

# Chameleon: Adaptive Fault Tolerance for Distributed Training via Real-time Policy Selection

Yuhang Zhou<sup>1</sup>, Zhibin Wang<sup>1,\*</sup>, Peng Jiang<sup>1</sup>, Haoran Xia<sup>1</sup>, Junhe Lu<sup>1</sup>,  
Qianyu Jiang<sup>1</sup>, Rong Gu<sup>1</sup>, Hengxi Xu<sup>2</sup>, Xinjing Huang<sup>2</sup>, Guanghuan Fang<sup>2</sup>,  
Zhiheng Hu<sup>2</sup>, Jingyi Zhang<sup>2</sup>, Yongjin Cai<sup>2</sup>, Jian He<sup>2</sup>, Chen Tian<sup>1</sup>

<sup>1</sup> State Key Laboratory for Novel Software Technology, Nanjing University, China    <sup>2</sup> Huawei, China

**Abstract**—Training large language models faces frequent interruptions due to various faults, demanding robust fault-tolerance. Existing backup-free methods—redundant computation, dynamic parallelism, and data rerouting—each incur performance penalties, whether from ongoing overhead, lengthy reconfigurations, or post-recovery inefficiencies. We propose *Chameleon*, an adaptive fault-tolerant system that intelligently selects optimal recovery strategies when a failure occurs. *Chameleon* achieves this through a unified performance model, quick execution plan search, accurate performance estimation, and efficient communication optimizations. Experiments on a 32-card cluster show that *Chameleon* has a performance gap of within 11.00% between post-recovery and failure-free training, while preserving model convergence and efficient memory usage. Compared to state-of-the-art methods, *Chameleon* achieves up to 1.229 $\times$  and 1.355 $\times$  higher average throughput than Oobleck and Recycle, respectively.

**Index Terms**—Distributed training, fault tolerance, performance modeling, communication optimization.

## I. INTRODUCTION

In recent years, large language models have gained significant popularity because of their impressive capabilities in various tasks, such as natural language understanding, code generation, and dialogue systems [1], [2]. With the increasing complexity and scale of these models, *long-term and large-scale training* has become a common practice. For example, training the 405B Llama 3.1 model required roughly 30.8 million GPU hours [1]. The long-term, large-scale training process can be interrupted on average every few hours due to various faults [3], such as hardware failures, software bugs, or network issues. Thus, fault-tolerant training techniques are crucial for maintaining model performance and stability, especially in distributed and resource-constrained environments.

In practice, the entire fault-tolerant training process typically consists of three phases. (i) *Fault-free phase*: The system trains according to the given configuration when no faults occur. (ii) *Fault-handling phase*: When a fault occurs, training is interrupted, and the system takes measures to recover, such as restoring from checkpoints [4], [5], backup servers [6], or adjusting the configuration using only the available resources [7]–[10]. (iii) *Post-recovery phase*: After recovery, due to the absence of backups and possible node loss, training efficiency may decrease compared to the fault-free phase. Given that many users operate under resource constraints, backup-free fault-tolerant solutions have become a focal point.

However, existing backup-free fault-tolerant training methods are often tailored to specific training phases, leading to inherent limitations in broader scenarios. First, *redundant computation* such as Bamboo [7], where each stage node in pipeline parallelism additionally stores the layers of its successor and performs gradient computation. When the successor fails, its predecessor can take over its tasks seamlessly. However, due to additional computation and memory overhead for each stage, the training throughput in the fault-free phase is reduced. Second, *dynamic parallelism* like Oobleck [8], which provides predefined pipeline templates. Upon failure, the system can switch to a template with fewer nodes, dynamically adjusting the parallelism to match the available training resources. Although it does not introduce additional overhead during the fault-free phase, it takes a long time to reconfigure the pipelines when a fault occurs, leading to a significant drop in throughput during the fault-handling phase. Finally, *data rerouting* is exemplified by Recycle [9], which reroutes the micro-batches assigned to failed nodes to peer nodes in the same pipeline stage but in different data parallel groups, without parameter reconfiguration. However, its training throughput in the post-recovery phase is related to pipeline bubbles, which can lead to inefficiencies if not managed properly. With an increasing number of failures, the degradation of performance becomes more pronounced.

Therefore, a comprehensive and general fault-tolerant system is urgently needed that is compatible with different training phases. However, achieving this goal requires overcoming several key challenges. (i) How to accurately model the performance of different fault-tolerant strategies across all phases to provide a theoretical basis for strategy selection? (ii) How to efficiently search for and determine the optimal fault-tolerant strategy and execution plan under various configurations and resource constraints? and (iii) How to further optimize communication or computation performance for different strategies and training phases to maximize overall training efficiency?

To address these challenges, *Chameleon* introduces a unified framework that integrates performance modeling, strategy selection, and communication optimization for fault-tolerant training. Specifically, *Chameleon* builds on a comprehensive performance model that covers all phases and strategies, leveraging both analytical modeling and profiling data to accurately estimate key metrics. Guided by these estimations, *Chameleon* efficiently searches for the optimal execution plan to maximize

\* Corresponding author: wzbwangzhibin@nju.edu.cn

overall training throughput under given resource constraints. In addition, *Chameleon* incorporates targeted optimizations for model transfer during recovery and synchronization communication in data parallelism, further improving the efficiency of the selected strategy. With these enhancements, *Chameleon* can achieve efficient and stable fault-tolerant training.

The main contributions of *Chameleon* are as follows.

**Performance Modeling.** To determine the direction of performance optimization for fault-tolerant training, we first model the performance of different strategies across various training phases, thereby establishing the system’s optimization objective. In particular, the reconstruction overhead during the fault-handling phase and the training performance after recovery are key factors in our strategy selection. We also consider performance modeling in complex scenarios such as multiple failures and asymmetric parallelism, enabling performance evaluation under extreme conditions. Finally, we provide accurate estimations of per-step execution time and memory usage, offering a solid theoretical foundation for strategy search while effectively avoiding out-of-memory (OOM) issues.

**Execution Plan Search.** Before selecting a strategy, it is essential to ensure that each candidate strategy is configured for optimal performance. For different fault-tolerant strategies, factors such as parallelism degree, failed node distribution, model partitioning, and data allocation can significantly affect post-recovery training efficiency. To address this, *Chameleon* employs a heuristic search that comprehensively considers these factors while controlling search overhead, enabling rapid identification of feasible execution plans. *Chameleon* selects the optimal plan based on estimated performance metrics.

**Communication Optimization.** Changes of the execution plan not only affect pipeline computation but also inevitably impact communication. These impacts fall into two categories: (i) weight transfer communication during training reconstruction in dynamic parallelism; (ii) synchronization communication in data parallelism, which may be affected by asymmetric parallel execution. Both types of communication overhead are dynamic and, if not handled properly, can incur high costs. Therefore, we optimize both types of communication: the former is abstracted as a bipartite graph matching problem, and the latter as a graph coloring problem, achieving optimal communication performance with minimal overhead.

*Chameleon* was extensively evaluated in both simulated and real-world environments, utilizing a cluster of 32 Ascend 910B AI training accelerators (NPUs) [11]. Compared to fault-free training, *Chameleon*’s post-recovery performance maintained a gap of less than 11.00%. In simulations, *Chameleon* consistently outperformed baselines like Ooblock and Recycle, achieving an average throughput of  $1.229\times$  and  $1.355\times$  higher, respectively. A detailed evaluation of *Chameleon*’s key techniques shows that its estimator achieves an accuracy error within 8.02%. Optimized weight transfer reduces recovery time by up to 26.79%, while asymmetric communication optimization shortens step time by up to 15.44%. Memory, convergence, and scalability analyses show that *Chameleon* performs well in all three aspects.

## II. BACKGROUND

In this section, we introduce the necessary background of distributed model training, state-of-the-art backup-free work for fault tolerance, and the limitations of these methods.

### A. Distributed Model Training

With the rapid growth of model sizes, distributed training leverages strategies including Data (DP) [12]–[14], Pipeline (PP) [15]–[17], Tensor (TP) [18], and Expert Parallelism (EP) [19]–[21]. *DP* involves splitting the dataset across multiple nodes, where each node processes a subset of the data and computes the gradients independently. *PP* divides the model into stages, with each stage running on a different node, while *TP* splits the model’s tensors across multiple nodes, allowing for parallel computation of large tensors. And *EP*, often used in the mixture of experts (MoE) models [22], assigns different experts across nodes, enabling the model to handle more complex tasks by activating only a subset of experts. These parallel strategies have different communication patterns, which determine their typical deployment scenarios. Specifically, *TP* and *EP* require frequent and fine-grained communication, making them more suitable for intra-node deployment [23]. In contrast, *DP* and *PP* are more suitable for inter-node scaling due to their lower communication overhead [24], [25]. In practice, these strategies can be organized properly to achieve hybrid parallelism [26], [27]. While *TP* and *EP* are crucial for scaling model capacity and throughput, their fine-grained communication and tight cross-device coupling make fault isolation and recovery particularly challenging. Consequently, most fault-tolerant systems [7]–[9], [28], [29] focus on *DP* and *PP*, utilizing their natural state partitioning for flexible fault isolation with minimal disruption.

### B. Fault Tolerance in Distributed Training

Long-running large-scale distributed training jobs are susceptible to various faults [16], [30], [31], such as hardware failures, network issues, and software bugs. For example, during the 54-day Llama 3.1 training [1], there were 419 unplanned interruptions caused by faults, averaging 3 hours between each interruption. Gemini [3] noted that during the OPT-175B model training on 992 A100 GPUs, approximately 110 faults occurred over two months. Minder [32] showed that when training tasks involved about 1,000 machines, approximately 2 faults occurred daily on average, with each fault potentially halting training for several hours. To mitigate the impact of faults, various fault-tolerance strategies have been proposed. A common approach is *checkpointing* [4], [5], which periodically saves the model state to disk, allowing recovery from the last checkpoint in case of interruptions. Another strategy to ensure rapid recovery is *warm backup* [6], which maintains a pool of idle nodes ready to take over tasks at any time. However, it incurs significant resource waste due to the low utilization of standby nodes. As a result, this paper focuses on the *backup-free* methods, which leverage inherent redundancy and dynamic reconfiguration within the active training cluster and are orthogonal to checkpoint- and backup-based solutions.

### C. Limitations of Backup-free Methods

In practice, state-of-the-art backup-free methods can be categorized into three main categories:

- **Redundant computation.** Bamboo [7] employs redundant computation within the training pipeline for fault tolerance. Each node maintains a replica of its successor and hides the computation overhead within the pipeline bubbles, enabling rapid recovery for a node failure.
- **Dynamic parallelism** means adjusting the parallelism to adapt to the available resources. Varuna [28] proposed adjusting the pipeline and data parallelism to minimize communication overhead and recover by continuously checkpointing the model states. Oobleck [8] introduced pipeline templates to run multiple replicas of the pipeline and reconstruct the lost stage from the redundant replicas.
- **Data rerouting.** Recycle [9] exploits the consistent parameters across different pipelines and reroutes the micro-batches on the failed node to its data-parallel peers. Further, it proposed a scheduling mechanism to allow rerouted micro-batches to execute within the inherent pipeline bubbles, minimizing the throughput degradation.

We analyze the performance impact of these backup-free methods throughout the training process, which includes three critical phases: (i) *Fault-free phase*. A fault-tolerance policy may also introduce additional overhead when no faults occur. (ii) *Fault-handling phase*. This phase typically includes searching for a new execution plan and restoring the training state. (iii) *Post-recovery phase*. After recovering from the failure, the model's training efficiency may be degraded. The time and memory impact of different fault-tolerance policies varies. Redundant computation minimizes fault-handling time but introduces a prohibitive overhead (nearly double the computation and memory) during the fault-free phase. Dynamic parallelism methods avoid overhead during fault-free training. The primary drawback is the high reconstruction overhead during fault-handling, such as restarting from checkpoints or copying from live replicas, especially with frequent failures. Data rerouting avoids huge reconstruction overhead by rerouting micro-batches on the fly. However, its effectiveness relies on complex scheduling to hide additional computation within pipeline bubbles. If the bubbles are highly optimized (e.g., zero-bubble parallelism [33]), the performance loss after resuming training is not negligible. In summary, existing methods often only consider performance in certain phases and lack a comprehensive solution.

The challenge hinges on balancing immediate fault-handling overhead with post-recovery performance. Data rerouting is preferable for individual failures due to its low handling cost. However, as failures accumulate, training performance after recovery can become unsatisfactory. Conversely, while dynamic parallelism has a higher reconstruction cost, it may achieve better long-term throughput post-recovery by a new optimal parallel plan. Therefore, we consider selecting the optimal fault-tolerance strategy in real time based on specific failure scenarios to ensure efficient and stable training.

### III. PERFORMANCE MODELING

In this section, we first model the performance of targeted fault-tolerance methods, including data rerouting and dynamic parallelism, and then determine the optimization goal. Based on these, we present our system *Chameleon*, which adaptively selects the most suitable fault-tolerance policy.

*Definition 1 (State):* The state of a training system, denoted as  $\mathcal{S}$  includes the following information:

- **Cluster Status:** The number of nodes in the cluster.
- **Execution Plan:** (i) the fault-tolerance policy, which can be either dynamic parallelism or data rerouting; (ii) the parallel configuration, including the number of data parallel groups  $N_{dp}$  and the number of pipeline stages  $N_{pp}$ ; (iii) the data distribution among different data parallel groups, where the sum equals the global batch size; (iv) the model layer distribution across pipeline stages; (v) the distribution of failed nodes across the stages. Detailed information is provided in Section IV-A.

Therefore, when a fault occurs, the system needs to switch from the current state  $\mathcal{S}_1$  to a new state  $\mathcal{S}_2$ . Generally, considering a series of faults or recovery operations, we can define the state transition as a sequence of states  $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_n$ .

Considering the whole training process, the number of processed samples  $S_{\text{total}}$  can be calculated as follows:

$$S_{\text{total}} = \sum_{i=1}^n S_i, \quad (1)$$

where  $S_i$  is the number of samples processed in state  $\mathcal{S}_i$  and  $t_{\mathcal{S}_i}$  is the time spent in state  $\mathcal{S}_i$ . Moreover, the execution time  $t_{\text{total}}$  of the whole training process can be calculated as:

$$t_{\text{total}} = \sum_{i=1}^n t_{\mathcal{S}_i} + \sum_{i=1}^{n-1} t_{\mathcal{S}_i \rightarrow \mathcal{S}_{i+1}}, \quad (2)$$

where  $t_{\mathcal{S}_i \rightarrow \mathcal{S}_{i+1}}$  is the time spent in transitioning from state  $\mathcal{S}_i$  to  $\mathcal{S}_{i+1}$ .

**Optimization Problem.** Straightforwardly, we aim to maximize the throughput of the whole training process, which can be formulated as:

$$\arg \max_{\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_n} \frac{S_{\text{total}}}{t_{\text{total}}} \quad (3)$$

However, it is impractical to optimize the whole training process, as it requires knowledge of the entire training process, including when the next fault will occur. Fortunately, we observe that *the duration between two faults, i.e.,  $t_{\mathcal{S}_i \rightarrow \mathcal{S}_{i+1}} + t_{\mathcal{S}_{i+1}}$ , remains constant<sup>1</sup> regardless of the execution plan, and changing the execution plan only affects the throughput of the current state  $\mathcal{S}_i$  and the transition time  $t_{\mathcal{S}_i \rightarrow \mathcal{S}_{i+1}}$ .*

Therefore, the optimization goal can be simplified to maximizing the throughput of each duration between two faults, which can be expressed as:

$$\forall i \in [1, n-1], \quad \arg \max_{\mathcal{S}_i} \frac{S_{i+1}}{t_{\mathcal{S}_i \rightarrow \mathcal{S}_{i+1}} + t_{\mathcal{S}_{i+1}}}. \quad (4)$$

<sup>1</sup>Note that for different durations, e.g.,  $j \neq i$ , the duration can be different, but the duration between two given faults is constant.

Note that for the first state  $S_1$ , its throughput can be determined by the initial configuration, which has various work to discuss, such as Alpa [34], and we omit the discussion here.

For the number of processed samples  $S_i$ , it can be calculated as follows:

$$S_i = T_i \cdot t_{S_i}, \quad (5)$$

where  $T_i$  is the throughput of state  $S_i$ .

By substituting  $S_i$  into the optimization goal, we can obtain:

$$\forall i \in [1, n-1], \quad \arg \max_{S_{i+1}} \frac{T_{i+1} \cdot t_{S_{i+1}}}{t_{S_i \rightarrow S_{i+1}} + t_{S_{i+1}}}. \quad (6)$$

Considering the  $T_i$  in state  $S_i$  is calculated as:

$$T_i = \frac{B_i}{t_{\text{step}, S_i}}, \quad (7)$$

where  $B_i$  is the given global batch size in state  $S_i$ , and  $t_{\text{step}, S_i}$  is the time for each step in state  $S_i$ .

Therefore, we can further express the optimization goal as:

$$\forall i \in [1, n-1], \quad \arg \max_{S_{i+1}} \underbrace{\frac{B_{i+1}}{t_{\text{step}, S_{i+1}}}}_{\text{throughput}} \cdot \underbrace{\frac{t_{S_{i+1}}}{t_{S_i \rightarrow S_{i+1}} + t_{S_{i+1}}}}_{\text{effective time ratio}}, \quad (8)$$

which mainly consists of two parts: the throughput of the next state  $S_{i+1}$  and the effective time ratio, which represents the proportion of time spent in state  $S_{i+1}$  relative to the total time between two faults. Regarding the throughput, since the batch size  $B_{i+1}$  is given by users, the key to maximizing the throughput is to minimize the step time  $t_{\text{step}, S_{i+1}}$ , which represents the time for each training step (a.k.a., iteration). For the effective time ratio, since the total time between two faults is  $t_{S_i \rightarrow S_{i+1}} + t_{S_{i+1}}$ , which is constant, the key to maximizing the effective time ratio is to minimize the state transition time  $t_{S_i \rightarrow S_{i+1}}$ . We need to investigate the factors that affect the state transition time  $t_{S_i \rightarrow S_{i+1}}$  and the per-step execution time  $t_{\text{step}, S_{i+1}}$ . First, the fault-tolerance strategy in the execution plan allows for a choice between data rerouting and dynamic parallelism. The transition time  $t_{S_i \rightarrow S_{i+1}}$  varies significantly between these strategies. For example, data rerouting does not require training reconstruction, so the transition time is negligible. In contrast, dynamic parallelism incurs considerable transition time due to weight transfer between nodes and restart overhead. The factors affecting  $t_{\text{step}, S_{i+1}}$  are more complex, involving parallel configuration, as well as distribution of failed nodes, data, and model layers.

In summary, we can abstract the problem of fault-tolerant training as follows: Given the current state  $S_i$ , how can we determine the next state  $S_{i+1}$  (specifically, its execution plan  $E_i$ ) such that both the state transition time  $t_{S_i \rightarrow S_{i+1}}$  and the per-step execution time  $t_{\text{step}, S_{i+1}}$  are minimized? To address this, we design the *Chameleon* system, which adaptively finds the most suitable execution plan in response to failures. Specifically, *Chameleon* tackles the following challenges.

- **Given  $S_i$ , how do we determine the optimal  $S_{i+1}$ ?** (§IV-A) We employ a heuristic search that rapidly explores candidate execution plans considering multiple

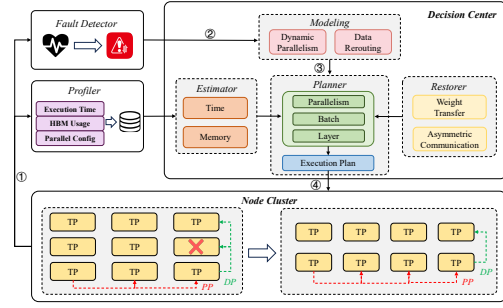


Fig. 1: The overall workflow of *Chameleon*.

factors, including parallelism, data, and layer distribution. We select the optimal execution plan by evaluating the estimated  $t_{\text{step}, S_{i+1}}$  and  $t_{S_i \rightarrow S_{i+1}}$  of each plan.

- **How do we minimize the state transition time  $t_{S_i \rightarrow S_{i+1}}$ ?** (§IV-B) During dynamic parallelism,  $t_{S_i \rightarrow S_{i+1}}$  includes the weight transfer overhead incurred by training reconstruction, which varies with the execution plan. We abstract this process as a bipartite matching problem to minimize transfer cost, thus reducing  $t_{S_i \rightarrow S_{i+1}}$ .
- **How do we minimize the step time after recovery  $t_{\text{step}, S_{i+1}}$ ?** (§IV-B) The  $t_{\text{step}, S_{i+1}}$  consists of both pipeline computation time and synchronization communication time. While the former can be estimated by the estimator, the latter, especially under asymmetric parallelism, can be optimized as a graph coloring problem.
- **How do we estimate the step time after recovery  $t_{\text{step}, S_{i+1}}$ ?** (§IV-C) *Chameleon* estimates both step time and memory usage for each candidate plan. For the former, it predicts pipeline time by applying analytical formulas for symmetric pipelines (data rerouting) and dynamic programming for asymmetric pipelines (dynamic parallelism). For the latter, it identifies the peak memory of each pipeline stage based on layer-wise profiling, ensuring that no OOMs occur.

#### IV. SYSTEM DESIGN

In this section, we will introduce the core techniques of *Chameleon* and explain how they enable fast recovery and efficient training. As shown in Figure 1, *Chameleon* consists of three main components: a profiler, a fault detector, and a decision center. The decision center integrates a planner, estimator, and restorer to manage the recovery process. The workflow is as follows: ① *Monitoring*: The profiler continuously collects runtime metrics such as execution time, HBM usage, and parallel configurations from the cluster. Meanwhile, the fault detector monitors the health of each node. ② *Fault Trigger*: Upon detecting a failure, the fault detector immediately triggers the decision center to initiate the fault-tolerance mechanism. ③ *Decision Making*: For each failure, the planner generates potential execution plans considering parallelism, layer, and batch configurations. Then it queries the estimator, which uses profiling data to predict the performance and memory usage of each plan. Meanwhile, the restorer is responsible for minimizing the overhead of transferring model

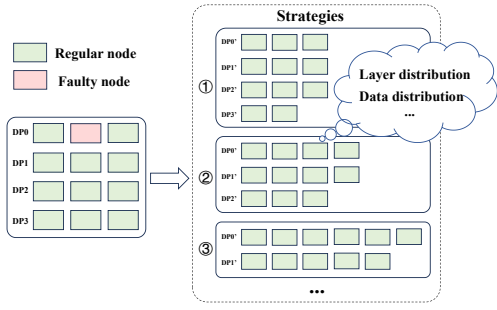


Fig. 2: Search for optimal execution plan.

weights and synchronization communication in asymmetric parallelism. Based on these, the planner selects the optimal execution plan. ④ *Plan Execution*: The planner sends the finalized execution plan to the cluster, which isolates the failed node and resumes training with minimal disruption.

#### A. Planner

In the planner, we need to determine the execution plan that achieves the best performance.

**Execution Plan Search.** If data rerouting is selected, considering that pipeline bubbles during training are usually well-optimized and the proportion is limited, the upper and lower bounds of performance do not differ significantly. Therefore, when determining the execution plan, we evenly distribute the micro-batches from the failed nodes to other data-parallel peers, aiming to overlap them with bubbles as much as possible. However, if dynamic parallelism is used, the performance differences between different execution plans can be substantial. As shown in Figure 2, we need to search for the optimal execution plan based on the following factors.

- **DP/PP Parallelism.** Unlike directly reducing one DP or PP degree, our goal is to utilize all available nodes as much as possible to maximize throughput. Thus, we provide a larger parallelism search space that supports asymmetric parallelism and is not restricted by predefined PP templates as in Oobleck. Based on our experience, the new DP degree often differs from the original by less than 2. We employ a heuristic search by limiting the parallelism range to reduce the search overhead.
- **Batch Distribution.** For load balancing of different DP groups and to avoid stragglers, we first pre-allocate micro-batches according to the proportion of nodes. If there are remaining unallocated micro-batches, we recursively assign them one by one to all pipelines. If some partitions have no data finally, we reassign one micro-batch from the largest partition to fill these. This check is repeated to ensure that no partition is left idle.
- **Layer Distribution.** For a given pipeline, special attention must be paid to load balancing between stages. Therefore, we first evenly split the model layers across all stages. For the remaining layers that cannot be evenly divided, we enumerate all possible allocation schemes. After filtering out options that may cause OOM via

memory estimation, we select the splitting scheme with the lowest pipeline execution time by the estimator.

Algorithm 1 illustrates the steps to identify the optimal execution plan, with *get\_execution\_plan* serving as entry. Lines 1-7 demonstrate the function *get\_parallel\_strategy* to enumerate all candidate parallel strategies ( $S_{cand}$ ), taking into account possible failure numbers and valid DP/PP combinations. Specifically, it recursively factorizes the current number of nodes to determine the possible degrees of DP/PP that meet the required range in *integer\_partition*. In the *get\_execution\_plan* function, for each candidate strategy in  $S_{cand}$ , lines 13-14 determine the balanced distribution of micro-batches (*distribute\_batch*) and the optimal splitting of layers into stages (*split\_layers*). Line 15 estimates the execution time of the candidate by *time\_estimator*. The function then tracks the plan ( $P_{best}$ ) with the lowest estimated execution time and returns it in lines 16-19.

---

#### Algorithm 1: Search for the Best Execution Plan

---

**Input:** Number of Nodes  $N$ , Max Number of Faults  $N_f$ , Range of DP  $R_{dp}$ , Range of PP  $R_{pp}$ , Micro Batches  $N_m$

**Output:** The Best Execution Plan  $P_{best}$

```

1 Function get_parallel_strategy( $N, N_f, R_{dp}, R_{pp}$ )
2    $S_{cand} = \text{null}$ 
3   for  $i \leftarrow 1$  to  $N_f$  do
4     for each  $dp$  in  $R_{dp}$  do
5        $L_{pp} = \text{integer\_partition}(N - i, dp, R_{pp})$ 
6        $S_{cand} += (dp, L_{pp})$ 
7   return  $S_{cand}$ 
8 Function get_execution_plan( $N, N_f, R_{dp}, R_{pp}, N_m$ )
9    $T_{best} = \infty$ 
10   $P_{best} = \text{null}$ 
11   $S_{cand} = \text{get\_parallel\_strategy}(N, N_f, R_{dp}, R_{pp})$ 
12  for each  $(dp, L_{pp})$  in  $S_{cand}$  do
13     $batch = \text{distribute\_batch}(N_m, L_{pp})$ 
14     $layers = \text{split\_layers}(L_{pp}, \text{memory\_estimator})$ 
15     $T_{est} = \text{time\_estimator}(L_{pp}, layers, batch)$ 
16    if  $T_{est} < T_{best}$  then
17       $T_{best} = T_{est}$ 
18       $P_{best} = (dp, L_{pp}, layers, batch)$ 
19  return  $P_{best}$ 

```

---

The Planner's overhead involves three steps: (i) Parallel strategy search iterates through valid  $(dp, L_{pp})$  combinations based on the divisors of the node count  $N$ . The complexity is  $O(\tau(N))$ , where  $\tau(n)$  denotes the divisor function. Even for large clusters (e.g.,  $N = 1024$ ), the number of divisors is small (e.g.,  $\tau(1024) = 11$ ). (ii) The complexity of batch distribution is linear with respect to the number of data-parallel groups,  $O(N_{dp})$ . (iii) Due to the even-distribution-first policy, layer distribution is limited to allocating the remaining layers  $r = L \bmod N_{pp}$  across stages. The complexity is bounded by the binomial coefficient  $\binom{N_{pp}}{r}$ . Since the pipeline



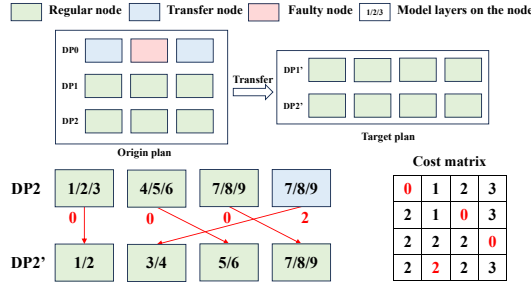


Fig. 3: Optimization of weight transfer.

depth  $N_{pp}$  is typically small (e.g.,  $\leq 16$ ) in practice, the number of candidates remains in the range of ten to hundreds. Furthermore, *Chameleon* pre-calculates and caches the optimal plans for potential failure scenarios (e.g., loss of 1 to  $k$  nodes) during the fault-free training phase. When a fault occurs, the Restorer directly retrieves the pre-computed plan from the cache, making the decision-making latency negligible.

### B. Restorer

There is significant room for optimization of  $t_{S_i \rightarrow S_{i+1}}$  and  $t_{step, S_{i+1}}$ . For the former, it suffers from the communication overhead brought by weight transfer, while the latter may suffer from asymmetric synchronous communication.

**Weight Transfer.** To achieve higher training throughput, we prefer to use dynamic parallelism to rearrange the training nodes. However, the high overhead of model weight transmission can become a bottleneck in practical deployment. As shown in Figure 3, the original training uses a configuration of DP = 3 and PP = 3, with 9 layers of model weights evenly distributed in each stage. When a node failure occurs, the planner identifies an execution plan, i.e., DP = 2 and PP = 4, with the model layers distributed as (2,2,2,3). To migrate the layers from the original to the new configuration, how can we minimize the amount of data transferred? We can model this as a bipartite graph matching problem.

Taking the original  $DP2$  group and the new  $DP2'$  group as an example, there may be different correspondences between the two, resulting in varying amounts of weight data that need to be transferred. Assuming the number of remaining nodes is  $N$ , we can construct an  $N \times N$  cost matrix  $Cost$  based on the different layer distributions, where  $Cost[i][j]$  represents the cost for node  $i$  to migrate to the  $j$ -th node under the new plan. For example, if the first node in  $DP2$  corresponds to the first node in  $DP2'$ , the layers change from (1,2,3) to (1,2); layer 3 can be directly discarded without any additional data transfer, so the cost is 0. However, if this node corresponds to the second node in  $DP2'$ , the layers become (3,4); in addition to discarding layers 1 and 2, layer 4 must be transferred from another node, resulting in a cost of 1. In this way, we can further use the Kuhn-Munkres algorithm [35] to compute the migration scheme with the minimum total cost.

**Asymmetric Communication.** During dynamic parallelism, asymmetric parallelism may introduce additional communication overhead for  $t_{step, S_{i+1}}$ . As shown in Figure 4, in symmetric parallel training, AllReduce communication between DP

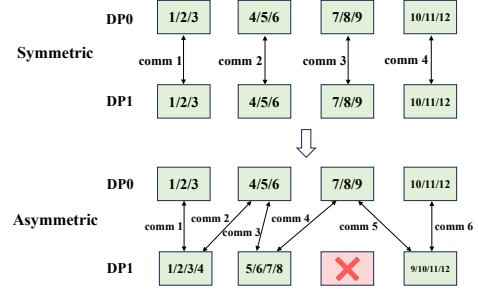


Fig. 4: The asymmetric DP gradient update communication.

groups occurs for the same model layer. However, when a node failure causes the PP configuration in one DP group to change, the model layers are redistributed. To ensure communication between the same layers, we need to establish asymmetric communication domains, which mainly leads to two changes as shown in Figure 4: (i) the number of communications increases (e.g., from 4 AllReduce operations to 6); (ii) there are dependencies between multiple DP communications, causing originally parallel communications to be executed serially. For example, in the new comm2, layer 4 is distributed on the first node, so comm2 must wait for comm1 on that node to finish before it can proceed.

To minimize the total time for asymmetric communication, the key is to maximize the parallelism of DP communications that do not have dependencies. We model this problem as a graph coloring problem: each model layer is treated as a vertex, and if two layers are located on the same device, there is a communication dependency, represented by an edge. The goal is to assign different colors to adjacent vertices, using the minimum number of colors. Layers with the same color can perform DP communication in parallel, and the number of colors corresponds to the number of communication rounds required. We adopt a greedy algorithm to find the minimum number of communication rounds, with a time complexity of  $O(L^2)$ , where  $L$  is the number of model layers.

Besides model weights, our Restorer also incorporates the transfer and reconstruction of optimizer states. For ZeRO parallelism, we only focus on scenarios where state redundancy exists; in such cases, lost optimizer states are recovered by directly transferring replicas from peer nodes in other healthy DP groups, thereby ensuring lossless precision.

### C. Estimator

During the selection of fault-tolerance policy, especially given a specific execution plan, we need to estimate its performance in terms of time and memory overhead. In practice, its time should be minimized as much as possible, while its memory usage needs to stay within the hardware limit.

**Time Estimator.** *Dynamic parallelism.* When faced with failures, the system can adjust the data and pipeline parallelism sizes to better utilize available resources. Two scenarios exist: (i) *Symmetric parallelism:* In Varuna and Parcae, the new parallelism configuration requires the number of nodes to be a multiple of the new DP and PP sizes, ensuring an even

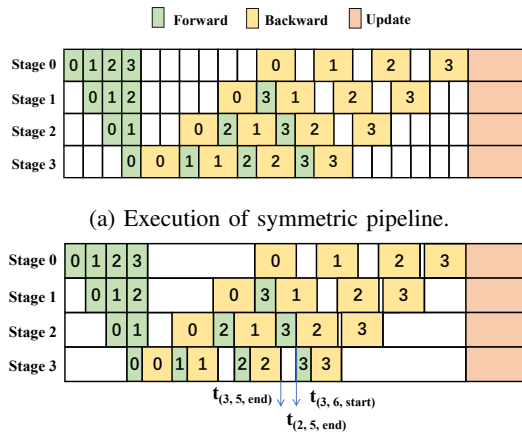


Fig. 5: Execution of different pipeline scenarios.

distribution of data and model across nodes to maximize throughput. (ii) *Asymmetric parallelism*: Oobleck provides predefined pipeline templates, each specifying a different number of pipeline stages. In certain parallel configurations, multiple templates may coexist, resulting in asymmetric parallelism. For an irregular number of nodes, this approach can effectively utilize all available nodes.

Different parallelism strategies yield different  $t_{\text{step}, S_{i+1}}$ . Take the 1F1B pipeline parallelism as an example, as shown in Figure 5a, the training consists of  $N_{dp}$  pipelines, each pipelined into  $N_{pp}$  stages. In each iteration, every stage processes  $N_m$  micro-batches, with forward and backward pass times per micro-batch denoted as  $T_f$  and  $T_b$ , respectively. The execution of pipelines shows different patterns under symmetric and asymmetric parallelism. For the former, the step time can be calculated as follows:

$$t_{\text{step}, S_{i+1}} = (N_{pp} + N_m - 1) \cdot (T_f + T_b) \quad (9)$$

However, for the latter, the step time is determined by the slowest pipeline due to the synchronous update across all pipelines. This relationship is formulated as:

$$t_{\text{step}, S_{i+1}} = \max_{p_i \in \mathcal{P}} \{t_{p_i}\} \quad \text{where } \mathcal{P} = \{p_1, p_2, \dots, p_P\} \quad (10)$$

where  $\mathcal{P}$  represents the set of all pipelines.

In asymmetric parallelism, estimating the execution time of each pipeline  $t_{p_i}$  is not straightforward. First, uneven data distribution across pipelines leads to different numbers of micro-batches for each pipeline. Meanwhile, unbalanced layer allocation among stages within the same pipeline directly affects the forward and backward times of each stage. As illustrated in Figure 5b, asymmetric pipeline parallelism leads to varying forward and backward computation times across stages, resulting in numerous pipeline bubbles and making estimation challenging. We observe that the start time of the  $j$ -th computation in the  $i$ -th stage ( $t_{i,j,\text{start}}$ ) depends on both the end time of the previous computation in the same stage ( $t_{i,j-1,\text{end}}$ ) and the end time of its depended computation (often has the same micro-batch index) in the previous stage

( $t_{f_1(i), f_2(j), \text{end}}$ ). Therefore, we propose a dynamic programming algorithm to simulate the pipeline time  $t_{p_i}$ . The general transition function during the ready phase is defined below.

$$t_{i,j,\text{start}} = \max(t_{i,j-1,\text{end}}, t_{f_1(i), f_2(j), \text{end}}) \quad (11)$$

where  $f_1(i)$  denotes a mapping from the current stage index  $i$  to the previous stage index  $i'$  depended by the current computation, for example, if the current computation is forward,  $f_1(i) = i - 1$ , and if it is backward,  $f_1(i) = i + 1$ ; and  $f_2(j)$  denotes a mapping from the current computation index  $j$  to the depended computation index  $j'$  in the previous stage, which can be profiled from the actual execution order.

For dynamic parallelism, the state transition introduces non-negligible time overhead  $t_{S_i \rightarrow S_{i+1}}$ . This overhead consists of three parts: (i) the cost of searching for the optimal execution plan; (ii) the cost of transferring model weights; and (iii) the overhead of restarting training. It is important to note that the search can be performed in advance and overlapped with training time, while the restart overhead depends only on the training scale. However, the weight transfer cost varies with the execution plan and is difficult to predict. Detailed optimization of transmission is discussed in Section IV-B.

*Data rerouting*. For this policy, since no reconfiguration of the training is required, the state transition time  $t_{S_i \rightarrow S_{i+1}}$  can be considered negligible. The main time overhead occurs during post-recovery training, as some nodes take on additional computation tasks of the failed nodes, resulting in a longer per-step execution time  $t_{\text{step}}$ . Recycle optimizes through techniques such as decoupled backward propagation and staggered optimization, reducing rerouting latency via sophisticated scheduling. To simplify the analysis, we assume that the computational tasks of the failed nodes are evenly distributed among the remaining functional nodes in the same DP group when a failure occurs. The per-step execution time of the 1F1B pipeline in the data rerouting under a single failure can be calculated as follows.

$$t_{\text{step}, S_{i+1}} = (N_{pp} + N_m - 1 + \frac{N_m}{N_{dp} - 1}) \cdot (T_f + T_b) \quad (12)$$

However, it is worth noting that as the number of failed nodes increases, the per-step time after recovery also increases. Specifically, the per-step time  $t_{\text{step}, S_{i+1}}$  with  $N_f$  failed nodes is related to the distribution of failed nodes across the pipeline stages. And its calculation is shown as follows.

$$t_{\text{step}, S_{i+1}} = (N_{pp} + N_m - 1 + \sum_{i=0}^{N_{pp}-1} [F_i > 0] \cdot \frac{N_m \cdot F_i}{N_{dp} - F_i}) \cdot (T_f + T_b) \quad (13)$$

where  $F_i$  is the number of failed nodes in the  $i$ -th stage, and the sum of  $F_i$  is equal to the total number of failed nodes  $N_f$ . But if any  $F_i$  is larger than  $N_{dp}$ , the training cannot be recovered, and we must switch to dynamic parallelism.

**Memory Estimator**. To evaluate whether a given model layer splitting is appropriate, we need to determine whether OOM occurs during training. In practice, we observe that the peak memory of the  $i$ -th stage appears during the steady-state phase

of the pipeline. We can decompose the peak memory for the  $i$ -th pipeline stage,  $M_{\text{peak},i}$ , into two primary components: static memory, which includes memory for components whose size is relatively fixed during an iteration, such as model weights and optimizer states; and dynamic memory, which consists of the stored activations from the forward pass. Its peak depends on both the number of layers in the stage and the stage's position in the pipeline. The final formula is as follows.

$$M_{\text{peak},i} \approx \underbrace{N_{l_i} \cdot (m_p + m_o + m_g)}_{\text{Static Memory}} + \underbrace{(N_{pp} - i) \cdot N_{l_i} \cdot m_a}_{\text{Peak Dynamic Memory}} \quad (14)$$

where  $N_{l_i}$  is the number of layers in the  $i$ -th stage,  $m_p$  is the average memory for the parameters of a single layer,  $m_o$  is the average memory for the optimizer states of a single layer,  $m_g$  is the average memory for the gradients of a single layer ( $m_g = m_p$ ), and  $m_a$  is the average memory for the activations produced by a single micro-batch for a single layer. All of the last four parameters can be estimated by profiling.

**Computational Complexity.** (i) Time Estimator. For symmetric parallelism and Data Rerouting, we employ analytical formulas with  $O(1)$  complexity. For asymmetric parallelism, the complexity of the dynamic programming method is  $O(N_{pp} \times N_m)$ . (ii) Memory Estimator only calculates the peak memory usage from profiling, and its complexity is  $O(1)$ .

## V. EVALUATION

In this section, we present the evaluation of our fault-tolerant system *Chameleon*. We will detail the experimental setup, including the cluster setup, baselines, and workloads. Following this, we will present the results obtained from our experiments, including real-world and simulation results, as well as an ablation study to understand the impacts of the core techniques of *Chameleon*. Finally, we will analyze the memory usage and convergence of *Chameleon*.

### A. Experimental Setup

We evaluated *Chameleon* with the following setup.

**Cluster Setup.** In real-world experiments, we used a cluster consisting of 32 Ascend 910B AI training accelerators [11], [36]. The cluster is composed of 4 nodes, each equipped with 8 NPUs, and each NPU has 64 GB of memory. Note that *Chameleon*'s core mechanisms—execution plan search, weight transfer, and communication scheduling—are algorithmic abstractions independent of the underlying hardware. Thus, the effectiveness demonstrated on Ascend is generalizable to other platforms, such as NVIDIA GPUs. To compare the efficiency of *Chameleon* with other systems, we built an event-driven simulator, with details provided in Section V-B.

**Baselines.** *Chameleon* is implemented based on PyTorch 2.1.0 and Ascend chips. We compare the performance of *Chameleon* with the original training in the real-world cluster. Then, we include two state-of-the-art baselines, dynamic parallelism exemplified by Ooblock and data rerouting exemplified by ReCycle, in the simulator for comparison, as they are implemented on different hardware. We exclude redundant computation

methods like Bamboo and traditional checkpointing as baselines, as recent literature has demonstrated their prohibitive overhead compared to these backup-free approaches [9].

**Workloads.** We used the 7B Llama-2 model [37] and the WikiText dataset for training. In real-world experiments, we conduct tests on 8, 16, and 32 NPUs, with parallel configurations (PP, DP, TP) including (4, 2, 1), (4, 4, 1), and (2, 2, 8). The batch size ranges from 16 to 64, while the micro-batch size is always set to 1. To ensure that *Chameleon* can maintain stable fault-tolerant training over long periods, the total training time lasted up to 9 hours.

### B. Experimental Results

The real-world results demonstrate the practical performance of *Chameleon* in the Ascend cluster, while the simulation results provide insights into its comparative performance against other systems.

**Real-World Results.** Figure 6 shows the performance comparison between the original training (8, 16, and 32 NPUs) and the training after failure recovery (7, 15, and 24 NPUs). After recovery, *Chameleon* achieves 96.22%, 89.00%, and 74.38% of the original training performance, respectively. Notably, for the 32-NPU case (TP=8), a single fault leads to the loss of 8 NPUs, resulting in a larger performance gap compared to the original. However, compared to the theoretical maximum for 24 NPUs, *Chameleon* reaches 99.17% of the performance, demonstrating its ability to fully utilize available resources.

**Simulation Results.** In simulation experiments, we evaluated *Chameleon*, ReCycle (data rerouting), and Ooblock (dynamic parallelism). By randomly injecting failures at a specified NPU failure rate (10% per hour), we simulated the operation of all three systems for 9 hours of training with 32 NPUs. Figure 7 shows the number of active nodes over time, while Figure 8 presents the training throughput over time. The results show that *Chameleon* consistently outperforms both Ooblock and ReCycle, achieving an average throughput that is  $1.229\times$  and  $1.355\times$  higher, respectively. While Ooblock exhibits significant throughput fluctuations, ReCycle's performance steadily degrades as the number of failures increases.

### C. Performance Breakdown

To evaluate the efficiency and necessity of the core techniques in *Chameleon*, we performed the following tests.

**Estimation Accuracy.** To verify the accuracy of the estimator, we compared the estimated time with the actual training time before and after recovery in real-world experiments under different configurations. As shown in Figure 9, the estimation error is always kept within 8.02%, demonstrating the effectiveness of our estimation mechanism.

**Weight Transfer.** To understand the impact of weight transfer optimization on training performance, we conducted an ablation study comparing training performance with and without optimization. Taking single-node 8-card training as an example (DP=4, PP=2, TP=1), as shown in Figure 10, when the number of layers is relatively small (e.g., 4 or 8), the weight transfer optimization does not yield significant performance



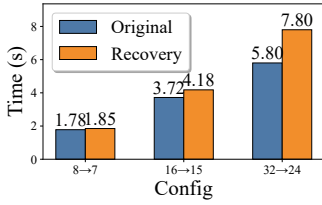


Fig. 6: Real-world training.

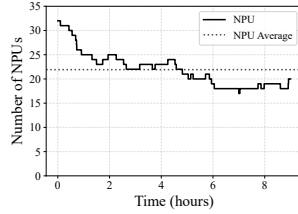


Fig. 7: Num of active NPUs

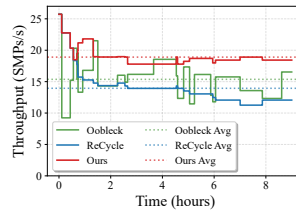


Fig. 8: Training throughput

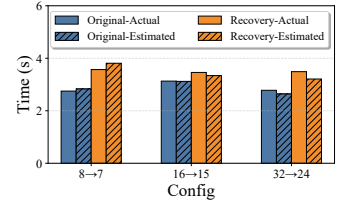


Fig. 9: Estimation results

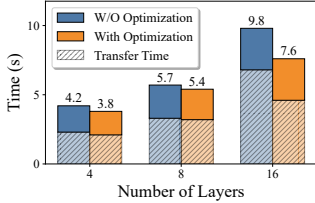


Fig. 10: Weight transfer

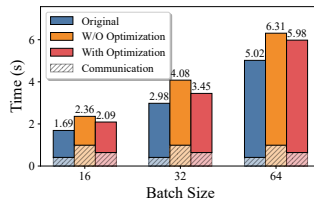


Fig. 11: Asymmetric comm

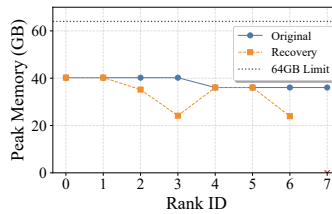


Fig. 12: Memory

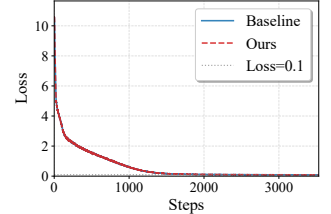


Fig. 13: Convergence

improvements. However, as the number of layers increases to 16, the transfer time during recovery is reduced by 32.35%. Correspondingly, the total recovery time decreases from 11.2 s to 8.2 s, representing a reduction of 26.79%, thereby enhancing overall training efficiency.

**Asymmetric Communication.** We conducted an ablation study to evaluate the impact of our asymmetric communication optimization on training performance. As shown in Figure 11, in the same 8-card training, the batch size varies from 16 to 64, asymmetric communication optimization significantly reduces the overhead of AllReduce and other synchronizations, maintaining a reduction rate of 35.35%. Although training performance decreases to some extent compared to fault-free scenarios, step time decreases by 11.44%, 15.44%, and 5.52%, respectively, compared to cases without optimization. Notably, when the batch size is small, the optimization effect is more pronounced due to the relatively high communication ratio.

#### D. Memory Analysis

To ensure that *Chameleon* can handle large-scale model training without running into OOM issues, we conducted a memory usage analysis. Taking single-node 8-card training as an example, when an NPU failure occurs, the parallelism changes from symmetric parallelism (DP=4, PP=2) to asymmetric pipelines with length [2, 2, 3]. As shown in Figure 12, the peak memory per NPU in *Chameleon* does not increase; in fact, it decreases in some cases and remains well below the device memory limit (64GB). This is due to changes in DP and the possible increase in the number of nodes per pipeline, resulting in fewer layers assigned to each node. These results demonstrate that *Chameleon* can effectively manage memory usage during fault-tolerant training without OOMs.

#### E. Convergence Analysis

To verify whether *Chameleon* will affect the model convergence, we conducted a convergence analysis on 32 Ascend NPUs by comparing our training loss with the original training. As shown in Figure 13, during training for up to

3500 steps, the training loss of *Chameleon* remains almost identical to the original training, and both ultimately reach the convergence criterion ( $\leq 0.1$ ). This indicates that *Chameleon* does not negatively impact model convergence.

#### F. Scalability Analysis

To validate our theoretical scalability analysis, we measured the decision-making overhead across varying cluster sizes in simulation. For small to medium-scale clusters (up to 256 cards), the search latency is negligible ( $\leq 1.152$ s). Even at large scales, the overhead remains acceptable (19.25s for 1024 cards and 40.48s for 2048 cards), while this search process can be pre-computed asynchronously.

## VI. CONCLUSION

This paper presents *Chameleon*, a novel fault-tolerance system for distributed training that adaptively selects recovery policies based on real-time system states. The core components of *Chameleon*—the Planner, Estimator, and Restorer—enable fast and comprehensive policy search, accurate performance modeling and estimation, and low-overhead communication optimization. These features allow *Chameleon* to employ the optimal execution plan, achieving more robust and efficient training compared to single-strategy approaches. Our evaluations in both simulated and real-world environments demonstrate that *Chameleon* significantly improves training efficiency and fault tolerance, while ensuring good memory management, model convergence, and scalability.

## ACKNOWLEDGMENT

This work was supported by the Key Program of the National Natural Science Foundation of China under Grant Nos. 62325205 and 62502198, the Natural Science Foundation of Jiangsu Province under Grant Nos. BK20243053 and BK20251224, the Nanjing “U35” Talent Cultivation Program (No. U (2024) 001), and the Nanjing University-China Mobile Communications Group Co., Ltd. Joint Institute.

## REFERENCES

- [1] A. Grattafiori, A. Dubey, A. Jauhri *et al.*, “The llama 3 herd of models,” *arXiv preprint arXiv:2407.21783*, 2024.
- [2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altschmidt, S. Altman, S. Anadkat *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [3] Z. Wang, Z. Jia, S. Zheng, Z. Zhang *et al.*, “Gemini: Fast failure recovery in distributed training with in-memory checkpoints,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 364–381. [Online]. Available: <https://doi.org/10.1145/3600006.3613145>
- [4] A. Eisenman, K. K. Matam, S. Ingram *et al.*, “Check-N-Run: a checkpointing system for training deep learning recommendation models,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 929–943. [Online]. Available: <https://www.usenix.org/conference/nsdi22/presentation/eisenman>
- [5] Z. Jiang, H. Lin, Y. Zhong, Q. Huang *et al.*, “Megascall: Scaling large language model training to more than 10,000 gpus,” 2024. [Online]. Available: <https://arxiv.org/abs/2402.15627>
- [6] L. Xie, J. Zhai, B. Wu, Y. Wang *et al.*, “Elan: Towards generic and efficient elastic training for deep learning,” in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, 2020, pp. 78–88.
- [7] J. Thorpe, P. Zhao, J. Eyoifson, Y. Qiao, Z. Jia, M. Zhang, R. Netravali, and G. H. Xu, “Bamboo: Making preemptible instances resilient for affordable training of large {DNNs},” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 497–513.
- [8] I. Jang, Z. Yang, Z. Zhang, X. Jin, and M. Chowdhury, “Oobleck: Resilient distributed training of large models using pipeline templates,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 382–395.
- [9] S. Gandhi, M. Zhao, A. Skiadopoulos, and C. Kozyrakis, “Recycle: Resilient training of large dnn using pipeline adaptation,” in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024, pp. 211–228.
- [10] M. Wagenländer, G. Li, B. Zhao, L. Mai, and P. Pietzuch, “Tenplex: Dynamic parallelism for deep learning using parallelizable tensor collections,” in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024, pp. 195–210.
- [11] H. Liao, J. Tu, J. Xia, H. Liu, X. Zhou, H. Yuan, and Y. Hu, “Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 789–801.
- [12] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis *et al.*, “Accurate, large minibatch sgd: Training imagenet in 1 hour,” 2018. [Online]. Available: <https://arxiv.org/abs/1706.02677>
- [13] A. Sergeev and M. D. Balso, “Horovod: fast and easy distributed deep learning in tensorflow,” 2018. [Online]. Available: <https://arxiv.org/abs/1802.05799>
- [14] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, “Imagenet training in minutes,” 2018. [Online]. Available: <https://arxiv.org/abs/1709.05011>
- [15] S. Smith, M. Patwary, B. Norick, P. LeGresley *et al.*, “Using deepspeed and megatron to train megatron-turing nlq 530b, a large-scale generative language model,” 2022. [Online]. Available: <https://arxiv.org/abs/2201.11990>
- [16] Y. Huang, Y. Cheng, A. Bapna *et al.*, “Gpipe: Efficient training of giant neural networks using pipeline parallelism,” 2019. [Online]. Available: <https://arxiv.org/abs/1811.06965>
- [17] T. Kim, H. Kim, G.-I. Yu, and B.-G. Chun, “BPipe: Memory-balanced pipeline parallelism for training large language models,” in *Proceedings of the 40th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, 23–29 Jul 2023, pp. 16639–16653. [Online]. Available: <https://proceedings.mlr.press/v202/kim23l.html>
- [18] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, “Zero: Memory optimizations toward training trillion parameter models,” 2020. [Online]. Available: <https://arxiv.org/abs/1910.02054>
- [19] W. Cai, J. Jiang, L. Qin *et al.*, “Shortcut-connected expert parallelism for accelerating mixture-of-experts,” 2025. [Online]. Available: <https://arxiv.org/abs/2404.05019>
- [20] Y. Qian, F. Li, X. Ji *et al.*, “Eps-moe: Expert pipeline scheduler for cost-efficient moe inference,” 2025. [Online]. Available: <https://arxiv.org/abs/2410.12247>
- [21] D. Liu, Z. Yan, X. Yao, T. Liu *et al.*, “Moe parallel folding: Heterogeneous parallelism mappings for efficient large-scale moe model training with megatron core,” 2025. [Online]. Available: <https://arxiv.org/abs/2504.14960>
- [22] W. Fedus, B. Zoph, and N. Shazeer, “Switch transformers: scaling to trillion parameter models with simple and efficient sparsity,” *J. Mach. Learn. Res.*, vol. 23, no. 1, Jan. 2022.
- [23] C. Jin, Z. Jiang, Z. Bai *et al.*, “Megascall-moe: Large-scale communication-efficient training of mixture-of-experts models in production,” 2025. [Online]. Available: <https://arxiv.org/abs/2505.11432>
- [24] W. Li, X. Liu, Y. Li *et al.*, “Understanding communication characteristics of distributed training,” in *Proceedings of the 8th Asia-Pacific Workshop on Networking*, ser. APNet ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1–8. [Online]. Available: <https://doi.org/10.1145/3663408.3663409>
- [25] D. Moolchandani, J. Kundu *et al.*, “Amped: An analytical model for performance in distributed training of transformers,” in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 306–315.
- [26] V. Korthikanti, J. Casper, S. Lym, L. McAfee *et al.*, “Reducing activation recomputation in large transformer models,” 2022. [Online]. Available: <https://arxiv.org/abs/2205.05198>
- [27] S. Li and T. Hoefler, “Chimera: efficiently training large-scale neural networks with bidirectional pipelines,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’21. ACM, Nov. 2021, p. 1–14. [Online]. Available: <http://dx.doi.org/10.1145/3458817.3476145>
- [28] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra, “Varuna: scalable, low-cost training of massive deep learning models,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 472–487.
- [29] X. Kang, G. Xiang, Y. Wang, H. Zhang, Y. Fang, Y. Zhou, Z. Tang, Y. Lv, E. Maman, M. Wasserman, A. Zameret, Z. Bian, S. Chen, Z. Yu, J. Wang, X. Wu, Y. Zheng, C. Tian, and X. Chu, “Elaswave: An elastic-native system for scalable hybrid-parallel training,” 2025. [Online]. Available: <https://arxiv.org/abs/2510.00606>
- [30] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, “Failures in large scale systems: Long-term measurement, analysis, and implications,” in *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.
- [31] Q. Weng, W. Xiao, Y. Yu, W. Wang *et al.*, “MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 945–960. [Online]. Available: <https://www.usenix.org/conference/nsdi22/presentation/weng>
- [32] Y. Deng, X. Shi, Z. Jiang, X. Zhang *et al.*, “Minder: Faulty machine detection for large-scale distributed model training,” in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. Philadelphia, PA: USENIX Association, Apr. 2025, pp. 505–521. [Online]. Available: <https://www.usenix.org/conference/nsdi25/presentation/deng>
- [33] P. Qi, X. Wan, G. Huang, and M. Lin, “Zero bubble pipeline parallelism,” 2023. [Online]. Available: <https://arxiv.org/abs/2401.10241>
- [34] L. Zheng, Z. Li, H. Zhang, Y. Zhuang *et al.*, “Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 559–578. [Online]. Available: <https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin>
- [35] H. Kuhn, “The hungarian method for the assignment problem,” *Naval Research Logistic Quarterly*, vol. 2, 05 2012.
- [36] B. Wróblewski, G. Gottardo, and A. Zouzias, “Parallel scan on ascend ai accelerators,” 2025. [Online]. Available: <https://arxiv.org/abs/2505.15112>
- [37] H. Touvron, L. Martin, K. Stone, P. Albert *et al.*, “Llama 2: Open foundation and fine-tuned chat models,” 2023. [Online]. Available: <https://arxiv.org/abs/2307.09288>