GroPipe: A Grouped Pipeline Hybrid Parallel Method for Accelerating DCNNs Training

Bin Liu, Yongyao Ma, Zijian Hu, Zeyu Ji, Zhenli He, *Senior Member, IEEE*, and Keqin Li, *Fellow, IEEE*

***Abstract*—Training large Deep Convolutional Neural Networks (DCNNs) with increasingly large datasets to improve model accuracy has become extremely time-consuming. Distributed training methods, such as data parallelism (DP) and pipeline model parallelism (PMP), offer potential solutions but face challenges like load imbalance and significant communication overhead. This paper introduces GroPipe, a novel architecture that synergistically integrates PMP and DP, markedly improving training speeds. GroPipe employs an automatic model partitioning algorithm based on a performance projection technique, ensuring load balance and facilitating quantitative performance evaluation in PMP. Additionally, it adopts a group-based delayed asynchronous communication strategy to efficiently reduce communication overhead in DP. Using the ResNet and VGG models with the ImageNet dataset, extensive experiments are performed on an 8-GPU server and demonstrate GroPipe's effectiveness. GroPipe achieves substantial improvements in time to accuracy, showing an average improvement of 42.2% and 14.0% on the ResNet series, and 79.2% and 43.9% on the VGG series, without compromising Top-1 accuracy.**

***Index Terms*—Delayed asynchronous communication, deep convolutional neural networks, hybrid parallelism, pipelined model parallelism.**

Received 24 February 2024; revised 2 April 2025; accepted 25 April 2025. Date of publication 5 May 2025; date of current version 11 June 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62376226 and Grant 62362068, in part by Shaanxi's Key Research and Development Program under Grant 2023-ZDLNY-63, in part by Xianyang's Key Research and Development Program under Grant L2022-ZDYF-NY-019, in part by the Natural Science Basic Research Program of Shaanxi Province under Grant 2022JQ-690, and in part by the Open Foundation of Yunnan Key Laboratory of Software Engineering under Grant 2023SE208. Recommended for acceptance by M. Kandemir. *(Corresponding authors: Zeyu Ji; Zhenli He.)*

Bin Liu is with the College of Information Engineering, Northwest A&F University, Yangling 712100, China, also with the Key Laboratory of Agricultural Internet of Things, Ministry of Agriculture and Rural Affairs, Shaanxi Engineering Research Center of Agricultural Information Intelligent Perception and Analysis, Shaanxi 712100, China, and also with Shaanxi Key Laboratory of Agricultural Information Perception and Intelligent Service, Shaanxi 712100, China (e-mail: liubin0929@nwafu.edu.cn).

Yongyao Ma, Zijian Hu, and Zeyu Ji are with the College of Information Engineering, Northwest A&F University, Yangling 712100, China (e-mail: mtyty2@nwafu.edu.cn; 2021056015@nwafu.edu.cn; zeyu.ji@nwafu.edu.cn).

Zhenli He is with Yunnan Key Laboratory of Software Engineering, Yunnan University, Kunming 650091, China, and also with the School of Software, Yunnan University, Kunming 650091, China (e-mail: hezl@ynu.edu.cn).

Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

This article has supplementary downloadable material available at https://doi.org/10.1109/TC.2025.3566869, provided by the authors.

Digital Object Identifier 10.1109/TC.2025.3566869

I. Introduction

DEEP convolutional neural networks (DCNNs) have a wide range of applications and significant contributions in the fields of image classification, autonomous driving, smart medicine, and smart agriculture [1], [2], [3]. These successes depend on the complex structure of the model and massive datasets. However, the boom in demand for high-quality intelligent services requires the development of large-scale models trained on massive datasets [4], [5], [6]. This demand has exceeded the computational limits of traditional stand-alone or host-based training methods.

To meet these computational demands, scholars have proposed distributed architectures that leverage multi-GPU servers for parallelized training. Notable works in this domain have creatively and significantly influenced the field by enhancing computational power through distributed clusters. Pipeline model parallelism (PMP), data parallelism (DP), and hybrid parallelism are the three major strategies for DCNNs' training.

PMP [7], [8], [9], [10], [11], [12], [13] firstly divides the large model into separate sub-models and assigns them to each node for training individually. It is similar to traditional model parallelism, as illustrated in Fig. 1. Due to the under-utilization of computational resources caused by traditional model partitioning [9], PMP divides mini-batches into micro-batches. After the first stage has finished processing the first micro-batch, the activation of this stage is immediately passed to the next stages. PMP is particularly advantaged when dealing with models too large to fit into a single device's memory. By splitting the model across multiple devices, it facilitates the training of significantly larger models. This method, however, leads to inefficiencies due to pipeline idling and the difficulty in balancing the workload across different model segments.

DP [14], [15], [16], [17], [18], [19], [20] involves dividing the training input data among multiple workers. Each worker maintains a local copy of the model weights and computes local gradients independently while using either parameter servers or all-reduce primitive synchronization [17], as illustrated in Fig. 2. With synchronized gradients, global weights are updated by executing loop iterations until they reach convergence. DP is optimal for scenarios where the model size is manageable within individual device memory limits, and the primary goal is to leverage multiple processors to handle large datasets efficiently. Its simplicity and scalability make it popular for many standard deep learning tasks. However, it faces

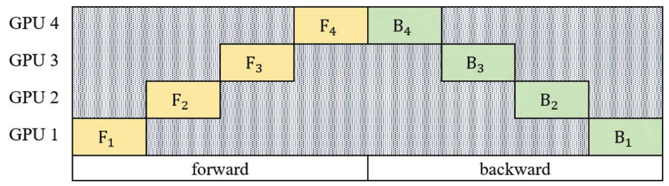


Fig. 1. Traditional model partitioning with 4 GPUs.

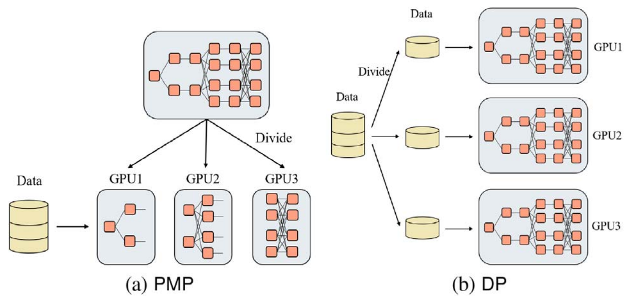


Fig. 2. Pipelined model parallelism and data parallelism.

challenges with large models due to its high communication overhead and memory constraints.

PMP and DP provide solutions for the efficient training of DCNNs. However, PMP suffers from poor scalability, and DP encounters storage limitations. To overcome these challenges, hybrid parallel methods [21], [22], [23] that combine the strengths of multiple parallel algorithms have emerged. It offers a well-rounded solution for handling large models and datasets, by providing a balanced approach to resource utilization.

In the academic community, this method is recognized as a practical solution to the imbalance among model size, data volume, and computational capacity. However, it also introduces greater complexity in its implementation and can lead to increased communication overhead. More specifically, researchers are currently struggling with two major challenges:

- **Load Balancing**. Using hybrid parallelism, it is difficult to achieve an even division of computational workload in distributed systems. Different model segments have different computational requirements, which may lead to insufficient resource utilization or slower convergence.
- **The Cost of Communication**. Coordination among various compute units, each executing different parallelism methods, introduces significant communication overhead. Efficient and timely communication is crucial for performance but remains a complex issue in practice.

The paper introduces GroPipe, a novel architecture designed to tackle the challenges of model partitioning and fine-grained parallelism. GroPipe ensures balanced workload distribution across GPUs, enhancing resource efficiency and reducing the detrimental effects of load imbalance on training performance. Its innovative group-based delayed asynchronous communication strategy further decreases communication overhead, boosting training efficiency. The architecture's strength lies in its enhanced convergence speed and resource utilization, making it an ideal solution for handling large-scale models and datasets in distributed training. GroPipe is implemented by PyTorch and specializes in training deep learning models quickly and efficiently. The demonstrated practical applicability and effectiveness of GroPipe confirm its status as a dependable tool for efficiently training deep learning models on prevalent datasets.

The research proposes a fresh perspective on hybrid distributed training for DCNNs, highlighting the importance of performance analysis and evaluation in achieving load balancing. The main contributions of our work are as follows:

- A hybrid parallel training architecture is constructed to speedup the training of DCNNs. This architecture synergistically combines PMP with DP and introduces a novel grouped pipeline hybrid parallel method. This method enhances the computational resource utilization and enables superior scalability of DCNNs across a multitude of devices or processors.
- An Automatic Model Partitioning Algorithm (AMPA) is presented to facilitate optimal load distribution. AMPA employs a performance projection technique to quantitatively evaluate and adjust processing times for different model partitions. The optimal partitioning scheme derived from this assessment is then applied to map model segments to the appropriate hardware resources, promoting equitable load distribution and augmented GPU efficiency.
- A group-based delayed asynchronous communication strategy is proposed to refine the efficiency of communication processes. The strategy is activated during backward propagation to reduce the gradient synchronization time of global weights. The adoption of group-based delayed asynchronous communication effectively reduces communication latency and enhances the throughput of GPUs.

In summary, this research advances parallel training methodologies for large-scale models by specifically addressing inherent computational inefficiencies. The remainder of this paper is organized into the following sections: Section II reviews existing research relevant to our study.

Section III presents the GroPipe architecture and the implementations in detail. In Section IV, the experimental results are shown to validate the efficiency of the proposed hybrid parallel method. In Section V, we summarize the findings of this paper and offer insights into our future research directions.

II. Related Work

In this section, the relevant literature about the study is reviewed, with a focus on DP and PMP strategies in DCNNs distributed training methods. Additionally, the differences and uniqueness of this research compared to existing studies are emphasized.

For DCNNs' distributed training, hybrid parallelism combines data and model parallelism's features, offering a flexible solution. It addresses memory constraints in DP and "bubbles" in PMP by merging the strengths of both techniques. This integrative approach provides a more balanced and flexible solution, surpassing the constraints encountered when each technique is applied in isolation. Generally, hybrid parallelism can be categorized based on how these techniques are integrated:

TABLE I
COMPARISON OF DIFFERENT METHODS

Method	Training Efficiency	Memory Overhead	Load Imblancing	Communication Overhead	Quantitative Analysis
GroPipe	✓		✓	✓	✓
Hippie [24]	✓	✓		✓	
PipeDream [10]	✓	✓	✓		
HetPipe [25]	✓			✓	
Alpa [26]	✓		✓		
DAPPLE [27]	✓	✓		✓	

- **Operator Parallelism with DP**: This approach involves splitting the operators across different devices (model parallelism) and simultaneously processing different batches of data on each of these segments (DP). Alpa [26] presents an innovative approach to automating the parallelization process for distributed deep learning. It is designed to automatically generate model-parallel execution plans, leveraging inter-operator and intra-operator parallelisms. This hierarchical optimization allows Alpa to manage the computational graph and device cluster efficiently, enabling effective and scalable training of large-scale neural networks with minimal manual intervention. 3D parallel technology in DeepSpeed [28] is an advanced training approach that combines data and model parallelism. This method distributes the model layers and the data batches among various workers, enhancing memory efficiency. This design facilitates scaling AI models across multiple GPUs and nodes, optimizing memory use, cutting down communication overhead, and boosting the overall training speed.
- **PMP with DP**: This variant combines PMP (where different stages of the model's computation are done on different devices) with DP. Ye et al. addressed the issue of excessive GPU memory consumption during distributed training and introduced a hybrid parallel method called Hippie [24]. This approach introduces communication and computation scheduling operations to mitigate memory consumption and hide communication overhead. HetPipe [25] trains large DCNN models on heterogeneous GPU clusters, which reduces training time and improves model convergence speed. This strategy utilizes dynamic scheduling algorithms to ensure model convergence, reduce GPU memory consumption, and identify the optimal parallelization scheme. Narayana et al. propose a general hybrid parallel training system PipeDream [10], which combines inter-batch pipeline and intra-batch parallelism to achieve high throughput.

The existing research has significantly pushed the boundaries of what we can achieve with DCNN models. However, two notable areas require further attention. The first is load imbalance. While distributing workloads across devices, it's often observed that the distribution is not always even, leading to inefficiencies in the system, particularly in hybrid parallelism. The second is high communication overhead. Communication among devices in distributed training can become a bottleneck, which results in increased training time. This issue is even more pronounced in hybrid parallelism scenarios, where the need for dense communication can substantially increase computational overhead. Table I illustrates the distinctions between GroPipe and other methods in addressing primary issues.

To emphasize the innovation and originality of this paper, the differences between our research and existing studies in DCNN training are presented as follows:

- **Innovative Approach to Load Balancing:** In contrast to existing research, our study rigorously evaluates the impact of efficient load balancing on DCNN training by integrating AMPA and performance prediction methods. This approach accurately minimizes the time difference among GPUs. This is quite different from traditional approaches, which focus on optimizing resource allocation to improve training efficiency.
- **Group-Based Delayed Asynchronous Communication Strategy:** This study introduces a novel group-based delayed asynchronous communication strategy, a technique that has never been employed in existing DCNNs training methods. The strategy is initiated during backward propagation to shorten the gradient synchronization duration. This innovation not only reduces the communication overhead but also enables the GPU to perform computational tasks with higher efficiency.

III. GROPIPE: GROUPED PIPELINE PARALLELISM

This section introduces GroPipe, a hybrid parallel training architecture that combines PMP and DP to improve model scalability and training efficiency. **Note:** To help readers better understand the key concepts, fundamental knowledge on distributed training, including DP, model partitioning, and parameter update strategies, along with a detailed discussion on DP and Torchgpipe, has been provided. Due to space constraints, this content has been relocated to Section I of the supplementary file, available online.

A. GroPipe Architecture for Large DCNNs

To efficiently train large models on a single server with multiple GPUs, the proposed GroPipe architecture combines the advantages of PMP and DP. Fig. 3 shows the architecture of the proposed hybrid parallel method. GroPipe first assigns computing resources into N groups based on grouping configurations, with each group consisting of k homogeneous GPUs (e.g., $k = 3$ as shown in Fig. 3). Then, for the given large DCNN model and allocated k GPUs, the model partitioner automatically divides the model into k partitions such that the speedup performance of the pipeline executed within each group can be maximized.

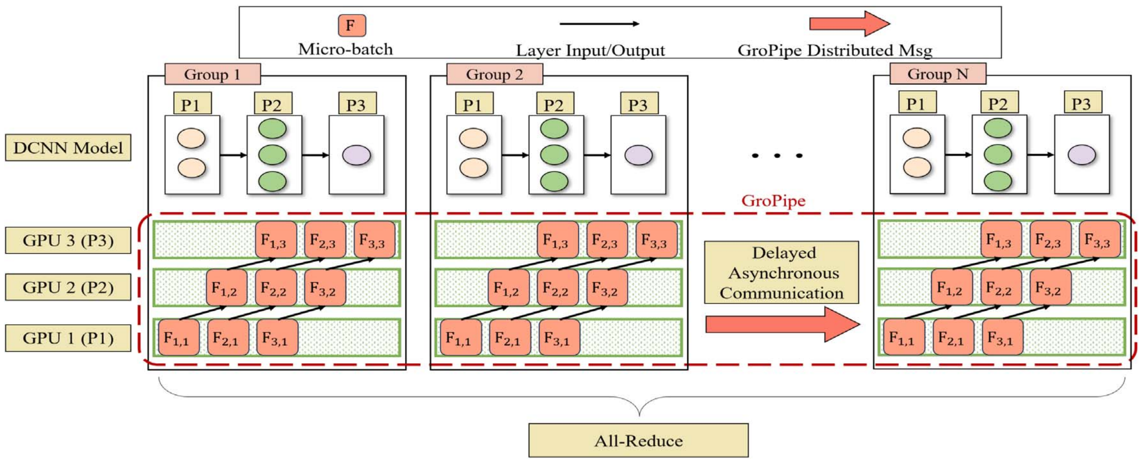


Fig. 3. GroPipe utilizes pipeline model parallelism within groups and data parallelism between groups. Asynchronous delayed communication is employed between groups to reduce communication overhead.

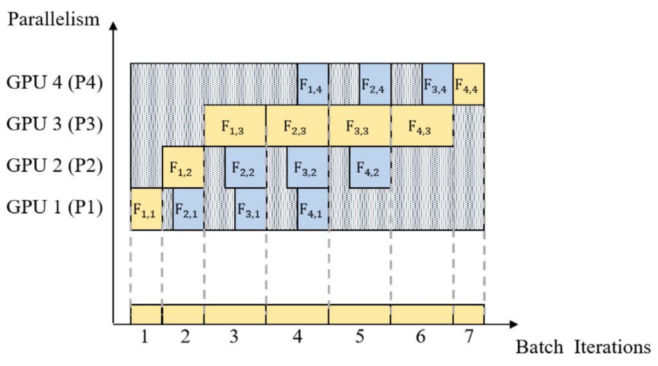


Fig. 4. Unevenly divided pipeline model with 4 GPUs.

Finally, DP is employed among the groups to achieve efficient model training.

B. Performance Analysis

This paper proposes an advanced performance analysis method called performance projection to ensure load balancing in model partitioning. Performance projection provides quantitative performance metrics for partitioning. It ensures an even distribution of computational loads across layers, preventing resource wastage and performance bottlenecks.

Fig. 4 shows an example of a forward pass diagram for an unevenly divided pipeline model. Assume that the model is divided into k partitions and the mini-batch is divided into m micro-batches. $F_{i,j}$ denotes the forward pass task in the *jth* partition with the *ith* micro-batch, $B_{i,j}$ denotes the backward pass task in the *jth* partition with the *ith* micro-batch. The total minimum latency of the entire computation T^* is written as Equation (1):

$$T^* = \min(T_F + T_B), \tag{1}$$

where T_F represents the forward pass time and T_B represents the back pass time. Assuming $T_{M,K}$ represents the total time required to train m micro-batches on k GPUs, and $t_{m,k}$ represents the time required to train the *mth* micro-batch on the *kth* GPU. In the pipeline model, the total time T_F is determined by the duration required to finish the final task on the last GPU. This time frame is influenced by the predecessor task on the same GPU as well as the corresponding task on a preceding GPU. This sequence of dependencies extends regressively to the very first task undertaken on the initial GPU, and we have:

$$T_F = T_{M,K} = t_{m,k} + \max(T_{m,k-1}, T_{m-1,k}). \tag{2}$$

Given that the study is carried out in a homogeneous environment, the duration required for a single GPU to process different batches is the same in this case. Consequently, the total time can be seen as the longest path from $F_{1,1}$ to $F_{M,K}$. Equation (2) can be further refined and expressed in an altered form Equation (3), allowing for a more streamlined analysis and interpretation of the data:

$$T_F = T_{stable} + T_{rising}. \tag{3}$$

In Equation (3), the model training time can be divided into two parts: T_{stable} and T_{rising}. T_{stable} refers to the training time required by the GPU with the longest training time, as shown by GPU 3 in Fig. 4. T_{rising} represents the time required for each of the remaining GPUs to train one micro-batch. Equation (4) is utilized to compute the GPU number n associated with the longest path T_{stable}:

$$n = arg \max_{j=1}^{k} \left(\sum_{i=1}^{m} t_{i,j} \right). \tag{4}$$

Subsequently, the value n is substituted into Equation (5), and Equation (6) to calculate T_{stable} and T_{rising}:

$$T_{stable} = \sum_{i=1}^{m} t_{i,n}, \tag{5}$$

$$T_{rising} = \sum_{p=n+1}^{k} t_{k,p} + \sum_{q=1}^{n-1} t_{1,q}. \tag{6}$$

Ultimately, under the assumption that the computational process of forward propagation is equivalent to that of backpropagation, arranging the aforementioned equations culminates in

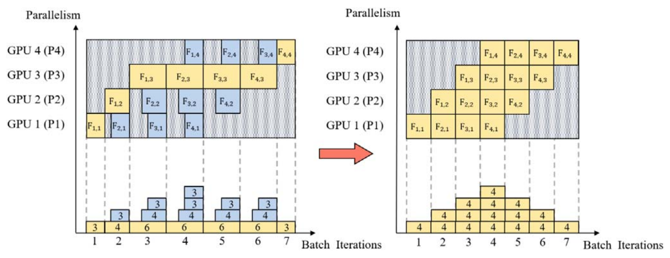


Fig. 5. Performance mapping of the pipeline with 4 GPUs enables load balancing across the GPUs.

the derivation of the final Equation (7):

$$T_B = T_F = \sum_{i=1}^{m} t_{i,n} + \sum_{p=n+1}^{k} t_{k,p} + \sum_{q=1}^{n-1} t_{1,q}. \tag{7}$$

Fig. 5 illustrates the performance projection method. It keeps track of the training time for each partition in each batch iteration and records the differences among them. This method quantitatively analyzes the performance differences of each partition compared to the average value from a statistical perspective. When the performance dispersion of each partition is smaller than a specified threshold, the AMPA can identify the optimal partitioning approximately. Assuming the model is divided into k partitions, with each partition having a runtime of T_i, the average runtime per partition is μ. The performance dispersion (PD) can be summarized as follows:

$$PD = \frac{\sum_{i=1}^{k} (T_i - \mu)^k}{k\mu}. \tag{8}$$

C. Automatic Model Partitioning for PMP

To address the issue of load imbalance in PMP, AMPA is proposed. It consists of three components. Firstly, the AMPA is used to divide a large DCNN model into multiple partitions to solve the programmers' burden of partitioning manually. Secondly, it utilizes the performance projection method to assess the time differences among partitions quantitatively and tries to minimize these differences. Finally, the PMP algorithm is applied to the micro-batches divided by a mini-batch to improve the degree of training parallelism.

Algorithm 1 describes the AMPA. This algorithm aims to minimize Equation (1), allowing for load balancing across different partitions of the model. The algorithm divides the different layers of a large model into k partitions and puts them on k GPU devices within a group, respectively. In this paper, the volume of parameters for different DCNN models is investigated, as shown in Table II. The theoretical analysis shows that the parameters of the full-connection layer are a small percentage of the DCNN model. Therefore, this algorithm does not require any partition on the full-connection layer. To maximize the efficiency of the pipeline, the execution time of each partition should be as equal as possible. The inputs of the algorithm are an L-layer DCNN model, a mini-batch, and the number of partitions. The output is a list of the number of layers in each partition. Lines 1-3 in the algorithm are independent pre-training processes of the DCNN model. The execution time of each layer is first collected

Algorithm 1: Automatic Model Partitioning Algorithm.

```
Input:
    model: the sequential DCNN model;
    mini-batch: the input data;
    $k$: the number of partitions;
Output:
    partitions: a layer count list for each partition;
1  collect the execution time for each layer based on
     mini-batch, i.e. $T = [t_1, t_2, ..., t_L]$, which $t_i$ indicates
     the sum of execution time (including both forward
     pass and backward pass) of $layer_i$ (unit:$\mu$s) and $L$ is
     the number of layers of the model;
2  normalize the $t_i$ of T to [0,1] with Equation (9);
3  divide $T$ evenly into $k$ partitions, i.e.
     $P = [p_1, p_2, ..., p_k]$ which
     $p_i = [t_{(i-1)\times s+1}, t_{(i-1)\times s+2}, ..., t_{i\times s}]$ $(s = \frac{L}{k})$;
4  while True do
5      find the maximal partition in $P, p_{max}$;
6      while True do
7          find the minimal partition in $P, p_{min}$;
8          if the performance projection method meets
             the criteria then
9              return $len(p_1)$, $len(p_2)$, ..., $len(p_k)$;
10         end
11         if $max \leq min$ then
12             $h = min - 1$;
13             move the last element in $p_h$ to $p_{min}$;
14         else
15             $h = min + 1$;
16             move the first element in $p_h$ to $p_{min}$;
17         end
18         if $max = h$ then
19             break;
20         end
21     end
22 end
```

TABLE II
THE VOLUME OF PARAMETERS

Model	Full-Connection Layer Params	Total Params
ResNet-50	1.9M	24.4M
ResNet-101	1.9M	42.5M
ResNet-152	1.9M	57.4M
VGG-16	3.9M	131.9M
VGG-19	3.9M	137M

based on the mini-batch in pre-processing. Then, to eliminate the undesirable effects caused by singular time values, all the times are normalized to [0,1], as shown in Equation (9):

$$t_i = \frac{t_i - min(T)}{max(T) - min(T)}. \tag{9}$$

At last, the times are divided into k partitions. Lines 4-22 are the partition iteration process with a double loop. The maximal partition representing the longest-running partition p_{max} is found in

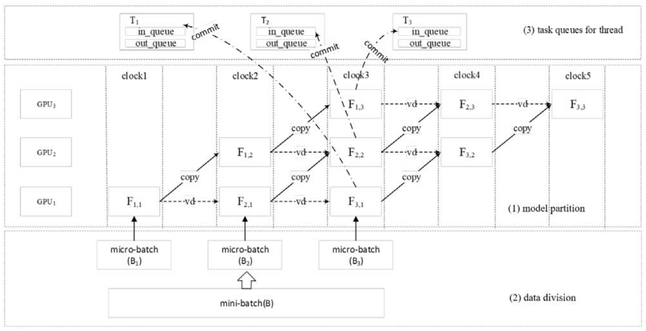


Fig. 6. Execution diagram of PMP. The solid arrows indicate copy dependencies between devices, the dotted arrows represent virtual dependencies (vd) between batches and the dash-dot arrows represent committing the task of $F_{i,j}$ to the *in_queue*$_i$.

the outer loop. The minimal partition p_{min} with the minimum runtime and the p_h next to p_{min} between p_{max} and p_{min} are found in the inner loop. On each iteration, the current partition P is changed to another one P', and the only difference between P and P' is that either the last element of p_h is moved to p_{h+1} or the first element of p_h is moved to p_{h+1} for $h \in \{1, 2, ..., k\}$. The algorithm introduces the performance projection method to quantitatively measure the time differences among partitions and aims to achieve load balancing by minimizing these differences. The sum of the maximal partition in P is gradually smaller in each iteration, so the algorithm eventually terminates and returns a partition result as shown in Lines 8-10. The total time complexity of the algorithm is only $O(k \times L^3)$ where k is the number of partitions and L is the number of layers. Experiments show that the algorithm takes about ten seconds for all models in the experiment and has a negligible impact on the training time.

Building upon the load balancing partitions, a pipeline model parallelism algorithm based on intra-group algorithm (PMPIA) is proposed. As shown in Fig. 6, the main idea of PMPIA is to split a large DCNN model into multiple partitions with similar training times. Then a mini-batch is divided into multiple micro-batches, which are fed into the multiple partitions. These partitions are trained in a pipelined-parallel manner within a group. The algorithmic details of PMPIA are presented in Algorithm 2.

PMPIA is shown in Algorithm 2, its inputs are a mini-batch and the result of k-partition in Algorithm 1. The outputs are the trained parameters of the model. Lines 1-2 in the algorithm are initialization work. The k host threads (T_1,T_2, … , T_k) are first created, and each thread has a pair of task queues (*in_queue*$_i$,*out_queue*$_i$,$i = 1, 2, ..., k$). Each thread is responsible for scheduling and executing the tasks submitted to the corresponding *in_queue*$_i$. In addition, multiple partitions of the large model need to be copied to the corresponding GPU devices, respectively. Lines 3-9 are the training iteration process. A mini-batch is first divided into m micro-batches and then fed into the corresponding partitions one after the other in a pipelined manner for executing forward pass and backward pass. Lines 11-19 are the forward scheduling process of the pipeline. All tasks $F_{i,j}$ per clock cycle are submitted to the *in_queue*$_i$ ($i =$

Algorithm 2: Pipelined Model Parallelism Based on Intra-Group Algorithm.

```
Input:
    mini-batch: the input data;
    partitions: model partitioning result in Algorithm 1;
Output:
    model_params: the trained parameters of the model;
1  Create $T_1, T_2, ..., T_k$ threads, $T_i$ contains a pair of
     queues $(in\_queue_i,\ out\_queue_i)$;
2  copy $partitions[i]$ of the model to $device_i$,
     $i = 1, 2, 3..., k$;
3  for iteration from 1 to the last mini-batch do
4      divide a mini-batch $(M)$ into $m$ micro-batches,
         $[M_1, M_2, ..., M_m]$;
5      PIPELINE();
6      calculate the loss of all micro-batches;
7      backward(loss);
8      update(model_params);
9  end
10 return model_params;
11 function PIPELINE():
12 for clock from 1 to $m + k - 1$ do
13     for $i, j$ such that $i + j = clock + 1$ do
14         build a virtual dependency: $F_{i,j}$ depends on
             $F_{i-1,j}$;
15         build a copy dependency: copy the output of
             $F_{i,j-1}$ from $out\_queue_{j-1}$ to $device_j$;
16         commit $F_{i,j}$ to the $in\_queue_j$, then put the
             output into $out\_queue_j$;
17     end
18     put an exit message into all $in\_queues$;
19 end
```

$1, 2, ..., k$) for asynchronously processing the tasks, and then put the results of the tasks to the corresponding *out_queue*$_i$ ($i = 1, 2, ..., k$). Due to PyTorch's dynamic computation graph, the automatic differentiation (autograd) engine of PyTorch may not run exactly in the reverse order of execution as in the forward pass during the backward pass. Hence two key dependencies structure of the graph have to be built to force the pipeline to work as desired. First, the dependencies of the micro-batches need to be established (as line 14). It is enforced that $F_{i-1,j}$ must be completed before executing $F_{i,j}$ and $B_{i,j}$ must be completed before executing $B_{i-1,j}$. Second, the dependencies of the partitions (devices) need to be established (as line 15), it is enforced that $F_{i,j}$ must be completed before executing $F_{i,j+1}$ and $B_{i,j}$ must be completed before executing $B_{i,j-1}$.

D. Group-Based Delayed Asynchronous Communication for DP

As illustrated in Fig. 7, to reduce communication further overhead to speed up the training of large DCNN models, a novel DP is presented in GroPipe. In this paper, DP is supported by a notion of a group that consists of k GPUs. A group is allowed to load a large DCNN model by aggregating multiple

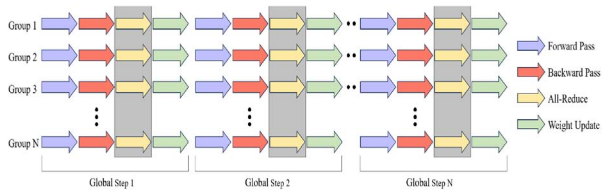


Fig. 7. Training process of DP among the groups.

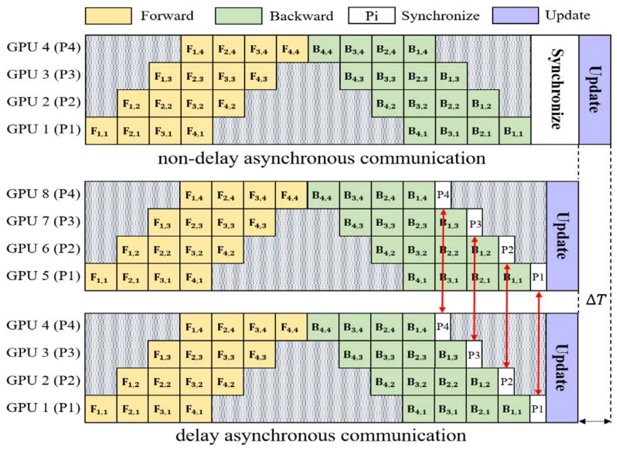

Forward
Backward
Pi Synchronize
Update
non-delay asynchronous communication
delay asynchronous communication
ΔT

Fig. 8. Comparison of training time with and without delayed asynchronous communication.

GPUs when an individual GPU may be resource-limited. DP is used among multiple groups, each group with multiple GPUs to load the whole large DCNN model. By distributing the data across multiple devices and processing them independently, DP reduces the amount of data that needs to be communicated among devices. Instead of exchanging all the intermediate activations and gradients among devices for each training iteration, only the parameter updates need to be communicated. This reduction in communication volume helps mitigate the communication overhead and improves training efficiency.

However, DP requires all GPUs to perform all-reduce operations simultaneously, leading to high communication overhead and network congestion. Fig. 8 presents a group-based delayed asynchronous communication strategy for DP to address the mentioned challenges. Considering that all-reduce collective communication is time-consuming, especially on large models with massive parameters. GroPipe should not communicate all gradients in a single all-reduce. Otherwise, there is no opportunity to conceal communication latency. As illustrated in Fig. 8, a naive solution inserts a gradient synchronization phase after the backward pass and before updating parameters. Instead of traditional delayed asynchronous communication, GroPipe leverages the advantages of its architecture by implementing delayed asynchronous communication based on different groups, further achieving higher throughput and lower latency. The process includes four steps: First, GroPipe registers one autograd hook for each partition within all groups. Second, PMP and DP are simultaneously employed for training. Third, the corresponding hook fires after the partition completes a backward pass for the current

TABLE III
EXPERIMENTAL ENVIRONMENT

Configuration	Value
Graphics processor units	NVIDIA Tesla T4 $\times$ 8
GPU memory	16GB $\times$ 8
Operation system	Ubuntu 16.04.2 LTS (64-bit)
Deep learning framework	PyTorch
CUDA version	CUDA 10.1
Dataset	ImageNet

TABLE IV
BENCHMARK MODELS

Model	Global Batch Size	Forward/Backward Pass Size	Memory
ResNet-50	256	71.64GB	71.88GB
ResNet-101	256	107.43GB	107.74GB
ResNet-152	256	151.65GB	152.02GB
VGG-16	256	54.70GB	55.35GB
VGG-19	256	59.67GB	60.35GB

mini-batch. Finally, the all-reduce operation is launched asynchronously to synchronize all gradients of the partition. For example, as shown in Fig. 8, when the task that belongs to the last partition P4 (Group 1 and Group 2) is completed, then the all-reduce operation is started to synchronize the gradients of the partition P4 and the following tasks are executed at the same time. So, DP can utilize group-based delayed asynchronous communication during the backward pass, effectively reducing the ΔT latency and preventing network congestion caused by hardware bandwidth limitations.

IV. EXPERIMENT AND DISCUSSION

This section explores three key experimental areas: performance comparison, load balancing, and communication efficiency. To further evaluate GroPipe's effectiveness, a memory occupancy experiment assesses the partitioning algorithm, while an ablation experiment examines the impact of removing the DP component by comparing throughput results. Due to space constraints, the details of these experiments have been relocated to Sections II and IV of the supplementary file, available online.

A. Experiment Setup

In this section, the accuracy-to-time of GroPipe is proposed in this paper, and DP [19], Torchgipe [29], Dapple [27], and DeepSpeed [28] are compared for ResNet-50, ResNet-101, ResNet-152, VGG-16, and VGG-19, respectively, using the ImageNet dataset, with a global batch size of 256. DP and Torchgipe are well-established methods, widely used across various model architectures and tasks, providing strong baselines due to their simplicity and effectiveness in distributed training. To converge to the expected results faster, the experiments use the same cosine learning rate schedule for GroPipe, DP, and Torchgipe training. The initial learning rate starts at 0.1 for the ResNet series and 0.01 for the VGG series, followed by a cosine drop adjustment strategy before 32 epochs and a linear drop adjustment strategy after 32 epochs, reducing the learning rate to 0.0001.

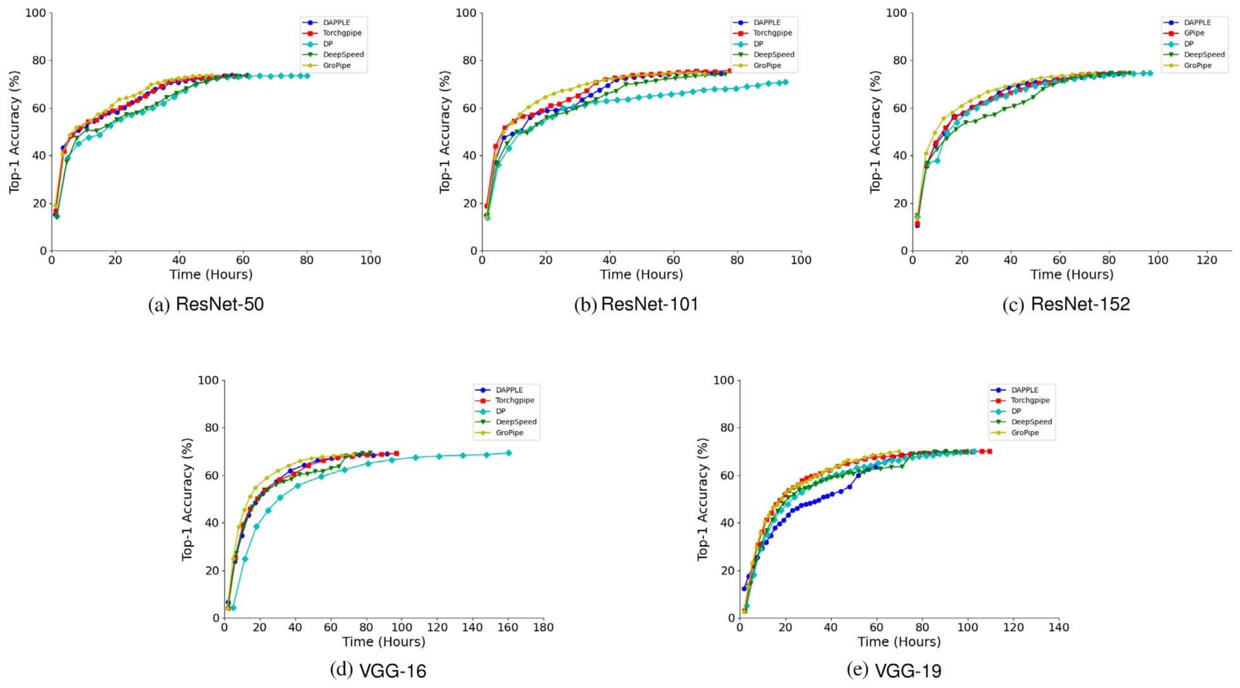


Fig. 9. Time-accuracy comparison of GroPipe with 4 other methods across 5 different models.

B. Performance Evaluations

In this section, the accuracy-to-time of GroPipe is proposed in this paper, DP [19], Torchgipe [29], Dapple [27], and DeepSpeed [28] are compared for ResNet-50, ResNet-101, ResNet-152, VGG-16, and VGG-19, respectively, using the ImageNet dataset, with a global batch size = 256. To converge to the expected results faster, the experiments use the same cosine learning rate schedule for GroPipe, DP, and Torchgipe training. The initial learning rate starts from 0.1 for the ResNet series and 0.01 for the VGG series, respectively, followed by a cosine drop adjustment strategy before 32 epochs and a linear drop adjustment strategy after 32 epochs to reduce the learning rate to 0.0001.

For the configuration of GroPipe in the experiment, we selected the setup with N = 4 and K = 2, where N represents the number of data-parallel groups and K represents the pipeline depth within each group. This configuration was chosen to optimally balance memory efficiency, parallelism, and communication overhead, ensuring efficient resource utilization across the GPUs.

As illustrated in Fig. 9, the accuracy after every two epochs is plotted. At the beginning of the training, the learning rate is larger, so the accuracy rises quickly. Then, as the learning rate decreases, the accuracy increases more and more slowly and finally stabilizes. In addition, GroPipe, Dapple, DeepSpeed, and Torchgpipe complete each corresponding epoch significantly faster than DP during the entire training process. It clearly shows that the high communication overhead of gradient synchronization of DP leads to a long training time per iteration. Moreover, GroPipe is slightly faster than Dapple, DeepSpeed, and Torchgpipe during each epoch. One explainable reason is that the group-based delayed asynchronous communication causes GroPipe to have lower GPU communication overhead than Torchgipe. Fig. 9 shows the comparison of the time to achieve the same Top-1 accuracy by the five approaches of GroPipe, Dapple, DeepSpeed, Torchgipe, and DP on the ResNet-50, ResNet-101, ResNet-152, VGG-16, and VGG-19. Therefore, the GroPipe can achieve almost the same convergence accuracy and faster convergence speed than Torchgipe and DP.

TABLE V
TRAINING TIME AND PERFORMANCE IMPROVEMENT OF DIFFERENT METHODS ACROSS VARIOUS MODELS USING THE IMAGENET DATASET

	GroPipe	DP	DAPPLE	Torchgipe	DeepSpeed	Performance Improvement
ResNet50	50.154	80.045	61.226	57.326	60.892	14.3% $\sim$ 59.6%
ResNet101	69.735	94.959	74.796	77.769	76.084	7.3% $\sim$ 36.2%
ResNet152	73.965	96.746	83.639	85.947	88.356	13.1% $\sim$ 30.8%
VGG16	78.784	166.392	91.717	103.128	80.273	1.9% $\sim$ 111.2%
VGG19	65.689	96.695	90.411	103.067	99.459	37.6% $\sim$ 56.9%

Based on the time-accuracy comparison experiments, the performance of GroPipe compared to Dapple, DeepSpeed, Torchgpipe, and DP can be obtained. As depicted in Table V, it can be observed significant performance improvements with GroPipe. When the optimal value of k is chosen for different models based on the subsequent experiment, GroPipe outperforms DP by 59.6%, 36.2%, 30.8%, 111.2%, and 47.2% on the ResNet-50, ResNet-101, ResNet-152, VGG-16, and VGG-19 models, respectively. In both the ResNet and VGG series, the performance has shown an average improvement of 42.2% and 79.2%, respectively.

Furthermore, comparing with Torchgipe, it can be found that GroPipe achieves a speedup performance improvement of 14.3%, 11.5%, 16.2%, 30.9%, and 56.9% on the ResNet-50, ResNet-101, ResNet-152, VGG-16, and VGG-19 models, respectively. The average performance improvement is 14.0% in the ResNet series

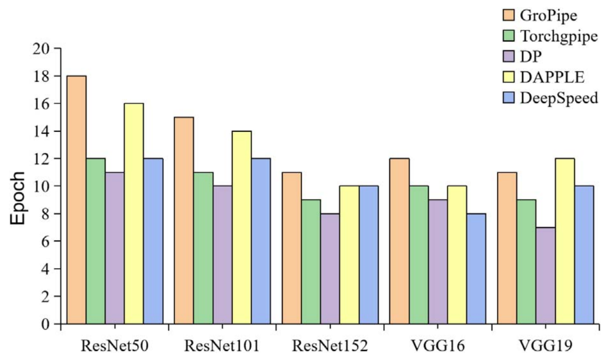


Fig. 10. Epoch completion within a 20-hour timeframe on ImageNet.

and 43.9% in the VGG series. Similarly, when compared to DAPPLE, GroPipe demonstrated improvements of 22.1%, 7.3%, 13.1%, 16.4%, and 37.6% on the same five models, with an average speedup of 14.2% for the ResNet series and 27.0% for the VGG series. Additionally, against DeepSpeed, GroPipe showed significant enhancements, achieving speedups of 21.4%, 9.1%, 19.5%, 1.9%, and 51.4%, with average gains of 16.7% for the ResNet series and 26.7% for the VGG series. These substantial performance gains highlight GroPipe's effectiveness in improving the efficiency of deep learning model training.

An interesting observation is that GroPipe achieves a greater speedup over Torchgipe on the VGG model compared to ResNet. This difference can be explained by the architectural characteristics of the two models, which directly influence their time to accuracy, the key performance metric in our experiments. Time to accuracy measures the duration required for a model to reach a target accuracy during training, and it is highly sensitive to both model complexity and convergence speed. VGG, with its deeper architecture and numerous convolutional and fully connected layers, has a higher computational burden and slower convergence, making it more prone to the vanishing gradient problem due to the absence of residual connections. This results in a longer time to accuracy for VGG. GroPipe, with its hybrid of data and pipeline parallelism, significantly reduces this extended training time, leading to a much larger performance improvement for VGG. In contrast, simpler pipeline parallelism is less effective for such a deep and computationally heavy model. On the other hand, ResNet, which features residual blocks, benefits from faster convergence and fewer parameters, resulting in a shorter time to accuracy. Since ResNet already achieves quicker convergence due to its architecture, the relative impact of GroPipe's optimizations is smaller. As a result, while GroPipe still provides a speedup for ResNet, the improvement is less dramatic than that seen with VGG. In summary, the experimental results highlight that GroPipe delivers more significant speedup for VGG compared to ResNet, validating the effectiveness of GroPipe's optimizations, particularly for models with higher computational demands like VGG.

In addition, the paper also tests the number of training epochs achieved by GroPipe, Torchgipe, and DP when training the ImageNet dataset within a 20-hour timeframe. As shown in Fig. 10, the experimental results indicate that GroPipe achieves a higher number of training epochs within the 20-hour duration. This could be attributed to the advantages of GroPipe's parallel computation and memory allocation mechanisms, enabling more frequent updates of model parameters within the given time frame.

In continuation of the previous experiments, this paper examines the influence of different micro-batch sizes on the training rate and the subsequent impact on training speed. Specifically, the performance of GroPipe is evaluated by measuring the training time for one epoch using various micro-batch sizes on the mini-ImageNet dataset. The observed results, as illustrated in Fig. 11, indicate that GroPipe achieves higher training efficiency when the micro-batch size ranges from 8 to 12.

By comparing the training time for different micro-batch sizes, it can be observed the effect of micro-batch size on the overall training efficiency of GroPipe. When a mini-batch is divided into more micro-batches, the computational workload per micro-batch decreases, allowing them to be processed more quickly in the pipeline. This enables different stages of the pipeline to receive the next micro-batch faster, reducing idle time and improving parallel efficiency.

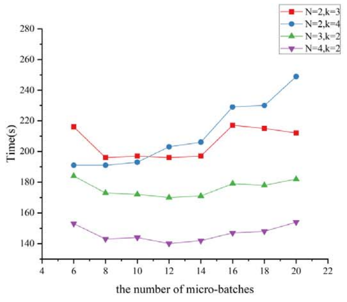
However, dividing a mini-batch into too many micro-batches can also have some negative effects. Firstly, micro-batch partitioning introduces additional overhead, such as the time for partitioning and merging micro-batches, as well as communication overhead. Secondly, excessive micro-batches may lead to workload imbalance among stages in the pipeline, causing certain stages to take longer to compute and limiting overall parallel efficiency improvement.

As shown in Fig. 11, it also can be seen that these five models bring more benefits from N than from k for the same number of GPUs. For example the optimal throughput for $N = 4$, $k = 2$ is higher than the optimal throughput for $N = 2$, $k = 4$. This is because PMP tends to reach a performance bottleneck when scaling across multiple GPUs.

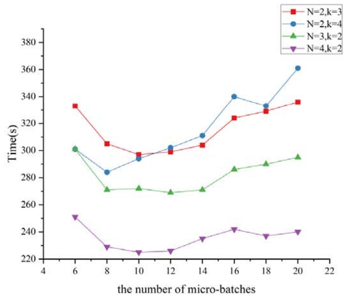
Therefore, in practical applications, it is necessary to choose the number of micro-batches based on the specific task and computational resources, to take the trade-off between parallel efficiency and additional overhead.

C. Evaluation of Load Balancing in PMP

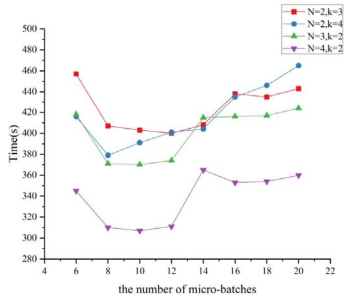
Throughput experiments are conducted on ResNet-50, ResNet-101, ResNet-152, VGG-16, and VGG-19, using various grouping configurations and batch sizes in this section. These experiments aim to find the highest throughput under the corresponding grouping configuration. The throughput serves as an indicator of pipeline load balancing, with a higher throughput indicating a more balanced model load. As shown in Tables VI and VII, N represents the number of groups, k represents the number of stages in each group, and m represents the number of micro-batches. The optimal throughput of the model is shown in the bolded data in Table VI. The experimental results are analyzed from the vertical and horizontal dimensions. From the vertical dimension, for the same models with different grouping configurations, the throughput first increases and then decreases with m increasing. As shown in Fig. 12, the peaks are



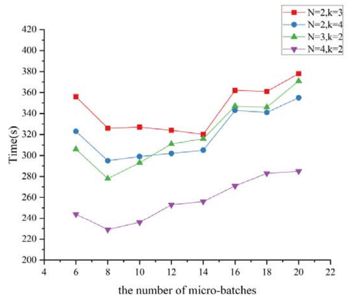
(a) ResNet-50



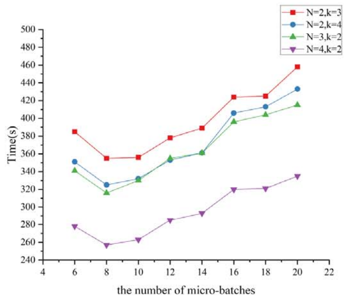
(b) ResNet-101



(c) ResNet-152



(d) VGG-16



(e) VGG-19

Fig. 11. GroPipe training time for one epoch across different models using Mini-ImageNet with various configurations.

TABLE VI
THROUGHPUT RESULTS OF GROPIPE ACROSS DIFFERENT CONFIGURATIONS ON THE RESNET SERIES

	ResNet-50					ResNet-101					ResNet-152				
	N = 2			N = 3	N = 4	N = 2			N = 3	N = 4	N = 2			N = 3	N = 4
m	k = 2	k = 3	k = 4	k = 2	k = 2	k = 2	k = 3	k = 4	k = 2	k = 2	k = 2	k = 3	k = 4	k = 2	k = 2
1	161.332	155.808	156.566	230.394	290.740	99.664	98.284	95.106	140.985	174.504	70.340	69.496	67.134	99.312	122.744
2	198.494	216.618	230.784	287.625	357.504	127.494	139.496	145.980	177.390	217.808	89.038	99.808	104.246	125.364	153.364
4	222.374	249.136	293.256	310.623	385.288	131.138	155.748	173.344	181.146	223.316	90.612	109.820	121.046	126.654	154.836
6	226.146	262.460	**313.516**	315.303	391.504	134.852	165.074	184.316	184.275	226.644	92.750	116.988	129.446	128.778	158.652
8	243.632	292.414	312.962	337.083	417.128	**152.364**	181.888	**196.510**	207.720	251.404	**106.406**	132.720	**143.424**	146.568	178.624
10	247.632	291.054	309.290	339.243	414.160	148.098	**187.014**	189.336	206.298	**256.696**	105.642	134.046	138.406	**147.123**	**180.448**
12	**248.836**	**293.194**	295.000	**344.985**	**425.860**	150.618	186.148	183.610	**208.944**	255.668	104.160	**135.134**	134.684	145.431	178.152
14	246.758	292.530	290.450	339.699	419.952	148.614	182.560	178.100	207.108	243.544	96.074	132.168	133.806	129.702	149.348
16	235.824	261.472	261.220	325.056	406.764	141.556	169.836	161.368	195.279	236.292	95.656	122.612	123.528	129.741	154.840

TABLE VII
THROUGHPUT RESULTS OF GROPIPE ACROSS DIFFERENT CONFIGURATIONS ON THE VGG SERIES

	VGG-16					VGG-19				
	N = 2			N = 3	N = 4	N = 2			N = 3	N = 4
m	k = 2	k = 3	k = 4	k = 2	k = 2	k = 2	k = 3	k = 4	k = 2	k = 2
1	96.052	97.1	101.168	139.257	176.396	86.732	86.768	88.344	121.146	157.404
2	127.6	135.162	147.102	184.353	230.14	116.202	123.814	130.858	160.383	203.584
4	137.48	157.232	173.044	195.954	246.38	120.232	140.82	156.1	174.93	219.516
6	130.776	153.202	170.894	181.227	233.696	116.45	140.894	155.706	161.019	201.628
8	**142.246**	168.442	**188.886**	**201.726**	**251.804**	**128.462**	**153.852**	**169.65**	**174.96**	**220.172**
10	133.746	168.314	186.052	190.137	243.196	116.948	153.246	165.762	166.503	214.416
12	122.922	**170.036**	183.79	177.756	224.676	108	143.756	154.94	154.026	195.94
14	121.06	166.108	181.546	174.726	221.152	104.082	139.442	151.188	151.209	189.812
16	108.762	150.74	159.796	157.842	201.076	98.414	126.746	132.876	136.548	172.428

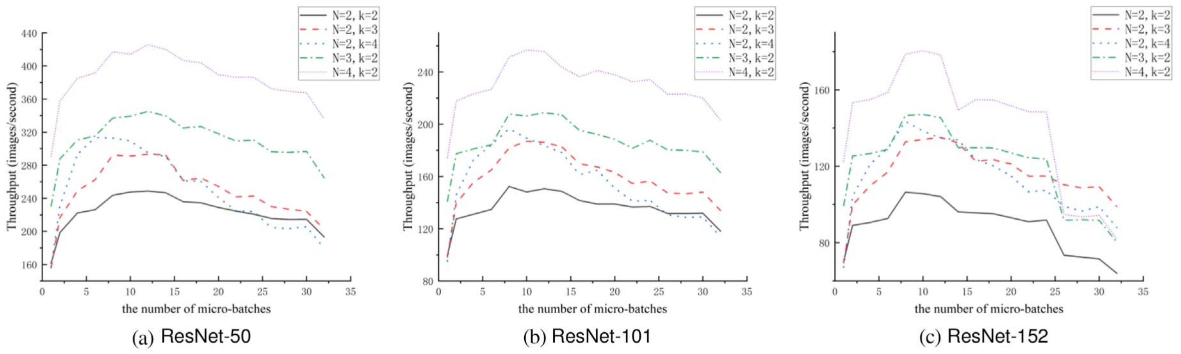
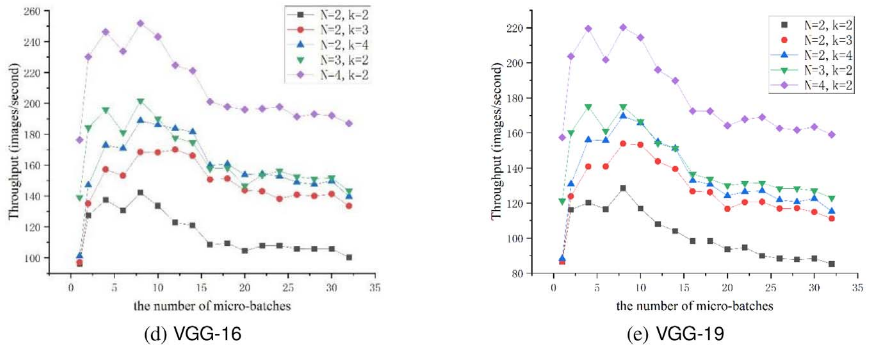



Fig. 12. GroPipe throughput across different models with various configurations.

concentrated between $m = 6$ and $m = 12$. Before the throughput rate reaches its peak value, the idle time in the pipeline gradually decreases and the throughput shows an upward trend with the increase of m. Throughput reaches its maximum value when idle time and communication overhead are balanced. As m continues to increase, the throughput begins to decrease gradually, because the benefits of DP are not enough to offset the communication overhead. From the horizontal dimension, for different models with the same grouping configurations, the throughput decreases gradually as the number of model parameters increases, this is because communication overhead becomes a bottleneck that affects the training speed.

The throughput is influenced not only by load balancing but also by the number of micro-batches. From Fig. 12 and Table VI, it is evident that for the same configuration of the model (with k and N held constant), the training speed first increases and then decreases as the number of batches increases. Fig. 12 shows that when m is small, due to the limited PMP algorithm, the throughput is low. At this time, only the computation operation from one device can be performed at any specific time, which leads to an under-utilization of GPU. Then, as m increases, the throughput grows accordingly, but the growth rate also gets lower and lower. Finally, after the peak throughput is reached, the throughput becomes lower and lower as m increases, because the communication overhead within the group becomes larger and larger, and gradually becomes a performance bottleneck that dominates the training speed. At this time, the benefits brought by PMP are not enough to offset the communication cost.

To further validate the effectiveness of the partitioning algorithm, a memory occupancy experiment is conducted based on the throughput experiments. A balanced memory occupancy

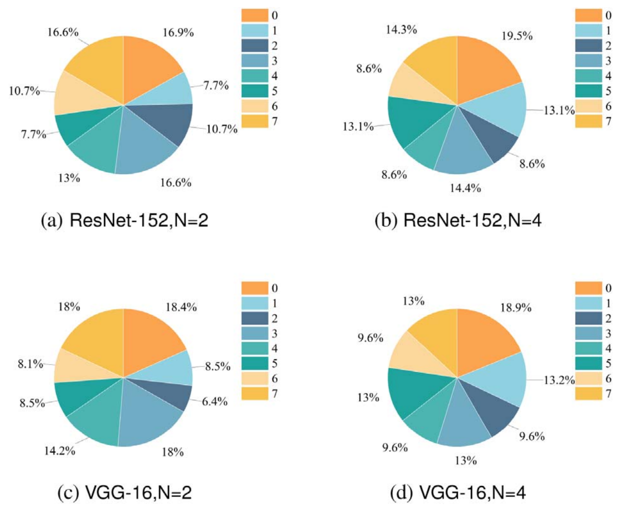


Fig. 13. GroPipe GPU memory occupancy on ResNet-152 and VGG-16 models with different configurations.

indicates that the partitioning algorithm successfully achieves load balancing.

The experiment focuses on examining the memory usage of GroPipe under the ResNet and VGG series. GroPipe divides the model into N groups, with PMP running among the groups. Throughout the execution process, the memory usage of each GPU is monitored and recorded. As shown in Fig. 13, the experimental results shows that under different partitioning scenarios, a relatively balanced distribution of GPU memory usage is observed, except for the first GPU. This disparity in memory usage arises because the first GPU is responsible for processing the data sent by other GPUs, necessitating additional memory to

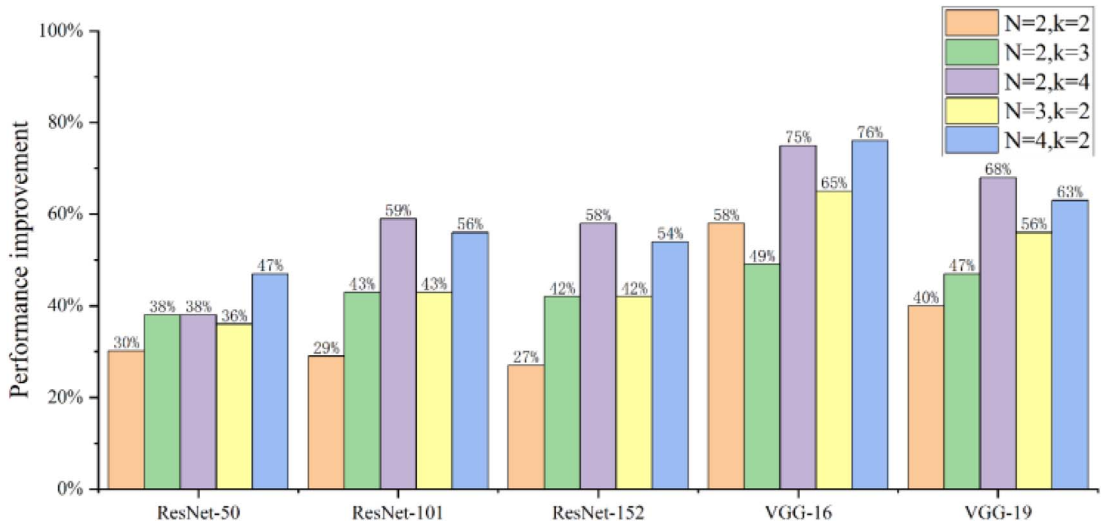


Fig. 14. Speedup performance improvement with different configurations.

store and handle this extra data. Consequently, the first GPU tends to exhibit higher memory occupancy compared to the other GPUs. This balanced distribution indicates efficient memory utilization and helps prevent memory bottlenecks during model training.

By analyzing these factors and interpreting the data obtained from the experiments, valuable insights can be gained into the memory utilization patterns of GroPipe and make informed decisions regarding resource allocation and memory management for optimal performance.

D. Evaluation of the Group-Based Delayed Asynchronous Communication Strategy

In this section, the group-based delayed asynchronous communication among multiple groups with mini-ImageNet dataset is evaluated in detail combined with PMP within a group, on ResNet-50, ResNet-101, ResNet-152, VGG-16, and VGG-19. Here, the m is set to the optimal value, which means that the specific m makes the highest throughput. Then, how the group-based delayed asynchronous communication affects the speedup is tested over different models with different configurations. As illustrated in Fig. 14, it can be seen that the speedup performance of GroPipe improves by 76% (the maximum value) when using VGG-16 with the configuration of $N = 4$ and $k = 2$. When N remains constant, the speedup increases with increasing k. This is mainly because the total amount of gradient communication for each partition in all the groups is smaller during the backward pass. As shown in Fig. 8, it can be theoretically inferred that the communication overhead for each all-reduce is lower and the iteration speed is faster. Moreover, ResNet-152 has almost similar speedup to ResNet-101 in the same configuration, and the speedup of ResNet-50 is slightly lower than ResNet-101 and ResNet-152 in the same configuration, except for $N = 2$, $k = 2$. The explanation for the above appearance is that ResNet-101 and ResNet-152 have more parameters than ResNet-50, which can better play the effect of group-based delayed asynchronous communication. Finally, it can be found

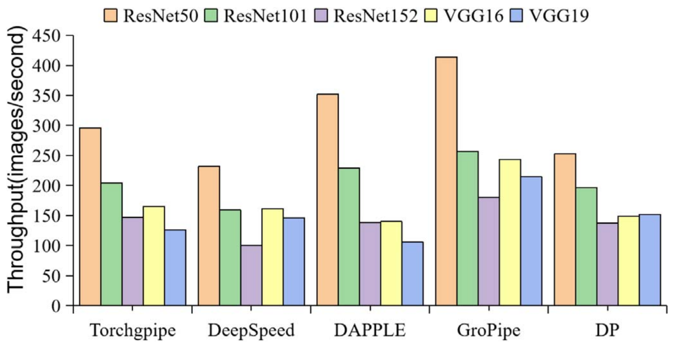


Fig. 15. Throughput comparison of five methods across five models.

that the acceleration performance of the VGG series is more significant than that of the ResNet series under the same configuration. The main reason is that the ratio of the parameter amount to the calculation amount of the two models of the VGG series is larger than that of the ResNet series, and the communication of parameters is undoubtedly much larger than the computational time overhead of the model. As a result, the acceleration effect of the VGG series is more obvious. This indicates that the group-based delayed asynchronous communication strategy has a significant acceleration effect and promising application prospects in hybrid parallelism.

We also evaluated the throughput of GroPipe against SOTA methods across five models, as shown in Fig. 15. The results demonstrate that GroPipe consistently outperforms other methods, achieving a performance lead of 12.1% to 79.3% on the ResNet series and 41.2% to 102.3% on the VGG series.

Memory and Computation Balance: GroPipe smartly partitions layers based on both execution time and memory usage, mitigating load imbalance issues that are common in other methods, thereby improving overall performance.

E. Language Model Experiment

In this section, we conducted a comparison of five methods using the BERT-base language model, a widely recognized architecture in natural language processing. GroPipe performs exceptionally well, achieving up to a 51% performance improvement. Due to space limitations, the detailed experimental results are moved to Section III of the supplementary file, available online.

V. Conclusion and Future Work

This paper introduces GroPipe, a novel grouped pipeline hybrid parallel training architecture that combines the strengths of PMP and DP. By achieving pipeline load balancing and minimizing communication overhead, GroPipe accelerates DCNN training. It utilizes AMPA for model partitioning and applies the performance projection method to ensure quantitative load balancing. Additionally, GroPipe incorporates group-based delayed asynchronous communication among the groups. Experimental results show that GroPipe outperforms DP and Torchgpipe, with an average acceleration of 42.2% and 14.0% on the ResNet series and 79.2% and 43.9% on the VGG series, respectively, when reaching the same Top-1 accuracy. Future improvements may focus on reducing GPU memory consumption through techniques like model compression, gradient aggregation, and memory-efficient algorithms, which will further optimize GroPipe's memory utilization. We are eager to engage in academic discussions and share insights regarding the implementation and experimental results. Interested researchers are encouraged to contact the authors for further details, and we look forward to future collaborations and knowledge exchange.

Acknowledgment

The authors express their sincere gratitude to the anonymous reviewers and editors for their invaluable insights and constructive comments, which have significantly contributed to improving the quality of this manuscript.

References

[1] Y. Xu and H. Zhang, "Convergence of deep convolutional neural networks," *Neural Netw.*, vol. 153, pp. 553–563, Sep. 2022.
[2] A.-A. Tulbure, A.-A. Tulbure, and E.-H. Dulf, "A review on modern defect detection models using DCNNs–Deep convolutional neural networks," *J. Adv. Res.*, vol. 35, pp. 33–48, Jan. 2022.
[3] H. Zhao, H. Du, S. Yang, and F. Yao, "Rec-RN: User representations learning over the knowledge graph for recommendation systems," in *Proc. 4th Int. Conf. Mach. Learn., Big Data Bus. Intell.*, 2022, pp. 228–233.
[4] K. S. Chahal, M. S. Grover, K. Dey, and R. R. Shah, "A Hitchhiker's guide on distributed training of deep neural networks," *J. Parallel Distrib. Comput.*, vol. 137, pp. 65–76, Mar. 2020.
[5] S. Ouyang, D. Dong, Y. Xu, and L. Xiao, "Communication optimization strategies for distributed deep neural network training: A survey," *J. Parallel Distrib. Comput.*, vol. 149, pp. 52–65, 2021.
[6] T. Nakamura, S. Saito, K. Fujimoto, M. Kaneko, and A. Shiraga, "Spatial-and time-division multiplexing in CNN accelerator," *Parallel Comput.*, vol. 111, 2022, Art. no. 102922.
[7] U. U. Hafeez, X. Sun, A. Gandhi, and Z. Liu, "Towards optimal placement and scheduling of DNN operations with Pesto," in *Proc. 22nd Int. Middleware Conf.*, 2021, pp. 39–51.
[8] Y. Huang et al., "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 103–112.
[9] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," 2014, *arXiv:1404.5997*.
[10] D. Narayanan et al., "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 1–15.
[11] F. Qararyah, M. Wahib, D. Dikbayır, M. E. Belviranli, and D. Unat, "A computational-graph partitioning method for training memory-constrained DNNs," *Parallel Comput.*, vol. 104–105, 2021, Art. no. 102792.
[12] S. Li and T. Hoefler, "Chimera: Efficiently training large-scale neural networks with bidirectional pipelines," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–14.
[13] X. Shi, X. Peng, L. He, Y. Zhao, and H. Jin, "Waterwave: A GPU memory flow engine for concurrent DNN training," *IEEE Trans. Comput.*, vol. 72, no. 10, pp. 2938–2950, Oct. 2023.
[14] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Accurate, efficient and scalable training of graph neural networks," *J. Parallel Distrib. Comput.*, vol. 147, pp. 166–183, Jan. 2021.
[15] Y. Xu et al., "SSD-SGD: Communication sparsification for distributed deep learning training," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 1, pp. 1–25, 2022.
[16] L. Wang et al., "FFT-based gradient sparsification for the distributed training of deep neural networks," in *Proc. 29th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2020, pp. 113–124.
[17] P. Goyal et al., "Accurate, large minibatch SGD: Training Imagenet in 1 hour," 2017, *arXiv:1706.02677*.
[18] C. Yang et al., "PerfEstimator: A generic and extensible performance estimator for data parallel DNN training," in *Proc. IEEE/ACM Int. Workshop Cloud Intell.*, 2021, pp. 13–18.
[19] S. Li et al., "Pytorch distributed: Experiences on accelerating data parallel training," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3005–3018, 2020.
[20] M. Jang, J. Kim, H. Nam, and S. Kim, "Zero and narrow-width value-aware compression for quantized convolutional neural networks," *IEEE Trans. Comput.*, vol. 73, no. 1, pp. 249–262, Jan. 2024.
[21] Z. Cai et al., "TensorOpt: Exploring the tradeoffs in distributed DNN training with auto-parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 8, pp. 1967–1981, Aug. 2022.
[22] S. Zhao et al., "vPipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel DNN training," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 3, pp. 489–506, Mar. 2022.
[23] Z. Lai et al., "Merak: An efficient distributed DNN training framework With automated 3D parallelism for giant foundation models," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 5, pp. 1466–1478, May 2023.
[24] X. Ye et al., "Hippie: A data-paralleled pipeline approach to improve memory-efficiency and scalability for large DNN training," in *Proc. 50th Int. Conf. Parallel Process.*, 2021, pp. 1–10.
[25] J. H. Park et al., "HetPipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2020, pp. 307–321.
[26] L. Zheng et al., "Alpa: Automating inter-and intra-operator parallelism for distributed deep learning," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 559–578.
[27] S. Fan et al., "DAPPLE: A pipelined data parallel approach for training large models," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 431–445.
[28] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discovery & Data Mining*, 2020, pp. 3505–3506.
[29] C. Kim et al., "Torchgpipe: On-the-fly pipeline parallelism for training giant models," 2020, *arXiv:2004.09910*.

Bin Liu was born in Shaanxi, in 1981. He received the Ph.D. degree in electronic and information engineering from Xi'an Jiaotong University, China, in 2014. Since 2024, he has been a Professor with the College of Information Engineering, Northwest A&F University, China, where he has received a Postdoctoral Fellow with the College of Mechanical and Electronic Engineering. His research interests include parallel computing and deep learning. He currently serves as a Reviewer for IEEE Transactions on Computers, *The Journal of Supercomputing*, and *Future Generation Computer Systems*, among other journals.

Yongyao Ma was born in Henan. He received the B.S. degree in information security from China University of Geosciences, Wuhan, in 2020. He is currently pursuing a master's degree in computer science and technology with the Northwest A&F University, China. His research interests include parallel computing, computer vision, and distributed computing.

Zijian Hu was born in Jiangxi. He received the B.S. degree in software engineering from the East China University of Technology, in 2021. He is currently pursuing a master's degree in electronic information with the Northwest A&F University, China. His research interests include parallel computing, computer vision, and distributed computing.

Zeyu Ji received the B.S. degree from the School of Information Engineering, Zhengzhou University, the M.S. degree from the Polytechnic University of Tours, France, and the Ph.D. degree from Xi'an Jiaotong University. Currently, he is an Assistant Professor with the College of Information Engineering, Northwest A&F University, Yangling, China. His research interests include computer architecture, high-performance computing, and deep learning.

Zhenli He (Senior Member, IEEE) received the M.S. degree in software engineering and the Ph.D. degree in systems analysis and integration from Yunnan University (YNU), Kunming, China, in 2012 and 2015, respectively. He was a Postdoctoral Researcher with Hunan University (HNU), Changsha, China, from 2019 to 2021. Currently, he is an Associate Professor and the Head of the Department of Software Engineering with the School of Software, Yunnan University, Kunming, China. His research interests include edge computing, energy-efficient computing, heterogeneous computing, and edge AI techniques.

Keqin Li (Fellow, IEEE) received the B.S. degree in computer science from Tsinghua University, in 1985, and the Ph.D. degree in computer science from the University of Houston, in 1990. He is a SUNY Distinguished Professor with the State University of New York and a National Distinguished Professor with Hunan University, China. He has authored or co-authored more than 1110 journal articles, book chapters, and refereed conference papers. He holds over 75 patents announced or authorized by the Chinese National Intellectual Property Administration. Since 2020, he has been among the world's top few most influential scientists in parallel and distributed computing regarding single-year impact (ranked #2) and career-long impact (ranked #4) based on a composite indicator of the Scopus citation database. He is listed in Scilit Top Cited Scholars (2023–2024) and is among the top 0.02% out of over 20 million scholars worldwide based on top-cited publications. He is listed in ScholarGPS Highly Ranked Scholars (2022–2024) and is among the top 0.002% out of over 30 million scholars worldwide based on a composite score of three ranking metrics for research productivity, impact, and quality in the recent five years. He received the IEEE TCCLD Research Impact Award from the IEEE CS Technical Committee on Cloud Computing in 2022 and the IEEE TCSVC Research Innovation Award from the IEEE CS Technical Community on Services Computing in 2023. He won the IEEE Region 1 Technological Innovation Award (Academic) in 2023. He was a recipient of the 2022–2023 International Science and Technology Cooperation Award and the 2023 Xiaoxiang Friendship Award of Hunan Province, China. He is a member of the SUNY Distinguished Academy. He is an AAAS Fellow, an AAIA Fellow, an ACIS Fellow, and an AIIA Fellow. He is a member of the European Academy of Sciences and Arts. He is a member of Academia Europaea (Academician of the Academy of Europe).