ECCHECK: Enhancing In-Memory Checkpoint with Erasure Coding in Distributed DNN Training

Guicheng Qi
HKU
u3010274@connect.hku.hk

Zongpeng Li
Tsinghua University
zongpeng@tsinghua.edu.cn

Chuan Wu
HKU
cwu@cs.hku.hk

Zhuwei Peng, Yi Zheng
Huawei
{pengzhuwei,zhengyi29}@huawei.com

***Abstract*—Distributed large model training is intensively time and resource consuming. Failures during the long training period are often inevitable, and can incur substantial recovery costs. Checkpointing has been the standard fault tolerance approach, which periodically stores the latest model states at remote persistent storage. This process can be time-consuming due to limited network bandwidth, and adversely affects training throughput. In-memory checkpointing addresses this issue by saving checkpoint data into host memory instead of remote storage. However, host memory is non-persistent, and may not provide sufficient resilience in case of machine failure. We propose ECCHECK, a novel in-memory checkpoint system that employs erasure coding to enhance fault tolerance in distributed deep neural network training. ECCHECK advocates serialization-free encoding and decoding in model checkpointing. Several techniques are proposed to minimize computation and communication overhead incurred by erasure coding. Extensive experiments demonstrate that ECCHECK achieves superior fault tolerance compared to state-of-the-art solutions, while maintaining high checkpointing frequency, low checkpointing stalls, and fast recovery from failures.**

***Index Terms*—distributed training, fault tolerance, in-memory checkpoint, erasure coding**

I. INTRODUCTION

The emergence of large deep learning models such as GPT-4 [23], LLaMA [34], and PaLM-E [8] has unlocked new possibilities in natural language processing, computer vision, and beyond. Training these models, with hundreds of billions of parameters, requires vast computation resources: often utilizing hundreds or even tens of thousands of GPUs for weeks or even months [33]. This protracted training process inevitably encounters numerous faults, including software glitches and hardware breakdowns. For example, the training of Llama 3.1 405B experienced 419 unexpected failures over 54 days, encompassing various issues such as GPU device malfunctions, network infrastructure failures, server hardware problems, and software bugs. This averaged to approximately one failure every 3 hours, with 78% of these failures attributed to confirmed or suspected hardware issues [33]. Upon failure, training often needs to be restarted, incurring substantial recovery costs in time and resources.

Checkpointing has been the *de facto* standard to enable failure recovery in distributed model training, periodically storing training states (*e.g.*, model parameters, optimizer states, dataloader states) on remote persistent storage [24]. By loading the latest checkpoint from the storage, it allows training to resume from the corresponding model states. To ensure model consistency, training is paused during this recovery process, and loading checkpoint from the remote storage is often time-consuming due to network bandwidth constraints. To maintain acceptable training throughput, checkpointing frequency is typically kept relatively low, once every few hours [1]. Consequently, failures may cause the model to roll back to a rather old state, wasting significant time and computation resources. For example, 178,000 GPU hours are wasted due to various training failures in the training of OPT-175B [36].

To address the low checkpointing frequency, CheckFreq [20] adaptively adjusts checkpointing frequency in real time, to strike a balance between checkpoint overhead and failure recovery time. It also implements a two-phase checkpointing process, snapshot and persist, to reduce training interruptions, but still transfers the checkpoint to remote storage. Check-N-Run [9] uses incremental checkpointing for deep recommendation models and employs quantization techniques to reduce the checkpoint size, expediting data transfer to remote storage. However, quantization may compromise model accuracy. GEMINI [35] adopts in-memory checkpointing, where checkpoints are stored directly in CPU memory of host machines. This enables high checkpointing frequencies due to much higher I/O bandwidth. However, GEMINI's node grouping strategy restricts its ability to handle consecutive node failures effectively, and it still relies on remote storage when concurrent failures happen in the same group, due to the non-persistent nature of host memory.

Addressing these challenges, we propose ECCHECK, an in-memory checkpoint system for distributed large model training that leverages erasure coding for distributed checkpoint storage and efficient failure recovery. By harnessing the high bandwidth of inter-node connections, ECCHECK can achieve significantly higher checkpointing frequency compared to remote storage-based checkpointing. Although CPU memory is non-persistent, erasure coding can provide more fault tolerance than replication, under the same level of redundancy.

ECCHECK can tolerate multiple node (*aka* machine) failures while maintaining high checkpoint frequency and fast training recovery, without compromising model accuracy or training throughput. To efficiently integrate erasure coding into distributed DNN training, we design *a serialization-free encoding and decoding protocol* for model checkpointing and

This work was supported in part by the Huawei Cloud Computing Technologies Co., Ltd. under Grant TC20240629018 and by Hong Kong Research Grants Council (RGC) under Contracts HKU 17204423(GRF) and C7004-22G (CRF).

recovery. To alleviate the additional computational and communication overhead due to checkpoint coding, we propose several techniques to accelerate checkpointing and to avoid interference with distributed model training. Our contributions can be summarized as follows:

▷ We design ECCHECK, an innovative in-memory checkpointing system that utilizes erasure coding for fault tolerance to multiple machine-level failures in distributed DNN training. The system strategically classifies all n training nodes into k data nodes and m parity nodes. By applying erasure coding to process checkpoints in a distributed manner, ECCHECK enables the training system to withstand up to m concurrent failures. ECCHECK leverages CPU resources for checkpoint encoding and communication, and does not compete with model training for GPU computation and memory resources. ECCHECK can be applied in distributed training using any type of parallelism (tensor parallelism, pipeline parallelism, fully sharded data parallelism).

▷ We design a serialization-free encoding and decoding protocol for processing model checkpoints. It decomposes the checkpoint into distinct components for tailored processing, reducing overhead and improving efficiency.

▷ We advocate multiple optimization techniques in ECCHECK to minimize both computational and communication overhead incurred by erasure coding. ECCHECK employs the efficient Cauchy Reed-Solomon [2] code as its encoding scheme, leveraging CPU thread pool to accelerate encoding. ECCHECK utilizes a sweep line algorithm to optimize the selection of data nodes and parity nodes, and determines the ideal XOR reduction target (a crucial step in ECCHECK's checkpointing process), to reduce communication volume. Furthermore, ECCHECK exploits network idle time slots for communication tasks and employs a pipelined approach to overlap checkpoint computation and communication.

▷ We implement ECCHECK on top of Megatron-LM [32] [22] [16]. We conduct extensive experiments to demonstrate the scalability, fault tolerance, and performance benefits of ECCHECK under various distributed training configurations. ECCHECK not only provides enhanced fault tolerance compared to replication-based in-memory checkpointing, but also features reduced checkpointing time (by up to $5.2\times$) and faster recovery (by up to $13.9\times$) compared to remote storage-based checkpointing.

II. Background and Motivation

A. Distributed Model Training and Checkpointing

Distributed training represents the *status quo* of training large deep learning models, with computation distributed across multiple workers/devices. Several key parallelism strategies have been widely adopted. Data parallelism [12] replicates the entire model at each device, with each device processing a different mini-batch of training data. Tensor parallelism [32] partitions tensors among devices, allowing parallel computation on different parts of the same model layer. Pipeline parallelism [13] [21] divides the model into stages, running different layers on different devices, creating a computation pipeline among microbatches of input data. The parallelism strategies can be combined to optimize resource utilization and training scalability under different model architectures and hardware configurations. An example of hybrid parallelism is given in Fig. 1, with data, pipeline, and tensor parallelism combined to train a DNN model.

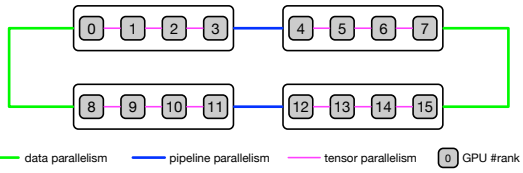
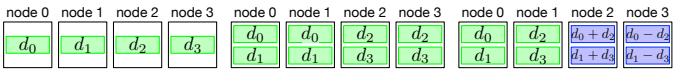


Fig. 1: Hybrid parallelism in a 4-node distributed training system, where each node contains 4 GPUs. The DNN model is divided into 2 pipeline-parallel stages, with each stage partitioned among 4 GPUs, and each stage having a replica.



(a) Each of the four nodes stores one chunk. (b) Replication-based Approach. (c) Erasure coding-based Approach.

Fig. 2: Replication vs. erasure coding of checkpoint. With the same amount of redundancy, erasure coding provides stronger fault tolerance.

A typical checkpoint includes model states, optimizer states, dataloader states, and additional metadata (such as iteration count and checkpoint version), all structured as a dictionary. The checkpointing process generally involves several steps: synchronizing model states across distributed workers to ensure model consistency, serializing checkpoint data into compact byte objects, and storing the byte objects to persistent storage. In distributed environments, effective checkpointing must carefully balance fault tolerance and training performance.

Since the interconnection bandwidth among nodes is significantly higher than that to remote storage, in-memory checkpointing can potentially achieve a high checkpointing frequency. Nonetheless, CPU memory is non-persistent and in-memory checkpoint would be lost when the machine fails. GEMINI [35], a state-of-the-art in-memory checkpointing system, employs a replication-based method to ensure fault tolerance. It divides all nodes into groups of the same size; within a group, each node stores replications of all checkpoint data in that group, thereby providing fault tolerance. In Fig. 2, four nodes in a distributed training system each store one checkpoint data chunk. Nodes 0-1 and 2-3 are organized into two separate groups. With the inter-group replication design of GEMINI (Fig. 2b), the system can tolerate up to two concurrent node failures, one in each group. Using a larger group size may allow tolerating more concurrent failures, but may incur significant communication and memory overhead, as each node needs to broadcast its checkpoint to all nodes within the same group.

B. Erasure Coding

Erasure coding is a classic method for fault tolerance with data redundancy, and has been well studied/applied in

the fields of distributed storage [7], cloud computing [17], and network communications [30]. Compared to replication, erasure coding requires less redundancy to achieve the same level of fault tolerance, at the cost of additional computational overhead.

The core idea of erasure coding is to use an encoding matrix (also known as a generator matrix) to transform k original data packets into $n = k + m$ encoded packets, such that any k of these n packets can reconstruct the original k data packets. The encoding matrix is of size $(k+m) \times k$, with any k rows being linearly independent. An erasure code can be either *systematic* or *non-systematic* [25]. Systematic erasure codes, widely adopted in practice, preserve the original k data packets and append m parity packets. The encoding matrix is $E = [I_k, E']^T$, where I_k is a $k \times k$ identity matrix, E' is a $k \times m$ matrix. Non-systematic codes transform the entire message, resulting in an encoded output that does not contain the original data. Systematic codes have lower encoding and decoding complexity, while non-systematic codes can provide enhanced error correction capabilities and improved security. We focus on systematic codes due to their lower computational overhead; security considerations are beyond the scope of this work.

In Fig. 2c, node 0 and node 1 store the original data chunks $[d_0, d_1]$ and $[d_2, d_3]$, respectively. In contrast, node 2 and node 3 store the encoded parity chunks $[d_0 + d_2, d_1 + d_3]$ and $[d_0 - d_2, d_1 - d_3]$. The system can tolerate any two concurrent failures. For instance, if node 0 and node 1 fail concurrently, the checkpoint chunks stored on node 2 and node 3 can be used to recover d_0, d_1, d_2, d_3. For both replication (Fig. 2b) and erasure coding (Fig. 2c), each node needs to store twice its original checkpoint data size. Erasure coding can provide better fault tolerance than replication under the same amount of redundancy.

On the other hand, erasure coding incurs additional computation (for chunk encoding and decoding) and communication (for transferring chunks among nodes for distributed storage) overhead. Is it worth adopting erasure coding which trades off extra computation for better fault resilience? Consider a 2000-node cluster, which can be divided into 500 groups, each containing 4 nodes. Let p denote the probability of a single node experiencing failure, assuming node failures are independent [31] [11]. If we use a replication-based checkpointing system, for each 4-node group, the recovery rate (*i.e.*, the probability of recovering all checkpoint data from node failures) is given by:

$$R_{rep} = (1-p)^4 + \binom{4}{1} p(1-p)^3 + (\binom{4}{2} - 2)p^2(1-p)^2 \quad (1)$$

If we use an erasure coding-based method for each 4-node group, as illustrated in Fig. 2c, the group can tolerate any two concurrent failures. Under this arrangement, the recovery rate is:

$$R_{era} = (1-p)^4 + \binom{4}{1} p(1-p)^3 + \binom{4}{2} p^2(1-p)^2 \quad (2)$$

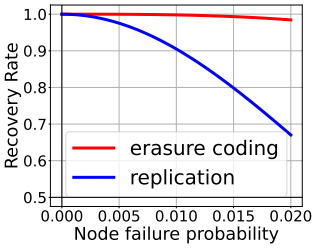


Fig. 3: Recovery rate comparison between replication-based and erasure coding-based methods in a 2000-node cluster.

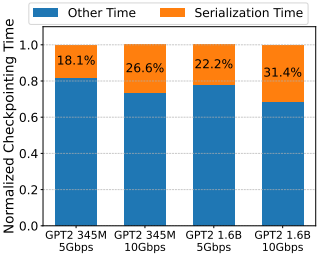


Fig. 4: Serialization overhead in checkpointing for GPT2 models trained on 4 NVIDIA A100 40G GPUs. Checkpoints stored to remote storage; aggregated bandwidth marked under each case.

We obtain that $R_{era} - R_{rep} = 2p^2(1-p)^2$. For the 2000-node cluster divided into 500 groups, considering that any group failure renders recovery impossible, the overall recovery rate in the cluster is R_{rep}^{500} with replication-based checkpointing, and R_{era}^{500} with the erasure coding-based method. Fig. 3 compares the fault tolerance capacity between the two methods. With the increase of node failure rate, the better failure resilience of erasure coding is more pronounced. If we can optimize erasure coding to incur only marginally more overhead than replication, without significantly interfering with training, then adopting erasure coding is worthwhile due to substantially improved fault tolerance in large-scale distributed systems.

C. Opportunities and Challenges

Existing research has explored some approaches to applying erasure coding to in-memory checkpointing. For instance, a study presented at HASE [4] investigates diskless checkpointing in distributed scenarios; however, it is not well-suited for DNN training workloads. ECRM [37] is specifically tailored for recommendation models, which limits its applicability to other types of DNNs and broader distributed training scenarios. Furthermore, it can only tolerate a single node failure, making it less effective in scenarios requiring higher fault tolerance. The application of erasure coding to in-memory checkpointing presents promising opportunities for improving failure resilience of distributed training systems, while maintaining high checkpointing and recovery efficiency. Nevertheless, several challenges need to be addressed to fully realize the benefits of erasure code-enhanced in-memory checkpointing.

Challenge 1: Designing an efficient encoding and decoding protocol. In distributed model training, each worker maintains a sharded `state_dict` (*i.e.*, the checkpoint data), of which model and optimizer states are maintained in GPU memory, Random Number Generator (RNG) states in the dataloader and other training metadata like iteration count are stored in CPU memory. Since encoding is applied to a contiguous block of memory, it is necessary to transfer the non-contiguous checkpoint into a contiguous block of memory in the CPU. Therefore, an intuitive first step is to serialize the checkpoint into compact byte objects, treating them as data blocks to

be subsequently encoded. However, serialization can incur significant overhead [19], as shown in the examples in Fig. 4, constituting a significant proportion of the overall checkpointing time. Notably, as the aggregated bandwidth to remote storage increases, the relative serialization overhead grows in comparison to the overall checkpointing time, as transferring the checkpoint takes less time while the serialization time remains unchanged. Furthermore, when encoding after serialization, the process of serialization is on the critical path of checkpointing, which cannot be executed with encoding in a pipelined manner. Given these limitations, a serialization-free encoding and decoding approach is more desirable for processing the checkpoint.

Challenge 2: Minimizing the overhead incurred by erasure coding. Erasure coding inevitably introduces additional computation and communication overhead. If the overhead significantly hinders training throughput or reduces checkpointing frequency, the approach becomes less appealing. Fortunately, existing research has developed techniques to accelerate encoding and decoding, achieving impressive throughput exceeding 40 Gbps [38]. Additionally, communication during distributed training typically follows predictable patterns [29]. For example, in tensor parallelism, the communication involves frequent all-gather and reduce-scatter operations to reconstruct full tensors or aggregate partial computations; in pipeline parallelism, the communication resembles a pipeline, with activations flowing forward and gradients flowing backward between adjacent stages, and an all-reduce operation is applied at the end of a mini-batch to update gradients. Many idle network time slots exist during distributed training [29], *e.g.*, in tensor parallelism, idle time occurs between consecutive all-gather or reduce-scatter operations; in pipeline parallelism, network idling happens when some stages are under computation while others are waiting for data. They can be utilized for the communication incurred by erasure coding, reducing its interference with the training process. We propose several optimization techniques to minimize the computation and communication overhead introduced by erasure coding, developing robust and efficient in-memory checkpointing solutions that exploit the advantages of erasure coding while mitigating potential overheads.

III. ECCHECK

A. Overview

We propose ECCHECK, an in-memory checkpointing system that utilizes erasure coding. As illustrated in Fig. 5, ECCHECK adopts an erasure coding-based in-memory checkpointing scheme. All training nodes follow 4 steps during checkpointing. First, each worker (on a GPU) offloads the tensor data (*i.e.*, GPU states, including model and optimizer states) onto CPU memory (step 1), also known as CUDA `DtoH` copy. Upon completion, training continues. The following steps are executed asynchronously. After offloading, all checkpoint data resides in CPU memory, where the CPU performs encoding, according to the encoding protocol (step 2). Then the encoded chunks are transferred to other nodes following a predetermined communication strategy (step 3). After that, each node stores enough coded checkpoint chunks to ensure that any subset of them can recover the original checkpoint data. Additionally, all checkpoint data are transferred to a remote storage at a low frequency (step 4) to counter catastrophic failures in which surviving nodes cannot recover the original checkpoint.

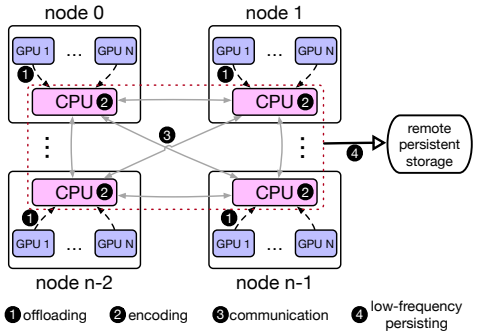


Fig. 5: Checkpointing architecture of ECCHECK.

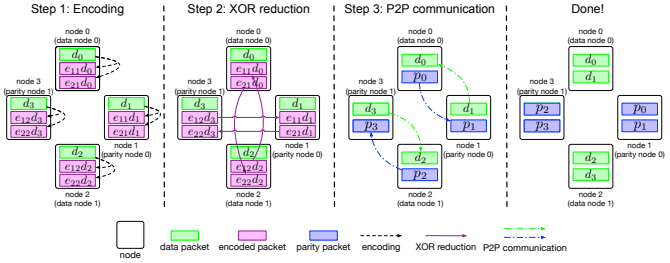


Fig. 6: ECCHECK: example of checkpointing.

ECCHECK can be applied to a distributed training system using any type of parallelism. Since data parallelism naturally provides model replicas, ECCHECK is more useful with pipeline-parallel, tensor-parallel, and fully sharded data parallel (FSDP) training, where no full replicas are available for checkpoint recovery. In distributed training, each worker maintains a sharded `state_dict` of the model, which is the data to be checkpointed. The sharding of `state_dict` is determined by the specific parallelism strategy of choice. In tensor parallelism, `state_dict` is sharded across devices based on tensor dimensions, with each shard containing a portion of the model's parameters, typically split along the hidden dimension for linear layers and attention heads for transformer models. In pipeline parallelism, `state_dict` is sharded by model layers, with each shard containing the parameters of one or more consecutive layers of the model. Each shard typically includes model parameters, optimizer states, and other necessary training metadata specific to that portion of the model. ECCHECK utilizes erasure coding to encode the sharded `state_dict` at each node, and the encoded dictionary is referred to as `encoded_state_dict`.

B. Design Principle

We abstract each sharded `state_dict` (*i.e.*, the checkpoint) on each worker as a data packet. Consider n nodes in a

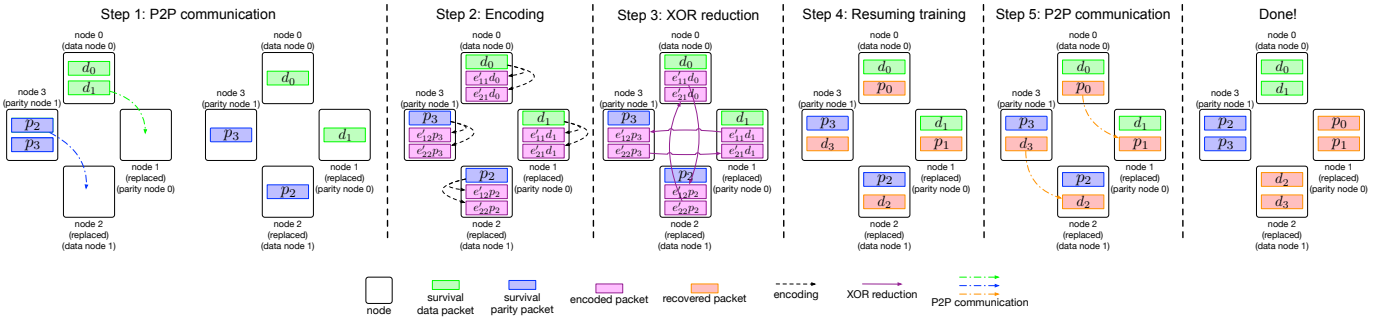


Fig. 7: ECCHECK: example of recovery.

distributed training system, each node has g devices (GPUs), and there will be ng checkpoint data packets, which altogether constitute the complete checkpoint of the DNN model. We evenly divide the ng checkpoint data packets into k chunks, and encode the k chunks into m parity chunks ($m = n - k$), so that we can obtain n chunks in total. We allow each node to store one chunk, and any k of the n chunks can recover the original k chunks (*i.e.*, the original ng checkpoint data packets). In this way, the distributed training system can tolerate any m concurrent node failures.

For example, when there are 4 nodes and each node has one device, there are 4 data packets, d_0, d_1, d_2, d_3. We evenly divide the four data packets into 2 chunks D_0 and D_1, each containing 2 data packets, *i.e.*, $D_0 = [d_0, d_1]$ and $D_1 = [d_2, d_3]$. Let E be a 4×2 encoding matrix (any two rows of E are linearly independent). Let P_0 and P_1 denote the 2 parity chunks. The encoding is performed as follows:

$$\begin{bmatrix} D_0 \\ D_1 \\ P_0 \\ P_1 \end{bmatrix} = E \begin{bmatrix} D_0 \\ D_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ e_{11} & e_{12} \\ e_{21} & e_{22} \end{bmatrix} \begin{bmatrix} D_0 \\ D_1 \end{bmatrix} = \begin{bmatrix} D_0 \\ D_1 \\ e_{11}D_0 + e_{12}D_1 \\ e_{21}D_0 + e_{22}D_1 \end{bmatrix} \tag{3}$$

As a result, we have

$$\begin{aligned} P_0 &= \left[e_{11}D_0 + e_{12}D_1\right] = \left[e_{11}d_0 + e_{12}d_2, \quad e_{11}d_1 + e_{12}d_3\right] \\ P_1 &= \left[e_{21}D_0 + e_{22}D_1\right] = \left[e_{21}d_0 + e_{22}d_2, \quad e_{21}d_1 + e_{22}d_3\right] \end{aligned} \tag{4}$$

Then we distribute D_0, D_1, P_0, P_1 among the four nodes, so that any two nodes can recover the original D_0 and D_1. The decoding process, which recovers D_0 and D_1, *e.g.*, when only D_0 and P_1 are available, is illustrated as follows:

$$\begin{aligned} &\begin{bmatrix} D_0 \\ P_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ e_{21} & e_{22} \end{bmatrix} \begin{bmatrix} D_0 \\ D_1 \end{bmatrix} \\ \Longrightarrow &\begin{bmatrix} D_0 \\ D_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ e_{21} & e_{22} \end{bmatrix}^{-1} \begin{bmatrix} D_0 \\ P_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -\frac{e_{21}}{e_{22}} & \frac{1}{e_{22}} \end{bmatrix} \begin{bmatrix} D_0 \\ P_1 \end{bmatrix} \end{aligned} \tag{5}$$

To integrate the checkpoint encoding process of Eqn. 3 into a distributed training system, ECCHECK first categorizes the n nodes into two types, k data nodes and m parity nodes with $n = k + m$, so that the k data nodes store data chunks and the m parity nodes store parity chunks. The selection of data nodes is crucial and will be detailed as part of the communication strategy in Sec. IV-B. A three-step procedure is then carried out for distributed checkpoint encoding and storage among the nodes: encoding, XOR reduction, and P2P communication. In the example in Fig. 6, there are two data nodes ($k = 2$) and two parity nodes ($m = 2$) in a 4-node distributed training system, with one worker per node. Each worker first encodes its checkpoint data packet into m encoded packets. Next, XOR reduction operations are conducted across different subsets of nodes, with each operation storing the XOR result (the parity packet) of m encoded packets on a target node. For instance, the XOR reduction is conducted between node 1 and node 3, and the XOR result of $e_{11}d_1$ (in node 1) and $e_{12}d_3$ (in node 3) is p_1, which is stored in node 1 (the target node). Finally, the nodes perform P2P communication to transfer data packets or parity packets to the respective data or parity nodes, so that node 0 stores D_0, node 1 stores P_0, node 2 stores D_1, and node 3 stores P_1.

Let

$$\begin{aligned} p_0 = e_{11}d_0 + e_{12}d_2,\ p_1 = e_{11}d_1 + e_{12}d_3, \\ p_2 = e_{21}d_0 + e_{22}d_2,\ p_3 = e_{21}d_1 + e_{22}d_3 \end{aligned} \tag{6}$$

Then we obtain:

$$D_0 = \left[d_0, d_1\right],\ D_1 = \left[d_2, d_3\right],\ P_0 = \left[p_0, p_1\right],\ P_1 = \left[p_2, p_3\right] \tag{7}$$

We will discuss in Sec. IV-A that we use a bitmatrix $B(E)$ as the encoding matrix, which is computed over $GF(2)$, so addition is equivalent to bit-wise XOR – the reason why the second step is called XOR reduction. The above derivation is related to the three steps in Fig. 6. For example, node 0 initially has d_0; computing $e_{11}d_0$ corresponds to the encoding step, and $e_{11}d_0 + e_{12}d_2$ corresponds to the XOR reduction step, resulting in p_0; then node 0 sends p_0 to node 1, corresponding to the P2P communication step.

To recover the training states from the distributed, encoded checkpoint chunks following Eqn. 5, there are two key tasks: (1) Each surviving or replaced node should obtain the original checkpoint for resuming training; (2) Each surviving and replaced node should store the same data or parity chunk as it did before the failure, to restore the same fault tolerance capacity. ECCHECK includes two distinct recovery workflows, corresponding to two failure cases:

- When all data nodes survive, the original checkpoint data remains intact. The recovery workflow goes as follows: Data nodes send the replaced nodes their corresponding parts of the checkpoint data. Upon completion, each node can use its checkpoint data to resume training. Then the lost parity packets are encoded and stored in the corresponding parity nodes.

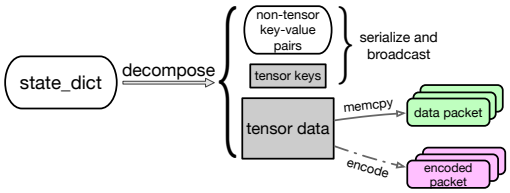


Fig. 8: Serialization-free encoding protocol.

• When some data nodes fail, resulting in the loss of some original checkpoint chunks, we first decode the surviving packets to reconstruct all data packets, enabling each node to have its own checkpoint data and resume training immediately. After that, the decoded data or parity packets are transferred to the corresponding data or parity nodes, to restore the fault tolerance capacity.

Fig. 7 illustrates an example recovery scenario where nodes 1 and 2 are down, following the same setting as in Fig. 6. First, node 0 sends d_1 to the replaced node 1, and node 3 sends p_2 to the replaced node 2, distributing the decoding workload across all nodes to speed up recovery. Node 0 still stores d_1 in its memory, so does node 3, and this is omitted in the figure for clarity. Each node then encodes its data packet to obtain two encoded packets using a decoding matrix E' (E' is based on the inverse of a submatrix of encoding matrix E and Eqn. 5 is an example). After encoding and XOR reduction, each node stores a data packet and a parity packet. Training can then continue. Finally, the recovered data packets and parity packets are sent to their corresponding data or parity nodes.

C. Encoding and Decoding Protocol

An intuitive approach to encode checkpoint is to first serialize its data into compact byte objects, treating them as data packets to be subsequently encoded. However, serialization can incur significant overhead [19] and cannot be executed in a pipelined manner with subsequent encoding and communication operations. In ECCHECK, we eliminate the need for serialization, and propose a serialization-free encoding and decoding protocol to save and load the checkpoint, based on the following observations:

• `state_dict` can be decomposed into three distinct components: non-tensor key-value pairs (such as training metadata like iteration count), tensor keys, and tensor data (the latter two components collectively form the tensor key-value pairs, and the tensors are model states, optimizer states, and RNG states in the dataloader). Non-tensor key-value pairs and tensor keys are relatively small in size, while tensor data constitutes the vast majority. For instance, in `state_dict` of GPT2-345M, non-tensor key-value pairs and tensor keys each occupy approximately 52KB (less than 0.001% of the total checkpoint size), while tensor data account for around 6.5GB (over 99.99% of the total).

• The majority of data to checkpoint (including model and optimizer states) resides in GPU memory, while others (non-tensor key-value pairs, tensor keys, a small part of tensor data) are in CPU memory.

• Each tensor's data is stored contiguously in memory, while their sizes can vary significantly.

Our serialization-free encoding protocol consists of the following steps (Fig. 8):

1. Analyze and decompose `state_dict` into three components: non-tensor key-value pairs (dict), tensor keys (list), and tensor data (list). Transfer tensor data residing in GPU memory to CPU memory.

2. Serialize and broadcast the non-tensor key-value pairs and tensor keys to all other workers, as their communication traffic is negligible compared to tensor data.

3. For the list of tensor data (now all on CPU memory): (a) Copy the original tensor data to reserved data buffers for data packets. (b) Encode each tensor and store the encoded result into reserved encoding buffers for encoded packets. (c) When a buffer is full, it becomes a data or encoded packet. XOR reduction and P2P communication operations are performed on the corresponding data and encoding buffers (packets).

Upon completion of checkpoint encoding, each node stores serialized non-tensor key-value pairs, serialized tensor keys, and data chunk or parity chunk, separately in CPU memory.

The decoding protocol follows the same three-step procedure above for encoding to decode the data packets or parity packets on surviving and replaced nodes, except for replacing the encoding matrix by the decoding matrix. That is, it also computes 2 'parity' chunks from 2 'data' chunks, just like the encoding process. In the example shown in Eqn. 5 and Fig. 7, the 'data' chunks are D_0 and P_1, while the 'parity' chunks are D_1 and P_0. Then the original tensor data are obtained according to the decoded data packets. By combining the tensor data with the non-tensor key-value pairs and tensor keys, the original `state_dict` can be reconstructed.

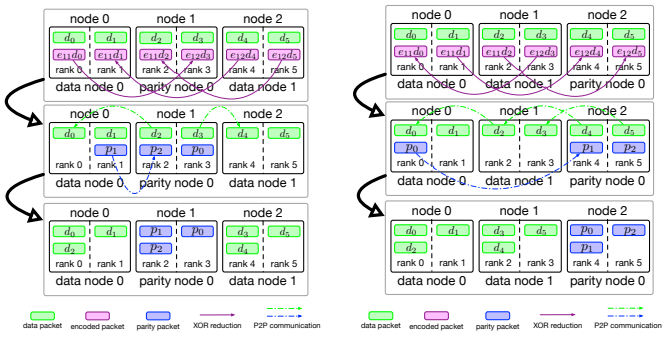
This design significantly improves checkpoint efficiency by eliminating the need for full serialization of `state_dict`, focusing computational efforts on the most data-intensive components. Furthermore, once decomposition of `state_dict` is done and tensor data has been transferred to host memory, DNN training can immediately continue, and ECCHECK executes the remaining checkpointing steps asynchronously, allowing less training stalls.

IV. Overhead Minimization

A. Encoding Optimizations

ECCHECK's computational overhead primarily occurs during checkpoint encoding. We adopt an efficient encoding scheme, Cauchy Reed-Solomon code, with low computational complexity, and the thread pool technique to accelerate encoding implementation.

Cauchy Reed-Solomon Code. Cauchy Reed-Solomon code [2] [38] [18] is a variant of Reed-Solomon code based on the Cauchy matrix. Its encoding can be implemented by using XOR operations exclusively [2]. Due to space limitation, we omit the details of Cauchy Reed-Solomon code. The encoding process is computed over $GF(2)$. Consider the example in Eqn. 3, where $P_0 = e_{11}D_0 + e_{12}D_1$. The multiplication



(a) node 1 be the parity node (b) node 2 be the parity node

Fig. 9: Examples on different communication volumes resulting from different data and parity nodes selection.

of $e_{11}D_0$ corresponds to computing the XOR result of a bitmatrix $B(e_{11})$ and the binary form of D_0, while the addition of $e_{11}D_0 + e_{12}D_1$ is equivalent to a bitwise XOR operation. This choice of erasure code mitigates the coding overhead, maintaining high training throughput and enabling more frequent checkpointing.

Thread Pool Technique. We adopt Jerasure [26], a widely-adopted library implementing various erasure codes on CPUs, to generate encoding matrices over finite fields $GF(2^\omega)$. To enhance performance, we employ a thread pool technique to parallelize the encoding process. We divide an encoding task, which typically involves encoding a contiguous block of memory into another contiguous block of the same size, into multiple sub-tasks. Each sub-task is responsible for encoding a portion of the memory. We assign threads in the threadpool to execute these sub-tasks concurrently, effectively parallelizing the encoding process across multiple CPU cores. This parallelization significantly improves the encoding speed.

B. Communication Strategy

ECCHECK's communication overhead mainly occurs during XOR reduction and P2P communication. Our communication strategy, which optimally selects data/parity nodes and XOR reduction target, inherently reduces communication volume.

1) Optimal Selection of Data and Parity Nodes: The decision of which nodes serve as data nodes or parity nodes directly impacts the communication traffic during checkpointing. In the example in Fig. 9, there are 3 nodes in a distributed training system, each containing two devices, and we are to decide two data nodes ($k = 2$) and one parity node ($m = 1$). Assume each worker has 1 data packet (1 unit size), and then each worker has a parity packet (1 unit size) as well. In Fig. 9a, when node 1 is designated as the parity node, the checkpoint encoding process requires 3 XOR reduction operations and 3 P2P communication operations, resulting in total communication traffic of 6 units. Especially, node 2 is selected as data node 1 and should eventually store d_3, d_4, d_5, while node 2 already contains d_4, d_5 initially, necessitating the transfer of only one additional data packet. In Fig. 9b, when node 2 is the parity node, there are 3 XOR reduction operations and 4 P2P communication operations, leading to total communication traffic of 7 units. Especially, node 1 is selected as data node 1 which initially stores only d_3, requiring the transfer of two additional data packets. Consequently, the data/parity node selection in Fig. 9a leads to lower communication overhead.

We compute the best data and parity node selection by formulating the problem as a maximum overlap interval pairing problem. Consider two arrays of intervals, $origin\_group$ and $data\_group$, where $origin\_group$ represents the physical distribution of GPUs across machines in a distributed training setup, grouping GPUs by their host machines, while $data\_group$ is a logical partitioning of all GPUs across machines into equal-sized groups based on k (*i.e.*, the number of data nodes). In Fig. 9, $origin\_group = [[0, 1], [2, 3], [4, 5]]$ and $data\_group = [[0, 1, 2], [3, 4, 5]]$. The goal is to find, for each interval in $data\_group$, the interval in $origin\_group$ that has the maximum overlap. The index of this maximally overlapping interval in $origin\_group$ is a corresponding data node. In the above example, for interval $[0, 1, 2]$ in $data\_group$, the maximally overlapping interval in $origin\_group$ is $[0, 1]$, and the index of $[0, 1]$ is 0; thus, node 0 is selected as a data node. After processing all k intervals in $data\_group$, the k data nodes are identified, and the other nodes become parity nodes. It ensures that each designated data node requires the minimum number of additional data packets from other nodes, thereby minimizing the communication traffic in P2P communication. We adopt the sweep line algorithm [3], an efficient algorithm for interval problems, to solve it. The algorithm uses a sweep line that moves from left to right across all interval endpoints in $origin\_group$ and $data\_group$. It maintains a set of active intervals from $origin\_group$ and processes events (interval starts and ends) in chronological order. When encountering a $data\_group$ interval, it compares it with all active $origin\_group$ intervals to find the maximum overlap. This approach efficiently handles all intervals in a single pass, avoiding redundant comparisons. The time complexity of our data/parity node selection is $O((n + m)\log(n + m))$, where n and m are the lengths of $origin\_group$ and $data\_group$, respectively.

2) Optimal Selection of XOR Reduction Target: The number of XOR reduction operations is determined by the `world_size` W (*i.e.*, the number of workers), k and m, and remains constant regardless of which nodes are designated as the parity nodes. Specifically, ECCHECK evenly divides the W workers into k data groups, each containing W/k workers. Consequently, there are W/k reduction groups, each comprising k workers from the k data groups that share the same relative index within their respective data groups. Within each reduction group, there are m XOR reduction operations to generate m parity packets. Therefore, $\frac{W}{k} \cdot m$ XOR reduction operations are performed in total.

While the number of reduction operations is fixed, we can optimize the selection of the reduction target. The reduction target is desirably a parity node to eliminate the need of data transfer in subsequent P2P communication, thereby min-

imizing overall communication traffic. Given W/k reduction groups and g workers per parity node, there are $\frac{W}{k} - g$ reduction groups without parity workers (*i.e.*, workers on a parity node). For groups containing parity workers, we simply assign the XOR reduction target to parity workers, eliminating the need for additional P2P communication. For each reduction group without a parity worker, we must distribute the m XOR reduction results across its k workers and then transfer them to the corresponding parity workers through the P2P communication. The selection of reduction targets in this reduction group depends on the relative values of k and m:

- When $k = m$: the sweep line algorithm typically designates odd-indexed nodes as data nodes and even-indexed nodes as parity nodes. There are $W/k = 2g$ reduction groups, each with k workers and m XOR reduction operations. Since $k = m$, the m XOR reduction results can be directly stored on the k workers.

- When $k > m$: $k - m$ workers in each group are not assigned as reduction targets for any XOR reduction operation, exempting them from P2P communication. ECCHECK assigns m reduction targets sequentially at an interval of $\lfloor k/m \rfloor$ within each reduction group.

- When $k < m$: at least $m - k$ workers in each reduction group are selected as reduction targets multiple times. ECCHECK assigns m reduction targets in a round robin manner within each reduction group.

3) Communication Scheduling: While checkpoint encoding is executed on the CPU, avoiding direct interference with GPU computation, the associated communication shares the same network infrastructure as training communication. Such network resource sharing could potentially impede the training process. Fortunately, training communication typically follows predictable patterns [29]. We can exploit such predictability to strategically schedule checkpointing communication operations during network idle periods, thereby minimizing disruption to the training process.

To identify network idle periods, we profile the time slots for inter-node communication during the first 50 training iterations. Then we synchronize checkpointing communications with these identified idle periods. In tensor-parallel training, we leverage the idle times between synchronization of partial results across devices. For pipeline-parallel setups, we exploit the bubble periods when some stages are waiting for inputs or have finished processing.

C. Pipelined Execution

Further, we execute the three steps (encoding, XOR reduction, and P2P communication) in a pipelining manner, exploiting multi-threading to maximize efficiency: as soon as a data packet is encoded by an *encoding thread*, it is immediately passed on to the *XOR reduction thread*, which performs the necessary XOR operations. Meanwhile, the encoding thread continues uninterrupted with encoding the next data buffer. Upon completion of XOR reduction, the *P2P communication thread* immediately initiates data transfer. This pipelined execution works in concert with the idle time slot scheduling discussed earlier. The *encoding thread* operates continuously, preparing data packets and encoded packets for transmission. However, the *XOR reduction thread* and *P2P communication thread* are synchronized with the identified network idle periods. These threads buffer their operations and execute them only during the profiled idle time slots, ensuring that checkpointing communications do not interfere with training traffic. Such pipelined execution effectively overlaps computation and communication, substantially reducing the overall execution time.

TABLE I: Model configurations

Model	Hidden size	#AH	#Layers	Model size
GPT-2	1600	32	48	1.6B
	2560	40	64	5.3B
	5120	40	64	20B
BERT	1600	32	48	1.6B
	2560	40	64	5.3B
	5120	40	64	20B
T5	1600	32	48	1.6B
	2560	40	64	5.3B
	5120	40	64	20B

Note: AH is short for attention heads.

V. EVALUATION

A. System Implementation

We implement ECCHECK on top of Megatron-LM [32], a state-of-the-art framework for training large models, in 5.2k lines of C++ and Python code. ECCHECK primarily comprises three functions: `eccheck.initialize`, `eccheck.save`, and `eccheck.load`. When training starts, `eccheck.initialize` determines the encoding matrix and the communication strategy, and allocates necessary buffers. It also identifies network idle time slots by online profiling. During training, `eccheck.save` periodically stores the `encoded_state_dict` in host memory. In the event of failures, `eccheck.load` reconstructs the latest version of the original `state_dict` from available `encoded_state_dict` entries. The distributed training utilizes NCCL [5] as the communication backend, and Adam optimizer [15] for parameter updates. We adopt the Gloo [10] communication backend for ECCHECK communication operations, as we use CPU resources for checkpoint encoding and communication, enabling parallel processing without interfering with GPU training.

B. Experimental Setup

Testbed. We experiment on four machines, each equipped with two AMD EPYC 7H12 CPUs and four NVIDIA Tesla A100 GPUs (40 GB) interconnected via NVLink. The host memory of each machine is 512 GB. We train each DNN model using hybrid parallelism: tensor parallelism with a degree of 4 among GPUs on each machine and pipeline parallelism across 4 stages on the four machines. Inter-machine network bandwidth is 100 Gbps. For persistent checkpoint storage, we utilize a remote storage system with aggregated bandwidth of 5 Gbps from the machines to it. To evaluate

scalability, we scale up to a maximum of 32 NVIDIA Tesla V100 GPUs (32 GB).

Benchmarks. We train representative large deep learning models, including GPT-2 [27], BERT [6], and T5 [28], on the Code Parrot dataset [14]. We vary the number of layers and hidden sizes, generating a diverse set of model scales, as shown in Table I. We maintain a consistent vocabulary size of 50,257 tokens in all experiments.

Baselines. We compare ECCHECK with three baselines: (1) **base1**, which uses `torch.save()` from PyTorch and is the conventional choice for checkpointing in distributed training [24]. It serializes `state_dict` and then transfers the result to remote persistent storage. Its checkpointing is synchronous, blocking training until checkpointing is complete. (2) **base2**, a 2-phase checkpoint scheme inspired by CheckFreq [20]. It first copies the model states from GPU to CPU, and then asynchronously writes the checkpoint data to remote persistent storage. (3) **base3**: replication-based in-memory checkpointing according to GEMINI (which is not open-sourced) [35]. In our four-node testbed, it groups node 1 and node 2 into one group and node 2 and node 3 into another. Each worker first copies the model states from GPU to CPU, and then broadcasts the checkpoint data to all other nodes in the same group.

Settings. In our settings, $k = 2$ and $m = 2$. Each worker reserves 12 data buffers and 24 encoding buffers in the CPU, and each buffer is 64 MB in size.

C. Checkpointing Time

Fig. 10 compares the checkpointing time for models of different sizes. The time is measured from when `eccheck.save` is called until its completion. In-memory checkpointing, **base3** and our ECCHECK, significantly reduce the checkpointing time compared to remote storage-based checkpointing. Checkpointing time limits the maximum checkpointing frequency, as the next checkpointing process cannot begin until the current one finishes. A higher checkpointing frequency helps prevent training from resuming from a much earlier state after a failure, thereby reducing wasted GPU work. While ECCHECK exhibits a modest increase in checkpointing time compared to **base3** (approximately 1.6 times) due to the overhead of encoding, it offers a substantially enhanced fault tolerance capacity. Specifically, it can tolerate any two concurrent failures, while **base3** is unable to recover if both node 0 and node 1 (or both node 2 and node 3) fail. We posit that this minor increase in checkpointing time is a judicious trade-off, considering the much improved resilience against failures that ECCHECK provides. Furthermore, as we will demonstrate next, the additional time required by ECCHECK has a negligible impact on overall training progress.

D. Training Efficiency

Fig. 11 presents a time breakdown of ECCHECK checkpointing when training GPT-2 models of varying sizes. Step 1 (i.e., analyzing and decomposing the `state_dict`, and meanwhile transferring tensor data from device memory to host memory) only blocks training for a short duration. Step 2 (i.e., broadcasting non-tensor key-value pairs and tensor keys) takes negligible time. Step 3 (i.e., the pipeline execution of the encoding, XOR reduction, and P2P communication) takes the majority of the time. This step occurs asynchronously, allowing training to progress unimpeded.

Fig. 12 compares the average training iteration time across different checkpoint frequencies (i.e., 1/#checkpoint interval). We observe that the runtime overhead for **base1** is significant due to its reliance on synchronous checkpointing, which blocks training until the checkpointing process is complete. As the checkpoint frequency increases, the runtime overhead for **base2** becomes more pronounced. Although **base2** writes checkpoint data asynchronously to remote persistent storage to reduce checkpoint stalls, it fails to reduce the actual checkpointing time, thereby limiting its ability to support high-frequency checkpointing. In contrast, both **base3** and ECCHECK demonstrate similarly outstanding performance. This can be largely attributed to their use of high-bandwidth inter-node connections and effective communication strategies.

E. Fast Recovery

Fig. 13 presents the recovery time under two failure scenarios, i.e., from when `eccheck.load` is called to training resumption. In Figure 13a where all data nodes 0 and 2 survive, ECCHECK can resume training rapidly by leveraging high-bandwidth inter-node connections for transferring checkpoint data packets. In contrast, remote storage-based checkpointing, **base1** and **base2**, require significant recovery time, due to relatively low bandwidth and the inability to execute asynchronously.

In Fig. 13b where nodes 0 and 1 survive (data node 0 fails), **base3** fails to recover the original model states due to the failure of both node 2 and node 3. While ECCHECK requires additional time to resume training compared to the first scenario (due to decoding to recover lost data packets), it still substantially outperforms **base1** and **base2**.

F. Scalability

We first show that the communication overhead for each worker remains constant as the number of nodes increases. Consider a general scenario with n nodes, each containing g workers (with a total of $W = ng$ workers). These are divided into k data nodes and m parity nodes, where $n = k + m$. Each worker stores data of size s, meaning the total model size is $s \cdot W$. As we described in Section IV-B, there are W/k reduction groups, each comprising k workers. Each reduction group performs m XOR reduction operations, generating m parity packets that are distributed among the k workers. The total communication volume during the XOR reduction is $\frac{W}{k} m(k-1)s$. For P2P communication, under our optimal data and parity nodes selection strategy, each data node already holds g data packets. Given that there are W data packets in total, this means $W - kg$ data packets need to be transferred. Each parity node holds g parity packets, and there are $\frac{Wm}{k}$ total parity packets. Therefore, the total communication

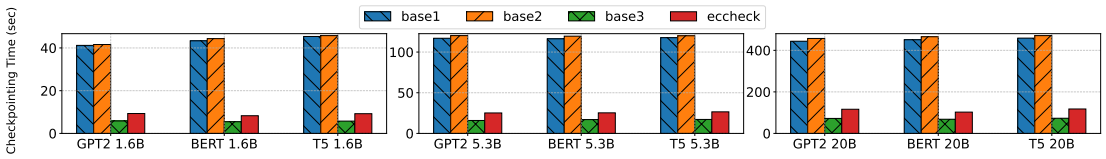


Fig. 10: Comparison of checkpointing time.

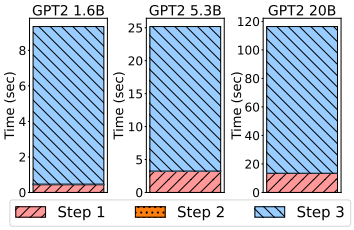


Fig. 11: Time breakdown of ECCHECK checkpointing.

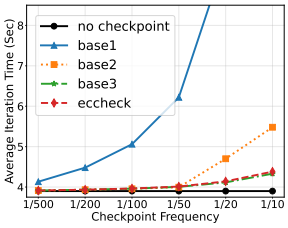


Fig. 12: Checkpointing overhead for GPT2 5.3B training.

(a) Node 0 and node 2 survive, i.e., all data nodes survive.

(b) Node 0 and node 1 survive, i.e., not all data nodes survive.

Fig. 13: Comparison of recovery time.

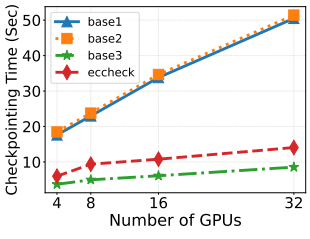


Fig. 14: Scalability of checkpointing time with varying numbers of GPUs.

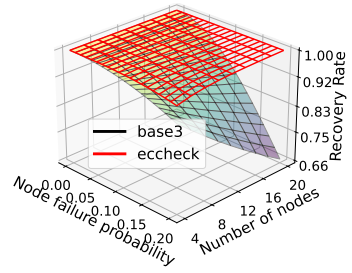


Fig. 15: Comparison of fault tolerance capacity between **base3** and ECCHECK under identical redundancy conditions ($k = m = \frac{n}{2}$).

volume for one checkpointing process is $\frac{Wm(k-1)s}{k} + (W - kg)s + (\frac{W}{k} - g)ms = msW$, which is m times the total model size. Thus, the average communication overhead per device is ms, meaning it does not increase as the number of nodes grows, it depends only on the number of parity nodes (which determines fault tolerance) and the size of the model shard each worker holds. This illustrates ECCHECK's excellent scalability. In Fig. 14, we evaluate the scalability of our approach using up to 32 NVIDIA V100 32GB GPUs. A GPT-2 model with a fixed hidden size of 1024 is trained, while the number of layers is varied from 16 (for 4 GPUs) to 128 (for 32 GPUs) to ensure that the model parameters per GPU remain nearly constant. The number of nodes is fixed at $n = 4$, with $k = m = 2$. The figure demonstrates the excellent scalability of **base3** and ECCHECK, attributed to their fully distributed design. In contrast, for **base1** and **base2**, the checkpointing time scales linearly with the number of GPUs. This is because the volume of data increases linearly, while the aggregated bandwidth to the remote storage system remains constant.

In large clusters, it is common to increase the number of parity nodes to enhance fault tolerance, which will raise the communication overhead. To mitigate this, a group-based checkpointing approach can be employed, where nodes are divided into smaller groups, and ECCHECK is applied within each group. This allows the system to scale to a larger number of nodes without a proportional increase in checkpointing overhead, while still maintaining sufficient fault tolerance compared to replication-based methods. An example of this approach is illustrated in Fig. 3.

G. Fault Tolerance Capacity

Consider our experimental setting where $k = m = n/2$. The successful recovery rate of ECCHECK is $\sum_{i=0}^{\frac{n}{2}} \binom{n}{i} p^i (1-p)^{n-i}$ according to Eqn. 2, demonstrating that ECCHECK can tolerate up to $n/2$ concurrent failures. In contrast, for **base3**, recovery becomes impossible if more than one failure occurs within the same group. To ensure recoverability, any i failures must occur in distinct groups. In this case, the probability of **base3** successfully recovering the original data is $\sum_{i=0}^{\frac{n}{2}} \binom{n}{i} p^i (1-p)^{n-i} \cdot \frac{\binom{\frac{n}{2}}{i} 2^i}{\binom{n}{i}}$ according to Eqn. 1.

Fig. 15 illustrates the fault tolerance capacities of both systems. ECCHECK achieves superior fault tolerance compared to **base3** under identical redundancy conditions (i.e., identical CPU memory usage), and this advantage becomes more pronounced as the number of nodes increases.

VI. Conclusion and Discussion

We introduced ECCHECK, an innovative in-memory checkpointing system for distributed deep learning training. Leveraging erasure coding, ECCHECK significantly enhances fault tolerance of in-memory checkpointing while maintaining high training throughput, low checkpointing stalls, and fast recovery. We judiciously design a series of techniques to optimize computation and communication efficiency during the erasure coding-based checkpointing procedure. Extensive experiments show ECCHECK's advantages: reduced checkpointing time (by up to 5.2×) and fast recovery (by up to 13.9×) compared

to remote storage-based checkpointing. Moreover, it demonstrates enhanced fault tolerance compared to replication-based in-memory checkpointing.

For better scalability, we can use a group-based checkpointing approach, by dividing nodes into groups, with ECCHECK applied for checkpointing within each group. The group size can be tailored to specific requirements of the training job, balancing communication efficiency and fault tolerance. Computing the optimal group size is part of our future work.

REFERENCES

[1] "Bloom chronicles," https://github.com/bigscience-workshop/bigscience/blob/master/train/tr11-176B-ml/chronicles.md, 2022.

[2] J. Bloemer, M. Kalfane, R. Karpz, M. Karpinski, M. Luby, and D. Zuckermank, "An xor-based erasure-resilient coding scheme," International Computer Science Institute, University of California at Berkeley, Berkeley, CA, USA, Tech. Rep. TR-95048, 1995.

[3] F. Chang, Y.-P. Cheng, T. Pavlidis, and T.-Y. Shuai, "A line sweep thinning algorithm," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, 1995, pp. 227–230 vol.1.

[4] Z. Chen and J. Dongarra, "A scalable checkpoint encoding algorithm for diskless checkpointing," in *2008 11th IEEE High Assurance Systems Engineering Symposium*, 2008, pp. 71–79.

[5] N. Corporation, "Nccl: Nvidia collective communication library," https://github.com/NVIDIA/nccl, 2017.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[7] A. Dimakis, V. Prabhakaran, and K. Ramchandran, "Decentralized erasure codes for distributed networked storage," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2809–2816, 2006.

[8] D. Driess, F. Xia, M. S. M. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, W. Huang, Y. Chebotar, P. Sermanet, D. Duckworth, S. Levine, V. Vanhoucke, K. Hausman, M. Toussaint, K. Greff, A. Zeng, I. Mordatch, and P. Florence, "Palm-e: An embodied multimodal language model," 2023.

[9] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram, "Check-N-Run: a checkpointing system for training deep learning recommendation models," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 929–943.

[10] Facebook, "Gloo: A collective communications library," https://github.com/facebookincubator/gloo, 2017.

[11] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. USA: USENIX Association, 2010, p. 61–74.

[12] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," 2018.

[13] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, *GPipe: efficient training of giant neural networks using pipeline parallelism*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[14] HuggingFace, "Code parrot dataset," https://huggingface.co/datasets/codeparrot/github-code, 2022.

[15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[16] V. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," 2022.

[17] J. Li and B. Li, "Erasure coding for cloud storage systems: A survey," *Tsinghua Science and Technology*, vol. 18, no. 3, pp. 259–272, 2013.

[18] C. Liu, Q. Wang, X. Chu, and Y.-W. Leung, "G-crs: Gpu accelerated cauchy reed-solomon coding," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1484–1498, 2018.

[19] F. Lu, X. Wei, Z. Huang, R. Chen, M. Wu, and H. Chen, "Serialization/deserialization-free state transfer in serverless workflows," in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 132–147.

[20] J. Mohan, A. Phanishayee, and V. Chidambaram, "CheckFreq: Frequent, Fine-Grained DNN checkpointing," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 203–216.

[21] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–15.

[22] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[23] OpenAI, "GPT-4," https://openai.com/gpt-4, 2023.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: an imperative style, high-performance deep learning library*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[25] W. W. Peterson, *Error-Correcting Codes*. Cambridge, MA: MIT Press, 1961.

[26] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications – version 1.2," Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, USA, Tech. Rep. CS-08–627, 2008, citeseer.

[27] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.

[28] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, jan 2020.

[29] S. Rajasekaran, M. Ghobadi, and A. Akella, "CASSINI: Network-Aware job scheduling in machine learning clusters," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1403–1420.

[30] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 2, p. 24–36, apr 1997.

[31] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," in *International Conference on Dependable Systems and Networks (DSN'06)*, 2006, pp. 249–258.

[32] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020.

[33] L. team, "The llama 3 herd of models," 2024.

[34] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[35] Z. Wang, Z. Jia, S. Zheng, Z. Zhang, X. Fu, T. S. E. Ng, and Y. Wang, "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.

[36] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," 2022.

[37] T. Zhang, K. Liu, J. Kosaian, J. Yang, and R. Vinayak, "Efficient fault tolerance for recommendation model training via erasure coding," *Proc. VLDB Endow.*, vol. 16, no. 11, p. 3137–3150, jul 2023.

[38] T. Zhou and C. Tian, "Fast erasure coding for data storage: A comprehensive study of the acceleration techniques," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 317–329.