impact, we sampled two LLM training jobs for different models at different scales in our production clusters for a week and the result is shown in Table 1. We can see that the time wastage caused by cold migration (unsaved progress, job initialization, cleanup, and reschedule) contributes nearly 90% of the cluster downtime. This is because the job restart triggers complicated and sequential management operations on many components. For example, in a shared cluster, destroying the virtual network requires confirmations from all servers that all services require networking (*e.g.* monitoring, storage) have finished. A single unresponsive operation on one server not only delays the current step, but also delays all followup steps. When job stops, a critical step of the monitoring system is to push unsaved monitoring data on each server to the logging service. The large scale synchronized write burst easily causes stragglers and delays the followup cleanup steps. Scheduler can be trapped in a restart loop if it does not identifies the culprit correctly and continuously reschedules the job to the same malfunctioning server. This creates huge resource wastage, especially during night times when the operator cannot intervene. Slowness contributes only around 10% of the downtime because operators usually choose to migrate the job when slowness is detected and deemed persistent.

We promote to replace the traditional **cold migration** primitive with the **live migration** primitive. The live migration primitive shifts the workload from anomaly machines to healthy standbys without restarting the workload on other healthy machines. It maintains the original optimal hyperparameter setting and parallelization strategies to sustain high training efficiency. Live migration only requires minimal cluster changes and avoids unnecessary scheduler operations. The healthy machines do not go through the restart process, which reduces pressure on peripheral services such as checkpoint storage and logging.

With the help of live migration primitive, the operators have more deployment flexibility and longer maintenance window without suffering from the high restart overhead. They can isolate the stragglers, failed nodes or switches, rebalance a suboptimal cluster allocation, and shift workload to perform maintenance or balance the electric load without stopping and rescheduling the job. It avoids job rescheduling failures. Also, because live migration bypasses job restart on healthy machines, the lower pressure on peripheral services allows a more responsive cleanup and initialization on anomaly machines and healthy standbys, respectively.

## 2.3 Naive Live Migration Implementation

The basic requirement to implement live migration is **zero memory overhead** because we observe that current large-scale LLM training jobs commonly keep GPU memory consumption full. Indeed, as shown in Figure 1, the average memory consumption per GPU of three model training jobs at

| Category | Cluster scale | |
|---|---|---|
| | 4800 | 8192 |
| Healthy | 92.44% | 81.39% |
| Unsaved progress | 3.44% | 6.34% |
| Job initialization | 1.12% | 2.00% |
| Job cleanup | 0.98% | 3.37% |
| Job reschedule | 1.37% | 3.94% |
| Slowness | 0.65% | 2.96% |

Table 1: Cluster status of two training jobs at different scales. Slowness is defined as a 10% iteration time increase.

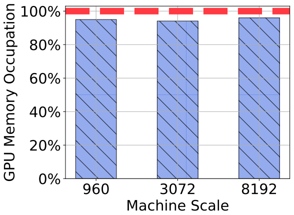


Figure 1: GPU memory utilization across machine scales

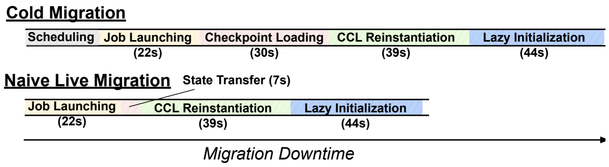


Figure 2: Migrating ine machine to another in a GPT-10B model on an 8-GPU machine setting

different scales sampled from our production cluster is higher than 94% per job. This is not a coincidence. Operators observe higher training throughput when increasing the job's memory ratio, for example, by increasing the batch size.

Based on this requirement, we derived a naive live migration system from the cold migration process. Instead of re-scheduling and rebooting all machines and retrieving the training checkpoint, a naive live migration system operates as follows: (1) all existing training participants are notified about the decision to migrate machines (*migration leavers*) to new machines (*migration joiners*), typically triggered by an event such as a straggler; (2) the *migration joiners* launch the training job from scratch; (3) the *migration leavers* transfer training states (e.g., model parameters and optimizer states) to the *migration joiners* once the joiners complete the job booting process; (4) all communication-related groups (e.g., data parallelism CCL groups, pipeline parallelism CCL groups) are destroyed and re-instantiated due to CCL's static limitation (§4); (5) the *migration leavers* disconnect, and their role is replaced by the *joiners* in the training job; and (6) after the *joiners* are fully replaced into the training process with the existing machines, the first few iterations are unavoidably very slow due various lazy initialization and computational resource ramp up (§5).

Unfortunately, although naive live migration avoids the complete teardown and restart of all jobs across machines, the critical path remains largely unchanged. This is because the *migration joiners* must undergo the same procedures as those in cold migration, as illustrated in Figure 2. This figure shows the restart procedures of running a GPT-10B job on a 64-GPU, 8-machine cluster with our naive live migration implementation. Regardless of whether there is a single *joiner* or all *joiners*, the downtime remains similar—135 seconds for one *joiner* and 105 seconds for all *joiners*. This is because all participants must wait for every participant to finish each procedure sequentially.

To reduce migration downtime, one approach is to overlap these procedures. However, this can easily incur additional memory overhead on both the communication and computation sides. Take CCL as an example—while overlapping and coexisting old and new CCL groups during migration can reduce downtime associated with destroying and re-instantiating CCL groups, it incurs significant GPU memory overhead, as every CCL group instance requires dedicated GPU buffers.

warm up in the background

Furthermore, overlapping without careful design can negatively impact ongoing training performance. Taking computation as an example, overlapping the joiner's lazy initialization requires existing machines to spare GPU cycles and incurs additional synchronization costs to participate, as lazy initialization can only be triggered during the *actual training iteration* due to runtime constraints. These constraints result in the sequential costs illustrated in Figure 2.

The downtime-memory dilemma highlights the difficulty of achieving the goals of **minimizing JCT overhead while simultaneously avoiding any memory overhead** and motivates us to address these challenges through the development of a new live migration system — TrainMover.

## 3 TrainMover Overview

TrainMover is a resilient LLM training framework that provides the live migration primitive. TrainMover develops two key techniques, collective communication group (CCG) replacement and sandbox lazy initialization, to move communication and computation migration overhead to the background. The CCG replacement algorithm (§4) enables existing machines to recycle their current CCGs, replacing only the new delta CCG connections while guaranteeing the new CCGs maintain identical performance. Additionally, it divides the joiners' new CCG setup into two parts: overlappable setup (e.g., intra-machine connections) and non-overlappable setup (e.g., inter-machine connections), minimizing the non-overlap setup as much as possible to reduce downtime. Sandbox lazy initialization (§5) creates emulated environments for joiners and warms them up in the background so that they can compute at full pace after joining the new cluster. By moving most events away from the migration critical path, TrainMover migrates the workload with minimum JCT overhead and zero memory overhead (§6).

## 4 Communication Member Replacement

Current state-of-the-art CCLs do not support member replacement, *i.e.*, replacing one or multiple members in an established collective communication group (CCG). This inflexibility brings high migration overhead since we can only destroy the old CCL group and re-initialize a new one to change the membership. Replacing the member in a CCG can bypass redundant setup steps and reduce the migration time. In this

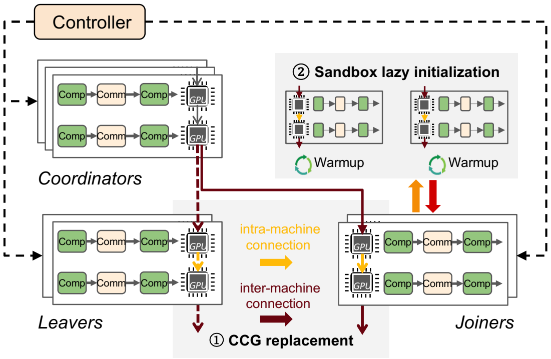


Figure 3: TrainMover Overview

| NCCL Setup Component | | CUDA_VISIBLE_DEVICE Flag | |
|---|---|---|---|
| | | With | Without |
| Network bootstrap | | 2.50s (3.24%) | 2.48s (6.86%) |
| Topology discovery and computation | | 8.48s (11.01%) | 9.1s (25.16%) |
| Connection Establishment | Intra-machine | 62.03s (80.53%) | 20.52s (56.73%) |
| | Inter-machine | 4.02s (5.22%) | 4.07s (11.25%) |

Table 2: Time breakdown of NCCL setup components with and without the CUDA_VISIBLE_DEVICES flag.

section, we first deep-dive into the CCG initialization process in NCCL, one of the most adopted CCLs (§4.1), and explain how TrainMover implements CCG member replacement with the minimum downtime (§4.2).

## 4.1 NCCL CCG Initialization

Given the participants, a NCCL CCG initialization goes through the following steps:

- **Network bootstrap.** One of the CCG participants collects all other participants' network addresses and guides them in constructing a TCP-based ring connection channel.
- **Topology discovery and computation.** Each participant collects local device metadata (*e.g.,* NIC bandwidth, NVLink capability) and shares it using the ring all-gather algorithm. Given the gathered information from everyone, each participant decides its predecessors and successors.
- **Connection establishment.** According to the graph, each participant reserves GPU memory for buffering and creates the *inter-machine* and *intra-machine* connections. *Inter-machine* connections commonly use GPU Direct RDMA (GDR), while *intra-machine* ones use NVLink.

Table 2 presents the setup overhead of each NCCL component on an 8-machine cluster with 64×40GB A100 GPUs, training a GPT-10B model configured with TP=4, PP=2, DP=4, and distributed optimizer enabled. This cost involves setting up 7 communication groups (DP, PP, and TP groups) and demonstrates that the total setup time ranges from 30s to 80s in the 8-machine setting. Intra-machine connection establishments take longer because there are more intra-machine connections than inter-machine connections in a ring or tree-based topology for an 8-GPU machine.

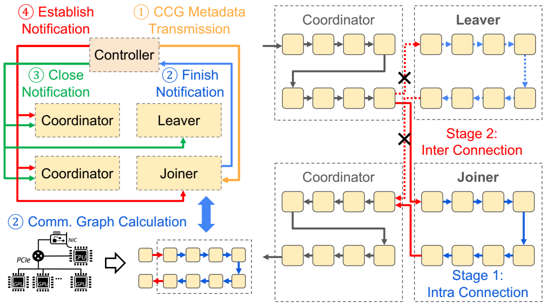


Figure 4: CCG replacement workflow

We noticed that the cost of establishing intra-machine connections can be exceptionally high when `CUDA_VISIBLE_DEVICES` is enabled. This flag is commonly used to pin a specific process to a GPU. When enabled, this flag stops the GPU from seeing others on the same server. Therefore, when the GPU wants to create an intra-machine connection with other GPUs, CUDA has to go through a 3x slower, trial-and-error-based code path to establish a connection with every other GPU until one succeeds, compared with the previous query-and-establish code path.

CCL's long initialization time creates huge JCT overhead for each migration. Our idea is to reuse the existing established CCG and focus on the delta introduced by the replacement. The CCG's communication graph is critical to its performance. The goal of TrainMover's replacement algorithm is (1) to keep the replaced computation graph the same as the one reconstructed from scratch to avoid performance overhead and (2) to minimize replacement downtime and avoid unnecessary JCT overhead.

## 4.2 CCG Member Replacement

**Communication graph generation.** We observed that NCCL's communication graph exhibits a regular pattern. Participants first construct intra-machine subgraphs individually, based on their host architecture. These subgraphs are then connected according to the global ranks assigned by the operator. For the sake of simplicity and performance [3][1], NCCL does not take the inter-machine topology (e.g., oversubscription ratio) into account.

TrainMover introduces a two-step replacement algorithm to leverage such a pattern and avoid communication between joiners and leavers to reduce replacement downtime. Firstly, the joiners compute the intra-machine subgraph using NCCL's original algorithm, which yields the same result as the original subgraph since the machines are identical. Secondly, for each joiner, TrainMover controller finds its global rank in the CCG by matching it with the corresponding leaver and connects the joiner's subgraph with the up and downstream ranks. This algorithm guarantees the new graph to be the same as the ground truth.

**Two-stage connection establishment.** Table 2 shows that the

[1] Supporting more advanced algorithms is discussed in §9.

阻塞方式执行

intra-machine connection establishment consumes 5 times more initialization time than the inter-machine ones. For the joiners, this is a necessary and crucial step and cannot be bypassed. We noticed that it is self-contained and does not require interaction with other participants. Following this insight, TrainMover splits the intra and inter-connection establishment separately into two stages and exposes explicit APIs to the training framework. TrainMover can initialize the intra-machine connections on the joiners while the original CCG is untouched and still functional in the first stage and only the second stage, inter-machine connection establishment, is on the critical path. The `CUDA_VISIBLE_DEVICES` flag no longer delays CCG initialization.

**CCG replacement workflow.** Now we present the CCG Member replacement workflow, as shown in Figure 4.

1. Given a CCG and the leavers and joiners in the CCG, TrainMover controller first sends each joiner the CCG metadata, *i.e.*, the previous CCG participants and the leavers.
2. Each joiner calculates the communication graph, initializes the intra-machine communication connections, and notifies the controller when finishes.
3. The controller notifies the leavers and coordinators to remove the inter-machine connections between them.
4. The controller notifies the joiner and coordinators to establish new connections.

Each of the steps can fail because a joiner or leaver could be unresponsive. Therefore, TrainMover executes each step in a blocking manner, ensuring it does not proceed until confirmation is received from all involved participants. During execution, TrainMover monitors the CCG's status on each participant. When any step fails, TrainMover can re-issue the command if the CCG on all participants is still alive. Otherwise, TrainMover falls back to re-initializing the entire CCG.

# 5 No-Time-to-Be-Lazy

Migration joiners introduce unforeseeable computational lazy initialization bottlenecks (§5.1) that can slow down the training process after migration. To address this issue, we propose sandbox lazy initialization (§5.3 and §5.2) to minimize downtime and eliminate interference with ongoing training.

## 5.1 Lazy Initialization

Lazy initialization is essential in distributed training, particularly in Python-based frameworks like PyTorch. These initialization processes are typically one-time, value-independent operations, triggered during the first execution. For example, computational graph compilation, where the computation graph is constructed lazily during the first model forward pass; model parameters and optimizer initialization, where optimizer memory is allocated only when the first model update occurs, as evaluation-based forward passes do not require optimizer initialization; CUDA initialization overhead, which involves GPU-specific setups like context creation and kernel loading; and JIT compilation optimizations, runtime hooking, and other caching behaviors, which enhance runtime performance by dynamically compiling frequently used operations during execution. In the same GPT-10B experiment used in Table 2, where the normal iteration time (from the second iteration onward) is approximately 6.8 seconds, the first iteration time can increase by more than 6×, reaching around 44 seconds due to lazy initialization, excluding the NCCL instantiation time.

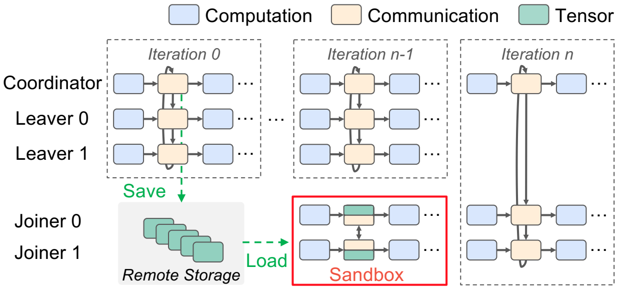


Figure 5: Sandbox lazy initialization workflow

Yet, these lazy initialization processes are difficult to identify or predict—they occur across different programming layers, exhibit varying behaviors, and depend heavily on the underlying libraries. The only practical way to uncover them is to run a real training iteration. However, running an iteration is not a single-handed task—it requires other existing machines to spare valuable GPU cycles and incurs additional synchronization costs to manage the coordination process.

## 5.2 Overlapped Lazy Init within Sandbox

To enable the joiner to run an actual training iteration *independently* and achieve the goal of minimizing training downtime, we introduce sandbox lazy initialization. This approach overlaps and triggers lazy initialization during ongoing training, avoiding interaction with existing machines and preventing degradation in ongoing training performance.

Running a successful training iteration independently is challenging, as any training iteration execution requires: (1) *valid training states* to ensure that code execution aligns with user-defined paths—using fake training states may result in NaN tensors or assertion failures in the training scripts—and (2) *communication operations* during the iteration, as even a barrier communication call requires others to respond.

Our solution for sandbox-based lazy initialization addresses these two challenges and is illustrated in Figure 5. When the training job is launched, we intercept NCCL collective calls on every GPU and record the tensors during the first iteration. These recorded tensors are stored alongside the checkpoint data in the remote storage. The recording occurs only during the first iteration; afterward, the training proceeds as normal, and the recording interception hook is removed.

When a migration occurs, all new joining machines are

第一个iteration，存储张量

placed into a sandbox, isolating them from the existing hosts. The initial state of model parameters, optimizer states, and the recorded tensors are loaded into the sandbox. During the first iteration of the new joiner (e.g., Joiner 0 and 1), we intercept communication calls intended for existing machines and emulate the recorded tensors to directly respond to these calls. For barrier calls, we ignore them and return immediately, allowing the process to proceed seamlessly. This approach enables the joiner to complete the lazy initialization process individually because (1) the emulated data is valid, as it was recorded during the first iteration when training began, and (2) all communication is handled through emulation, avoiding any halting issues.

After the warmup is complete, we replace the Joiner 0 and 1 with the original Leaver 0 and 1 at the end of iteration $n-1$, *enabling a machine transition with no lazy initialization required after the migration is finished.*

### 5.3 What to Record; What to Emulate

Recording and emulating tensors incur overhead in both storage usage and tensor loading time. If all communications are recorded, the storage requirements can become substantial (e.g., reaching TB levels for a 39.1B model), leading to increased storage pressure and significant I/O overhead.

Fortunately, not all communications need to be recorded and emulated. During a training iteration, two types of communication take place: intra-machine communication and inter-machine communication. For intra-machine communication, these connections are already established and operational during the CCG first stage (§4), allowing the new joining machine to use intra-machine connections during the first iteration directly. This eliminates the need to store communication tensors for intra-machine connections. Additionally, as illustrated in the lower part of Figure 5, only the communication touching the edges of the sandbox in the training graph relevant to the new joiner needs to be loaded into the sandbox and replay. This design is particularly beneficial when multiple machines are migrated together as a batch (e.g., an entire PP group). Combining these, the tensor storage and loading requirements for a 39.1B model are reduced to approximately 300GB.

Beyond that, lazy initialization only requires the emulated data to be *valid*. Certain data, such as gradients, can be set to zero during warmup, as gradients primarily indicate the direction of training and do not affect the correctness of the training process at this stage. This approach can further push tensor emulation overhead close to zero — although this tradeoff introduces the risk of warmup failures due to the use of fake (zero) values, TrainMover is designed to fall back and discard the lazy initialization entirely if such failures occur, ensuring robustness. In our experiments, even when tensors are not recorded at all and are instead emulated with zero values, Megatron-LM training continues for over 100 iterations without issues or halting.

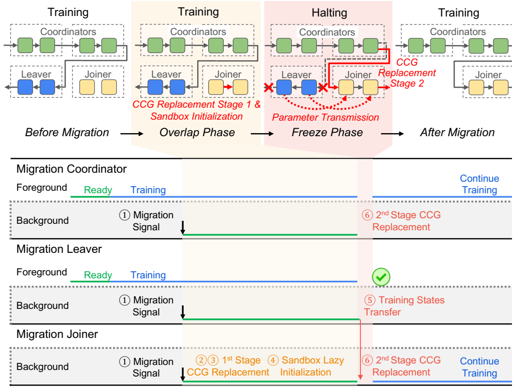


Figure 6: TrainMover workflow for live migration

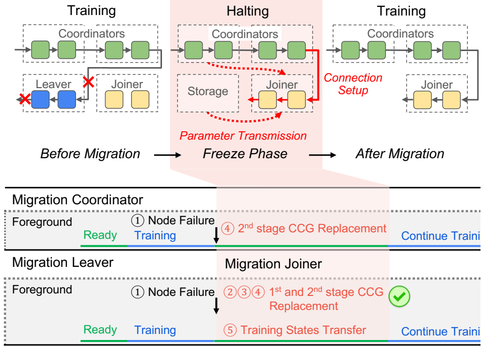


Figure 7: TrainMover workflow for unexpected failure

## 6 TrainMover Workflow

We describe TrainMover's workflow, which combines collective communication group replacement (§4) and sandbox-based lazy initialization (§5) to minimize downtime and achieve memory-overhead-free migration, in Section §6.1. Additionally, we demonstrate how TrainMover handles unexpected failures (§6.2) and leverages pre-heated machines to reduce expected downtime (§6.3).

### 6.1 Migration Lifecycle

The entire migration process consists of two phases: the *overlap phase* and the *freeze phase*, as depicted in Figure 6 with orange and red colors. Migration begins with the *overlap phase*, during which the migration coordinator and leaver continue the training process while simultaneously assisting the joiner in handling live migration tasks in the background. The *freeze phase* follows the *overlap phase*, requiring the foreground training process to pause temporarily to complete procedures that cannot be overlapped. This pause contributes to TrainMover's downtime.

The overlap phase begins immediately after receiving the migration signal ①. At this stage, the migration joiner initiates the *two-stage NCCL instantiation* (§4), during which it receives the communication collective group (CCG) metadata from the controller and sets up the first-stage CCL processes ②. These processes include establishing intra-machine com-

**分页借用（Page Borrowing)**: TrainMover利用目标节点上未使用的内存页来存储迁移数据，从而避免了在目标节点上为迁移作业分配额外内存的需要。
**增量检查点（Incremental Checkpointing）**:TrainMover在迁移过程中定期保存训练状态的增量检查点，以便在迁移失败时能够快速恢复。
**内存映射（Memory Mapping）**:TrainMover使用内存映射技术，使得源节点和目标节点能够共享相同的内存页，从而减少了数据传输量。

munication, calculating the topology, and allocating memory ③. Subsequently, the joiner begins the *sandbox lazy initialization* (§5), allowing it to independently warm up ④ while the coordinator and leaver continue with frontend training. This method effectively eliminates the cold-start overhead associated with adding a new machine.

After the joiner completes the first-stage NCCL setup and lazy initialization, the migration transitions into the *freeze phase*. During this phase, the migration leaver begins transmitting training states, such as model parameters and optimizer states, to the joiner ⑤. Concurrently, the joiner and coordinator establish the remaining CCL *inter-machine* connections, replacing the original inter-machine connections with those linked to the migration joiner while ensuring zero memory overhead ⑥. These leaver-joiner mappings are 1-to-1, meaning the state transmission and connection establishment processes for different joiners are independent. As a result, the overhead does not scale with the training size or the number of machines being migrated. Once these steps are completed, the migration process concludes, with the leaver exiting and the joiner taking over the training role to continue training.

### 6.2 Handling Unexpected Failure

TrainMover also handles unexpected failure events. Once the failed event is identified by monitoring systems, the controller launches a replacement machine from the pool of idle machines to restart the job from scratch (an accelerated recovery using preheated machines is discussed in §6.3).

The recovery procedure begins as depicted in Figure 7. While it largely aligns with the live migration process, there is a key difference: all tasks originally performed during the *overlap phase* are shifted to the *freeze phase*, as all remaining machines must stall and wait for the recovery process to complete before resuming training.

The migration joiner first establishes the initial stage of the CCL setup ②③. Since the recovery path lies entirely within the critical path, there is no need for sandbox lazy initialization before proceeding to the second stage of CCL initialization. The CCL initialization is completed as a unified process ④. Subsequently, the machine begins the training state recovery procedure ⑤.

During the recovery procedure, the approach varies depending on whether the data is recoverable. TrainMover checks for redundancy among the existing machines. If redundancy is available (e.g., when the distributed optimizer is disabled), the system performs a fast recovery by retrieving model parameters and optimizer states from the redundant machines. If redundancy is not available, TrainMover falls back to retrieving the data from a remote storage checkpoint. This adaptive strategy ensures robust recovery across diverse scenarios.

### 6.3 Preheat Standby Machine

All procedures originally performed in the *overlap phase* are shifted to the *freeze phase* during unexpected failures, leading to increased downtime. However, enabling tasks to be completed on standby (preheated) machines in advance can significantly reduce this downtime.

Our two main components—two-stage CCG replacement and sandbox lazy initialization—are designed to operate independently, relying primarily on machine local information. This independence makes them well-suited for use on standby preheated machines. Specifically, the first stage of CCG replacement sets up intra-machine communication, which can be completed ahead of time, as training job patterns are fixed and inter-machine communication typically follows a predetermined ring structure. Similarly, sandbox lazy initialization only requires recorded tensors, which can be utilized immediately after the first iteration of training begins.

Standby machines are commonly available in cloud environments or from GPU machines launched with other low-priority or preemptible jobs [8,41]. Since failures affecting multiple machines are rare—and typically, only a single machine fails at a time [13,34]—using a few spare machines to enhance throughput reliability is a reasonable trade-off. This approach ensures that migration procedures remain in the *overlap phase* without shifting to the *freeze phase*, thereby significantly minimizing downtime.

## 7 Implementation

TrainMover is implemented in C and Python, consisting of two main components: the training node and the controller. The code will be open-sourced after publication.

**Training Node.** TrainMover 's training node builds on Megatron-LM, with modifications to Megatron-LM, PyTorch, and NCCL. In Megatron-LM, we added a backend agent to manage migration signals and maintain NCCL ordering during overlapping training and migration, preventing blocking or interference. PyTorch's `c10d` layer was updated to support multiple global NCCL groups, enabling flexible setup and management. We extended a two-step initialization API and introduced a TrainMover intercept layer to aid the warmup process by recording, replaying, or bypassing tensors, depending on the lazy initialization mode. Additionally, new APIs allow fine-grained NCCL control, including inheriting or replacing channels from existing communicators.

**Controller.** The controller assigns roles, synchronizes migrations, and detects node failures. Migration agents establish channels with the controller at startup. During migration, the controller helps the joiner, leaver, and coordinator set up CCGs, ensuring proper ordering and avoiding NCCL deadlocks from dependency conflicts.

# 8 Evaluation

Our experiments show that TrainMover reduces downtime by up to 16× compared to baselines across model scales and parallelism settings, maintaining stable performance across migration scales (§8.2). It outperforms in handling stragglers, rebalancing, and failure recovery, improving training efficiency by up to 15% (§8.3). Detailed breakdowns reveal *zero* memory overhead and the benefits of each design component (§8.4).

## 8.1 Experiment Setup

Our testbed consists of 4 GPU machines, each with 8 Nvidia A100 GPUs, totaling 32 GPUs. Each machine connects to an 800 Gbps training network (8 Mellanox CX-6 adapters) and a 200 Gbps management and checkpointing network (2 Mellanox CX-6 adapters). By default, training runs on 3 machines, with 1 machine migrated to the 4th during experiments unless stated otherwise. The checkpoint storage is mounted remotely, with network bottlenecks eliminated unless specified. We disable `CUDA_VISABLE_DEVICE` flag by default.

**Baseline.** Our primary baseline is Megatron-LM, which includes built-in checkpointing for saving and loading models. We consider the following checkpointing approaches:

*Megatron-LM Per-iteration*: A per-iteration checkpointing system that assumes checkpoint saving is cost-free and can always be overlapped within a single iteration [15]. It also assumes that no progress is lost at any point of shutdown. This represents an idealized checkpointing approach. However, this method is known to incur very high I/O overhead [38].

*Megatron-LM Save-and-Restart*: A more practical baseline. Before shutdown, training stops and waits for the checkpoint to be saved. Subsequently, the reboot process begins, followed by loading the checkpoint and resuming training. Since the checkpoint is saved prior to shutdown, we also assume no progress is lost in this system. Checkpointing with different frequencies is explored in the experiments (§8.3).

We also compare TrainMover with two state-of-the-art reconfiguration-based systems: Oobleck [17] and Parcae [8]. Both Oobleck and Parcae reconfigure the training profile when failures or machine changes occur. Parcae does not support tensor parallelism, and neither Oobleck nor Parcae support distributed optimizers [30], as their designs rely on redundancy within data parallelism to reconfigure online.

**Model Settings and Dataset.** We evaluate four models—GPT-Medium [6], GPT-2.7B [6], GPT-20B [15], and GPT-39.1B [21]—to cover a range of model sizes, with GPT-20B and GPT-39.1B as primary representatives. Models are tested with various tensor parallel (TP), data parallel (DP), and pipeline parallel (PP) configurations, including (TP1, PP8, DP3), (TP4, PP8, DP3), and (TP8, PP8, DP3). Default profiles are: GPT-Medium and GPT-2.7B use TP1, PP8, DP3, a global batch size of 96, and a microbatch size of 2; GPT-20B uses TP1, PP8, DP3, the distributed optimizer, a global batch size of 36, and a microbatch size of 1; GPT-39.1B uses TP4, PP2, DP3, the distributed optimizer, a global batch size of 36, and a microbatch size of 1. The Wikitext dataset [23] is used, aligning with prior work [8, 17].

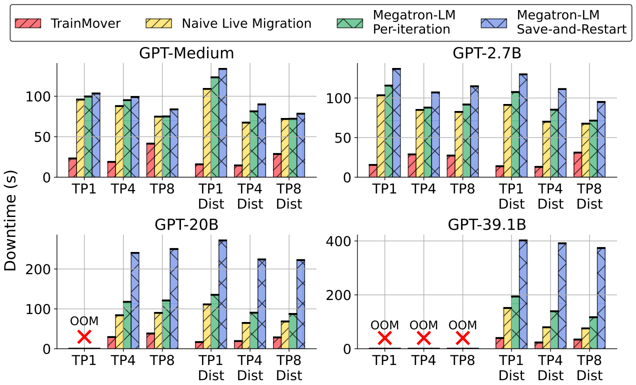


Figure 8: Machine downtime with different models and parallel settings

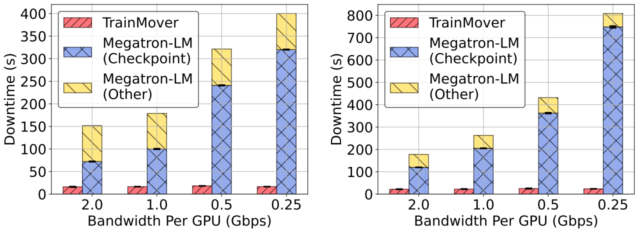
## 8.2 System Performance

We present the downtime comparison in Figure 8. *Naive live migration* refers to the implementation described in Section §2.3. The difference between *naive live migration* and checkpointing systems lies in whether the model states are directly loaded from an existing machine or retrieved from remote checkpoints. Two checkpointing baselines are also included for comparison.

Megatron-LM Save-and-Restart performs the worst among all baselines due to significant checkpoint saving overhead, which escalates with model size, reaching 400 seconds for the GPT-39.1B model. Naive live migration surpasses Megatron-LM Per-iteration by directly transferring model states from migration leaver to joiner machines, achieving up to a 1.745× speedup on the GPT-39.1B model with TP4 and DO enabled. *Naive live migration* outperforms Megatron-LM Per-iteration by enabling direct transfer of model states from migration leaver machines to migration joiner machines. This optimization achieves up to a 1.745× speedup on the GPT-39.1B model with TP4 and DO enabled.

TrainMover delivers the best performance by overlapping NCCL instantiation and warmup processes with the non-critical path. This design achieves up to a 16× speedup compared to the Megatron-LM Save-and-Restart baseline for the GPT-39.1B model. Both TrainMover and the naive system maintain zero additional memory overhead during migration.

**Eliminating Network Bottleneck.** TrainMover significantly alleviates network bandwidth bottlenecks during machine changes. LLaMA reports [13] their storage system bandwidth ranging from 2 TB/s to 7 TB/s while serving approximately 7,500 machines, translating to an effective bandwidth of 0.267 GB/s to 0.933 GB/s per machine. In this experiment, we evaluate checkpointing overhead under bandwidth ranging from 0.25 GB/s to 2 GB/s per GPU, encompassing the bandwidth range reported by LLaMA.



(a) GPT-20B

(b) GPT-39.1B

Figure 9: Downtime comparison with different bandwidth per GPU configuration

In both GPT-20B (Figure 9a) and GPT-39.1B (Figure 9b) setups, TrainMover maintains stable overheads of 16–18 seconds, due to parallel one-to-one state transfers between migration leaver and joiner. By contrast, Megatron-LM, which requires all GPUs to pull checkpoints from remote storage, incurs increasing overhead as bandwidth decreases and model size grows. Specifically, checkpoint loading overheads reach 320 seconds for GPT-20B and  750 seconds for GPT-39.1B at 0.25 GB/s per GPU.

**Migrating Multiple Machines at Once.** Migrating multiple machines is essential for rapid large-scale operations like rebalancing and maintenance [1]. Figures 10a and 10b show the performance of GPT-20B and GPT-39.1B during migrations involving 4% to 33% of 32 GPUs, each treated as an individual machine.

Both models maintain stable downtime overheads:  27 seconds for GPT-20B and  36 seconds for GPT-39.1B. This consistency stems from TrainMover 's design, where each migration leaver and joiner operates in parallel, performing one-to-one data transfers. This approach ensures constant overhead, preserving efficiency regardless of migration scale.

In contrast, checkpointing systems like Megatron-LM are highly inefficient. Even migrating just 4% of machines forces all machines to reboot and retrieve checkpoints from remote storage, causing a 7× to 8× increase in overhead.

**Migration at Scale.** Figures 11a and 11b evaluate TrainMover's performance on 3 to 8 AWS p4d.24xlarge instances during a training job. In each run, one machine is migrated to a new instance. Given the 40 GB memory of AWS p4d.24xlarge GPUs—half the capacity of our primary testbed GPUs—we use a GPT-10B model (half of GPT-20B) while keeping all other settings unchanged. Migration downtime is also compared with and without the `CUDA_VISIBLE_DEVICES` flag.

Our findings reveal that in TrainMover, migration downtime consistently remains at approximately 17 seconds, regardless of whether the isolation flag is enabled. This consistency occurs because the isolation flag primarily impacts intra-machine connection establishment, which TrainMover effectively overlaps. The downtime also remains unaffected by the number of machines in the system, as TrainMover only updates the connections between the migration leaver and migration joiner, leaving all other connections unaffected, ensuring that the system's scale does not influence downtime.

In contrast, Megatron-LM restarts the entire job during migration, requiring NCCL group re-initialization across all machines. With the isolation flag enabled, this procedure adds an additional 100 seconds of overhead. While NCCL's connection establishment is highly parallel, overhead still increases slightly due to the growing number of connections, resulting in a 1.2× rise as shown in Figure 11b when scaling from 3 to 8 machines. As the system scales further, TrainMover's downtime is expected to remain constant, whereas Megatron-LM's overhead will continue to grow.

## 8.3 Use Cases

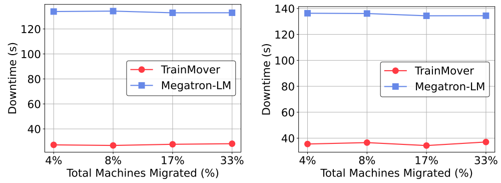
**Handling Straggler.** We examine the impact of different checkpoint handling strategies during straggler events, as illustrated in Figure 14. Alongside the previously evaluated *Per-iteration* and *Save-and-Restart* approaches, we include two additional strategies for broader developer consideration: deferred checkpointing and restart with progress loss.

During a straggler event, developers can choose from the following approaches. *Save-and-Restart*: save a checkpoint immediately and restart training. *Defer-50/100*: continue training and defer restarting until the next scheduled checkpoint (e.g., every 50 or 100 iterations). *Restart-50/100*: restart training immediately, with progress loss based on the most recent checkpoint at iteration 50 or 100.

Our experiments evaluate additional job completion time (JCT) for these approaches when a straggler causes a 20% slowdown [40] at the 75th iteration of a 100-iteration job. The results provide insights into how each checkpointing strategy affects overall training efficiency and recovery time in the presence of stragglers.

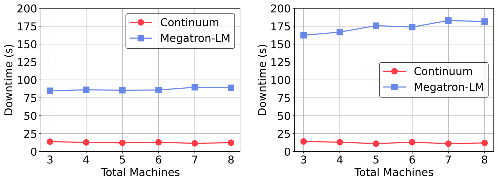
We demonstrate that TrainMover incurs only 37 seconds of additional job completion time (JCT), the lowest among all approaches, and achieves performance close to the ideal training scenario. During the straggler's slowdown period, TrainMover allows training to continue while the migration process is underway, minimizing downtime when switching from a straggler machine to a new machine. Defer-type baselines endure reduced performance until the next checkpoint, saving and restarting only afterward, causing delays. Restart-type baselines do not have checkpoint-saving overhead but lose progress based on the last checkpoint's distance, resulting in up more than 20× higher JCT than TrainMover.

Figure 13 illustrates the complete performance spectrum of training efficiency when a straggler occurs at any point from the 1st to the 100th iteration. TrainMover consistently outperforms all baselines, maintaining superior efficiency without relying on specific checkpointing strategies. Remarkably, TrainMover even exceeds the ideal per-iteration checkpointing system, improving training efficiency from 78% to 93%. For Restart-type baselines, performance peaks immediately after a checkpoint since progress loss is minimal. However,



(a) GPT-20B (b) GPT-39.1B

Figure 10: Downtime varying different migration scale



(a) Without flag (b) With flag

Figure 11: Downtime varying different total machines

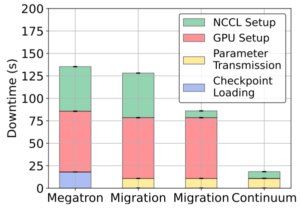


Figure 12: Design Breakdown

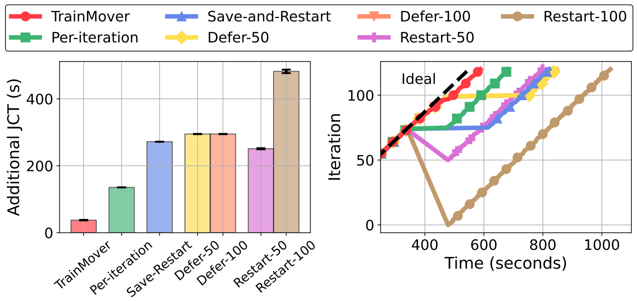
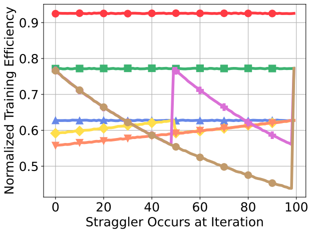


Figure 13: GPT-20B model with 20% slowdown starting at the 75th iteration



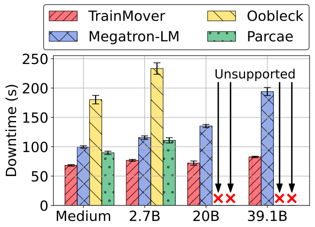


Figure 14: Straggler occurs at different iteration

Figure 15: Unexpected failure downtime

as the distance from the last checkpoint grows, performance degrades significantly due to increased progress losses.

**Rebalancing / Maintenance.** TrainMover handles machine changes swiftly, making it ideal for rebalancing or maintenance scenarios requiring frequent GPU or network updates. Table 3 compares the training efficiency of TrainMover, Megatron-LM per-iteration checkpointing, Oobleck, and Parcae across various model scales and training parameters.

During machine changes, online reconfiguration systems like Oobleck sometimes perform worse than the checkpointing system Megatron-LM. This is because reconfiguration involves starting new machines, tearing down and reestablishing NCCL groups, and performing warmup procedures, which saves little time compared to checkpoint-based systems and performs even worse if the system is not well-engineered. For instance, with the Oobleck 20B TP8 model, reconfiguration delays result in negligible progress with rebalncing every 10 minutes. Additionally, Oobleck and Parcae lack support for distributed optimizers, and Parcae also does not support tensor parallelism, limiting their scalability for larger models with the same GPU count.

TrainMover delivers the best performance among all approaches, maintaining a training efficiency of 0.93 when running the GPT-39.1B model with a distributed optimizer. In comparison, the efficiency drops to 0.68 when using Megatron-LM per-iteration checkpointing. This table highlights that even in high-frequency (10-minute) rebalancing scenarios, TrainMover loses no more than 10% of performance, making it a robust and efficient solution.

**Unexpected Failure Handling.** TrainMover can effectively handle unexpected failures. For the Medium and 2.7B models without the DO enabled, TrainMover achieves the best performance among all baselines. This is because, when a failure occurs, TrainMover overlaps the NCCL instantiation time, warmup time, framework initialization and state transmission. In contrast, Megatron-LM executes all components sequentially after rebooting, resulting in TrainMover having 1.46× less downtime.

Parcae and Oobleck utilize redundancy within the data parallelism group, enabling direct state transfer during failures and reducing checkpoint loading time compared to remote retrieval. However, both systems still incur additional overheads, including rebooting new machines, initializing framework components, reinstantiating NCCL groups, and performing warmup. These steps result in higher downtime compared to checkpoint-based approaches.

For GPT-20B and GPT-39.1B models, where the DO is enabled and redundancy is eliminated, the state can only be recovered from remote checkpoints during unexpected failures. TrainMover demonstrates 1.88× and 2.35× shorter downtime compared to Megatron-LM for the GPT-20B and GPT-39.1B models, respectively. This overlapping advantage becomes increasingly significant as model sizes grow.

## 8.4 Design Deep Dive

We next demonstrate zero memory overhead by detailing the migration procedure and breaking down our design benefits.

**Zero Memory Overhead Workflow.** Figure 16 illustrates the network (top) and memory (bottom) usage during a migration in TrainMover. A key step in the process is state transfer. As shown in the network breakdown, the leaver sends traffic to the joiner around the 530th second, resulting in high Rx bandwidth utilization on the joiner. However, model transfer is not without costs. Each leaver-joiner pair requires a new NCCL group for state delivery, which consumes GPU memory. To achieve zero memory overhead, TrainMover im-

| | | Enable Distributed Optimizer | | | | Disable Distributed Optimizer | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Med. | 2.7B | 20B | 39.1B | | Med. | 2.7B | 20B | 39.1B |
| TrainMover | TP1 | 0.97 | 0.98 | 0.97 | 0.93 | TP1 | 0.96 | 0.97 | OOM | OOM |
| | TP4 | 0.98 | 0.98 | 0.97 | 0.96 | TP4 | 0.97 | 0.95 | 0.95 | OOM |
| | TP8 | 0.95 | 0.95 | 0.95 | 0.94 | TP8 | 0.93 | 0.95 | 0.94 | OOM |
| Megatron-LM | TP1 | 0.79 | 0.82 | 0.77 | 0.68 | TP1 | 0.83 | 0.81 | OOM | OOM |
| | TP4 | 0.86 | 0.86 | 0.85 | 0.77 | TP4 | 0.84 | 0.85 | 0.92 | OOM |
| | TP8 | 0.88 | 0.88 | 0.85 | 0.81 | TP8 | 0.87 | 0.85 | 0.96 | OOM |
| Oobleck [17] | TP1 | / | / | / | / | TP1 | 0.70 | 0.61 | OOM | OOM |
| | TP4 | / | / | / | / | TP4 | 0.62 | 0.49 | 0.03 | OOM |
| | TP8 | / | / | / | / | TP8 | 0.57 | 0.49 | 0.00 | OOM |
| Parcae [8] | TP1 | / | / | / | / | TP1 | 0.85 | 0.82 | OOM | OOM |
| | TP4 | / | / | / | / | TP4 | / | / | / | / |
| | TP8 | / | / | / | / | TP8 | / | / | / | / |

Table 3: Training efficiency comparison with GPU/Network changes every 10 minutes

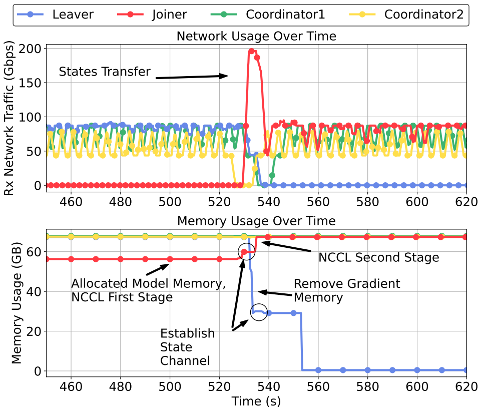


Figure 16: Network traffic and memory usage timeline during Migration

plements specific optimizations during state transmission:

The migration leaver does not need to continue training after the migration is complete, it can free up and repurpose the pre-allocated GPU gradient buffer for use as the NCCL transmission channel at around the 530th second. On the migration joiner, during the first stage of NCCL initialization, inter-machine connections are not fully established, creating temporary memory availability at the 530th second. This allows the state transfer channel to operate without exceeding memory limits. Once the transfer is complete, the channel is immediately destroyed, freeing memory before the second stage of NCCL initialization. This process ensures zero memory overhead during migration.

**Design Breakdown.** We break down our designs and evaluate their incremental impact on system performance in Figure 12, using the GPT-20B profile. In this setup, Megatron-LM's total downtime is approximately 130 seconds, with checkpoint loading, GPU warmup, and NCCL setup times accounting for 13%, 51%, and 36% of the total, respectively.

*Migration Naive*, represents a naive migration system that excludes the two-layer NCCL designs and lazy initialization designs. Unlike Megatron-LM, which loads checkpoints from remote storage, it retrieves parameters directly from the leaver to the joiner. This approach eliminates checkpoint loading time, reducing it by approximately 2×. The benefit becomes even more pronounced as model sizes increase.

*Migration NCCL$^+$* builds on the naive migration baseline by introducing the NCCL two-layer designs. This design significantly reduces the critical path, leaving only the second stage of NCCL initialization in the critical path. Consequently, NCCL time decreases from 50 seconds to just 6 seconds. More detailed on NCCL memory can be found in Appendix A.

The full system, *TrainMover*, further integrates the self-detached warmup design, effectively eliminating GPU warmup time. This enables training to resume immediately after migration, reducing downtime by more than half to less than 20 seconds. This final step completes the design.

## 9 Related Work and Discussion

**Fault Tolerance Systems.** Fault tolerance systems [8, 10, 12, 17, 34] handle node changes by adjusting configurations such as batch size, pipeline stages, or data parallelism. While flexible, these adjustments increase management costs, disrupt optimized large-scale training tailored to efficiency and network topology [11, 19, 27, 39], and add uncertainty for developers, reducing training efficiency. At large scales, significant adjustments can severely degrade performance. In contrast, TrainMover maintains training settings unchanged and operates transparently to users during live migration. Moreover, reconfiguration after failure often relies on redundancy, which is increasingly impractical as trends shift toward reducing redundancy (e.g., Microsoft Zero [30, 31], PyTorch FSDP [42]). Redundancy assumption no longer universally valid as systems evolve.

**Multiple-Iteration Lazy Initialization.** In Section §5, we discuss communication tensor recording for the first iteration. For complex models like MoE with branching code paths, we support user-defined multi-iteration recording and replaying to enhance code path coverage and lazy initialization. Uncovered code does not affect correctness, as this is a warmup procedure.

**Advanced CCL algorithms.** Recent works [22, 32] have proposed better CCL graph computation algorithms considering inter-machine topology and outperforming NCCL's ring and tree-based algorithm. In the worst case, the CCG graph for such algorithms can change drastically after the replacement and may require the establishment of intra-machine connections in the coordinators. On the one hand, production clusters are commonly homogeneous and the network provides full bi-sectional bandwidth, the worst case does not happen often. On the other hand, TrainMover can fall back to a full CCG rebuild if the delta is too large.

**GPU-granularity migration.** TrainMover's implementation supports GPU-granularity migration. The training framework can still benefit from sub-iteration level JCT overhead and zero memory overhead. However, we need to forgo the optimizations introduced in §5.3 and pay extra storage and I/O overhead.

## 10 Conclusion

We develop TrainMover, a live migration system that leverages CCL replacement and sandbox lazy initialization in the non-critical path to minimize downtime. TrainMover incurs no memory overhead and avoids interfering with ongoing training. It effectively handles stragglers, maintenance, rebalancing, and failures, outperforming SOTA checkpointing and reconfiguration systems by up to 16× in reducing downtime.

## References

[1] Maintaining large-scale AI capacity at Meta. https://engineering.fb.com/2024/06/12/production-engineering/maintaining-large-scale-ai-capacity-meta/, 2024.

[2] Megatron-LM Github Repository. https://github.com/NVIDIA/Megatron-LM, 2024.

[3] NCCL Github Issue 1340 - Network Topology awareness. https://github.com/NVIDIA/nccl/issues/1340, 2024.

[4] 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*, EuroSys '22, page 472–487, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[7] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[8] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. Parcae: Proactive, Liveput-Optimized DNN training on preemptible instances. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1121–1139, Santa Clara, CA, April 2024. USENIX Association.

[9] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, Renton, WA, April 2022. USENIX Association.

[10] 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*, SOSP '24, page 211–228, New York, NY, USA, 2024. Association for Computing Machinery.

[11] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez

Fernandez, Shashidhar Gandham, and Hongyi Zeng. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 57–70, New York, NY, USA, 2024. Association for Computing Machinery.

[12] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. Enabling parallelism hot switching for efficient training of large language models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 178–194, New York, NY, USA, 2024. Association for Computing Machinery.

[13] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily

Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models, 2024.

[14] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. Just-in-time checkpointing: Low cost error recovery from deep learning training failures. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 1110–1125, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. Just-in-time checkpointing: Low cost error recovery from deep learning training failures. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 1110–1125, New York, NY, USA, 2024. Association for Computing Machinery.

[16] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[17] 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, New York, NY, USA, 2023. Association for Computing Machinery.

[18] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei,

Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 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, Santa Clara, CA, April 2024. USENIX Association.

[19] Apostolos Kokolis, Michael Kuchnik, John Hoffman, Adithya Kumar, Parth Malani, Faye Ma, Zachary DeVito, Shubho Sengupta, Kalyan Saladi, and Carole-Jean Wu. Revisiting reliability in large-scale machine learning research clusters, 2024.

[20] Oleksii Kuchaiev, Jason Li, Huyen Nguyen, Oleksii Hrinchuk, Ryan Leary, Boris Ginsburg, Samuel Kriman, Stanislav Beliaev, Vitaly Lavrukhin, Jack Cook, Patrice Castonguay, Mariya Popova, Jocelyn Huang, and Jonathan M. Cohen. Nemo: a toolkit for building ai applications using neural modules, 2019.

[21] Shengwei Li, Zhiquan Lai, Yanqi Hao, Weijie Liu, Keshi Ge, Xiaoge Deng, Dongsheng Li, and Kai Lu. Automated tensor model parallelism with overlapped communication for efficient foundation model training, 2023.

[22] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. Rethinking machine learning collective communication as a multi-commodity flow problem. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 16–37, New York, NY, USA, 2024. Association for Computing Machinery.

[23] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

[24] 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*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 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*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[27] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 691–706, New York, NY, USA, 2024. Association for Computing Machinery.

[28] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[29] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[30] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[31] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[32] 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, Boston, MA, April 2023. USENIX Association.

[33] 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.

[34] 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, Boston, MA, April 2023. USENIX Association.

[35] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[36] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[37] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[38] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. Bytecheckpoint: A unified checkpointing system for large foundation model development, 2024.

[39] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, Boston, MA, April 2023. USENIX Association.

[40] 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, 2024.

[41] Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric Friedman, Scott Shenker, and Ion Stoica. Can't be late: Optimizing spot instance savings under deadlines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 185–203, Santa Clara, CA, April 2024. USENIX Association.

[42] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023.

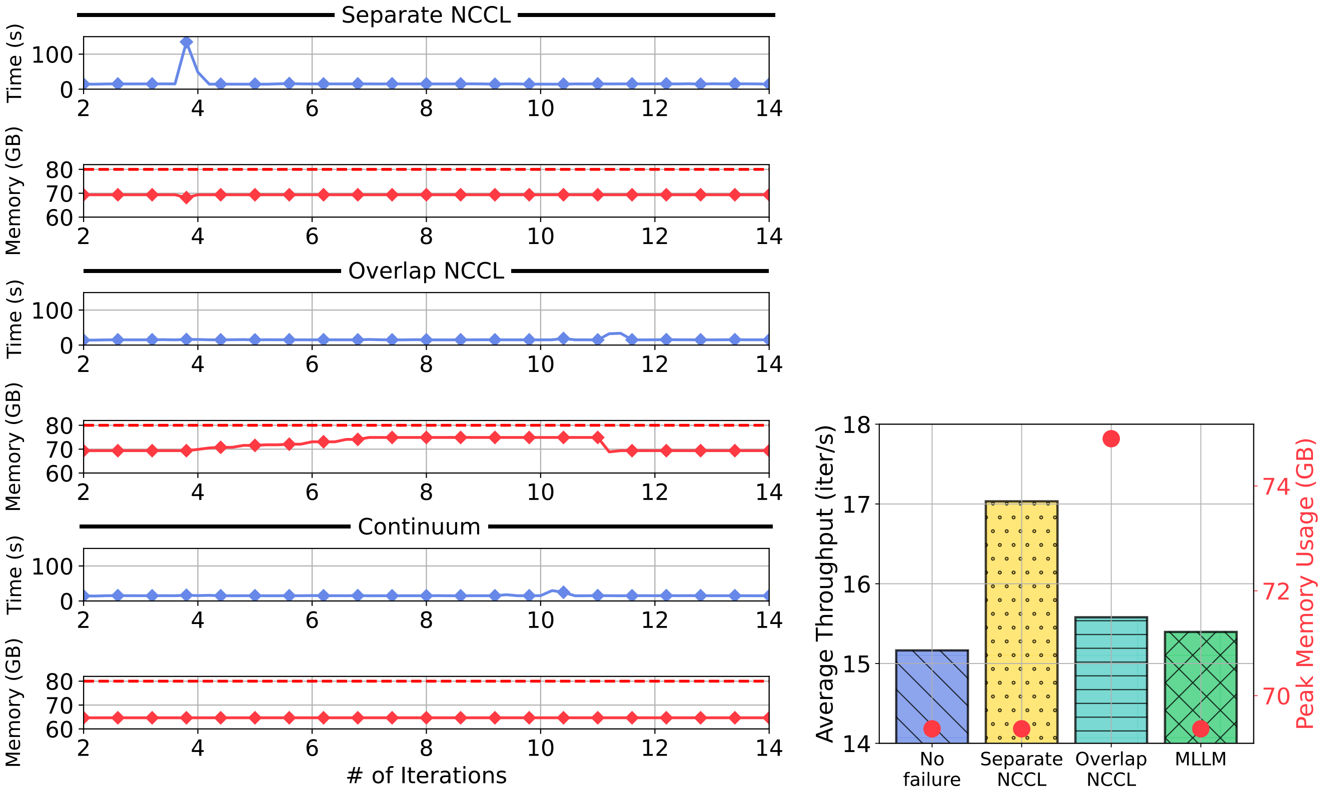
# A NCCL Decision Choice

Our NCCL design achieves zero memory and minimal downtime overhead by overlapping all non-critical path establishment costs. In Figure 17a, we show the iteration time (including NCCL re-setup time) and GPU memory usage across three different ways of handling changes in NCCL group participants during migration. (1) *Separate NCCL* involves completely destroying and re-instantiating NCCL groups whenever the group composition changes. This approach is used by Oobleck, Parcae, Bamboo, and all existing systems that rely on native PyTorch support. (2) *Overlap NCCL* is another baseline where we modify PyTorch to allow multiple global groups to exist simultaneously, enabling new members to join within a separate NCCL context. (3) TrainMover NCCL is our design, which incorporates a two-stage approach and reuses existing primitives.

For *Separate NCCL*, downtime increases to approximately 8× the duration of normal training around the 4th iteration, when migration occurs and participant changes take place. However, GPU memory usage remains unchanged, as new groups are initialized only after the old groups are destroyed. In contrast, *Overlap NCCL* experiences a small downtime around the 11th iteration, when the final migration completes and the old NCCL groups are removed (migration spans from the 4th to the 12th iteration in the backend). However, this design incurs a high memory overhead, increasing from 71GB to 77GB, because many new NCCL groups (e.g., DP/TP/PP groups) must be initialized, and there are two sets of groups (frontend and backend) existing simultaneously.

TrainMover's NCCL design achieves zero memory overhead and only a small downtime around the 10th iteration, when migration completes. This is because the second stage of NCCL instantiation (inter-machine connections) occurs on the critical path, requiring the old inter-machine NCCL connections to be destroyed and rebuilt with new participants within a short period. The NCCL reuse mechanism ensures zero memory overhead during this process.

In summary, as shown in Figure 17b, our approach performs closely to the no-failure case, without introducing any additional memory overhead. In contrast, both the separate and overlapping NCCL approaches fall short, either compromising throughput (0.88×) or incurring additional memory overhead.



(a) Timeline

(b) Throughput and Peak Memory

Figure 17: time and memory cost timeline for different NCCL design decision