dl^2: Detecting Communication Deadlocks in Deep Learning Jobs

Yanjie Gao
Microsoft Research
Renmin University of China
Beijing, China
yanjga@microsoft.com

Jiyu Luo[*]
University of Science and Technology of China
Hefei, China
luojiyu@mail.ustc.edu.cn

Haoxiang Lin[†]
Microsoft Research
Beijing, China
haoxlin@microsoft.com

Hongyu Zhang
Chongqing University
Chongqing, China
hyzhang@cqu.edu.cn

Ming Wu
Zero Gravity Labs
San Francisco, USA
ming@0g.ai

Mao Yang
Microsoft Research
Beijing, China
maoyang@microsoft.com

Abstract

In recent years, deep learning has seen widespread adoption across various domains, giving rise to large-scale models such as large language models. Training these models, particularly in distributed environments, presents substantial computational and communication challenges. A critical issue is the communication deadlock—a state in which processes become indefinitely stalled while awaiting network messages from others, which leads to resource wastage and reduced productivity. Current approaches to deadlock handling are either unsuitable for deep learning due to its unique hybrid programming paradigm or limit optimization opportunities. This paper presents dl^2, a novel dynamic analysis tool designed to detect communication deadlocks in deep learning jobs. dl^2 models the runtime trace of a job as an execution graph, detects unmatched communications, and constructs a wait-for graph to identify deadlock cycles. dl^2 can also handle nondeterministic communication behaviors, providing replay and diagnostic support for root cause analysis. We evaluate dl^2 using PyTorch with a combination of synthetic test cases and real-world deep learning workloads. The experimental results show that dl^2 successfully detects all communication deadlocks, achieving 100% precision and recall, which highlights its effectiveness.

CCS Concepts

• **Software and its engineering → Deadlocks**.

Keywords

deep learning, large language model, communication deadlock

ACM Reference Format:
Yanjie Gao, Jiyu Luo, Haoxiang Lin, Hongyu Zhang, Ming Wu, and Mao Yang. 2025. dl^2: Detecting Communication Deadlocks in Deep Learning Jobs. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3696630.3728529

[*]The work of Jiyu Luo was performed during the internship at Microsoft Research until April 2024.
[†]Haoxiang Lin is the corresponding author.

1 Introduction

Deep learning has achieved significant success in recent years across various application domains, including natural language processing, programming, gaming, drug discovery, and scientific computing. The demand for training complex large-scale models, particularly large language models (LLMs) such as the GPT [43], Llama [54], DeepSeek [9], and Qwen [67] series, has surged dramatically, driven by the growing adoption of artificial intelligence (AI). Distributed training has emerged as a practical and promising solution to manage the substantial computational and memory requirements of these large models. This approach leverages numerous or even clusters of computing resources to enable data, model, and pipeline parallelism [44]. As an example, Meta trained the Llama 3 models on two custom AI clusters, each consisting of 24K graphics processing units (GPUs) [25].

During distributed training, tens, hundreds, or even thousands of computing processes work together by continually synchronizing tensors through network messages. This extensive communication requires sophisticated timing coordination to ensure efficient progress. However, due to the predominantly manual design of distributed training programs and the lack of sufficient support in current distributed training frameworks, a group of processes may end up waiting indefinitely for messages from one another, leading to a state of impasse where no further progress is possible. We refer to this condition as a *communication deadlock*. For instance, a deadlock can occur when both processes in a deep learning job perform synchronous send-then-receive operations in the same order, preventing either from advancing. Communication deadlocks are a known issue in deep learning workloads. After consulting with the site reliability engineers (SREs) of Microsoft Platform-X, it was confirmed that communication deadlocks do occur in stalled training jobs submitted by various product and research teams. Furthermore, the absence of adequate tooling makes thorough identification and in-depth root cause diagnosis of such deadlocks particularly challenging for the SREs. Another example comes from a developer who reported and discussed a deadlock issue on GitHub [56]. The developer's job encountered a hang because one process failed to enter the synchronization barrier, causing the entire system to stall.

This highlights the common challenge of ensuring proper synchronization across processes in distributed training environments.

Communication deadlocks in deep learning jobs pose a significant challenge in contemporary deep learning workflows, with many practitioners and users reporting frequent encounters with these issues [56–64]. Such deadlocks not only lead to considerable resource wastage (e.g., GPU, CPU, storage, and network I/O) but also substantially hinder development productivity. The impact is even more severe in automated machine learning (AutoML) [18, 31] or automated training parallelization scenarios [22, 70], where numerous jobs from the same experiment run concurrently. In such cases, if one job encounters a communication deadlock, hundreds of other jobs with identical or highly similar communication patterns may also experience deadlocks, causing widespread stalls.

Extensive research has been conducted using static analysis [3, 4, 10, 11, 14, 17, 21, 48, 69] or dynamic analysis [7, 46, 47, 55] to address various types of deadlocks. These methods are typically applied to multi-threaded or Message Passing Interface (MPI) [26] programs written in traditional programming languages such as C, C++, C#, Java, or Rust. However, deep learning jobs differ significantly in their programming models and abstractions compared to those of traditional programs studied in prior research. Developers typically employ a hybrid programming paradigm, where deep learning models are constructed as tensor-oriented computation graphs using built-in operators (i.e., mathematical operations such as convolution and pooling) provided by training frameworks such as TensorFlow [1] and PyTorch [39]. Communication, on the other hand, is abstracted and hidden by the frameworks and underlying libraries, including Meta's Gloo [19], the NVIDIA Collective Communications Library (NCCL) [34], and the proprietary CUDA Toolkit [36]. As a result, existing approaches are not directly applicable to deep learning jobs. More recently, researchers [5, 8, 38, 49, 68] have proposed preventing deadlocks in deep learning jobs through global coordination or enforcing a predetermined execution order on operators. Nevertheless, these methods constrain the flexibility of execution plans, limiting opportunities for further optimization. Additionally, they do not provide capabilities for deadlock detection or assist developers with root cause analysis.

In this paper, we present dl^2, a dynamic analysis tool designed to detect communication **d**ead**l**ocks in **d**eep **l**earning jobs. The tool is built on the idea that the runtime trace of a job can be modeled as an *execution graph* (EG) [13], where nodes represent communication-related actions and edges depict the dependencies between these actions. Common actions include point-to-point message sending and receiving, collective communication such as AllReduce, and synchronization across processes and GPU kernels. A directed edge from action B to action A indicates a causal dependency, meaning that action B can only start once action A has finished. dl^2 traverses the EG, identifies unmatched communications (e.g., a logically incomplete point-to-point communication lacking a corresponding send or receive action), and constructs a *wait-for graph* (WFG) [29] among the involved processes based on these incomplete interactions. Cycles in the WFG signify potential communication deadlocks.

Deep learning jobs rely on well-defined APIs provided by training frameworks and lower-level libraries (e.g., Gloo and NCCL) to handle various types of communication. dl^2 is designed to identify all such APIs. It implements an interposition layer within the address space of each computing process, allowing it to intercept the invocations of these APIs. Upon interception, dl^2 captures relevant invocation details, such as process ID, API name, arguments, and results. This data is used to represent actions in the EG. Additionally, we thoroughly analyze the API semantics and establish a set of rules to determine action dependencies (e.g., a receive action depends on a prior matched send). To detect potential communication deadlocks, dl^2 first runs the job in a native environment. As actions are exposed, dl^2 updates the EG by adding edges according to the defined dependency rules. It then constructs an up-to-date WFG and checks it for cycles, identifying any as deadlocks. For developers seeking root cause diagnosis, dl^2 offers a replay capability, faithfully reproducing the entire communication to aid in debugging. As GPU computations dominate execution, dl^2 employs a stubbing technique that intercepts GPU kernels responsible for computations (e.g., those executing operators) and returns mock tensors without utilizing actual GPUs when they do not impact communication. This approach significantly reduces the overhead of trace collection. In cases where jobs exhibit nondeterministic communication behavior, deadlocks may not appear in the current recorded execution but could occur in other executions. To handle this, dl^2 tries to adjust the execution order of asynchronous actions or buffered synchronous actions in the EG while maintaining consistency. If an action can be reordered, it will erroneously participate in out-of-round communication, indicating the possibility of a communication deadlock.

We have implemented dl^2 for the PyTorch framework [39] and evaluated it on both synthetic test cases utilizing collective communication operations [32] (AllToAll, AllReduce, AllGather, Broadcast, and ReduceScatter) and real-world jobs involving the training of representative models (GPT-2 [43], mBART [23], MLP [53], OpenFold [2], and Swin Transformer [24]). Experimental results indicate that dl^2 successfully identifies all communication deadlocks and achieves 100% precision and recall [37], demonstrating its overall effectiveness.

In summary, this paper makes the following contributions:

(1) We identify a critical, underexplored, and challenging research problem in real-world development: communication deadlocks in deep learning jobs.
(2) We propose a novel dynamic analysis approach for detecting communication deadlocks.
(3) We develop a tool, dl^2, which models the runtime trace of a job as an execution graph and constructs a wait-for graph for deadlock detection. dl^2 also supports the detection of potential nondeterministic communication deadlocks.
(4) We demonstrate the practical utility and effectiveness of dl^2 through extensive experimental evaluation.

The remainder of this paper is structured as follows: Section 2 provides an overview of distributed training, collective communication primitives, and communication deadlocks. Section 3 outlines the methodology behind dl^2, while Section 4 delves into the implementation details. In Section 5, we present the experimental results. Section 6 addresses threats to validity, as well as the practicality, generality, and extensibility of our work. Related work is reviewed in Section 7, and we offer concluding remarks in Section 8.

2 Background

2.1 Deadlocks

In concurrent, parallel, or distributed computing, a deadlock occurs when a group of processes reaches a state of impasse in which none can make progress because each process is waiting for others to perform actions, such as releasing a lock or receiving a message. Deadlocks are generally categorized as either resource deadlocks or communication deadlocks. In the first, processes compete for one or more resources that are already held by other processes. A typical example involves processes acquiring different locks for resources, but an incorrect order of acquisition causes a halt. Communication deadlocks occur when every process is waiting to communicate with others in the same process group.

In 1971, E. G. Coffman, M. Elphick, and A. Shoshani [6] identified four necessary and jointly sufficient conditions for deadlock, now referred to as the *Coffman conditions*:

(1) *Mutual Exclusion*: Resources cannot be shared among multiple processes. That is, each resource can be utilized by only one process at a moment.
(2) *Wait For*: A process holds one or more resources while simultaneously waiting for additional resources that are already allocated to other processes.
(3) *No Preemption*: A process cannot forcibly seize an allocated resource unless the owner voluntarily releases it.
(4) *Circular Wait*: A circular chain of processes exists in which each process is waiting for a resource that another process in the chain is holding, forming a cycle of dependencies.

While these conditions were initially proposed for resource contexts, they can also be readily adapted for communication deadlocks. Since circular wait is a fundamental characteristic of such situations, various detection tools—including our dl^2—construct wait-for graphs (WFGs) [29] that illustrate wait dependencies between processes and check for cycles to identify potential deadlocks.

2.2 Distributed Deep Learning Training

With the increasing demand for complex large-scale models, especially large language models (LLMs) such as the GPT [43], Llama [54], DeepSeek [9], and Qwen [67] series, distributed training has become a crucial solution. This approach involves parallelizing the entire training process across multiple machines or even clusters of machines by strategically partitioning both the training data and the model architecture. Several distributed strategies, including data parallelism [49], tensor parallelism [50], and pipeline parallelism [16], have been developed to address the challenges posed by the growing size and complexity of these models.

Developers typically utilize established distributed training frameworks, such as Hugging Face Transformers [65], Microsoft DeepSpeed [44], and NVIDIA Megatron-LM [50], while manually implementing parallelism to achieve optimal runtime performance for their deep learning jobs. Recently, automated parallelization tools like nnScaler [22] and Alpa [70] have emerged. These tools aim to automatically generate efficient parallelization plans by exploring a vast optimization space.

Runtime performance is largely influenced by the underlying collective communication libraries, such as NCCL [34] and Gloo [19]. Below, we provide a brief overview of five key collective communication primitives, which are also utilized in our evaluation (see Section 5.3):

(1) AllToAll: Distributes a list of tensors across all processes, with each process receiving one tensor, and gathers tensors from all other processes.
(2) AllReduce: Performs a reduction operation on tensors from all processes and combines the results across all processes.
(3) AllGather: Collects tensors from all processes and consolidates them into an ordered sequence for each process.
(4) Broadcast: Sends a tensor from the source process to all other processes.
(5) ReduceScatter: Reduces tensors from all processes and distributes chunks of the reduced result to each process.

2.3 Communication Deadlocks in Deep Learning Jobs

While the aforementioned frameworks and tools provide significant advantages, they also introduce considerable communication challenges. Improper implementation may lead to circular waits, often due to misuse of synchronous communication operations or unexpected communication orders resulting from nondeterminism (e.g., asynchrony or buffering). As a result, communication deadlocks can occur, since GPU kernels—which execute actual communication—hold GPUs exclusively without allowing preemption.

Building on related empirical studies [12, 38, 68], GitHub issues, and our own experience, we classify communication deadlocks into three categories: point-to-point, collective, and hybrid, based on the communication operations involved. Point-to-point deadlocks arise from communication in which each message is exchanged between only two processes. For example, the PyTorch Distributed communication package [41] provides APIs such as `send`, `recv`, `isend`, and `irecv`, which support both synchronous and asynchronous point-to-point communication. The second category, collective deadlocks, involves multiple processes participating in a single communication operation, such as broadcasting a tensor to an entire process group or reducing a tensor across all GPUs. The hybrid category refers to deadlocks caused by a combination of point-to-point and collective communication. For example, this can occur when one process is waiting to receive a message while others in the same group are performing a collective operation like AllReduce.

2.4 Deadlock Examples

Figure 1 illustrates a simple distributed training program using the popular PyTorch distributed communication package [41]. The NCCL (NVIDIA Collective Communications Library) backend is selected for communication, with alternatives such as MPI, Gloo, or Unified Collective Communication (UCC) [66] also available. This program spawns two processes (also referred to as *ranks*), with their communication logic for tensor exchange (e.g., gradients) encapsulated in the `run` function. Process 0 sends its tensor to process 1 using the `dist.send` function and then waits for a tensor from process 1 using `dist.recv`. Concurrently, process 1 follows the same pattern, sending and receiving tensors to and from process 0. Because these point-to-point functions are synchronous, each

```python
import torch.distributed as dist

def run(pid, tensor):
    ...
    dist.init_process_group(backend = "nccl", rank = pid,
    ↪  world_size = size)
    ...
    if pid == 0:
        # Send the tensor to process 1
        dist.send(tensor = tensor, dst = 1)
        # Receive tensor from process 1
        dist.recv(tensor = tensor, src = 1)
    else:
        # Send the tensor to process 0
        dist.send(tensor = tensor, dst = 0)
        # Receive tensor from process 0
        dist.recv(tensor = tensor, src = 0)
```

Figure 1: A simple PyTorch distributed training program that results in a communication deadlock, as both processes block while waiting for the other to receive its tensor.

process blocks while waiting for the other to receive its tensor, which leads to a communication deadlock.

As outlined in Section 1, a developer identified and discussed a deadlock issue on GitHub [56] related to the use of the Hugging Face Transformers library. The problem stemmed from a misalignment in communication across multiple processes. Specifically, inconsistent handling of training metrics caused one process to prematurely exit the training loop and reach the synchronization barrier (PyTorch's `dist.barrier` function). In contrast, the remaining processes were still engaged in forward passes during the distributed training. This desynchronization led to a deadlock, as the process that had reached the barrier was left waiting indefinitely for the others to catch up. The issue was ultimately resolved by ensuring proper metrics aggregation and synchronization across all processes, which guaranteed that they progressed through their operations in a coordinated and uniform manner.

3 Methodology

3.1 Execution Graph

System analysis and diagnosis have traditionally treated runtime information, such as function calls and logs, as unstructured text or simple paths. This approach often obscures the essential causal relationships between events, making the identification of issues tedious and challenging. Our tool dl^2, in contrast, models the runtime trace of a deep learning job as an execution graph (EG) [13], which captures the complete execution for subsequent deadlock analysis. Formally, an EG is represented as a directed graph (or digraph):

$$EG = \langle V = \{act_i\}_{i=1}^{N}, E = \{(act_i, act_j)\}_{i \neq j}, P = \{p_i\}_{i=1}^{K} \rangle \, .$$

In this graph, each node act_i represents a communication-related action, such as sending or receiving messages, performing tensor reduction across GPUs, or waiting for the completion of GPU kernels. Such actions are modeled based on established APIs commonly used by developers for communication. P is a set of active processes that continuously execute various actions. A directed edge (act_i, act_j) signifies a causal dependency between two actions, where act_j can only begin after act_i completes. For example, a Recv action in process p_j depends on a preceding Send action from another process p_i, provided p_j has indeed received the message from p_i. These causal dependencies can form wait dependencies between processes at runtime. In the above scenario, p_i must wait for p_j to complete the Recv action before proceeding.

$\langle$***Identifiers***$\rangle$ id ::= $aid \mid pid \mid gid$ (action/process/group IDs)
  | src, dst ::= pid | sa ::= aid (aliases)

$\langle$***Actions***$\rangle$ act ::= $p2p\_act \mid col\_act \mid sy\_act$

$\langle$***Point-to-Point Actions***$\rangle$ $p2p\_act$ ::= s_p2p_act
  | as_p2p_act

$\langle$***Synchronous Point-to-Point Actions***$\rangle$ s_p2p_act ::=
  $Send\langle aid, pid, dst\rangle \mid Recv\langle aid, pid, src, sa\rangle$

$\langle$***Asynchronous Point-to-Point Actions***$\rangle$ as_p2p_act ::=
  $iSend\langle aid, pid, dst\rangle \mid iRecv\langle aid, pid, src, sa\rangle$

$\langle$***Collective Actions***$\rangle$ col_act ::= $s\_col\_act \mid as\_col\_act$

$\langle$***Synchronous Collective Actions***$\rangle$ s_col_act ::=
  | $AllReduce\langle aid, pid, gid, sa[\;\;]\rangle$
  | $AllGather\langle aid, pid, gid, sa[\;\;]\rangle$
  | $Scatter\langle aid, pid, src, gid, sa[\;\;]\rangle$
  | $Broadcast\langle aid, pid, src, gid, sa[\;\;]\rangle$
  | ...

$\langle$***Asynchronous Collective Actions***$\rangle$ as_col_act ::=
  | $iAllReduce\langle aid, pid, gid, sa[\;\;]\rangle$
  | $iAllGather\langle aid, pid, gid, sa[\;\;]\rangle$
  | $iScatter\langle aid, pid, src, gid, sa[\;\;]\rangle$
  | $iBroadcast\langle aid, pid, src, gid, sa[\;\;]\rangle$
  | ...

$\langle$***Synchronization Actions***$\rangle$ sy_act ::= $Wait\langle aid, pid, sa\rangle$
  | $Barrier\langle aid, pid, gid, sa[\;\;]\rangle$

Figure 2: Syntax of actions. "[ ]" refers to an array.

3.2 Action

We present the syntax of actions in Figure 2 [41], derived from the documentation of training frameworks and collective communication libraries, as well as our own experience. In our study, an identifier can represent either a unique action or process or a distinct group of processes, where *aid*, *pid*, and *gid* correspond to the IDs of actions, processes, and process groups, respectively. A process group may consist of all computing processes or a subset of them. Within a single process, action IDs increment based on the execution order. For convenience, we define several identifier aliases: *src* and *dst* represent the IDs of the source and destination processes, respectively, while *sa* refers to the source action ID.

Actions are classified into three categories: point-to-point, collective, and synchronization. These correspond to point-to-point communications (Send and Recv), collective operations (e.g., AllReduce, AllGather, Scatter, and Broadcast), and synchronization primitives (Wait on an asynchronous action until its completion and Barrier to synchronize all processes at this point), respectively. Refer to the PyTorch documentation [41] for a comprehensive list of collective actions. Each synchronous point-to-point or collective action has a corresponding asynchronous version, denoted by a prefix i (e.g., iSend for Send). Note that in training frameworks like PyTorch, both versions may share the same API name but are distinguished by different values of a parameter related to asynchrony. Every action carries a context encapsulated within angle brackets, which includes its action ID, process ID, and other relevant information. In certain collective actions (e.g., AllReduce), $sa[\ ]$ refers to an array containing the IDs of the actions on which they depend.

3.3 Causal Dependency

We identify three categories of causal dependencies between actions (i.e., edges in an execution graph). To indicate that action act_j causally depends on act_i, we use the notation $act_i \prec act_j$ instead of $(act_i, act_j) \in EG$.

The first two categories of dependencies are based on program logic. The *sequential dependency* ensures that actions executed later within a single process depend on those executed earlier, according to the program's order:

$$(act_i.pid = act_j.pid) \wedge (act_i.aid < act_j.aid) \models act_i \prec act_j\,.$$

In a model, the computation is represented as a graph of *operators* (also known as a computation graph), where each operator corresponds to a high-level mathematical operation such as matrix multiplication, 2D convolution, or attention. Since two operators may have an inherent algorithmic dependence, their respective actions exhibit *algorithmic dependency*:

$$(act_i \in OP_m) \wedge (act_j \in OP_n) \wedge (OP_m \prec OP_n) \models act_i \prec act_j\,.$$

The third category, *communication dependency*, arises from specific action types. For instance, a Wait action causes the current process to block until a particular action is completed. Consequently, a Wait action depends on the completion of the awaited action:

$$(act_j = Wait) \wedge (act_i.aid = act_j.sa) \models act_i \prec act_j\,.$$

In the case of point-to-point communications, the dependency is straightforward—a receiving action depends on its corresponding sending action:

$$\begin{aligned}&(act_i = (Send \vee iSend)) \wedge (act_j = (Recv \vee iRecv))\\ &\wedge (act_i.aid = act_j.sa) \models act_i \prec act_j\,.\end{aligned}$$

Collective communications are much more complex because they involve multiple processes. For example, all processes within the same group execute AllReduce to obtain the sum of tensors from each individual process. This type of communication results in a complete directed subgraph in the execution graph, where each distinct pair of processes sends and receives tensors from one another. Consequently, every process is both dependent on and depended upon by others, which can be expressed as follows:

$$\begin{aligned}&(act_i = AllReduce) \wedge (act_j = AllReduce)\\ &\wedge (act_i.pid \neq act_j.pid) \wedge (act_i.gid = act_j.gid)\\ &\wedge (act_i.aid = act_j.sa[act_i.pid]) \models act_i \prec act_j\,.\end{aligned}$$

The last condition emphasizes that the process executing act_j successfully receives the tensor transmitted by act_i.

Causal dependencies play a critical role in ensuring the correct matching of communications among actions. A *matched communication* is defined as a set of actions that jointly complete an anticipated communication. For example, a matched point-to-point communication consists of a send action and a corresponding dependent receive action, expressed as follows:

$$\begin{aligned}&(act_i = (Send \vee iSend)) \wedge (act_j = (Recv \vee iRecv))\\ &\wedge (act_i \prec act_j) \models \{act_i, act_j\} \text{ is matched}\,.\end{aligned}$$

Similarly, in the context of AllReduce actions, the requirement is that each process within the communication group performs one AllReduce action, and every pair of distinct AllReduce actions exhibits mutual dependency. Unmatched communications, on the other hand, indicate that some processes within the group might become blocked, waiting for others to complete their respective communication actions. To represent these incomplete interactions, we can construct a wait-for graph that illustrates dependencies among the participating processes.

3.4 Deadlock Identification

A wait-for graph (WFG) [29] is formally represented as a directed graph:

$$WFG = \langle V = \{p_i\}_{i=1}^{K}, E = \{(p_i, act, p_j)\}_{i\neq j}\rangle\,.$$

To construct an up-to-date WFG, we begin by identifying the unmatched communications and their associated actions in the execution graph. As mentioned in Section 3.3, such unmatched communications are determined based on the previously outlined rules for causal dependencies. Each node in the WFG represents a process, while a directed edge (p_i, act, p_j) (denoted as $p_i \prec_{act} p_j$) indicates that process p_j, executing action act, is waiting for process p_i to perform a corresponding action in order to complete the communication. For example, in point-to-point communication, p_i is set to receive a message from p_j. If p_i has not initiated or completed the receive operation, then the relation $p_i \prec_{Send} p_j$ holds. Conversely, if p_i is waiting for p_j to send the message, the relation becomes $p_j \prec_{Recv} p_i$. As another example related to AllReduce, if the $sa[\ ]$ array of process p_j does not include the corresponding action ID of process p_i within the same group, this implies that p_j has not yet received the tensor from p_i; therefore, the relation $p_i \prec_{AllReduce} p_j$ holds. A communication deadlock occurs at runtime if, and only if, a cycle is detected in the WFG.

3.5 Nondeterminism

Nondeterminism in deep learning jobs primarily stems from two sources. The first is asynchronous communication. GPU kernels can execute concurrently across different streams, and if proper inter-process synchronization mechanisms are not utilized, processes may receive messages out of order, resulting in unexpected behavior. The second source is communication buffering, which consolidates multiple messages into a single transmission, potentially altering the order of messages. Without sufficient context to determine the current communication round, out-of-round communication can lead to deadlocks or data corruption [45], depending on whether processes validate incoming messages.

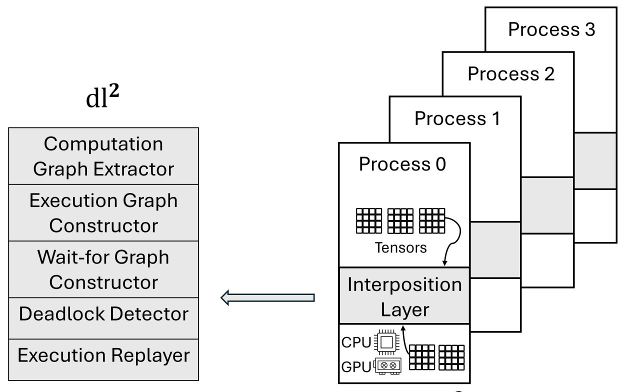

Figure 3: Architecture of dl^2.

Detecting communication deadlocks caused by nondeterminism is particularly challenging, as these issues tend to manifest only in rare, corner-case executions. dl^2 tackles this problem by identifying opportunities to reorder asynchronous actions or buffered synchronous actions within the current execution graph, while avoiding crossing operator boundaries, synchronization primitives, prior synchronous actions, and buffer limits to ensure consistency. When an action can be reordered, it may participate in out-of-round communication inadvertently, signaling the potential for a communication deadlock.

4 Implementation

We have implemented dl^2 on Linux by intercepting invocations to the PyTorch and NCCL APIs. These APIs are chosen due to their widespread use among developers for writing distributed training programs. Since GPU kernels handle actual communication and computation, we also intercept them to capture a comprehensive trace of job executions. Although dl^2 was developed on Linux, porting it to other operating systems, such as Windows and macOS, should be straightforward, as PyTorch is cross-platform. However, alternative CPU backends, such as Gloo, may be required. In the following sections, we provide a detailed explanation of dl^2's implementation, covering its architecture, execution interception, computation stubbing for accelerating trace collection, and communication replay for diagnosis.

4.1 Architecture

Figure 3 meticulously delineates the architecture of dl^2. In a native environment, the deep learning job initiates four processes for distributed model training. Each process loads and executes an interposition frontend of dl^2 within its own address space, which intercepts both communication and computation operations. While the processes are running, this interposition frontend continuously transmits per-process information about every action, later used for deadlock detection. dl^2 consists of five additional components: a computation graph extractor, an execution graph constructor, a wait-for graph constructor, a deadlock detector, and a communication replayer.

The computation graph extractor determines the number of operators and their dependencies. The execution graph constructor, a key component of dl^2, extracts actions, resolves inter-process dependencies, and builds a complete execution graph (EG). The wait-for graph constructor then uses the EG to determine unmatched communications and create a wait-for graph (WFG) by leveraging rules for causal dependencies. To store and manipulate the WFG, dl^2 utilizes "NetworkX" [30], a Python package for manipulating complex graphs and networks. We can efficiently detect cycles and identify potential communication deadlocks by leveraging the provided `find_cycle` function, which performs depth-first traversal. To assist developers in diagnosing the root causes of deadlocks or triggering potential deadlocks caused by nondeterminism, the communication replayer generates a simplified version of the training program, which faithfully executes those communication actions in the EG while preserving existing dependencies. Consequently, developers can easily rerun this program to reproduce the exact communication from the original run.

This design minimizes the impact of dl^2 on the target jobs, enabling the development of a generic interposition frontend and a tool that are compatible with operating systems beyond Linux. Currently, interaction between processes and dl^2 occurs via log files, but this can be upgraded to a remote procedure call (RPC) mechanism to support real-time deadlock detection.

4.2 Interposition

We implement the interposition frontend of dl^2 as a lightweight layer that exposes actions. This layer is positioned at the API boundary to avoid modifications to either the target deep learning program or the underlying operating system. The design choice enables us to create a generic frontend, as different training frameworks and collective communication libraries exhibit similar functionalities and APIs. The interposition frontend operates within the job's processes. To minimize complexity, we design it to be stateless, only recording the relevant information from API invocations while delegating analysis to the other five components of dl^2, thereby reducing the impact on the target job.

Specifically, dl^2 intercepts commonly used communication APIs from the PyTorch distributed communication package [41] (see Section 3.2) and NCCL (e.g., `ncclSend`, `ncclRecv`, `ncclAllReduce`, `ncclBroadcast`, and `ncclAllGather`). Since the former is implemented in Python, we utilize the "intercepts" package [51], which "allows developers to intercept function calls in Python and handle them in any manner they choose." To handle NCCL APIs, we use the "LD_PRELOAD" environment variable [20] to override the corresponding functions with our own implementations with the same names. When an API is invoked, the interposition frontend creates a context for the invocation, including a newly generated action ID, current process ID, target process ID, communication group ID if applicable, and other relevant information. This context is asynchronously stored in a per-process log file to minimize interception overhead. Initially, we planned to embed the action IDs of communication-initiating participants (e.g., Send) within the network payload (i.e., the tensors) to facilitate tracking causal dependencies across processes. However, this approach required modifying tensor types at runtime, leading to unnecessary tensor copies and undesirable job behaviors. Consequently, we now replace the action ID with a hash value for each tensor sent or received, derived from its shape and selected element values.

To obtain a computation graph, we use the built-in hook mechanism of PyTorch via the following four functions of class `torch.nn.Module`[1], intercepting the model's forward and backward passes:

(1) `register_forward_pre_hook`
(2) `register_forward_hook`
(3) `register_full_backward_pre_hook`
(4) `register_full_backward_hook`

Because GPU kernels handle the actual communication and computation, we employ the NVIDIA CUDA Profiling Tools Interface (CUPTI) [35] to capture GPU execution details, enabling us to correlate them with the high-level operators and communication actions.

4.3 Computation Stubbing and Communication Replay

GPUs are designed with thousands of cores specifically optimized for parallel processing. In distributed deep learning training, they play a pivotal role in performing millions to billions of matrix multiplications and other calculations over massive datasets. Consequently, GPU computations dominate the overall training process compared to other operations. Our observations indicate that the values of computed tensors should not affect subsequent communications. Therefore, if GPU computations can be simulated by generating mock tensors instead of utilizing actual GPUs, trace collection time can be significantly reduced. To achieve this, dl^2 leverages the stubbing technique, commonly used in software testing. This feature is made available through a configurable option, which is disabled by default. Developers can enable it once they ensure that the execution of their deep learning jobs is not affected by the real results of GPU computations. Given that dl^2 captures the computation graph and has a comprehensive understanding of operators, it intercepts the `cudaLaunchKernel` function [33] and identifies whether the GPU kernel is responsible for computations tied to a specific operator. When applicable, dl^2 bypasses the actual computation and returns mock tensors directly.

Replay is a powerful technique for debugging applications and diagnosing root causes. When communication deadlocks are identified, this technique can provide valuable evidence and assistance to developers, particularly for potential deadlocks arising from nondeterminism that have not yet manifested. dl^2 implements a simple yet effective communication replay feature. It produces a streamlined program where processes faithfully execute the exact communication actions in the precise order recorded in the log. Additionally, dl^2 inserts necessary synchronization barriers to align inter-process dependencies with those from the original run.

5 Evaluation

5.1 Experimental Design

We evaluated dl^2 on both synthetic test cases and real-world deep learning tasks using version 2.0.0 of PyTorch [39]. Our evaluation aims to address the following research questions (RQs):

RQ1: How effective is dl^2 in detecting deadlocks in real-world deep learning jobs?

RQ2: How effective is dl^2 in detecting deadlocks in deep learning jobs that utilize auto-synthesized parallelization execution plans?

RQ3: How effective is dl^2 in detecting deadlocks in deep learning jobs with nondeterministic communication patterns?

To determine the ground truth about the occurrence of a communication deadlock, we attached our interposition layer and ran the deep learning program for multiple iterations in a native environment. Each iteration corresponded to a single model update step, processing a batch of input data items. To prevent jobs from hanging indefinitely, we also defined a timeout threshold. After execution, we leveraged domain expertise to analyze the program and its execution trace to assess whether the job completed successfully or deadlocked due to a circular wait among communication processes. For successful jobs from RQ3 (which involved nondeterministic communication), we further assessed whether reordering actions might trigger a potential deadlock. If a communication deadlock was identified, the job was classified as a true positive; otherwise, it was considered a true negative. Next, we ran dl^2 on the job's execution trace, comparing its output with the ground truth to calculate the numbers of true positives, false positives (i.e., incorrectly reported deadlocks), true negatives, and false negatives (i.e., missed deadlocks). When dl^2 identified a deadlock, we replayed the buggy execution for cross-validation (see Section 4.3).

We used standard metrics of *precision* and *recall* to evaluate the effectiveness of dl^2, defined as follows [37]:

$$\text{Precision} = \frac{tp}{tp + fp} \times 100\%\,, \quad \text{Recall} = \frac{tp}{tp + fn} \times 100\%\,.$$

Here, tp, fp, tn, and fn represent the numbers of true positives, false positives, true negatives, and false negatives, respectively. Consequently, the total numbers of actual positives and actual negatives correspond to $tp + fn$ and $fp + tn$. Higher precision and recall values indicate a more effective tool.

The experimental setup utilized a high-performance workstation equipped with cutting-edge hardware, enabling efficient execution of computationally intensive tasks. Specifically, the system was powered by 56 Intel Xeon E5-2690 v4 CPUs (each operating at a base frequency of 2.60 GHz and featuring 35 MB of L3 cache) and 512 GB of main memory. It also included 8 NVIDIA Tesla P100 GPUs, which are PCIe-based and equipped with 16 GB of HBM2 memory per GPU. The system ran on Ubuntu Server 16.04.7 LTS.

5.2 RQ1: How effective is dl^2 in detecting deadlocks in real-world deep learning jobs?

In this section, we evaluate dl^2 on five real-world deep learning tasks, each involving the training of a representative model:

(1) MLP (Multilayer Perceptron) [53]: This foundational neural network model is configured with 16 linear layers, a batch size of 8, and a hidden size of 1024.
(2) GPT-2 [43]: This transformer-based language model features 6 layers, 12 attention heads, a batch size of 8, a hidden size of 768, and a sequence length of 512.
(3) mBART (Multilingual Bidirectional and Auto-Regressive Transformer) [23]: Designed for multilingual natural language processing, this sequence-to-sequence model employs

[1]https://pytorch.org/docs/stable/generated/torch.nn.Module.html

Table 1: Experimental results on real-world deep learning models.

Metrics \ Model	MLP	GPT-2	mBART	Swin Transformer	OpenFold
Total	128	128	128	128	128
True Positive	106	101	96	98	102
True Negative	22	27	32	30	26
False Positive	0	0	0	0	0
False Negative	0	0	0	0	0
Precision	100%	100%	100%	100%	100%
Recall	100%	100%	100%	100%	100%

2 layers, 8 attention heads, a batch size of 8, a hidden size of 128, and a sequence length of 512.

(4) Swin Transformer [24]: Tailored for computer vision tasks, this hierarchical vision transformer model—specifically the tiny variant [27]—operates with 4 layers, 8 attention heads, a batch size of 8, and a hidden size of 1024.
(5) OpenFold [2]: This advanced protein structure prediction model uses 2 layers, a batch size of 8, and hidden sizes of 128 for both pair representations and multiple sequence alignment (MSA).

Both data parallelism [49] and tensor parallelism [50] are applied to the above models, with maximum parallelism levels set to 4 for data and 2 for tensor.

To prevent out-of-memory errors on the GPU, we selected smaller batch sizes and, where necessary, reduced the number of layers. We began by setting the number of processes within a communication group to 2, 4, 6, and 8 to explore the effects of varying group sizes. For each specified group size, we created 32 separate instances of the training program. Next, we randomly selected a subset from these instances to undergo communication mutations, ensuring diverse communication patterns across different trials. This process yielded a total of 128 jobs per model, enabling a thorough assessment of how different group sizes and communication mutations affect deadlocks.

Table 1 shows the experimental results. dl^2 achieves 100% precision and recall in all the experiments, which demonstrates its effectiveness for real-world deep learning jobs. dl^2 incurs a performance slowdown ranging from 1.002× to 1.09× compared to end-to-end execution without interposition. The interposition mechanism introduces an overhead of approximately 0.0036 to 0.0064 seconds per communication. In the experiments described above, the deadlock detection time varies from 0.27 seconds for 386 trace events to 11.2 seconds for 8,328 trace events.

5.3 RQ2: How effective is dl^2 in detecting deadlocks in deep learning jobs that utilize auto-synthesized parallelization execution plans?

This section evaluates dl^2 in two common scenarios that utilize automated parallelization tools: high-performance communication and operator scheduling.

In the first scenario, we developed simple training programs that separately performed hierarchical versions [15] of five common collective communication operations: AllToAll, AllReduce, AllGather, Broadcast, and ReduceScatter. Each of these operations was automatically synthesized using a combination of low-level point-to-point communications (message sending and receiving) and plain collective primitives, aiming to optimize communication efficiency. We applied the same process group configuration and communication mutation strategy as in RQ1, yielding 64 jobs per collective communication test. The experimental results are shown in Table 2, where dl^2 consistently achieves 100% precision and recall across all experiments.

Table 2: Experimental results on collective communication.

Metrics \ Collective	AllToAll	AllReduce	AllGather	Broadcast	ReduceScatter
Total	64	64	64	64	64
True Positive	49	40	47	55	51
True Negative	15	24	17	9	13
False Positive	0	0	0	0	0
False Negative	0	0	0	0	0
Precision	100%	100%	100%	100%	100%
Recall	100%	100%	100%	100%	100%

In the second scenario, we utilized the Inception-V3 [52] model and explored different scheduling orders for a set of logically concurrent operators. We implemented data parallelism using Horovod [49] for the model and applied Megatron-LM's tensor parallelism [50] to the `Conv2d` (2D convolution) operators. The number of processes in the communication group remained the same as in the previous evaluation. By randomly reordering the scheduling of the four branches within an Inception block, we generated 64 different instances. dl^2 successfully detects all 59 deadlocks, maintaining 100% precision and recall.

5.4 RQ3: How effective is dl^2 in detecting deadlocks in deep learning jobs with nondeterministic communication?

In this section, we designed an experiment to evaluate job execution under nondeterministic communication patterns, specifically within the context of data-parallel distributed training. The nondeterminism arises from the use of tensor fusion, a key technique that combines multiple small AllReduce operations into a single asynchronous reduction. This approach consolidates all ready tensors at a given time, thereby reducing communication events effectively and improving runtime performance. However, due to variability in the buffer-filling rate, asynchronous fused reductions can inadvertently participate in out-of-round communication, potentially leading to deadlock conditions.

For this experiment, we employed the tiny variant [27] of the Swin Transformer [24] model, implementing data parallelism and tensor fusion based on Horovod [49] and PyTorch's Distributed Data Parallel (DDP) [42]. The training was conducted on the ImageNet dataset with a batch size of 32. To facilitate gradient synchronization, we registered hooks within PyTorch's automatic differentiation package (`torch.autograd`) [40] during model construction. These hooks were triggered when a gradient tensor was ready during backpropagation, allowing DDP to mark the tensor for reduction. Once all gradients in a fusion buffer were ready, an asynchronous AllReduce operation was launched to compute the mean of the gradients across all processes. We further investigated the

Table 3: Experimental results under nondeterministic communication patterns.

Buffer Num \ Buffer Size	1 MB	2 MB	4 MB	8 MB	64 MB
1	Yes (64)	Yes (55)	Yes (28)	Yes (19)	No (2)
2	Yes (61)	Yes (48)	Yes (28)	Yes (19)	No (2)
4	Yes (63)	Yes (50)	Yes (28)	Yes (18)	No (4)
8	Yes (60)	Yes (57)	Yes (32)	Yes (20)	No (8)

impact of varying fusion buffer counts (1, 2, 4, and 8) as well as buffer sizes (1 MB, 2 MB, 4 MB, 8 MB, and 64 MB) to simulate diverse execution scenarios.

Table 3 presents the experimental results. A "Yes" in a cell indicates that dl^2 detected a potential deadlock caused by nondeterministic communication, whereas a "No" signifies the absence of deadlocks. The number in parentheses represents the count of fused AllReduce operations executed by a process during a training iteration. Our findings show that smaller buffer sizes (ranging from 1 MB to 8 MB) result in a significantly more AllReduce operations, increasing the likelihood of out-of-round communication and subsequent deadlocks. Notably, dl^2 consistently detects all potential deadlocks across scenarios, reinforcing its robustness in handling nondeterminism.

6 Discussion

6.1 Threats to Validity

We identify three primary threats to the validity of our work.

First, we formulated the rules for causal dependencies from the documentation of the PyTorch distributed communication package [41] and NCCL, as well as our domain expertise. These rules are crucial for constructing both the execution and wait-for graphs. We also examined the source code to validate and refine the rules to ensure accuracy. However, due to the significant manual effort involved in rule formulation, inaccuracies are possible. To address this threat, we aimed for group consensus in decision-making and continuously refined our approach by cross-referencing the documentation and source code. In our experiments, we evaluated the performance of dl^2 by measuring its precision and recall, achieving 100% in both metrics, which confirms the validity of our causal dependency rules.

Second, dl^2 addresses nondeterminism arising from the use of asynchronous communication APIs and detects potential deadlocks before they occur. However, high-level nondeterminism from code logic in deep learning programs, such as variations in execution paths or communication patterns, remains challenging. For instance, developers may interact with the environment (e.g., generating random numbers) or monitor the values of computed tensors to influence training decisions. At present, dl^2 cannot handle this type of nondeterminism. To mitigate this threat, dl^2 can assist developers in implementing stubs that simulate all possible outcomes, thus enabling the exploration of different execution paths. Additionally, developers can utilize model checkers and testing tools to guide their programs through diverse execution scenarios.

Finally, our approach assumes that deep learning programs use only the well-defined communication APIs provided by PyTorch and NCCL. However, developers or automated parallelization tools may introduce custom communication operations that are unknown to dl^2. To mitigate this threat, dl^2 could collaborate with developers and tool maintainers to understand the semantics of these custom operations, formulate dependency rules for them, and implement interception stubs accordingly.

6.2 Practicality

The practicality of dl^2 lies in its self-contained, automated design, enabling the detection of communication deadlocks with minimal user intervention. By invoking its initialization API within a deep learning program and running the job for a few iterations, dl^2 simplifies deployment while ensuring robust runtime trace collection. Subsequently, it automatically analyzes the trace to identify potential deadlocks and generates a streamlined program in which processes reproduce the exact communication actions in the precise order, facilitating root cause analysis. Furthermore, the tool can seamlessly integrate into existing workflows of automated parallelization tools and deep learning platforms by pre-running jobs with dl^2 to detect deadlocks early. This practicality positions dl^2 as a versatile and effective solution for diverse computational scenarios.

6.3 Generality of Our Approach

Currently, dl^2 is designed to work with PyTorch and NCCL. However, we believe that its approach to detecting communication deadlocks is general and applicable to other deep learning frameworks like TensorFlow [1] and alternative collective communication libraries such as Gloo, Unified Collective Communication (UCC) [66], and Microsoft Collective Communication Library (MSCCL) [28]. This claim is supported by the structural similarities shared by PyTorch/NCCL and other frameworks and libraries. For example, TensorFlow offers distributed training support through its `tf.distribute` module, as well as various communication APIs (e.g., `tf.broadcast_to`, `tf.distribute.NcclAllReduce`, and `tf.distribute.ReplicaContext.all_gather`) that parallel those available in the PyTorch distributed communication package. Similarly, MSCCL provides a set of point-to-point and collective communication APIs with naming conventions and function signatures consistent with those of NCCL.

In addition, many widely used distributed training frameworks, such as Hugging Face Transformers [65], Microsoft DeepSpeed [44], and NVIDIA Megatron-LM [50], are all built on top of PyTorch. This compatibility allows dl^2 to be directly applied in these environments. Although this paper primarily discusses its use in training jobs, dl^2 is equally applicable to model inference, since inference uses the same communication APIs and does not involve the complexity of backpropagation.

6.4 Extensibility of dl^2

At present, dl^2 supports 30 communication APIs in total (23 from PyTorch and 7 from NCCL), covering the most commonly used cases. As noted earlier, adapting dl^2 to other frameworks (such as TensorFlow) or collective communication libraries requires minimal effort. Existing causal dependency rules can be largely reused with minor adjustments. Portability is a key design goal, and the interposition frontend, along with other core components of dl^2, has been implemented in a generic manner to minimize the overhead involved in adapting it to different operating systems.

Furthermore, dl^2 is designed to be extensible, allowing for the integration of new communication APIs. To add support for a new API, developers need to comprehend its semantics, define dependency rules for constructing execution and wait-for graphs, and implement an interception stub that captures relevant runtime information.

7 Related Work

Deadlock detection and prediction have long been active areas of research, with many studies exploring different types of deadlocks [3, 4, 7, 10, 11, 14, 17, 21, 46–48, 55, 69]. Existing approaches are traditionally classified into two main categories: static and dynamic analysis.

Static analysis-based techniques [3, 4, 10, 11, 14, 17, 21, 48, 69] aim to identify potential deadlocks by analyzing the source code of applications without executing them. These methods commonly use formal approaches, such as dataflow analysis, type systems, and model checking, to investigate synchronization patterns, lock acquisition orders, and dependencies between threads. This information is then used to construct dependency graphs (e.g., resource allocation or wait-for graphs), where cycles represent potential deadlocks. For instance, DLOS [3] statically detects deadlocks in large, complex OS kernels like Linux, using a summary-based lock-usage analysis and a reachability-based comparison to detect cyclic dependencies, coupled with a two-dimensional filtering process to minimize false positives. Similarly, Zhang et al. [69] developed a framework for deadlock detection in Rust, focusing on condition variable-lock relationships, employing pointer analysis for aliasing and lock graph analysis to detect dependency cycles in a field- and context-sensitive manner.

In contrast, dynamic analysis operates at runtime, monitoring application execution to detect potential deadlocks [7, 46, 47, 55]. Unlike static methods, dynamic analysis observes actual thread interactions, lock acquisitions, and resource usage, making it more accurate but at the cost of significant runtime overhead. Tunç et al. [55] tackled the challenge of deadlock prediction, demonstrating the intractability of achieving both soundness and completeness. They proposed two algorithms, SPDOffline and SPDOnline, that predict deadlocks efficiently in overall linear time under both offline and online scenarios. Designing thread-safe libraries is challenging due to concurrency-related defects, and traditional testing methods are often inadequate. In the realm of Java libraries, OMEN+ [46] detects deadlocks by automatically analyzing potential concurrency issues through synthesized tests, using test synthesis and execution trace analysis to identify deadlocks in multithreaded environments.

Recent research has also addressed deadlocks specific to MPI programs, often caused by circular communication dependencies. Huang et al. [17] presented a predictive analysis method for detecting deadlocks in single-path MPI programs, using a three-stage process to identify, test, and encode deadlock candidates as SMT problems. This method focuses on specific deadlock points, making it more efficient than general approaches. PCMPI [14] introduced an efficient approach for deadlock detection in MPI programs by applying path compression and focus matching techniques. It simplifies the analysis by merging consecutive identical send operations and pairing receive operations with potential send matches, effectively identifying deadlocks, particularly with wildcard receives. However, this approach is limited to point-to-point communication plus the barrier operation and does not model more complex collective operations such as Broadcast and Reduce.

The work mentioned above typically focuses on multi-threaded programs in traditional languages like C, C++, C#, Java, and Rust, which differ significantly from deep learning programs in terms of programming models and abstractions. In deep learning, high-level computations are expressed as graphs in Python, while the underlying computation and communication are executed by low-level libraries for performance, hidden from developers. Moreover, most prior research primarily addresses resource deadlocks related to lock acquisition, making it inapplicable to deep learning jobs. dl^2 is designed to address this gap, offering a dynamic analysis tool tailored for detecting communication deadlocks in deep learning jobs. It models a broader set of communication APIs and constructs wait-for graphs from runtime traces to identify deadlock cycles.

Communication deadlocks pose a major obstacle in modern deep learning workflows, prompting the development of various techniques to prevent them [5, 8, 38, 49, 68]. For example, the OneFlow framework [68] organizes collective communications statically to ensure that all GPUs execute them in a consistent order at runtime. Horovod [49] introduced a plugin for data parallelism, supervising AllReduce operations through a centralized coordinator. Pathways [5] uses gang-scheduling and global command dispatch via a centralized controller to prevent deadlocks. OCCL [38], a deadlock-free library for GPU collective communication, leverages dynamic decentralized preemption and gang-scheduling, allowing collectives to be invoked in any order, simplifying execution flow and enhancing performance. Microsoft's Collective Communication Language (MSCCLang) [8] offers a domain-specific language and runtime that enables optimized communication with inherent guarantees against deadlocks and data races. However, these methods often require global coordination, imposing constraints on execution flexibility and limiting further optimization opportunities. In contrast, dl^2 introduces no constraints on execution, allowing for deadlock detection at runtime and assisting developers with root cause analysis. It can also work with existing frameworks and automated parallelization tools to eliminate deadlock-prone plans before execution.

8 Conclusion

This paper addresses the critical issue of communication deadlocks in deep learning jobs. We propose dl^2, a dynamic analysis tool that models a deep learning job's runtime trace as an execution graph and detects deadlocks using a wait-for graph. Our tool provides an efficient means to identify deadlocks and offers valuable diagnostic capabilities to developers. Through extensive evaluation of both synthetic and real-world deep learning tasks, we demonstrate that dl^2 accurately identifies all communication deadlocks, making it a valuable addition to the development of deep learning and large language models.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283.

[2] Gustaf Ahdritz, Nazim Bouatta, Christina Floristean, Sachin Kadyan, Qinghui Xia, William Gerecke, Timothy J O'Donnell, Daniel Berenberg, Ian Fisk, Niccolò Zanichelli, Bo Zhang, Arkadiusz Nowaczynski, Bei Wang, Marta M Stepniewska-Dziubinska, Shang Zhang, Adegoke Ojewole, Murat Efe Guney, Stella Biderman, Andrew M Watkins, Stephen Ra, Pablo Ribalta Lorenzo, Lucas Nivon, Brian Weitzner, Yih-En Andrew Ban, Shiyang Chen, Minjia Zhang, Conglong Li, Shuaiwen Leon Song, Yuxiong He, Peter K Sorger, Emad Mostaque, Zhao Zhang, Richard Bonneau, and Mohammed AlQuraishi. 2024. OpenFold: Retraining AlphaFold2 yields new insights into its learning mechanisms and capacity for generalization. *Nature Methods* (2024), 1–11.

[3] Jia-Ju Bai, Tuo Li, and Shi-Min Hu. 2022. DLOS: Effective Static Detection of Deadlocks in OS Kernels. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 367–382. https://www.usenix.org/conference/atc22/presentation/bai

[4] James Brotherston, Paul Brunet, Nikos Gorogiannis, and Max Kanovich. 2022. A compositional deadlock detector for Android Java. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 955–966. doi:10.1109/ASE51524.2021.9678572

[5] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sashank Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2024. PaLM: scaling language modeling with pathways. *J. Mach. Learn. Res.* 24, 1, Article 240 (mar 2024), 113 pages.

[6] E. G. Coffman, M. Elphick, and A. Shoshani. 1971. System Deadlocks. *ACM Comput. Surv.* 3, 2 (June 1971), 67–78. doi:10.1145/356586.356588

[7] Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. 2018. Dynamic Deadlock Verification for General Barrier Synchronisation. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 1 (dec 2018), 38 pages. doi:10.1145/3229060

[8] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. 2023. MSCCLang: Microsoft Collective Communication Language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 502–514. doi:10.1145/3575693.3575724

[9] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[10] Jyotirmoy Deshmukh, E. Allen Emerson, and Sriram Sankaranarayanan. 2009. Symbolic Deadlock Analysis in Concurrent Libraries and Their Clients. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 480–491. doi:10.1109/ASE.2009.14

[11] Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. doi:10.1145/945445.945468

[12] Yanjie Gao, Xiaoxiang Shi, Haoxiang Lin, Hongyu Zhang, Hao Wu, Rui Li, and Mao Yang. 2023. An Empirical Study on Quality Issues of Deep Learning Platform. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 455–466. doi:10.1109/ICSE-SEIP58684.2023.00052

[13] Zhenyu Guo, Dong Zhou, Haoxiang Lin, Mao Yang, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. 2011. G2: A Graph Processing System for Diagnosing Distributed Systems. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. USENIX Association, Portland, OR. https://www.usenix.org/conference/usenixatc11/g2-graph-processing-system-diagnosing-distributed-systems

[14] Jiale Hao, Meng Wang, and Hong Zhang. 2024. Efficient Deadlock Detection in MPI Programs with Path Compression and Focus Matching. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware* (Macau, China) *(Internetware '24)*. Association for Computing Machinery, New York, NY, USA, 467–476. doi:10.1145/3671016.3674822

[15] Mert Hidayetoglu, Simon Garcia de Gonzalo, Elliott Slaughter, Pinku Surana, Wen mei Hwu, William Gropp, and Alex Aiken. 2024. HiCCL: A Hierarchical Collective Communication Library. arXiv:2408.05962 [cs.DC] https://arxiv.org/abs/2408.05962

[16] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc., 10. https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf

[17] Yu Huang, Benjamin Ogles, and Eric Mercer. 2021. A predictive analysis for detecting deadlock in MPI programs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 18–28. doi:10.1145/3324884.3416588

[18] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). 2019. *Automated Machine Learning - Methods, Systems, Challenges*. Springer.

[19] Meta Incubator. 2024. Gloo: collective communications library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo.

[20] Michael Kerrisk. 2024. ld.so(8) – Linux manual page. https://www.man7.org/linux/man-pages/man8/ld.so.8.html.

[21] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. 2016. Sound static deadlock analysis for C/Pthreads. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 379–390. doi:10.1145/2970276.2970309

[22] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, Mao Yang, Lintao Zhang, and Lidong Zhou. 2024. nnScaler: Constraint-Guided Parallelization Plan Generation for Deep Learning Training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 347–363. https://www.usenix.org/conference/osdi24/presentation/lin-zhiqi

[23] Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. 2020. Multilingual Denoising Pre-training for Neural Machine Translation. *CoRR* abs/2001.08210 (2020). arXiv:2001.08210 https://arxiv.org/abs/2001.08210

[24] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin Transformer: Hierarchical Vision Transformer using Shifted Windows. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. 9992–10002. doi:10.1109/ICCV48922.2021.00986

[25] LlamaTeam. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[26] Message Passing Interface Forum. 2023. *MPI: A Message-Passing Interface Standard Version 4.1*. https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

[27] Microsoft. 2022. Swin Tiny Patch4 Window7 224 Model. https://huggingface.co/microsoft/swin-tiny-patch4-window7-224.

[28] Microsoft. 2024. Microsoft Collective Communication Library (MSCCL). https://github.com/microsoft/msccl.

[29] Don P. Mitchell and Michael J. Merritt. 1984. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, British Columbia, Canada) *(PODC '84)*. Association for Computing Machinery, New York, NY, USA, 282–284. doi:10.1145/800222.806755

[30] NetworkX. 2024. NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. https://networkx.org.

[31] NNI. 2018. NNI (Neural Network Intelligence): a lightweight but powerful toolkit to help users automate Feature Engineering, Neural Architecture Search, Hyperparameter Tuning and Model Compression. https://github.com/microsoft/nni.

[32] NVIDIA. 2024. Collective Operations. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html.

[33] NVIDIA. 2024. CUDA Runtime API. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EXECUTION.html.

[34] NVIDIA. 2024. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[35] NVIDIA. 2024. NVIDIA CUDA Profiling Tools Interface. https://developer.nvidia.com/cupti.

[36] NVIDIA. 2024. NVIDIA CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit.

[37] David L. Olson and Dursun Delen. 2008. *Advanced Data Mining Techniques* (1st ed.). Springer Publishing Company, Incorporated.

[38] Lichen Pan, Juncheng Liu, Jinhui Yuan, Rongkai Zhang, Pengze Li, and Zhen Xiao. 2023. OCCL: a Deadlock-free Library for GPU Collective Communication. arXiv:2303.06324 [cs.DC] https://arxiv.org/abs/2303.06324

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, Vol. 32. Curran Associates, Inc., 8024–8035. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[40] PyTorch. 2024. Automatic differentiation package - torch.autograd. https://pytorch.org/docs/stable/autograd.html.

[41] PyTorch. 2024. Distributed communication package - torch.distributed. https://pytorch.org/docs/stable/distributed.html.

[42] PyTorch. 2024. Distributed Data Parallel. https://pytorch.org/docs/stable/notes/ddp.html.

[43] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[44] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. doi:10.1145/3394486.3406703

[45] Joshua Romero, Junqi Yin, Nouamane Laanait, Bing Xie, M. Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. 2022. Accelerating Collective Communication in Data Parallel Training across Deep Learning Frameworks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1027–1040. https://www.usenix.org/conference/nsdi22/presentation/romero

[46] Malavika Samak and Murali Ramanathan. 2014. Omen+: a precise dynamic deadlock detector for multithreaded Java libraries. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 735–738. doi:10.1145/2635868.2661670

[47] Malavika Samak and Murali Krishna Ramanathan. 2014. Trace Driven Dynamic Deadlock Detection and Reproduction. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 29–42. doi:10.1145/2555243.2555262

[48] Anirudh Santhiar and Aditya Kanade. 2017. Static deadlock detection for asynchronous C# programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 292–305. doi:10.1145/3062341.3062361

[49] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799 http://arxiv.org/abs/1802.05799

[50] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[51] David Shriver. 2024. Intercepts allows you to intercept function calls in Python and handle them in any manner you choose. https://github.com/dlshriver/intercepts.

[52] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2015. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9. doi:10.1109/CVPR.2015.7298594

[53] Hind Taud and Jean-Franccois Mas. 2018. Multilayer perceptron (MLP). *Geomatic approaches for modeling land change scenarios* (2018), 451–455.

[54] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[55] Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound Dynamic Deadlock Prediction in Linear Time. *Proc. ACM Program. Lang.* 7, PLDI, Article 177 (jun 2023), 26 pages. doi:10.1145/3591291

[56] GitHub User. 2022. Training hangs in the end while calling dist.barrier(). https://github.com/huggingface/transformers/issues/17478.

[57] GitHub User. 2023. Add validation for send/recv sizes. https://github.com/pytorch/pytorch/issues/113376.

[58] GitHub User. 2023. all_to_all_single stuck when using output_split_sizes = [1, 3] and input_split_sizes = [1, 3]. https://github.com/pytorch/pytorch/issues/117486.

[59] GitHub User. 2023. [Bug] dist.broadcast with multi GPU only works on torch.float32, but errors on int64, int32 and hangs on float16. https://github.com/pytorch/pytorch/issues/118696.

[60] GitHub User. 2023. Distributed hangs when doing hierarchical communication. https://github.com/pytorch/pytorch/issues/130102.

[61] GitHub User. 2023. FSDP hangs when combining MoE architecture. https://github.com/pytorch/pytorch/issues/126616.

[62] GitHub User. 2023. Interleaved isend and irecv causes hang. https://github.com/pytorch/pytorch/issues/109401.

[63] GitHub User. 2023. P2P operations hang when mixing the usage of default and non-default communication groups. https://github.com/pytorch/pytorch/issues/116590.

[64] GitHub User. 2024. For AllReduce communications with a shape mismatch, some cases will hang while others will not. https://github.com/NVIDIA/nccl/issues/1417.

[65] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Qun Liu and David Schlangen (Eds.). Association for Computational Linguistics, Online, 38–45. doi:10.18653/v1/2020.emnlp-demos.6

[66] Unified Communication X. 2024. Unified Collective Communication (UCC). https://github.com/openucx/ucc.

[67] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[68] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, Haoran Zhang, and Jie Zhao. 2021. OneFlow: Redesign the Distributed Deep Learning Framework from Scratch. *CoRR* abs/2110.15032 (2021). arXiv:2110.15032 https://arxiv.org/abs/2110.15032

[69] Yu Zhang, Kaiwen Zhang, and Guanjun Liu. 2024. Static Deadlock Detection for Rust Programs. arXiv:2401.01114 [cs.PL] https://arxiv.org/abs/2401.01114

[70] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin