FlowCheck: Decoupling Checkpointing and Training of Large-Scale Models

Zimeng Huang[1,2*], Hao Nie[2,3*], Haonan Jia[2], Bo Jiang[1†], Junchen Guo[2], Jianyuan Lu[2], Rong Wen[2], Biao Lyu[4,2], Shunmin Zhu[5,2], Xinbing Wang[1]

[1]Shanghai Jiao Tong University, Shanghai, China
[2]Alibaba Cloud, Hangzhou, China
[3]Peking University, Beijing, China
[4]Zhejiang University, Hangzhou, China
[5]Hangzhou Feitian Cloud, Hangzhou, China

{lukehuang,bjiang,xwang8}@sjtu.edu.cn,alibaba_cloud_network@alibaba-inc.com

Abstract

Checkpointing is becoming a hotspot of interest in both academia and industry as the primary fault-tolerance method for large model training. However, existing checkpoint designs are tightly coupled with the training process, leading to interruptions that reduce overall training efficiency. To reduce the impact of checkpoints on training, this paper presents FlowCheck, a novel checkpointing system that decouples checkpoint operations from the training process, enabling checkpoint saving without blocking the training. Specifically, FlowCheck updates the checkpoints by extracting complete gradient information from the network traffic of normal training. FlowCheck deploys a traffic-mirroring network to support this design. To utilize mirrored traffic for checkpointing operations, two key challenges need to be addressed. First, we need to achieve precise identification and extraction of gradient packets from training traffic. Second, the transmission on the mirror link is unreliable due to its inability to trigger retransmission upon packet loss. Through two key designs: (1) packet-counting-based traffic identification, and (2) packet redundancy recovery mechanism, FlowCheck implements an efficient checkpointing system using the existing training network and solves the above two challenges. Experiments and estimations verify that FlowCheck achieves checkpoint operations with zero impact on training, and demonstrate that FlowCheck achieves over 98% effective training time under practical fault conditions.

CCS Concepts: • **Networks** → **In-network processing**; **Network monitoring**; Data center networks; • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Computing methodologies** → **Machine learning**.

Keywords: Distributed Training, Fault Tolerance, In-network Checkpoint

ACM Reference Format:
Zimeng Huang, Hao Nie, Haonan Jia, Bo Jiang, Junchen Guo, Jianyuan Lu, Rong Wen, Biao Lyu, Shunmin Zhu, Xinbing Wang. 2025. FlowCheck: Decoupling Checkpointing and Training of Large-Scale Models. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3696088

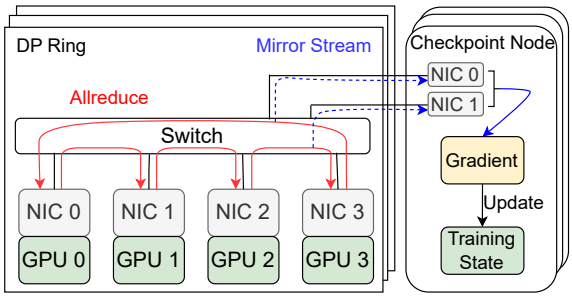

Figure 1. The architecture of FlowCheck. We use a 4-node DP ring to demonstrate how FlowCheck mirrors the training traffic and updates the checkpoint. The red curve represents the allreduce communication traffic, while the blue dashed curve represents the mirror stream traffic.

1 Introduction

With the continuous increase in the scale of deep learning (DL) models, the size of AI infrastructure providing services for DL tasks is also growing rapidly. Recent studies [12] have revealed that large language models (LLMs) are experiencing exponential growth, expanding at a staggering rate of 410

*Both authors contributed equally to this work.
†Bo Jiang is the corresponding author.

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696088

times every two years. Megascale [21], the cutting-edge production system for AI training at ByteDance, has surpassed the scale of 10,000 GPUs. However, faults in the AI accelerator cluster, such as overheating, power failure, or memory error, can lead to training interruption and the loss of all parameters in volatile accelerator memory [19]. This results in frequent interruptions of large-scale training tasks and the need to restart training from an intermediate state stored in checkpoints. According to the training logbook from OPT-175B[44], a cluster composed of 992 GPUs experienced approximately two failures and rollbacks per day on average.

Existing solutions integrate training and checkpointing in each worker, resulting in training efficiency reduction due to checkpointing. Methods such as Pytorch Checkpoint [24] and CheckFreq [26] initiate checkpoint communication actively from the training nodes, necessitating a pause in training during the checkpoint operation and thereby introducing overhead to the training process. This limitation prevents frequent checkpoint saving, leading to training failures and wastage of computational resources. According to the statistics from current training log books [4, 35], a checkpoint operation for a thousand GPU training task takes approximately ten to fifteen minutes to complete, and the optimal checkpoint saving frequency is approximately once per hour. Under these settings, approximately 178,000 GPU hours were wasted during the training of OPT-175B due to the blocking of checkpoint operations and rollback calculations for failures [44]. The checkpointing system Gemini [38] tries to eliminate the checkpoint overhead by hiding the checkpoint traffic in the network gaps of training nodes, and it increases the checkpoint frequency to once per iteration to enhance overall training efficiency. Although Gemini claims zero-overhead checkpointing, its method of scheduling checkpoint transmissions by analyzing traffic idle time windows still poses a risk of traffic conflicts and may slow down communication during training.

To reduce the impact caused by checkpoint operations in training, this paper proposes FlowCheck, a network-based checkpointing system that decouples checkpointing from training in large training clusters. Specifically, the design of FlowCheck is based on the observation that the allreduce communication in data parallelism (DP) contains gradient data used for model updates. In large-scale training scenarios, this allreduce traffic passes through the data center's leaf switches. This provides FlowCheck with the opportunity to leverage the mirror function of the switches to intercept the training traffic and extract the gradient information needed for checkpoint updates. As shown in Figure 1, FlowCheck deploys dedicated checkpoint nodes within the GPU cluster network to collect mirrored communication traffic between training nodes and maintain the latest checkpoints. These checkpoint nodes can be CPU-based and do not require any additional communication with the training nodes during the execution of checkpoint operations, thus avoiding the checkpointing burden on the training nodes.

Checkpointing through the capture and analysis of training traffic raises two key problems that need to be addressed. First, **how to extract complete gradient data from inter-node communication traffic?** In the normal training process, apart from the allreduce communication for transmitting gradient data, the communication between training nodes still includes many other data such as the traffic due to other types of parallelism (e.g., model parallelism). However, due to the limitation of the switch mirror function commonly adopted in current data centers [21, 35], the traffic mirroring operation cannot perform selective traffic mirroring. Therefore, it is necessary to analyze the packets in the mirror stream in order to extract gradient information from the mirrored data flow. Considering the fixed relationship between the traffic of each communication operation and the associated workload, FlowCheck differentiates between various communication streams by counting the total number of packets received. Additionally, the entire checkpoint processing workflow must be completed within a single iteration, to release occupied resources for newly arriving traffic from the subsequent iteration. To this end, FlowCheck employs a pipelined packet processing design.

Second, **how to handle packet loss on mirror links?** Since the mirror link lacks any retransmission mechanism in case of packet loss, it does not establish a reliable connection between the switch and the checkpoint node. When packet loss occurs, it is possible that we cannot recover the complete gradients, leading to checkpoint failure. Inspired by the transmission characteristics of the allreduce algorithm [13, 29], FlowCheck leverages partial overlapping gradients sent by multiple nodes, thereby mitigating the impact of packet loss. We demonstrate that utilizing this approach can reduce the expected mean checkpoint failure rate from once every 100 iterations to once every 10^9 iterations.

The design of FlowCheck leverages the ubiquity of data parallelism in large-scale model training. As long as data parallelism is utilized for training and its traffic passes through network switches, the design of FlowCheck remains effective. Since these requirements are consistently met in large-scale model training [21, 32, 44], FlowCheck should be widely applicable in such settings. This paper focuses on common practices for large-scale model training in the industry [4, 7, 9, 33, 39], where training is static and synchronous with fixed computing resources. FlowCheck does not make any assumptions about the accelerators used for training. In this paper, we conducted experiments specifically for NVIDIA A100 GPUs, but the design of FlowCheck still allows for the support of other accelerators such as AMD MI300X [5].

To sum up, FlowCheck makes the following contributions:

- To the best of our knowledge, FlowCheck is the first system that decouples checkpointing from training to achieve non-blocking checkpoint saving.

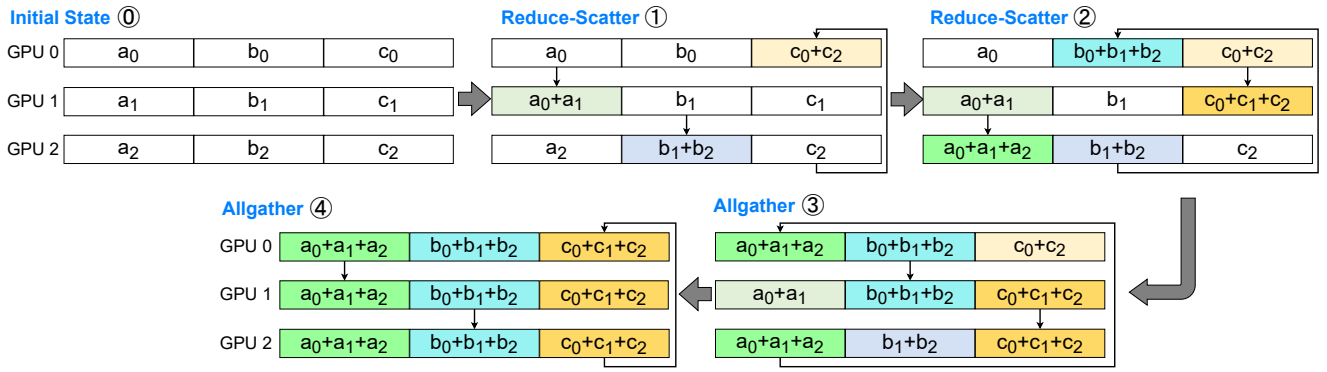

Figure 2. Illustration of ring allreduce. Assuming there are 3 GPUs, the parameters are divided into 3 chunks, i.e. a, b, and c. We use subcripts to label GPUs. For instance, a_i means the partial gradient for chunk a initially residing on GPU i. In each step, the concurrent data transfer follows the fixed pattern of a ring GPU 0 → GPU 1 → GPU 2 → GPU 0. In step ① of reduce-scatter, GPU 0 sends chunk a_0 to GPU 1, which aggregates a_0 and a_1 to produce $a_0 + a_1$. Chunks b and c are processed in a similar fashion. In step ②, GPU 1 sends $a_0 + a_1$ to GPU 2, which produces the full gradient $a_0 + a_1 + a_2$ for chunk a. Similarly, at the end of reduce-scatter, GPU 0 and GPU 1 have the full gradients for chunks b and c, respectively. During the allgather phase, the full gradients of different chunks are sent to other GPUs. For example, in step ③, GPU 2 sends $a_0 + a_1 + a_2$ to GPU 0, which then forwards it to GPU 1 in step ④. At the end of allgather, all GPUs have the full gradients of all chunks.

- A training traffic parsing algorithm, which extracts gradient data in complex training traffic while ensuring the efficiency of checkpoint operation execution.
- A redundancy-based recovery system tailored for mirror links, which ensures that FlowCheck can still recover complete gradient data from mirrored traffic under the current level of packet loss in the existing links.
- We open source the entire software implementation of FlowCheck at https://github.com/AlibabaResearch/flowcheck-eurosys25.

2 Background and Motivation

2.1 Distributed Training

In distributed training, DNN practitioners need to make two major decisions, i.e., determining how to divide tasks among workers (parallelism) and how to gather computing results from workers (collective communication). In this paper, we focus on data parallelism and its corresponding collective communication operation: allreduce.

Data parallelism. In data parallelism, each GPU operates on its assigned portion of the data by dividing the dataset into smaller chunks, distributing these chunks across the GPUs, and having each GPU perform the same computation on its subset. During each training iteration, GPUs calculate gradients on their local data and then communicate these gradients to update the model parameters. Data parallelism is often used because of its established efficiency [30, 46].

For large-scale models, data parallelism is often combined with other parallel strategies, such as pipeline parallelism [11, 17, 27] and tensor parallelism [28, 32]. In these scenarios, model parameters are partitioned across various DP groups, each group maintaining an identical subset of the model's parameters. A DP group consists of workers that process different data inputs while ensuring model parameters consistent through synchronization. After each iteration, an allreduce operation synchronizes the global gradient data within these DP groups. Subsequently, all DP groups receive the global gradients and update parameters concurrently, ensuring model parameter synchronization. The parameter update process of the t-th iteration can be expressed as follows:

$$\boldsymbol{W}_t = O(\boldsymbol{W}_{t-1}, \Delta_t), \tag{1}$$

where W_t is the model parameters at the t-th iterations, $O(\cdot)$ is the optimizer function for parameter update, e.g. SGD [48] or ADAM [23], and Δ_t is the global gradient of parameters.

Additionally, based on the existing training setup in large-scale workloads [36, 40], each DP group's communication traffic flows through the leaf switches in the data center network (DCN). This is primarily because DP groups are deployed across nodes, communication traffic between nodes cannot be directly transmitted via NVLink [3], and must be forwarded through the DCN switches. Consequently, in large-scale training scenarios, data center switches can commonly forward the DP traffic during the training process.

Ring allreduce. One of the most widely used algorithms for allreduce is the ring allreduce algorithm [13, 20, 34, 42]. In this paper, our focus is on the discussion of ring allreduce, whereas the exploration of other allreduce algorithms will be detailed in §9. As shown in Figure 2, the ring allreduce process is divided into two phases: reduce-scatter and allgather, each involving $N - 1$ communication steps, where N is the

number of nodes in the ring. Initially, each node holds a gradient tensor of identical size. The tensor is then evenly divided into N smaller chunks. During each step of the reduce-scatter process, node i sends one chunk to node $i+1$ and receives one from $i-1$. Then each node applies a reduction on its received chunk. After $N-1$ repetition, every node holds $\frac{1}{N}$ of the fully reduced data. In the allgather phase, each node collects all chunks through a similar pattern, but without reductions. Following $N-1$ exchanges, all chunks are reassembled, providing each node with the complete, aggregated gradient, thus completing the allreduce operation.

2.2 Existing Checkpoint Solutions and Limitations

Existing checkpointing methods all require the training nodes to initiate communication periodically for the storage of checkpoints, which imposes extra overheads. CheckFreq [26] uses the host memory to dump the checkpoint, but it still stops training while transmitting the checkpoint from the GPU to the host memory. For better performance, Gemini [38] utilizes the host memory of local and neighboring nodes for checkpointing, and schedules the inter-node checkpoint traffic to occur during the intervals between the training traffic of successive iterations. The intervals are estimated by profiling the first few iterations, but they may vary between different iterations, with a standard deviation of up to 10%. When the checkpoint traffic does not fit in the gaps, Gemini needs to reduce the checkpoint frequency to amortize the overhead incurred by hindering training.

In addition, some live migration efforts [31] provide solutions for migrating node states, but these transparent live migration methods are not checkpointing techniques and cannot handle unexpected failures.

2.3 Opportunities for Checkpointing in Network

During the allreduce communication process in distributed training of large models, all gradient data needed for model updates are transmitted across the network to other nodes. If we can capture all gradient data packets through the network and achieve reassembly of the gradient data, following Eq. (1), we theoretically could implement checkpointing that does not impede the normal training operations of the GPU. This observation presents an excellent opportunity to leverage network traffic for the implementation of checkpoint operations that are entirely decoupled from the training process. In turn, reducing the cost of checkpoints means they can be more frequent.

We can leverage the port mirroring functionality built into the data center network switches to capture communication traffic during the training process. By deploying a few dedicated checkpoint nodes under the network switches and configuring the switches' mirroring rules to direct training traffic to the checkpoint nodes, we create the opportunity to implement decoupled checkpoint operations. Notably, these checkpoint nodes can be CPU-based, as they do not require GPU resources. However, as illustrated in Figure 2, capturing only one node's inbound or outbound traffic in the ring, we would ultimately obtain only $\frac{N-1}{N}$ of the gradient data, where N is the total number of nodes in the ring. This is because the ring allreduce algorithm transmits fully aggregated gradient data through the network only during the allgather phase, in which each node sends $\frac{1}{N}$ of the gradient data per communication operation, with a total of $N-1$ such send operations being performed. For node i, monitoring solely its inbound traffic will result in the loss of the gradient data block with index $i+1$, while monitoring exclusively its outbound traffic will lead to the omission of the gradient data block with index $i+2$. Consequently, capturing the unidirectional traffic of a single node alone is insufficient to obtain the complete set of gradient data. To get the complete gradient data, it is essential to monitor at least two different inbound or outbound ports.

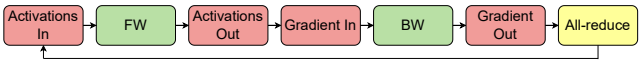

Figure 3. Various communication traffic during training. If model parallelism is enabled, inbound and outbound communications between stages are required before and after the forward and backward computations of each iteration.

3 Challenges

3.1 Gradient Packets Identification

Due to the limitations of port mirroring in current data center switches, which replicates all traffic without distinction, achieving FlowCheck's design requires extracting gradient data from all the communication traffic between GPU nodes. For communication operations between GPU nodes, the five-tuple in a data packet (source and destination IP addresses/port numbers, and protocol type) uniquely defines a network connection. Therefore, we can use five-tuple filtering to isolate training-related traffic from the overall GPU network traffic. As depicted in Figure 3, beyond allreduce traffic, there may exist traffic due to model parallelism during the actual training process, necessitating differentiation for node communication to gather gradient. Moreover, distinguishing between the reduce-scatter and allgather phases of allreduce is essential, as fully aggregated gradient data is only transmitted during the allgather phase (see §2.1). However, identifying relevant information from data packets is challenging, as they do not contain application-layer details, making it difficult to determine the transmission phase from a single packet.

Furthermore, to guarantee non-blocking checkpoint operations at the iteration level, there is a time deadline for CPU-only checkpoint server nodes to update checkpoints. These nodes are required to ensure that each checkpoint update is executed before the onset of communication for the next iteration. Failure to meet this deadline may result in the checkpoint nodes missing a portion of the communication

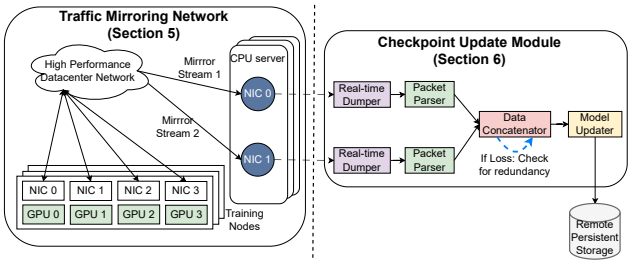

Figure 4. Overview of FlowCheck. FlowCheck consists of the mirroring network module and the checkpoint update module. A minimum of two mirror streams are required to capture the complete gradients. Additional mirror streams can increase the robustness against packet loss.

packets for the subsequent iteration. This requirement imposes performance demands on the speed of gradient packet dumping and parsing, requiring rapid processing to meet iteration deadlines.

3.2 Packet Loss in Mirroring

Mirrored checkpoint traffic packets are vulnerable to loss during transmission. Unlike traditional RDMA link packet loss, as the checkpoint node does not establish a direct RDMA connection with the monitored training node, we cannot rely on the retransmission mechanisms of the RDMA to resolve the issue. This could ultimately result in the inability of the checkpoint node to collect complete gradient information, thereby preventing normal checkpoint updates.

One straightforward approach is that checkpoint nodes verify an aggregation of the total length of gradient data received upon the completion of communication in each iteration. Based on the gradient data that is identified as missing, these checkpoint nodes issue retransmission requests to the GPU training nodes. Such an implementation, however, requires that all GPU nodes wait and guarantee the receipt of all gradient data by the checkpoint nodes after each iteration, which introduces additional synchronization overhead into the training process. Therefore, how to ensure data reliability without significantly impacting the normal operation of the GPU training cluster presents a challenge that needs to be addressed in the design of FlowCheck.

4 System Overview of FlowCheck

We propose FlowCheck, a system that decouples checkpoint saving operations from training by capturing and analyzing the communication during the training process. Figure 4 illustrates FlowCheck's architecture that consists of two modules: 1) the mirroring network module at the network level and 2) the checkpoint update module at the software level. The mirroring network module defines the networking design for the training GPU nodes and checkpoint server nodes. The checkpoint update module specifies how each checkpoint server node recovers gradients from the monitored training mirror stream and performs checkpoint updates.

Mirroring Network Module. FlowCheck employs a multi-level storage architecture for checkpointing. Unlike other multi-level checkpoint system designs [21, 35, 38], FlowCheck does not store checkpoint data in the local or neighboring node's host memory. Instead, it adds CPU-only checkpoint server nodes into the training network and utilizes these nodes for checkpoint storage through mirrored traffic. Specifically, each checkpoint server node stores the model state (i.e., model checkpoint) corresponding to the training nodes it monitors. Once the complete gradient data for each iteration is collected through mirrored traffic, these checkpoint nodes perform model updates locally, thereby ensuring the real-time consistency of the checkpoint.

Checkpoint Update Module. FlowCheck initializes the model parameters and optimizer states for each checkpoint node, ensuring consistency with the corresponding training node. After receiving mirrored traffic, the checkpoint server nodes generate checkpoint data via the checkpoint update module. Each checkpoint server node runs a FlowCheck worker agent which is responsible for loading the checkpoint update module. During training, the mirrored traffic is first analyzed by a state-machine-based packet parser to extract the corresponding model gradient data fragments from each packet (§6.1). To prevent packet loss in the mirrored stream, FlowCheck employs a method of capturing data across multiple nodes to achieve partial data redundancy, thereby further reducing the packet loss rate in the mirrored stream. This approach enhances the overall system reliability in the face of voluminous and complex network traffic (§6.2).

5 The Traffic Mirroring Network

FlowCheck requires the implementation of collecting gradient information from normal training traffic and completing the update of checkpoints on CPU-based checkpoint nodes. It utilizes network switches to mirror the training traffic. The network switch mirroring functionality works by duplicating all incoming or outgoing traffic from a specific port to a designated mirror port. After that, the CPU nodes dump the received mirrored data packets into the host memory in real-time for subsequent gradient data extraction and update. This design allows FlowCheck to achieve checkpoint updates without affecting the normal training of the GPUs. However, it raises two practical difficulties.

Difficulty 1: The Deployment Scale of Checkpoint Nodes. In existing AI data center networks, each GPU is connected to a leaf switch port to facilitate communication between GPUs on different servers. Utilizing the traffic mirroring functionality of network switches requires binding a monitoring port

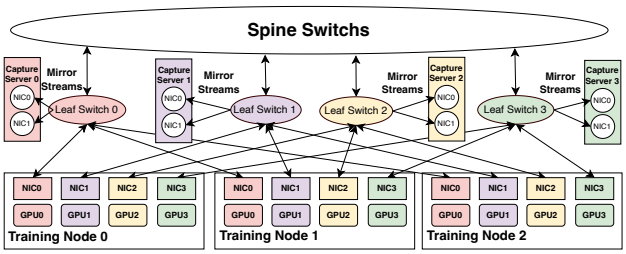

Figure 5. The traffic mirroring network design. We utilize the mirror function of switches to intercept training traffic based on the GPU multi-rail networking architecture.

to a mirroring port on the switch side. The switch generates a mirror stream of all traffic flowing through the monitoring port and delivers it to the mirroring port. Furthermore, considering the characteristics of the allreduce algorithm discussed in §2.3 and existing large model training settings [19, 21, 32], practical deployment must ensure that at least two training nodes' inbound or outbound traffic is captured in each DP group during model training. This implies that at least two checkpoint nodes must be connected to the same leaf switch as the DP group they are monitoring. However, when using tensor parallelism (TP) [28, 32], different GPUs on the same training node are often placed in different DP groups for efficiency, but they may connect to the same leaf switch. To ensure that the infrastructure design is compatible with different workload parallelization setups, a large number of mirror ports need to be configured under each leaf switch. In the worst-case scenario, this will require deploying at least twice the number of CPU nodes as normal training GPUs under each leaf switch, thereby introducing substantial additional costs to the system.

Solution 1: Using Multi-Rail Networking Architecture. We base the placement of CPU nodes on a multi-rail networking structure [16, 21], thereby alleviating the issue of excessive mirror port occupancy. Recent research [36, 40, 41] has demonstrated that employing this approach in GPU collective communication enhances communication efficiency, prompting its adoption in production data center environments [21]. In the GPU multi-rail networking architecture (Figure 5), GPUs with the same rank number from different training nodes are interconnected via leaf switches. In this network configuration, the allocation of DP and TP by current training frameworks [32] ensures that all GPUs within each DP group are connected to the same leaf switch. Notably, maintaining an optimal global batch size in data parallelism is crucial for training accuracy, as excessively large sizes hinder convergence [14, 22]. Existing data center network switches typically have an interface capacity of 64. Assuming a DP group size of 16 in the training setup, only three sets of CPU nodes are needed per leaf switch. Compared to the worst-case scenario mentioned earlier, this design reduces the number of switch ports occupied by CPU nodes and minimizes the impact on the scalability of the training network.

If the leaf network switches still lack sufficient ports to connect CPU checkpoint nodes, one potential approach is to configure the switch to forward packets based on predefined rules, directing packets to the specified locations in the RDMA network according to the quintuple information within the packets. Under this setup, CPU checkpoint nodes can be located anywhere within the RDMA network without the need to maintain a specific relative position with each DP group worker. However, this approach may introduce significantly additional link latency and higher packet loss rates. In §9, we will discuss the possibility of using optical splitters to completely resolve this issue.

Difficulty 2: Real-Time Mirror Packets Dump. Due to the lack of corresponding mapping between the virtual addresses in the mirrored packets and the physical addresses of the CPU checkpoint nodes, we cannot directly utilize the RDMA protocol stack for data offload operations. This results in reading information from each data packet in the NICs becoming a performance bottleneck. Traditional CPU-based data packet dumping approach, such as TCPdump [1], falls short in matching the performance demands of high-speed RDMA networks. Once the dump is not completed in time, the data packets in the receive queue of NICs will be overwritten, resulting in the checkpoint server node being unable to obtain the complete training traffic.

Solution 2: Multiple CPU-side DMA Dump. In FlowCheck, we leverage multiple DMA engines on the CPU side of the checkpoint server node to accelerate packet offload. On the checkpoint server node, we use large-page memory to store data packets of the mirror stream. Multiple DMA engines work collaboratively according to the order of receiving data packets to dump the content of all data packets of the mirror stream into the large-page memory. We experimentally tested the performance of multiple DMA engines conducting concurrent data packet dumping in §8.

6 Checkpointing from Training Traffic

Due to the complexity of mirrored traffic and its potential packet loss, performing checkpoint updates based on training traffic faces challenges outlined in §3. In this section, we explain how FlowCheck achieves reliable extraction of allreduce traffic to ensure the integrity of gradient data.

6.1 Mirror Packet Parser

FlowCheck addresses the challenge of gradient packet parsing mentioned in §3.1 by analyzing training communication patterns and tracking received data packets to identify allreduce packets within the mirrored traffic. Specifically, during the initialization process, FlowCheck calculates the number

of RDMA packets transmitted by each communication operation based on the workload and configures the five-tuple information for all checkpoint nodes to monitor training traffic. Upon receiving packets, FlowCheck employs five-tuple filtering to identify all training RDMA packets. By counting these received training RDMA packets, it determines the transmission phase and the data index of each packet. Once the corresponding position information of the gradient data within each packet is known, FlowCheck can recover the complete model gradient data from the mirrored data stream and complete the checkpoint update on the checkpoint node.

Based on the discussions and observations presented in §3.1, we can identify that the training traffic between GPU nodes during each iteration can be broadly classified into three states: non-allreduce traffic, DP reduce-scatter traffic, and DP allgather traffic. Other traffic between GPU nodes (e.g., health monitoring, usage logs) will be filtered out by five-tuple checking, as mentioned before. With a detailed analysis of the allreduce process depicted in §2.1, we can further break down the two states of allreduce into $2(N-1)$ independent transmission steps, where N represents the size of the allreduce communication group. The complete state transition of the training traffic is illustrated in Figure 6. By inspecting the checksum in the data packets, FlowCheck can detect transmission errors in the training traffic and wait for retransmitted packets to obtain the correct data. When packet loss occurs in the training communication link, FlowCheck is also capable of waiting for the retransmission of packets based on the index of the missing gradient data. FlowCheck sets a threshold for waiting on retransmitted packets (i.e. the number of packets in a complete allreduce step in our implementation). As long as the retransmitted packets are received within this threshold, FlowCheck can correctly retrieve the gradient data. We will discuss in detail the probability of packet loss in mirror links and its corresponding impacts in §6.2. Moreover, packet reordering during transmission does not affect processing, as FlowCheck automatically detects such reordering by analyzing the discontinuities in the packet sequence numbers (PSN) and adjusts the packet counter state accordingly during processing. For workloads that utilize only data parallelism, the total communication volume for non-allreduce traffic is zero. Therefore, this state does not exist in such scenarios. Since the size of model parameters on each worker is fixed, the amount of communication within each iteration is also of a fixed size. Therefore, we can determine the transition between the aforementioned states by analyzing the count of transmitted network packets.

We illustrate this process using the pseudocode provided in Algorithm 1. Assuming the model is trained using fp32 precision, each model parameter's gradient occupies 4 bytes under this setting. Therefore, the total size of the gradients to be transmitted is $4k$, where k is the number of parameters. FlowCheck maintains a packet counter to determine the transmission phase of a packet. The packet counter increments

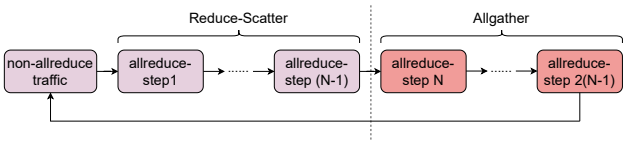

Figure 6. The finite state transition for training traffic analysis. Once the allgather phase is completed, the finite state machine will switch back to the initial state to parse the mirrored traffic for the next iteration.

Algorithm 1 Payload Parsing Algorithm

1: **Inputs:** packets counter c, the calculated num of non-allreduce packets t, the Maximum Transmission Unit (MTU) of NICs m, the Queue Pair (QP) id of current packet q, the size of allreduce group n, packet sequence number (PSN) of current packet p, the total layer num of model d, each layer's parameter size $L = \{l_1, l_2, \cdots, l_d\}$
2: **Outputs:** the layer and index of gradients in the payload, or -1 to indicates the other data
3: **if** $c < t$ **then**
4: **return** -1 {Non-allreduce pkts.}
5: **end if**
6: $total\_pkt\_num \leftarrow 0$
7: **for** i in $[1, 2, \cdots, d]$ **do**
8: $total\_pkt\_num \leftarrow total\_pkt\_num + \frac{l_i \times 4}{m} \times \frac{2(n-1)}{n}$
9: **if** $total\_pkt\_num > c$ **then**
10: $cur\_layer \leftarrow i - 1$
11: **break**
12: **end if**
13: **end for**
14: $single\_block\_size \leftarrow \frac{l_{cur\_layer} \times 4}{m \times n}$
15: $cur\_step \leftarrow \lceil \frac{c}{single\_block\_size} \rceil$
16: **if** $cur\_step < n$ **then**
17: **return** -1 {Reduce-scatter pkts.}
18: **end if**
19: $offset \leftarrow (p - cur\_step \times single\_block\_size) \times m$
20: $cur\_index \leftarrow (cur\_step - (n-1)) \times 4 + offset \times q$
21: **return** cur_layer, cur_index

by one upon receiving each packet. When the packet count reaches the total number of packets calculated by FlowCheck for a single iteration based on the workload, the counter is reset to zero to facilitate packet processing for the next iteration. If the current packet count is less than the total number of non-allreduce packets estimated from the workload size, the transmission state is identified as non-allreduce (lines 3-5). For allreduce traffic, the situation is more complex: since allreduce operations in NCCL [29] are executed layer by layer, we first need to determine which layer the current allreduce operation belongs to (lines 6-13), and then identify the specific transmission step of the allreduce process to distinguish the

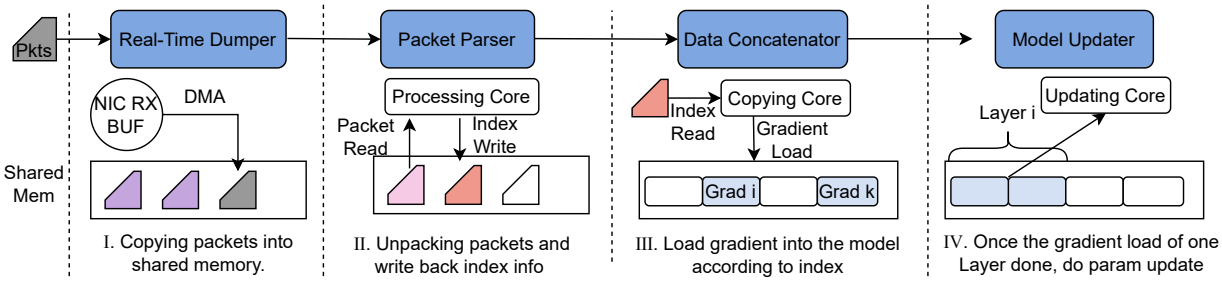

Figure 7. Details of CPU checkpoint updater design.

reduce-scatter traffic (lines 14-18). Once the current transmission step in allreduce is identified, we can infer the location of the gradient data blocks, enabling the accurate reconstruction of the gradient data (lines 19-21).

To expedite packet processing and meet the checkpoint operation deadline in §3.1, FlowCheck incorporates a pipeline-based design to ensure performance. FlowCheck performs several operations on intercepted training traffic: reading from the network card, parsing data packets, loading gradient data, and updating model parameters as shown in Figure 7. In FlowCheck, each pipeline stage is assigned to distinct CPU cores to prevent resource contention. This allows subsequent modules to process packets immediately after the preceding module finishes, without waiting for all packets to complete processing. This design ensures FlowCheck's checkpoint updates are completed within each iteration's deadline, avoiding training process overhead.

6.2 Packet Redundancy-Recover

In §2.3, we mentioned the necessity of monitoring the traffic of at least two training GPUs in each DP group. For each monitored GPU, we can capture $\frac{N-1}{N}$ of the gradient data, where N stands for the DP group size. The traffic sent by these monitored GPUs exhibits a certain level of redundancy, hence we can utilize redundant data to serve as data backup between checkpoint nodes, thereby reducing the likelihood of gradient data loss caused by packet loss in the network links.

As illustrated in Figure 8, if there exist N GPU nodes in a DP ring, when we monitor the inbound traffic of both node i_1 and node i_2 simultaneously, there exist 2 copies for $\frac{N-2}{N}$ of the data, and for $\frac{2}{N}$ of the data there is no redundancy. Further, if we concurrently monitor three nodes i_1, i_2, and i_3, then $\frac{N-3}{N}$ of the data has 3 copies, while $\frac{3}{N}$ of the data has 2 copies. As the mirror links of multiple monitored nodes are independent of each other, the probabilities of packet loss on their links are also mutually independent. Assuming that the packet loss rate of the monitoring link is p and the total number of gradient packets to be sent is K, then in the case where M nodes provide redundant backups for each other's data, the probability of gradient concatenation failure within

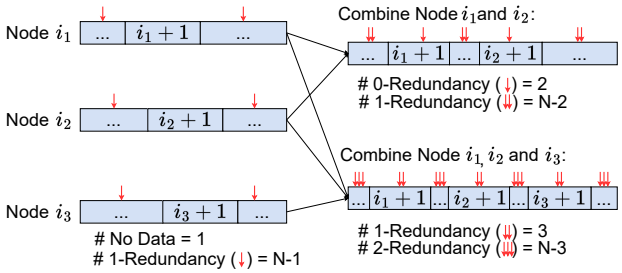

Figure 8. Redundancy of monitored stream. Assuming the DP communication group consists of N nodes indexed from 0 to $n-1$.

a single iteration due to link packet loss is:

$$P(fail)_M = 1 - (1 - p^{M-1})^{\frac{M}{N}K} \cdot (1 - p^M)^{\frac{(N-M)}{N}K} \quad (2)$$

In practical testing, we find that the packet loss rate of the monitoring link is approximately 10^{-9}. For a single A100 40GB GPU, assuming the total communication volume for allreduce reaches the GPU memory limit, under the default settings for NCCL transmission, approximately 16 million packets are sent during the allreduce phase. According to Equation (2), suppose the DP group size is 8, monitoring the traffic of two training GPUs can reduce the probability of gradient data loss in a single iteration to below 4×10^{-3}. Monitoring three GPUs further reduces this probability to 6×10^{-12}. This means gradient data loss occurs roughly once every 1000 iterations with two nodes, and once every 10^{12} iterations with three nodes. Considering that the entire training process of actual large models often requires tens of millions of iterations, in FlowCheck, we recommend capturing traffic from at least three training GPUs in each DP ring and performing mutual data backups to prevent potential packet loss on the mirror link.

During execution, FlowCheck provides consistent initialization of model parameters and optimizer states for all GPU workers and checkpoint servers. As long as the checkpoint server receives the complete gradient data within each iteration, according to Equation (1), FlowCheck can ensure that

both GPU workers and the checkpoint server perform consistent state updates in each iteration, maintaining checkpoint correctness throughout the training process. If FlowCheck still encounters a situation where gradient data cannot be recovered due to link packet loss even with the packet redundancy recovery strategy enabled, the CPU checkpoint node within FlowCheck will send a message to the training framework through the control plane network. This message will notify the framework to perform a regular checkpoint operation on the corresponding checkpoint nodes at the start of the next iteration, thereby ensuring the accuracy of checkpoints. Note that FlowCheck requires the gradients from each iteration, which is essential for ensuring the consistency between the local model states at the checkpoint nodes and those at the training nodes.

7 Implementation

FlowCheck needs to parse the training communication packets on specific CPU checkpoint nodes to support checkpoint updates. We utilized a dual-socket CPU server (featuring two Xeon 8369B CPUs) equipped with two NVIDIA ConnectX-6 NICs as the CPU checkpoint node. Thanks to the utilization of the DMA engine on CPU server, we have developed a DPDK-based application that successfully achieves real-time interception of 100Gbps training traffic. Through meticulous optimization, FlowCheck successfully executes both packet dumping and processing on the CPU within GPU nodes' iteration time, thereby avoiding the necessity for normal training to pause and wait during the checkpoint process.

Throughout training, the CPU checkpoint node by default only saves the latest version of model parameters and optimizer states as checkpoints for quick recovery. At regular intervals, e.g. one hour in our evaluation, these CPU checkpoint nodes transmit the maintained state to a remote storage pool for persistent storage and version control. As a result, the memory requirements for each checkpoint node are modest, and standard CPU servers are sufficient to meet the design requirements of FlowCheck. FlowCheck employs the regular recovery logic of a multi-level checkpoint system [10, 37]: it first attempts to retrieve the most recent level of checkpoints, and only when these are unavailable does it fall back to retrieving checkpoints from more distant levels. In this paper, we do not address specific strategies for setting the save frequency in multi-level checkpointing, as previous studies [8, 10] have already investigated frequency configurations for multi-level systems, which is orthogonal to our work.

Packet Dumper. FlowCheck offloads the workers' copy tasks to the Intel I/O Acceleration Technology (I/OAT) DMA engines [15] to accelerate this process. In our testbed, each Xeon processor is equipped with eight I/OAT DMA engines, collectively meeting the throughput demands of gradient processing. The DMA engines put all packets into queues in MMAP memory regions for each Queue Pair (QP), situated within huge pages. If encrypted communication is used during training, the checkpoint node must be equipped with the appropriate decryption algorithms or hardware to process the packets during the dumping process. A limitation of FlowCheck is that it does not account for encrypted communication, as the current implementation assumes unencrypted traffic.

Model	#Params	Hidden size	#Layer	#AH
BERT	110M	768	12	12
RoBERTa	330M	1024	24	16
GPT-3	1.3B	2048	24	16
Llama2-7B	7B	4096	32	32

Table 1. Models used for evaluation. AH is short for attention heads, M for million, and B for billion.

Pipeline-Based Updater. Upon completion of packet dumping, FlowCheck is tasked with unpacking and reconstructing the payload to form the gradient tensor. As illustrated in Figure 7, we implement these checkpoint updater modules in the code to facilitate the pipelined updating of model checkpoints. The modules within the pipeline communicate through shared memory, with a shared memory size of 50 GB in our implementation. We design a state machine followed in §6.1 as the Packet Parser module to extract the gradients from the received data packets. If packet retransmission occurs, the Packet Parser module will identify the error packets by checksum and only process the retransmitted packets. Once the Packet Parser has completed parsing a data packet, the Data Concatenator inserts the payload into the appropriate position within the model's gradient tensor, using the location information provided for each gradient data packet by the Packet Parser module. In the Data Concatenator module, we utilize the AVX512 instruction set to optimize the copying operation of data from shared memory to model memory space. Once the Data Concatenator has collected the gradient data for a layer, it will notify the Model Updater module to perform the parameter update for the corresponding layer. During this process, the PyTorch model and optimizer constructed during the initialization phase will update the model parameters and optimizer states based on the acquired gradient data. The Updater module incorporates both Python and Cython code, enabling support for native APIs for model updating while accelerating data processing speed.

8 Evaluation

8.1 Setup and Baseline

Testbed Setup. Our testbed includes a H3C S9850-32H switch, two training servers each with Intel XEON 4314 CPUs (16 cores)×2 and NVIDIA A100 (40GB) ×4, and checkpoint servers with Intel XEON 8369B (32 cores)×2. Each checkpoint node has two 100Gbps Mellanox CX-6 RDMA NIC to handle both in/outbound mirror flows and each training node has four 100Gbps Mellanox CX-6 RDMA NIC. All

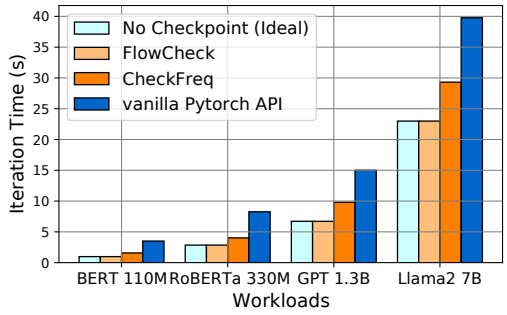

Figure 9. The iteration time under 8-node DP without checkpoint and with different checkpoint mechanisms. "No Checkpoint(ideal)" denotes the duration of a single iteration of normal training without checkpointing.

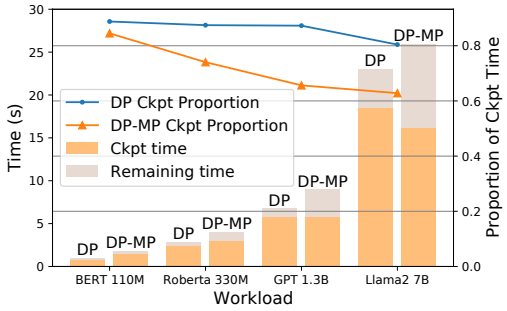

Figure 10. Proportion of FlowCheck's checkpoint time to the interval between two consecutive allreduce operations. Here "DP" refers to the DP-only scenarios and "DP-MP" refers to the DP-MP scenarios. The bar chart indicates the time spent on checkpoint operations and remaining time for the deadline of each iteration, while the line chart depicts the proportion of time spent on checkpoint operations in relation to the total iteration time.

servers are directly connected to the H3C switch. To simulate training scenarios on a larger scale to the greatest extent possible, we employ Docker in our experimental setup to isolate these GPU resources from each other and disable intra-node NVLink communication, thereby emulating the training communication among a maximum of 8 nodes. We will further discuss the scalability and applicability of FlowCheck with estimations in larger-scale scenarios in §8.5.

Workloads. We evaluate FlowCheck with popular and representative large deep learning models, including BERT, Ro -BERTa, GPT-3 and Llama2. We vary the number of layers, hidden sizes, and intermediate sizes in these models. Table 1 shows the detailed model configurations. We use the Adam [23] optimizer for different models. We set the micro-batch size to 32 and we enable the activation recomputation [30, 43] in evaluation. We employ NCCL v2.19.4 as the underlying communication library and utilize the NCCL LL communication protocol during training.

Baselines. We compare FlowCheck with two existing training checkpoint systems: vanilla Pytorch checkpointing and CheckFreq [26]. The Pytorch training framework [24] provides a simple and user-friendly checkpoint API, which is currently the first choice for many DL practitioners for model checkpoint saving. CheckFreq is currently the best performing checkpoint framework available in the open-source academic community. It utilizes the CPU memory space of training nodes as temporary storage for checkpoint data, thereby reducing the overhead of checkpoint operations. We do not compare with Gemini [38] since it is not open-source.

8.2 Training Efficiency

In this subsection, we evaluate the performance of FlowCheck, encompassing the scenario with only Data Parallelism (DP-only) and that with both Data Parallelism and Model Parallelism (DP-MP).

Performance for DP-Only Scenario. We first utilize data parallelism on 8 training GPUs to demonstrate the performance advantages of FlowCheck during checkpoint saving. In our evaluation, we set the checkpoint saving frequency to every iteration. Therefore, the iteration time comprises the duration of normal model training and the time taken to save the checkpoint. Figure 9 illustrates the duration of a single iteration for different baselines under the exclusive use of data parallelism. Unlike other baselines, in FlowCheck, the checkpoint saving operations do not block training, and the single iteration time for executing checkpoints remains the same as the iteration time without performing checkpoint operations. This is because the time interval between each two allreduce operations during training is sufficient to accommodate the completion of all checkpoint collection and saving operations by the designated checkpoint server node in FlowCheck. Figure 10 demonstrates that there is still some idle time remaining after the checkpoint node of FlowCheck completes all checkpoint operations, allowing it to wait for the arrival of the allreduce traffic for the next iteration. Concurrently, it can be observed that as the workload scale increases, the time constraints for FlowCheck to perform checkpoint operations become relatively less stringent. This is primarily attributed to the increasing proportion of time spent on forward and backward computations as the workload scale grows, thereby affording FlowCheck more time for checkpoint processing.

When using the standard PyTorch API to execute checkpoint saving, GPU training must be blocked while model and optimizer states are copied from the GPU to persistent storage, often SSDs with write speeds under 10GB/s, resulting in significant overhead. While CheckFreq aims to optimize this process by leveraging host memory for rapid data offload and overlapping GPU computation with checkpoint transmission,

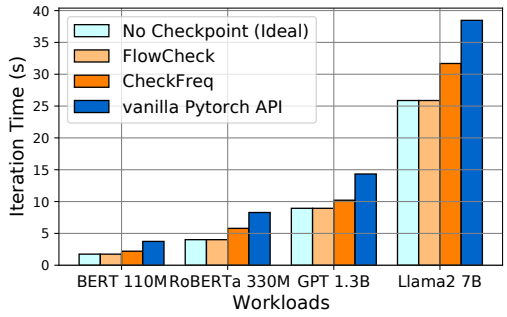

Figure 11. The iteration time under DP-MP without checkpoint and with different checkpoint mechanisms.

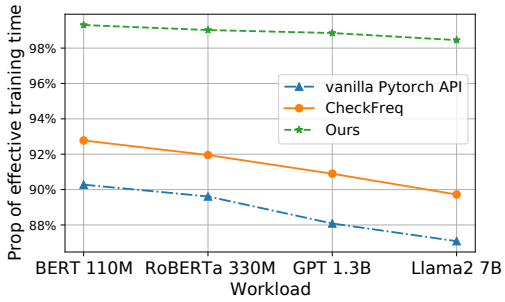

Figure 12. The proportion of effective training time under different workloads. The training utilizes 8 GPUs with data parallelism.

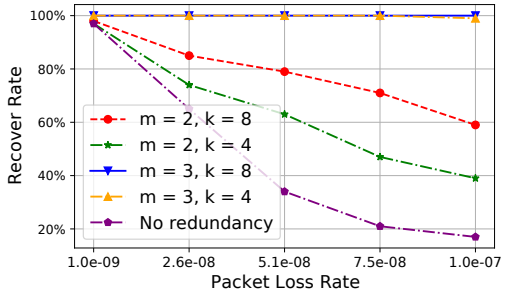

Figure 13. The recovery rate for gradient under different packet loss rates. "m" stands for the count of CPU nodes, and "k" stands for the DP group size.

it still cannot eliminate the blocking caused by GPU-to-host memory copies. This is because, constrained by factors such as CUDA's pinned memory operations and GPU cache [2], which affect PCIe transmission efficiency, the time required to transfer checkpoints between the GPU and CPU is longer than a single GPU iteration. Our tests and those of REFT [37] reveal that CheckFreq's actual GPU-CPU transfer bandwidth is around 2GB/s, significantly below modern PCIe links' capabilities. In multi-GPU server nodes, bandwidth contention from multiple GPU transmissions to host memory further degrades communication efficiency.

Performance for DP-MP Scenario. To evaluate the performance changes of FlowCheck in DP-MP scenarios, we deploy FlowCheck on 8 GPUs, with 2 MP stages configured separately. We partition the model into two similarly sized sections based on the number of layers, with each section consistently maintained across 4 GPUs. Between different MP stages, the training network transmits activation values and gradients of the loss function. Within the same MP stage, the training network transmits allreduce synchronization traffic. As shown in Figure 11, FlowCheck still ensures that checkpoint saving operations do not block training in DP-MP scenarios. Compared with the DP-only scenarios, FlowCheck has more available idle time to perform checkpoint operations in DP-MP scenarios (Figure 10), primarily due to the increased interval between two consecutive allreduce operations resulting from the inter-stage communication introduced by MP.

Performance for the Training Process. Based on the iteration time of each method measured above, we have also calculated the effective training duration proportion of different baselines corresponding to the training fault rate in the actual production environment during continuous training for 24 hours. Here effective training time refers to the duration of training, excluding the time spent on checkpoint saving, recovery, and redundant computations due to state rollback. We assume that the frequency of failures follows the data in the training log book in OPT [44] and utilize modeling in CPR [25] to determine the checkpoint saving frequency under different settings, ultimately calculating the proportion of effective training time under different baselines. The test result is shown in Figure 12. It can be observed that as the workload scale increases, the proportion of effective training time for other baselines rapidly decreases, primarily due to the continuous increase in checkpoint overhead, which leads to a lower optimal checkpoint frequency. In contrast, due to its decoupled design for checkpointing and training, FlowCheck ensures that checkpoint saving operations do not block the training process. At the same time, it significantly reduces the time wasted on redundant computations by saving checkpoints at every iteration, maintaining the proportion of effective training time above 98%.

8.3 System Reliability Analysis

Through real training traffic flow statistics, we observed a general link packet loss rate of approximately 10^{-9}. To simulate different link packet loss rates, we tested the reliability of our packet redundancy-recovery strategy in various system scenarios by manually dropping packets in the Packet Dumper module. As shown in Figure 13, under the GPT-3 1.3B workload, we manually varied the packet loss rate at the checkpoint nodes from 10^{-9} to 10^{-7}, and conducted 100 iterations of

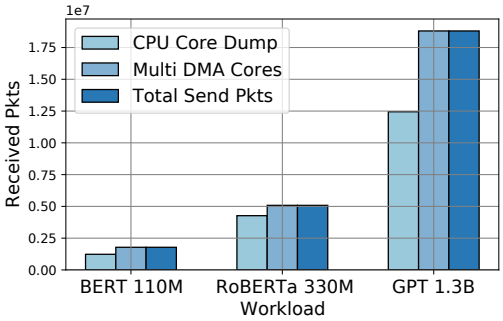

Figure 14. Multi DMA Capture Performance Improvement. We use 100Gbps RDMA NICs on checkpoint servers in this experiment.

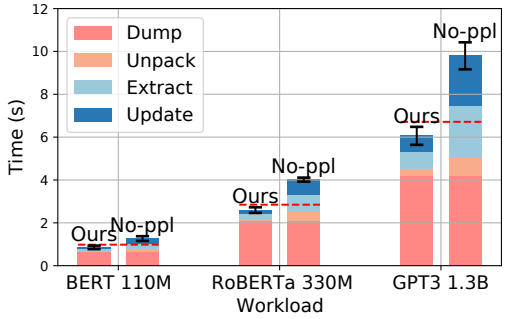

Figure 15. The performance of pipeline-based checkpoint updater. The red short dashed line represents the average duration per iteration during training. Tag "No-ppl" represents the non-pipeline method, where each module starts running only after the completion of processing all packets by the preceding module.

checkpoint saving tests under each packet loss rate setting. Finally, we statistically analyzed the proportion of successfully recovered checkpoint data from training traffic under different strategies. We can observe that when the packet loss rate increases, the recoverable probabilities for both the strategy without redundancy and the strategy with mutual redundancy using two monitoring nodes decrease significantly. However, the redundant scheme using three monitoring nodes can still maintain a single-iteration unrecoverable rate below 10^{-8}, indicating that the use of three monitoring nodes can maximize the effectiveness of FlowCheck's checkpoint design in ensuring system reliability.

8.4 Packet Processing Efficiency

In this subsection, we evaluate the packet processing efficiency of FlowCheck and conduct comparative tests on the specific performance of multi-DMA engines packet copying versus pipelined processing in our design.

Real-Time Dumper Performance We compare the packet processing speed between our design using multi-DMA cores and the approach utilizing the CPU cores for packet dumping. As depicted in Figure 14, relying solely on CPU cores for packet dumping results in the total number of received packets being less than the actual number of transmitted packets. This indicates that CPU core-based packet dumping alone is insufficient for real-time processing of high-speed traffic at 100 Gbps. In contrast, after employing multiple DMA engines as the data packet transmission accelerator, it can be observed that FlowCheck is capable of dumping all traffic data packets of varying model sizes into the host memory at a speed of 100Gbps, without experiencing any packet loss.

Pipeline-Based Checkpoint Updater Performance As described in §6.1, to maximize the utilization of CPU resources at checkpoint nodes and accelerate this process during the data packet processing, we employed a pipeline-based data packet processing approach to extract gradients from network traffic and perform model updates. To validate the effectiveness of the pipeline design in the packet processing procedure, we compared the total time required for packet processing with/without the pipeline design. We contrasted these results with the running time of each iteration, which is presented in Figure 15. Since the *Packet Dumper* module operates in real-time during model communication, we cannot further reduce the dumping time regardless of whether we implement pipeline-based overlap.

#Parameter	#GPU	#Layer	TP Size	MP Size	DP Size
175B	1536	96	8	12	16
530B	2240	105	8	35	8
1T	3072	128	8	64	6

Table 2. The Megatron-LM models used for estimation. Here "TP" stands for tensor parallelism and "MP" stands for model parallelism. The notation "B" stands for billion and "T" stands for trillion.

8.5 System Scalability

In this subsection, we use estimation to demonstrate that FlowCheck is scalable to scenarios with thousands of LLM training instances. We also further evaluate the performance changes of FlowCheck under more frequent fault scenarios.

Scaling to Thousands of GPUs. As the training scale increases, the total traffic generated during the training process also grows, necessitating more time for FlowCheck to update checkpoints from the training traffic. However, the larger training scale also leads to longer iteration times for normal training, resulting in extended intervals between consecutive allreduce operations. Based on the processing speeds of different components within the packet handling module for an individual packet, we can estimate the processing performance of FlowCheck at a larger scale and demonstrate its capability

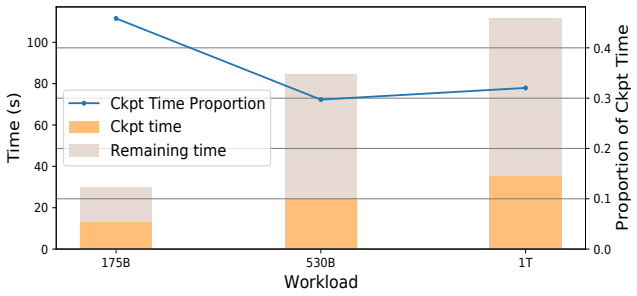

Figure 16. Estimation results of the proportion of checkpoint time within iteration time for larger workloads. The bar chart represents the specific time durations, while the line chart illustrates the proportions.

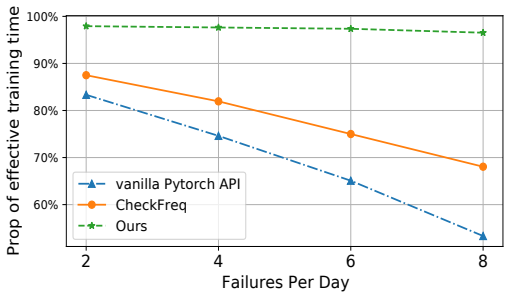

Figure 17. The proportion of effective training time under different failure rates. We use Megatron-LM 1T as the workload and follow the training settings in Table 2.

to ensure the completion of iteration-level checkpoint saving. Specifically, we estimate the total amount of data transmitted during the allreduce process based on different workloads, calculate the packet processing time of FlowCheck, and compare it with the time interval between the allreduce operations of the workload. For varying model scales from 175B to 1T, we adopt the specifications recommended by Megatron-LM [32] (total number of GPUs, parallel parameters) in our estimations and utilize data obtained from Calculon [18] simulations as proxies for the computational and communication durations consumed during model training. We conduct estimations for the workloads in Table 2.

As shown in Figure 16, throughout the estimation of workloads encompassing larger scales, extending up to 1T model size and encompassing a 3072-GPU cluster, FlowCheck ensures the completion of checkpoint operations within a single iteration of normal training, thereby achieving non-blocking checkpoint saving in the training process. As training scales increase, FlowCheck consistently ensures the completion of checkpoints earlier before each iteration's deadline. This is primarily because as the model size increases, the communication scale of a single DP communication group does not increase significantly, but the non-allreduce communication time introduced by other types of parallelism continues to increase. Therefore, in large-scale training scenarios, FlowCheck will have ample time for packet processing and checkpoint updating.

As FlowCheck introduces extra CPU nodes, we also briefly discuss its cost as the system scales. We use cloud providers' pricing for cluster nodes as a proxy. According to pricing on Alibaba Cloud in Sep 2024, renting an 8*A100 server costs $30 per hour, while renting a CPU server with an RDMA network card costs $1.37 per hour. Based on the training configuration provided in Table 2, when training a 175B model with 1536 GPUs, assuming a failure rate of twice per day, using native PyTorch for hourly checkpointing would result in approximately $5,760 of daily computational waste due to failures. In contrast, with FlowCheck, the combined daily cost of failure-related waste and additional hardware is around $526. This demonstrates that the extra hardware costs of FlowCheck remain acceptable, given the high-frequency, non-blocking checkpointing it provides.

Scaling to Frequent Failures. We assume that GPU nodes can retrieve data directly from checkpoint nodes after a failure, without accounting for checkpoint node failures. Relevant studies on checkpoint failure and multi-level checkpointing frequency are provided in §10. To evaluate the impact of failure rates, we manually adjusted various failure rates and estimated the proportion of effective training time as depicted in Figure 17. These estimations are based on the training settings of the Megatron-LM 1T model. We found that even if the frequency of failures reached 8 times per day, FlowCheck could still ensure more than 96% of the effective training time, while the effective training time ratio of other baselines would decrease rapidly with the increase of the failure rate. This is primarily due to the increasingly apparent computational waste caused by normal training when failures occur more frequently. However, FlowCheck ensures that checkpoints are updated at each iteration, thereby minimizing computational waste resulting from failures.

9 Discussion

Support for Other Allreduce Algorithms. In addition to the ring allreduce analyzed in detail in this paper, FlowCheck can be easily extended to support tree allreduce [29]. For tree allreduce, we only need to mirror the outbound traffic of the root node since the fully aggregated data are broadcasted from the root node in the allgather stage. In practical implementations, NCCL employs a double-binary tree approach to fully utilize bidirectional bandwidth. Therefore, mirroring and capturing the outbound traffic of the root nodes of both trees will be enough.

Support for Other Types of Parallelism. FlowCheck also has the potential to support other types of parallelism, such as FSDP [46] and ZeRO-3 [30]. In these cases, model parameters are transmitted over the network during the forward pass, so

we can capture and unpack all the traffic during this process to directly obtain the model parameter data corresponding to each iteration.

Support for Higher Network Bandwidth. As RDMA network bandwidth continues to increase, the packet dumping capability of FlowCheck will face potential challenges. With the NIC receiving packets at a faster rate, we must retrieve packets from the NIC buffer more quickly to prevent packet loss due to buffer overflow. Consequently, simply relying on the CPU-side DMA engine for packet dumping may cause performance bottlenecks. In such scenarios, deploying FlowCheck may require modifications to the NIC driver to leverage the NIC's hardware for packet dumping. A higher bandwidth will also impose a more stringent requirement on CPU for packet parsing, gradient data extraction, and checkpoint updates. Assuming the dumping rate matches the bandwidth, under the current system configuration, FlowCheck will be able to support a bandwidth of up to about 400 Gbps when each DP needs to transfer about 60 GB of data per iteration (e.g. for Llamma2-7B without model partitioning). With a larger shared memory, faster CPU, or smaller amount of data to transfer within each DP group (e.g. due to model parallelism), FlowCheck can support even higher bandwidth.

An Alternative Design to Mirroring Network. We can also use optical splitters for the mirroring network. An optical splitter divides a single optical signal into multiple paths, duplicating the signal without altering the network infrastructure or switch ports. Deployed beneath each leaf switch, these splitters bifurcate the link between the switch and monitored GPU nodes into two paths: one for normal training and the other for monitoring, connected to CPU nodes. This design does not require occupying switch ports. However, signal attenuation from the splitter can increase packet loss and retransmissions, reducing communication efficiency.

10 Related Work

Logging-based Checkpoint Mechanism. Apart from the improvement ideas mentioned in this article, there are some more attempts to reduce the operational overhead of checkpointing. Swift [47] proposed a logging-based method that records intermediate data during every iteration. This allows each GPU to replay the stored iterations from the latest checkpoint without redoing the backward and forward computations, which significantly reduces the time wasted on recovery after failures. However, this design still requires each training node to communicate for training logs actively, thus impacting and incurring overhead on the training process, ultimately limiting the frequency of checkpoint operations.

Checkpoint Saving for Specific Models. Some previous solutions have designed checkpoint systems tailored to specific models. CheckNRun [9] reduces the data volume during checkpoint saving process by implementing incremental checkpointing for recommender system models. EcRec [45] enhances the reliability of storing checkpoints on each training server through the design of erasure coding. These methods exploit characteristics relevant to the training of recommender system models to optimize the speed of checkpoint access and storage, yet they do not offer a generic design.

Optimization of Checkpointing Frequency for Multi-level Checkpoint System. FlowCheck ensures optimal save frequency (every iteration) from training nodes to the first-level checkpoint (checkpoint node) but does not increase the frequency between first- and second-level checkpoints (remote storage). To minimize save and rollback overhead considering varying node failure rates, proper save frequency configuration for multi-level checkpoint systems is needed. Sheng Di et al. [8] addressed this through mathematical modeling, while Masoud Gholami et al. [10] used a graph-based approach to capture node failure correlations. These studies are orthogonal to FlowCheck.

Checkpoint Compressing. Other efforts aimed to reduce the overhead of checkpoint operations by compressing checkpoint data. CheckNRun [9] utilizes CPU resources within the training nodes to compress checkpoints. DynaQuant [6] supports both lossy and lossless compression of checkpoints through the core approach of Quantization-Aware Delta Compression. These methods fundamentally reduce the size of checkpoints and are orthogonal to our design.

11 Conclusion

In this paper, we propose FlowCheck, a novel and efficient checkpoint system that decouples the checkpoint operation from training. Our primary objective is to decouple the checkpoint operation from training, which fundamentally reduces the blockage on training nodes. This enhancement increases model training efficiency and prevents resource wastage. As long as DP is used in parallel training and DP traffic passes through network switches, FlowCheck is applicable. Through two key designs: (1) packet-counting-based traffic identification, and (2) packet redundancy recovery mechanism, FlowCheck implements an efficient checkpointing system. Through experiments and estimations, we verify that FlowCheck can achieve checkpoint operations with zero impact on training, and demonstrate that FlowCheck can guarantee more than 98% effective training time under current fault frequency.

Acknowledgments

We are sincerely grateful to our shepherd John Wilkes and all anonymous reviewers for their valuable and constructive comments. This work was supported in part by the National Natural Science Foundation of China (No. 62072302, 61960206002), the Key R&D Program of Zhejiang Province (No. 2023R5202) and Alibaba Innovation Research Project (No. 2022010307).

References

[1] [n. d.]. Tcpdump. https://github.com/the-tcpdump-group/tcpdump.

[2] 2015. Memory Transactions. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/sourcelevel/memorytransactions.htm.

[3] 2016. NVIDIA NVLINK. https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/.

[4] 2023. BLOOM Chronicles. https://github.com/bigscience-workshop/bigscience/blob/master/train/tr11-176B-ml/chronicles.md,2022.

[5] 2024. AMD Instinct™ MI300X Accelerators. https://www.amd.com/en/products/accelerators/instinct/mi300/mi300x.html

[6] Amey Agrawal, Sameer Reddy, Satwik Bhattamishra, Venkata Prabhakara Sarath Nookala, Vidushi Vashishth, Kexin Rong, and Alexey Tumanov. 2023. DynaQuant: Compressing Deep Learning Training Checkpoints via Dynamic Quantization. http://arxiv.org/abs/2306.11800 arXiv:2306.11800 [cs].

[7] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[8] Sheng Di, Mohamed Slim Bouguerra, Leonardo Bautista-Gomez, and Franck Cappello. 2014. Optimization of multi-level checkpoint model for large scale HPC applications. In *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE, 1181–1190.

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

[10] Masoud Gholami Estahbanati and Florian Schintke. 2019. Multilevel checkpoint/restart for large computational jobs on distributed computing resources. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 143–14309.

[11] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.

[12] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. 2024. AI and memory wall. *IEEE Micro* (2024).

[13] Andrew Gibiansky. 2017. Bringing HPC techniques to deep learning. http://research.baidu.com/bringing-hpc-techniques-deep-learning.

[14] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch SGD: Training Imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[15] Andrew Grover and Christopher Leech. 2005. Accelerating Network Receive Processing. In *Linux Symposium*. Citeseer, 281.

[16] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. 2009. BCube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. 63–74.

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in Neural Information Processing Systems* 32 (2019).

[18] Mikhail Isaev, Nic Mcdonald, Larry Dennison, and Richard Vuduc. 2023. Calculon: a methodology and tool for high-level co-design of systems and large language models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Denver CO USA, 1–14. https://doi.org/10.1145/3581784.3607102

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

[20] Youhe Jiang, Huaxi Gu, Yunfeng Lu, and Xiaoshan Yu. 2020. 2D-HRA: Two-dimensional hierarchical ring-based all-reduce algorithm in large-scale distributed machine learning. *IEEE Access* 8 (2020), 183488–183494.

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

[22] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836* (2016).

[23] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[24] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[25] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, et al. 2021. Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. *Proceedings of Machine Learning and Systems* 3 (2021), 637–651.

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

[27] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

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

[29] NVIDIA. 2021. NCCL. https://developer.nvidia.com/NCCL.

[30] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[31] Lorenzo Rosa, Luca Foschini, and Antonio Corradi. 2024. Empowering Cloud Computing With Network Acceleration: A Survey. *IEEE Communications Surveys & Tutorials* (2024).

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

[33] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv preprint arXiv:2201.11990* (2022).

[34] Truong Thao Nguyen, Mohamed Wahib, and Ryousei Takano. 2021. Efficient MPI-AllReduce for large-scale deep learning on GPU-clusters.

Concurrency and Computation: Practice and Experience 33, 12 (2021), e5574.

[35] Qinlong Wang, Bo Sang, Haitao Zhang, Mingjie Tang, and Ke Zhang. 2023. DLRover: An Elastic Deep Training Extension with Auto Job Resource Recommendation. *arXiv preprint arXiv:2304.01468* (2023).

[36] Yuyang Wang, Dezun Dong, and Fei Lei. 2022. Understanding node connection modes in Multi-Rail Fat-tree. *J. Parallel and Distrib. Comput.* 167 (2022), 199–210.

[37] Yuxin Wang, Shaohuai Shi, Xin He, Zhenheng Tang, Xinglin Pan, Yang Zheng, Xiaoyu Wu, Amelie Chi Zhou, Bingsheng He, and Xiaowen Chu. 2023. Reliable and Efficient In-Memory Fault Tolerance of Large Language Model Pretraining. *arXiv preprint arXiv:2310.12670* (2023).

[38] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*. ACM, Koblenz Germany, 364–381. https://doi.org/10.1145/3600006.3613145

[39] Zhuang Wang, Haibin Lin, Yibo Zhu, and TS Eugene Ng. 2023. Hi-Speed DNN Training with Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 867–882.

[40] Noah Wolfe, Misbah Mubarak, Nikhil Jain, Jens Domke, Abhinav Bhatele, Christopher D Carothers, and Robert B Ross. 2017. Preliminary performance analysis of multi-rail fat-tree networks. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 258–261.

[41] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2023. Astra-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 283–294.

[42] Menglu Yu, Bo Ji, Hridesh Rajan, and Jia Liu. 2022. On scheduling ring-all-reduce learning jobs in multi-tenant GPU clusters with communication contention. In *Proceedings of the Twenty-Third International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*. 21–30.

[43] Longteng Zhang, Lin Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. 2023. LoRA-FA: Memory-efficient Low-rank Adaptation for Large Language Models Fine-tuning. *arXiv preprint arXiv:2308.03303* (2023).

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

[45] Tianyu Zhang, Kaige Liu, Jack Kosaian, Juncheng Yang, and Rashmi Vinayak. 2023. Efficient Fault Tolerance for Recommendation Model Training via Erasure Coding. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3137–3150.

[46] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch FSDP: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).

[47] Yuchen Zhong, Guangming Sheng, Juncheng Liu, Jinhui Yuan, and Chuan Wu. 2023. Swift: Expedited Failure Recovery for Large-Scale DNN Training. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 447–449.

[48] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. 2010. Parallelized stochastic gradient descent. *Advances in Neural Information Processing Systems* 23 (2010).