LIBPipe: Efficient Load Imbalance Pipeline Model Parallelism for Large Models Training

Bin Liu, *Member, IEEE*, Zhonghao Zhang, Hengzhao Li, Zeyu Ji, Hongming Zhang, *Member, IEEE*, and Keqin Li, *Fellow, IEEE*

***Abstract*—With the increasing size of datasets and the expansion of Deep Neural Networks (DNNs), the training process has become exceedingly time-consuming. Distributed training, specifically the Pipeline Model Parallelism (PMP) method, commonly mitigates this problem but suffers from bubble time delays. This paper proposes LIBPipe, a pipeline training framework that explicitly incorporates a load-imbalance method to reduce bubble time in PMP. Within LIBPipe, a model-unequal-partitioning method is designed from the perspective of load imbalance to reshape the pipeline execution pattern and significantly shorten idle periods during training. On top of this method, a performance-guided unequal-partitioning search algorithm is developed to efficiently identify near-optimal partitioning strategies under memory constraints. The paper theoretically proves the time efficiency of adopting load imbalance in pipeline models. Comprehensive experiments are conducted on an 8-GPU server to evaluate the efficiency of LIBPipe, using the IMDB and mini-ImageNet datasets as well as well-known models such as BERT and ResNet. The BERT-series models achieve a maximum throughput improvement of 60.3%, while the ResNet-series models achieve a maximum improvement of 74.1%.**

***Index Terms*—Deep neural network, distributed training, pipeline model parallelism, load imbalance, model partition.**

I. Introduction

DEEP Neural Networks (DNNs) have been widely used in numerous fields, including agriculture, healthcare, and aviation [1], [2], [3], as deep learning continues. It is common to broadly use large-scale models [4], [5], [6] with vast datasets [7], [8], [9] to achieve higher performance and accuracy, which requires extensive training. However, the computational and storage demands of large models often exceed the capabilities of traditional single-machine systems.

To mitigate the challenges associated with training large models on extensive datasets, scholars have proposed various distributed training methods [10], [11], [12], [13] that utilize multi-GPU servers to accelerate model training. It mainly involves two well-known parallelization techniques: data parallelism (DP) [14], [15], [16], [17] and pipeline model parallelism (PMP) [12], [14], [18], [19], [20], [21], as shown in Fig. 1. DP distributes the dataset among multiple devices, and each device holds an entire copy of the model and performs training on its respective dataset segment. Gradients are computed independently on each device and then aggregated to update the model parameters. Although DP is broadly implemented, it encounters limitations when the model's size exceeds the memory capacity of individual devices.

To address the above limitations of DP when training large models, researchers increasingly leverage pipeline model parallelism [12], [22], [23], [24]. PMP partitions a deep neural network into several segments and assigns each segment to a different GPU. A training batch is further divided into multiple mini-batches, which enter the pipeline sequentially. Each mini-batch first performs forward propagation, during which activations are passed from the first GPU through the pipeline to the last GPU. Once the forward pass of that mini-batch is completed, its backward propagation begins and gradients flow in the reverse direction.

In PMP, the pipeline must first be filled before all stages can operate concurrently and must be drained before completing

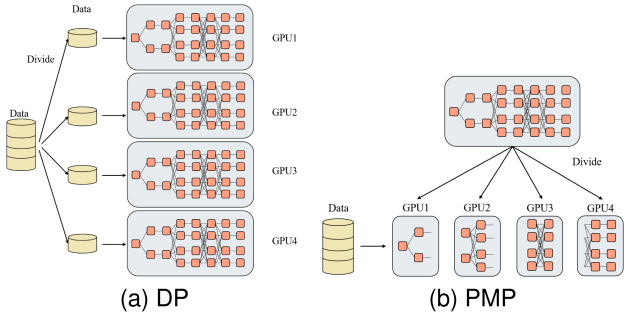

Fig. 1. Data parallelism and Pipeline model parallelism.

Received 22 October 2024; revised 24 November 2025; accepted 13 April 2026. Date of publication 22 April 2026; date of current version 14 May 2026. This research was supported in part by the National Natural Science Foundation of China under Grant 62376226, in part by the Shaanxi's Key Research and Development Program under Grant 2023-ZDLNY-63, in part by the Xianyang's Key Research and Development Program under Grant L2022-ZDYF-NY-019, and in part by the Natural Science Basic Research Program of Shaanxi Province under Grant 2022JQ-690. Recommended for acceptance by A. Anjum. *(Corresponding author: Zeyu Ji.)*

Bin Liu is with the College of Information Engineering, Northwest A&F University, Yangling 712100, China, and 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 Key Laboratory of Agricultural Information Perception and Intelligent Service, Northwest A&F University in Yangling, Shaanxi 712100, China (e-mail: liubin0929@nwafu.edu.cn).

Zhonghao Zhang, Hengzhao Li, and Zeyu Ji are with the College of Information Engineering, Northwest A&F University, Yangling 712100, China (e-mail: 2022051057@nwafu.edu.cn; lhz2023@nwafu.edu.cn; zeyu.ji@nwafu.edu.cn).

Hongming Zhang is with the Dean of the College of Information Engineering, Northwest A&F University, Yangling 712100, China (e-mail: zhm@nwsuaf.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/TPDS.2026.3686206, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2026.3686206

an iteration. The idle periods that occur during these filling and draining phases are collectively referred to as bubble time. Bubble time is a global metric that quantifies the total idle duration across all stages in a training iteration, and it directly limits pipeline utilization and overall throughput.

Beyond this global inefficiency, PMP also exhibits a characteristic inverted V-shaped bubble pattern within each mini-batch. Due to the forward–then–backward dependency, a mini-batch experiences two local idle regions: one after its forward pass completes and another before its backward pass begins. These two idle regions form an inverted V-shaped bubble structure on the execution timeline. While this pattern describes a local idle phenomenon, it contributes to the overall bubble time and further constrains performance.

While PMP is widely recognized for enabling the training of models significantly larger than the memory capacity of a single GPU and for improving throughput, its performance is fundamentally constrained by these bubble-induced idle times. This motivates the need for techniques that reduce bubble time and improve stage utilization.

Load balancing is employed to address this challenge, and extensive experiments have been conducted for this purpose. Our experimental results yield an unexpected discovery: model performance is significantly enhanced in specific scenarios by adopting load imbalance instead of traditional load balancing techniques. This finding prompts further exploration of the load imbalance technique, ultimately leading to the introduction of this innovative idea. However, it also presents two notable issues: the effectiveness of load imbalance is difficult to prove and results in a vast partitioning space, making it challenging to find an approximate optimal partitioning strategy.

This paper proposes LIBPipe, a pipeline training framework that incorporates a load imbalance method to significantly reduce idle time in PMP. The framework leverages the concept of load imbalance to improve training performance and efficiency. By redistributing workloads across stages, the method effectively utilizes computational resources, minimizes idle periods during training, and thereby accelerates the overall pipeline execution. To enable LIBPipe to efficiently identify an approximately optimal partitioning strategy, an unequal-partitioning search algorithm (GA-NNP) is designed, ensuring effective exploration of large partition spaces across different model configurations, even under memory constraints. This paper challenges the conventional assumption that load balancing is always superior to load imbalance, demonstrating that, under a properly designed load imbalance method, resource utilization can be further improved, offering a new solution for distributed large-scale model training. The main contributions of this work are summarized as follows:

- A novel load imbalance method is developed within the LIBPipe framework for PMP. Using a training-time evaluation model, it applies unequal model partitioning to minimize bubble time in pipeline execution.
- Mathematical derivations prove that adopting load imbalance in pipeline models can yield greater benefits than traditional even-load strategies, providing theoretical support for the effectiveness of load imbalance methods.
- An unequal-partitioning search algorithm based on performance evaluation is designed. LIBPipe abstracts the model into a computation graph and incorporates a time evaluation model to quantitatively estimate training time. The algorithm efficiently searches for approximately optimal unequal partitions within limited time.

The rest of this paper is organized as follows: Section II introduces the background and related work. Section III provides a detailed description of the LIBPipe framework and its load imbalance method. Section IV presents the experimental results to validate the efficiency of the proposed framework. Finally, Section V concludes the paper.

II. Related Work

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

PMP is an advanced distributed deep learning technique. It processes the model sequentially across multiple devices in distributed computing, helping to improve the efficiency of training large-scale models, commonly referred to as throughput. Fig. 2 illustrates some traditional classical PMP methods. Based on their focus, PMP can be broadly categorized into two types: performance optimization based on memory balance and training efficiency.

Performance optimization based on memory balance: These optimizations are designed to reduce the memory footprint of DNNs during training. By efficiently managing memory usage, it becomes possible to train larger models or to use more complex models on hardware with limited memory capacity. Building upon PipeDream, Narayanan et al. introduced a more innovative pipeline parallel training framework named PipeDream-2BW (Double-Buffered Weight Updates) [24]. It employs a One-Forward-One-Backward (1F1B) scheduling strategy and a weight double buffering mechanism to ensure high throughput while maintaining low memory consumption. Kim et al. proposed a novel method BPipe [25], for achieving memory balance in pipeline parallelism. BPipe employs an activation balancing method that transfers intermediate activations among GPUs during training, achieving a balance between memory usage and performance. Dreuning et al. devised mCAP [26], a system that employs a predictor to suggest optimal partitioning of a model for balancing peak memory usage across GPUs.

Performance optimization based on training efficiency: It aims to increase the speed and efficiency of training DNN models. It involves strategies like overlapping computation and communication, reducing bubble time among pipeline stages, and balancing the computational load across different devices. Huang et al. designed an extensible PMP framework based on the model, named GPipe [12]. GPipe employs a pipeline algorithm to synchronize gradients and divide mini-batches into finer-grained micro-batches, enhancing the model's parallelism. Narayanan et al. introduced the asynchronous pipeline parallelism framework PipeDream [18]. In PipeDream, each computation node can update the stage which is responsible

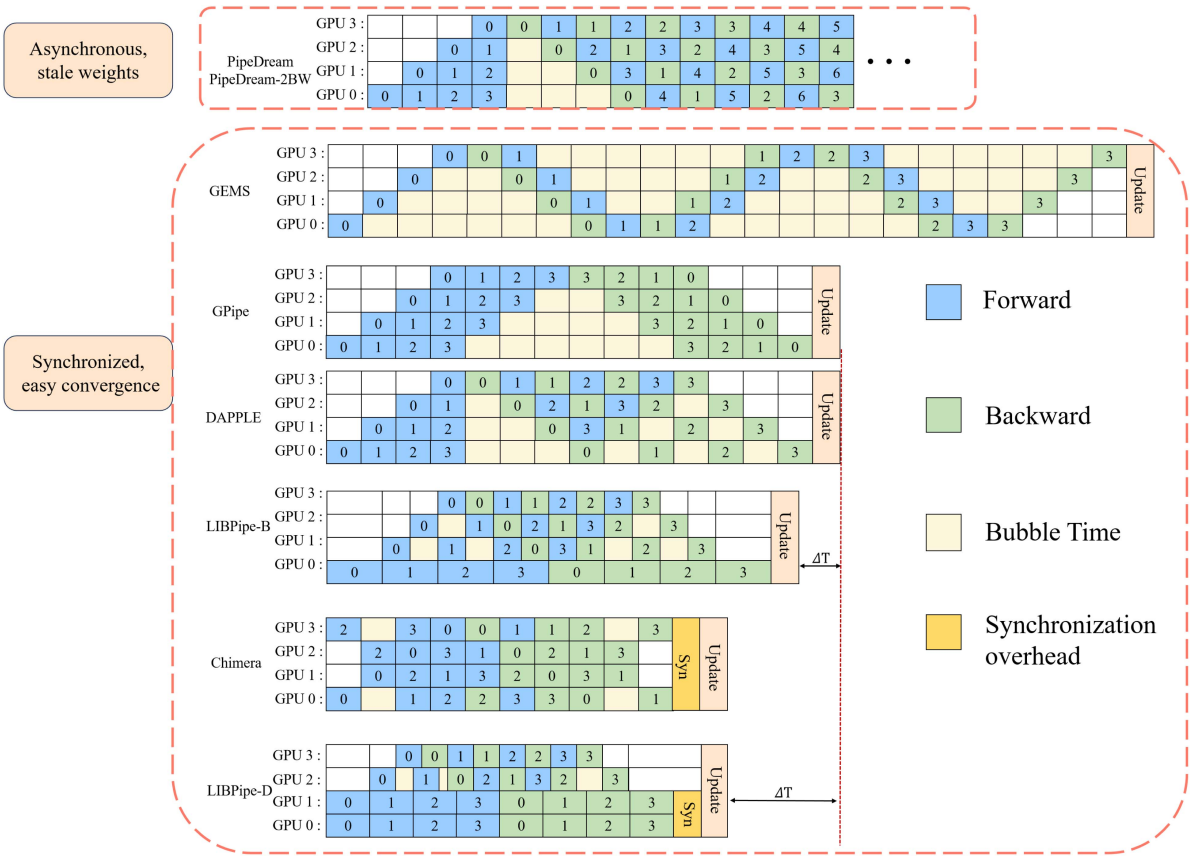

Fig. 2. Compared with other traditional pipeline methods, except for the asynchronous PipeDream and PipeDream-2BW, all other methods utilized 4 GPUs and 4 mini-batches.

for without necessarily waiting for other nodes. Li et al. proposed an innovative bidirectional PMP technique, Chimera [19]. Although it increases peak GPU memory usage, this method reduces idle time in PMP by 50%, significantly improving the training efficiency of large-scale models and offering a novel solution for PMP. Fan et al. designed a synchronous training framework called DAPPLE [27]. This framework utilizes a dynamic scheduling algorithm and adopts a 1F1B synchronous scheduling strategy to reduce GPU memory consumption while ensuring model convergence. It provides an efficient and convenient solution for distributed training. Zhao et al. developed BaPipe [28], a pipeline parallel framework that can automatically explore balanced partitioning strategies. Li et al. introduced TeraPipe[29], a pipeline parallelism algorithm for Transformer-based language models. This algorithm utilizes a novel dynamic programming-based approach to compute the optimal pipeline execution scheme based on specific model and cluster configurations. Zhang et al. put forth MixPipe[30], a novel synchronous bidirectional pipeline parallelism method. MixPipe achieves a better balance between pipeline and device utilization by initially flexibly regulating the number of micro-batches injected into bidirectional pipelines. Qi et al. presented a novel scheduling strategy [31] that splits the backward computation into two parts: one for calculating the gradient of the input and the other for calculating the gradient of the parameters, achieving zero pipeline bubbles in synchronous pipelines.

Existing research has made valuable contributions to the development of distributed training methods, primarily relying on the principle of load balancing. However,these methods often reach the theoretical limit of the load balancing strategy when dealing with more complex models and datasets, making it difficult to achieve better performance. In contrast, our work introduces intentional load imbalance in PMP. The key idea is not to enforce equal execution time across stages, but to exploit asymmetric partitioning to reshape the pipeline execution pattern and reduce global bubble time. To the best of our knowledge, this is the first study that treats load imbalance as a performance optimization resource, rather than a problem that must be minimized.

III. LIBPipe: Load Imbalance Pipeline Parallelism

This section presents the LIBPipe framework, a PMP training framework grounded in the principle of load imbalance. The framework addresses the primary challenge of high bubble time in PMP. The architecture of LIBPipe is illustrated in Fig. 3. In the planner phase, the model and hardware configuration are analyzed, and the results are then passed to a module that includes graph abstraction, training-time evaluation, and an unequal-partitioning search algorithm based on a customized genetic algorithm. This module automatically generates an approximately optimal partitioning strategy for DNNs. Finally, the

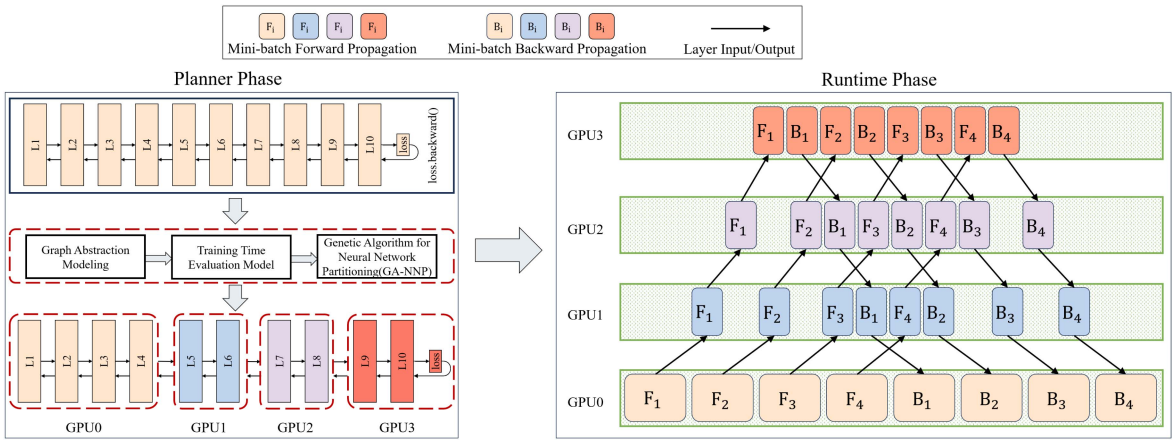

Fig. 3. The overall architecture of LIBPipe. LIBPipe first analyzes each layer of the model, then inputs the analysis results into a module consisting of Graph Abstraction Modeling, Training Time Evaluation Model, and Genetic Algorithm for Neural Network Partitioning, which is used to find the optimal partitioning strategy. Finally, the deployment and training are completed.

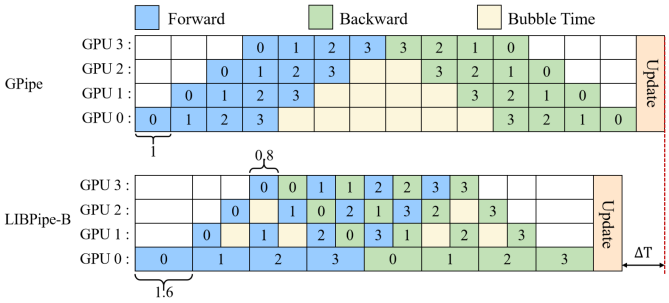

Fig. 4. Comparison of theoretical execution time between GPipe and LIBPipe-B under 4 GPUs and 4 mini-batches.

generated partitioning strategy is deployed and executed in the runtime phase.

A. Load Imbalance Strategy for Pipeline Training

A load imbalance strategy is designed to improve GPU utilization and reduce the bubble time generated in the existing PMP. The strategy divides the entire model unequally. As shown in Fig. 4, the first GPU is assigned a larger portion of the model than the remaining GPUs in LIBPipe-B. Assuming that the execution time of each forward/backward block in GPipe is one unit of time, the bubble time during the training process is 12 units. On the other hand, for LIBPipe-B, the execution time of the first forward block is 1.6 units, while the remaining blocks have an execution time of 0.8 units, resulting in a bubble time of 4.8 units. LIBPipe-B shows significantly better GPU utilization compared to GPipe.

The load imbalance strategy also adjusts the scheduling tasks for forward and backward propagation on different GPUs to reduce bubble time during training. As shown in Fig. 4, the first GPU must complete the forward propagation for all mini-batches before initiating backward propagation. Once the last GPU finishes forward propagation for a mini-batch, it immediately starts backward propagation. The closer the GPU is to the last one, the higher its priority for backward propagation. This prioritization enables earlier alternating forward and backward propagation, enhancing the efficiency of model training.

To find the most efficient unequal partitioning compared to GPipe, the load imbalance strategy aims to explore the minimum theoretical computation time for a pipeline and its corresponding approximate optimal partitioning scheme O. Therefore, as the core of load imbalance, the training time evaluation model is crucial to evaluate the pipeline execution time.

In cases of unequal partitioning, assuming the partition scenario P_i corresponds to the ith the pipeline can be divided into h parts, the overall execution time of the pipeline is represented as $T_i = \sum_{j=1}^{h} T_i^j$, where T_i^j represents the execution time of the jth part.

Since PMP has the property of overlapping computation times, this requires determining the computation time for each part that cannot be overlapped. Assuming that in jth part, at most k GPUs are running, their running times are $\{t_1^j, t_2^j, \ldots, t_k^j\}$, then $T_i^j = \max\{t_1^j, t_2^j, \ldots, t_k^j\}$. Combining the above conditions, LIBPipe-B can derive (1).

Assuming there are partitioning schemes $\{P_1, P_2, P_3, \ldots\}$, and their corresponding pipeline theoretical computation times are $\{T_1, T_2, T_3, \ldots\}$. Equation (2) can be derived. In Section III-C, this paper will develop a model unequal partitioning algorithm based on (1) and (2).

$$\boldsymbol{T}_i = \sum_{j=1}^{h} \max\{t_1^j, t_2^j, \ldots, t_k^j\} \quad (m \geq 1) \tag{1}$$

$$\boldsymbol{O} = \arg\min_{\mathrm{q}} \left\{ \sum_{j=1}^{h} \max\{t_1^j, t_2^j, \ldots, t_k^j\} | P_q \right\} \tag{2}$$

B. Proof of Time Effectiveness for Load Imbalance

Assume that the forward and backward propagation time for training one batch of training data in the whole model is T_F and T_B. The entire model is divided into k GPUs. The forward and backward propagation time on the first GPU is denoted as

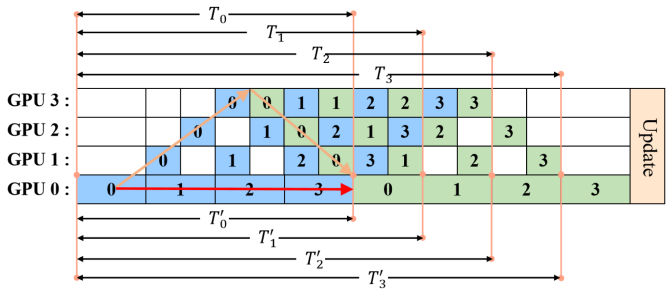

Fig. 5. T'_i is the execution time of the red arrow on GPU 0. T_i is the execution time of the orange arrows on all GPUs. When $T'_i \leq T_i$, there is no bubble time on GPU 0.

T_F^{first} and T_B^{first}, respectively. In the following derivation, the variable α represents the ratio of the first GPU's training time to the entire model's total training time. The definition of α is as shown in (3). A batch of training data is divided into m mini-batches. The total training time in the equal partitioned pipeline parallel training, T_{equ} is expressed in (4), while the total training time in the unequal partitioned pipeline parallel training, T_{LIB} is represented by (5). When (6) is satisfied, LIBPipe-B is optimized, which enhances GPU utilization and achieves a more efficient PMP.

$$\boldsymbol{\alpha} = \frac{T_F^{first} + T_B^{first}}{T_F + T_B} \tag{3}$$

$$\boldsymbol{T}_{equ} = \frac{(m+k-1) \times (T_F + T_B)}{k} \tag{4}$$

$$\boldsymbol{T}_{LIB} = m\alpha \times (T_F + T_B) \tag{5}$$

$$\boldsymbol{\alpha} < \frac{m+k-1}{mk} \tag{6}$$

As demonstrated in Fig. 5, T_i represents the time required for the ith mini-batch backward propagation task to complete its calculations before reaching the first GPU. T'_i represents the total time spent by the first GPU in performing calculations before executing the ith mini-batch backward propagation task. Intending to avoid excessive bubble time during the model training, the following conditions need to be met:

$$\boldsymbol{T}_i \leqslant T'_i \tag{7}$$

Through analysis of Fig. 5, the combined time of forward and backward propagation on the first GPU for each mini-batch can be denoted as $\frac{\alpha(T_F+T_B)}{m}$. Similarly, the combined time spent on forward-backward propagation tasks for each mini-batch across the remaining GPUs can be denoted as $\frac{(T_F+T_B)(1-\alpha)}{m(k-1)}$. T_i and T'_i can be represented by (8) and (9), respectively.

$$\boldsymbol{T}_i = \frac{(T_F + T_B) - \alpha T_B}{m} + (T_F + T_B) \times \frac{i(1-\alpha)}{m(k-1)} \tag{8}$$

$$\boldsymbol{T}'_i = \alpha T_F + \frac{i\alpha T_B}{m} \tag{9}$$

To satisfy (7), the range of α can be determined as follows:

$$\boldsymbol{\alpha} \geqslant \frac{(k+i-1)(T_F+T_B)}{(mk-m+i)T_F + (k+ki-1)T_B}, 0 \leqslant i \leqslant m-1 \tag{10}$$

Assuming that the right-hand side of the inequality in (10) is a function of i, it can be obtained the function $F(i)$ as follows:

$$\boldsymbol{F}(i) = \frac{(k+i-1)(T_F+T_B)}{(mk-m+i)T_F + (k+ki-1)T_B}, 0 \leqslant i \leqslant m-1 \tag{11}$$

It is generally required to satisfy $m \geqslant k \geqslant 2$ in PMP. Based on this constraint, it can be derived from (11) that $F(i+1) - F(i) \geq 0$. Therefore, it can be concluded that $F(i)$ is an increasing function within the range of i. By assigning the exponent variable i to $m-1$, the minimum value of the parameter α can be derived. From the preceding deductions, the domain over which the parameter α may be defined can be denoted as:

$$\frac{m+k-2}{mk-1} \leqslant \alpha < \frac{m+k-1}{mk} \tag{12}$$

In typical application scenarios, the values of m and k used to define the number of GPUs and mini-batches are usually required to satisfy the condition $m \geqslant k \geqslant 2$. Therefore, it can be derived that the value of $\frac{m+k-2}{mk-1}$ is always less than 1, which leads to the deduction of (13).

$$\frac{m+k-2}{mk-1} < \frac{m+k-1}{mk} \tag{13}$$

Therefore, under the condition of $m \geqslant k \geqslant 2$, a range of values exists for α such that LIBPipe-B achieves higher training efficiency compared to traditional PMP training.

Although the range of α can be determined based on (12), finding the approximate optimal partitioning strategy within this range is extremely time-consuming in practice. For example, deploying ResNet-152, which has 365 layers, on 8 GPUs using a brute force search algorithm based on recursion, the recursive depth can reach up to 7 layers, with an average of over 100 choices per layer. This complicates the distribution among the remaining GPUs. The more the number of GPUs and the deeper the model layers, the larger the recursive space. Therefore, this paper designs an automatic partitioning strategy, which will be explained in the following section.

C. Unequal Partitioning Algorithm Based on Performance Evaluation

LIBPipe-B employs an unequal partitioning algorithm to achieve load imbalance. This algorithm uses the training time evaluation model mentioned in Section III-A to assess the performance of different partitions. It combines graph abstraction modeling with a customized genetic algorithm for rapid searching. The specifics of graph abstraction modeling and the customized genetic algorithm will be further detailed in subsequent sections.

1) Graph Abstraction Modeling: Due to the unique nature of DNNs, LIBPipe-B can abstract a model as a graph with dependencies. The lines in the graph represent dependencies among nodes, and the nodes represent different mini-batches

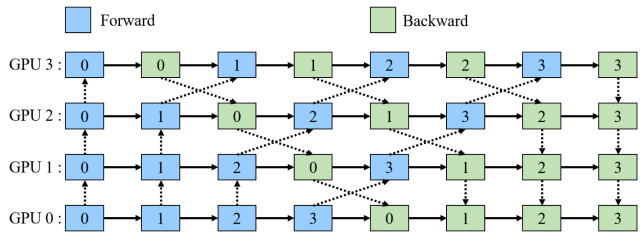

Fig. 6. The pipeline is abstracted as a graph, with blue blocks representing forward propagation and green blocks representing backward propagation. Dependencies ensure the correct execution of each block in the graph.

on various GPUs. Dependencies can be categorized into regular dependencies and virtual dependencies.

Regular dependencies define the critical execution sequence of mini-batches across GPUs. For instance, during the forward propagation, after mini-batch N is performed on GPU K, it is sent to GPU $K+1$, and GPU K proceeds to execute mini-batch $N+1$ after completing mini-batch N. In the backward propagation process, after mini-batch N is executed on GPU K, it is sent to GPU $K-1$, and GPU K proceeds to execute mini-batch $N+1$ after completing mini-batch N. Virtual dependencies refer to the requirement that the execution order of mini-batches must adhere to an underlying scheduling strategy.

As presented in Fig. 6, LIBPipe-B abstracts the LIBPipe-B pipeline based on the two types of dependencies mentioned above. In the graph, the solid line represents regular dependencies, and the dashed line represents virtual dependencies. Blue nodes represent mini-batches in the forward pass, while green nodes represent mini-batches in the backward pass. Each node has its value, representing the theoretical computation time for a specific mini-batch. Assuming that the forward propagation time and the backward propagation times are roughly equal, nodes have equal values on the same GPU. LIBPipe-B employs Algorithm 1 to achieve this functionality. In line 1 of the Algorithm 1, all nodes are wrapped. Each node possesses attributes including value, in-degree, GPU_id, and subsequent nodes. In line 2, the value and GPU_id attributes of all nodes are updated. From lines 3 to 5, dependencies are established for all nodes, and their in-degree as well as subsequent node attributes, are updated. Finally, a computational graph is returned that can evaluate the pipeline execution time proposed in this paper.

After completing the abstraction graph, it is inputted into the training time evaluation model.

2) Genetic Algorithm for Neural Network Partitioning(GA-NNP): In this section, a genetic algorithm specifically for neural network partitioning is presented to address the issue of the vast partitioning space in unequal partitioning. It can quickly find an approximate optimal partitioning, significantly enhancing the operational efficiency of LIBPipe-B. Algorithm 2 describes the implementation details of this genetic algorithm.

In line 1 of Algorithm 2, 40 sets of initial solutions are randomly generated, which is a compromise between accuracy and search time. In line 2, all hyperparameters are set. The crossover probability is set at 0.8, while the mutation probability is initially set at 0.3, and the number of iterations is established at 200. However, to prevent missing the approximate optimal solution

Algorithm 1: Graph Abstraction Modeling.

Input:
m: the number of mini-batches;
k: the number of partitions;
partition_time: the execution time of each partition;
Output:
graph_list:Directed Acyclic Graph (DAG), representing the model;

1 Initialize all nodes by encapsulating each node in a class, with each node possessing its attributes, including value, in-degree, GPU_id, and subsequent nodes;
2 Assign values to the attributes of each node;
3 **for** *iteration1 from 1 to* $k-1$ **do**
4 Establish regular dependencies for all nodes and update the in-degree and subsequent node attributes of each node;
5 Establish virtual dependencies for all nodes and update the in-degree and subsequent node attributes of each node;
6 **end**
7 **return** $graph\_list$

due to the initially high mutation probability, In line 4, GA-NNP dynamically adjusts the mutation probability, gradually reducing it by 0.005 every 10 iterations until it decreases to 0.1.

While solving with GA-NNP, it is common to encounter the problem of getting trapped in local optima. In such scenarios, it often happens that the encoding of the initial segment might have already converged to an approximate optimal state at a specific moment in the algorithm's execution. In contrast, the encoding of the subsequent segment has yet to achieve optimality. This predicament leads to continuous changes in the encoding of the earlier segment during the subsequent crossover and mutation operations, which may result in the loss of the approximate optimal solution. Intending to address this issue, in line 6, GA-NNP employs a bit-wise mutation method.

This bit-wise mutation method was gradually designed during our experimental process. Bit-wise mutation refers to the scenario where the mutation probability of each bit is related to its position. The higher the position, the higher the mutation probability. The mutation probability between two adjacent bits differs by Δx. The value of Δx is represented as shown in (14), where N represents the number of GPUs.

$$\boldsymbol{\Delta} x = \frac{\text{Mutation Probability}}{N+1} \tag{14}$$

In line 7, the fitness of each individual in the population is calculated. The calculation of fitness is intricately linked to the intricacies of an individual's partitioning strategy. Utilizing abstract graph modeling according to the partitioning strategy, the theoretical computation time for each individual is precisely determined with the aid of the training time evaluation model. Then fitness is calculated by taking the reciprocal of this time.

In line 8, individuals are selected based on their fitness. GA-NNP combines the roulette wheel selection with the elitist

Algorithm 2: Genetic Algorithm for Neural Network Partitioning.

Input: $sequence\_time\_list$ (the execution time for each layer of the model), k (the number of partitions), m (the number of mini-batches)
Output: An approximate optimal solution

```
1  Generate 40 sets of initial solutions init_list;
2  Set hyperparameters;
3  for iteration from 0 to epoch do
4      Adjust the mutation probabilities every 10 rounds;
5      cross_list = Crossover(init_list, cross_rate);
6      mutation_list =
         Mutation(cross_list, mutation_rate);
7      func_list =
         fitness(mutation_list, sequence_time_list, k, m);
8      new_solution_list = select(func_list);
9      init_list = new_solution_list
10 end
11 Select the best solution opt_list;
12 return opt_list
```

selection method to choose individuals. This method ensures that the potentially approximate optimal individuals are retained as much as possible while preserving a portion of the current non-approximate optimal individuals. This strategy maintains the diversity of individuals in the population, thereby preventing the algorithm from getting trapped in local optima. In line 11, after continuous iterations, GA-NNP outputs the approximate optimal individual, which represents the approximate optimal partition.

After implementing the two parts above, LIBPipe-B can obtain a theoretically approximate optimal solution. Even when finding the approximate optimal partition of 8 stages and 8 mini-batches in ResNet-152, the process can still be completed within a few minutes, in contrast to the tens of hours required for brute force solution. This significant efficiency improvement demonstrates the rapidity of the unequal partitioning algorithm in finding approximate optimal partitions, reducing the time cost of LIBPipe-B.

D. Custom Derivatives of the LIBPipe Framework

LIBPipe-B can be further extended to LIBPipe-Derived (LIBPipe-D). Since the key stage affecting the execution time of LIBPipe-B is the first stage, which also faces significantly greater memory pressure than other stages, allocating a GPU originally used for PMP operations to perform DP operations in the first stage can further compress the bubble time. However, this also introduces additional gradient synchronization time overhead. As shown in Fig. 7, the bubble time for LIBPipe-B is 4.8 units, whereas for LIBPipe-Derived, it is reduced to 1.5 units. The concept employed by LIBPipe-D is also based on load imbalance, requiring only minor modifications to the training time evaluation model. LIBPipe-D can then use the GA-NNP algorithm to find the optimal solution.

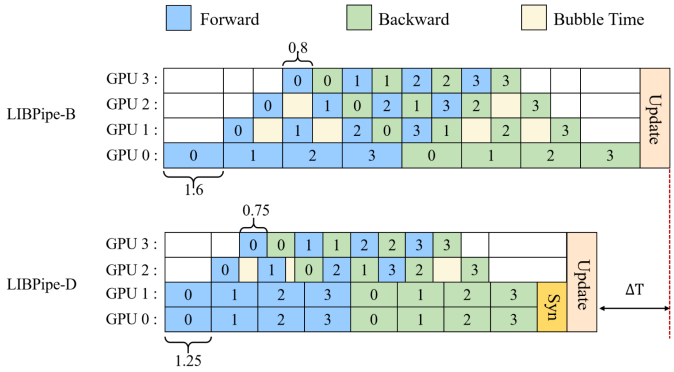

Fig. 7. Comparison of theoretical execution time between LIBPipe-B and LIBPipe-Derived under 4 GPUs and 4 mini-batches.

TABLE I
EXPERIMENTAL ENVIRONMENT

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

Due to the fact that the task load of LIBPipe-D is primarily concentrated on GPU 0 and GPU 1, an "Out of Memory" (OOM) error may occur when training large models. To address this, this paper introduces a memory estimator designed to estimate memory usage during training. The memory consumption during model execution is primarily composed of four components: model parameters, activations, gradients, and optimizer states. When partitioning the model, the estimator checks whether the memory usage of the partition exceeds the GPU memory. If it does, other partitioning results are sought. Under the constraints of the memory estimator, LIBPipe-D is able to find the optimal model partitioning strategy, ensuring that the memory usage does not exceed the hardware limits while improving training efficiency.

Due to space constraints, the detailed description of LIBPipe-D and the monotonicity analysis of $F(i)$ have been relocated to the supplementary file, available online.

IV. RESULTS & DISCUSSION

The experiments are conducted from three aspects. First, throughput comparisons between LIBPipe and other mainstream PMP methods are performed using the IMDB and mini-ImageNet datasets, with models including both language models and convolutional neural networks. All baseline methods (GPipe, DAPPLE, Chimera, PipeDream, etc.) were fully re-implemented on our hardware platform to ensure a fair and consistent comparison. Identical training configurations and hyperparameters were used across all methods unless a baseline inherently required a specific setting. Second, an analysis of

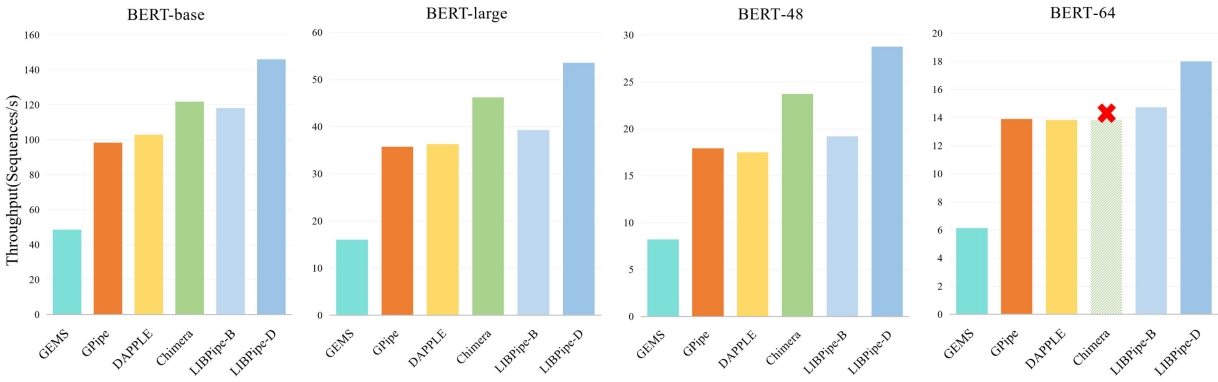

Fig. 8. Comparison of LIBPipe with other PMP methods on BERT series models with 4 GPUs and 4 mini-batches.

TABLE II
MODEL PARAMETERS

Model	Number of Parameters
ResNet-101	45M
ResNet-300	222M
ResNet-800	420M
ResNet-900	470M
BERT-base	110M
BERT-large	340M
BERT-48 (48-layer extension of BERT-large)	640M
BERT-64 (64-layer extension of BERT-large)	840M

memory usage on different models and GPUs is conducted to compare LIBPipe with other mainstream PMP methods. Lastly, brute force solutions are compared with GA-NNP to validate the effectiveness and feasibility of the proposed algorithm in searching for approximate optimal solutions.

A. Experiment Setup

The experiments utilize four convolutional neural network models and four language models: BERT-base, BERT-large, BERT-48, BERT-64, ResNet-101, ResNet-300, ResNet-800, and ResNet-900. Additional configuration parameters are listed in Table I, and the number of model parameters is shown in Table II.

B. Throughput Comparison

In this section, the paper designs throughput comparison experiments to demonstrate the efficiency of LIBPipe, categorized into language models and CNNs. For language models, the comparison methods remain consistent with the previous ones. All models use the IMDB dataset, running on 4, 6, and 8 GPUs, with a mini-batch size of 16, and a maximum sequence length of 128, and the number of mini-batches is consistent with the number of GPUs. For CNNs, all models use the mini-ImageNet dataset, running on 4, 6, and 8 GPUs, with a mini-batch size of 16, and the number of mini-batches is consistent with the number of GPUs. The methods compared for all models include GEMS, GPipe, DAPPLE, and Chimera.

1) Throughput Comparison of Language Models: As shown in Fig. 8, on 4 GPUs, LIBPipe-B outperforms GEMS by an average of 139.8%, with a maximum improvement of 143.9% across all unidirectional pipelines. It also surpasses GPipe by an average of 10.7%, with a maximum of 20.2%, and DAPPLE by an average of 9.8%, with a maximum of 14.9%. LIBPipe-D outperforms LIBPipe-B by an average of 32.9%, with a maximum improvement of 49.6%. For the bidirectional pipeline Chimera, an OOM error occurred on the BERT-64 model, so no data is available. LIBPipe-B consistently lags behind Chimera, while LIBPipe-D surpasses Chimera by an average of 18.9%, with a maximum of 21.2%.

As shown in Fig. 9, on 6 GPUs, LIBPipe-B outperforms GEMS by an average of 237.9%, with a maximum improvement of 246.0% across all unidirectional pipelines. It also surpasses GPipe by an average of 16.2%, with a maximum of 21.5%, and DAPPLE by an average of 11.2%, with a maximum of 13.1%. LIBPipe-D outperforms LIBPipe-B by an average of 25.2%, with a maximum improvement of 36.4%. For the bidirectional pipeline Chimera, LIBPipe-B consistently lags behind Chimera, while LIBPipe-D surpasses Chimera by an average of 8.1%, with a maximum of 10.6%.

As shown in Fig. 10, on 8 GPUs, LIBPipe-B outperforms GEMS by an average of 319.0%, with a maximum improvement of 363.1% across all unidirectional pipelines. It also surpasses GPipe by an average of 16.9%, with a maximum of 18.1%, and DAPPLE by an average of 14.4%, with a maximum of 15.7%. LIBPipe-D outperforms LIBPipe-B by an average of 24.1%, with a maximum improvement of 37.6%. For the bidirectional pipeline Chimera, an OOM error occurred on the BERT-64 model, so no data is available. LIBPipe-B consistently lags behind Chimera. However, LIBPipe-D leads Chimera by 6.2% on BERT-base but falls behind Chimera on all other models, with the largest gap being 11.3%.

The experimental results show that both LIBPipe-B and LIBPipe-D outperform traditional unidirectional pipelines. Although LIBPipe is a unidirectional pipeline, and it is not entirely fair to compare it with the bidirectional pipeline Chimera, a comparison was still made to validate the efficiency of LIBPipe. Chimera consistently outperforms LIBPipe-B. As for LIBPipe-D, it outperforms Chimera on 4 GPUs and 6 GPUs. However, on 8 GPUs, Chimera, with its two upward and two downward pipelines, outperforms LIBPipe-D on all models except BERT-base. This is because the computational workload of BERT-base is not large enough, and Chimera incurs significant overhead

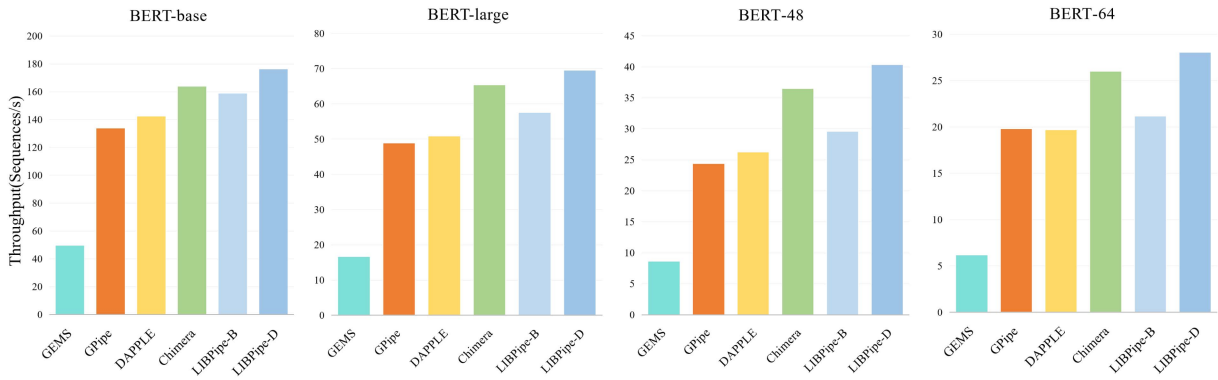

Fig. 9. Comparison of LIBPipe with other PMP methods on BERT series models with 6 GPUs and 6 mini-batches.

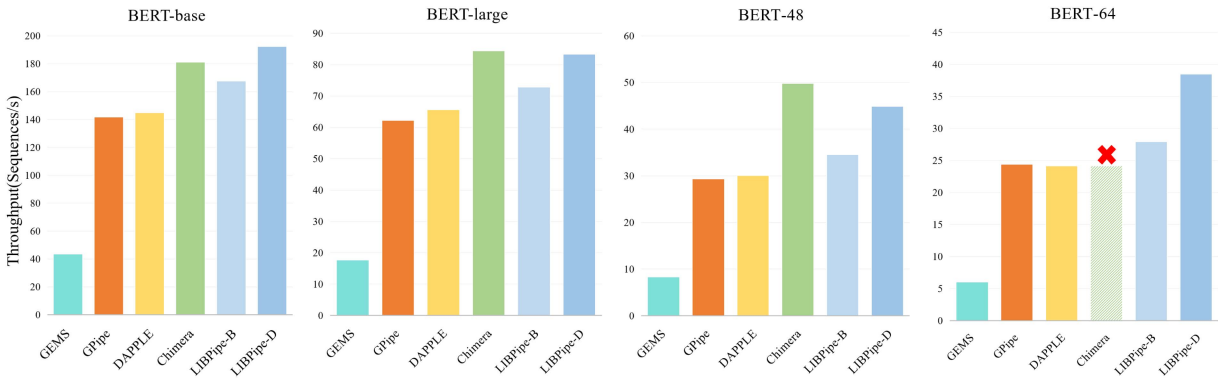

Fig. 10. Comparison of LIBPipe with other PMP methods on BERT series models with 8 GPUs and 8 mini-batches.

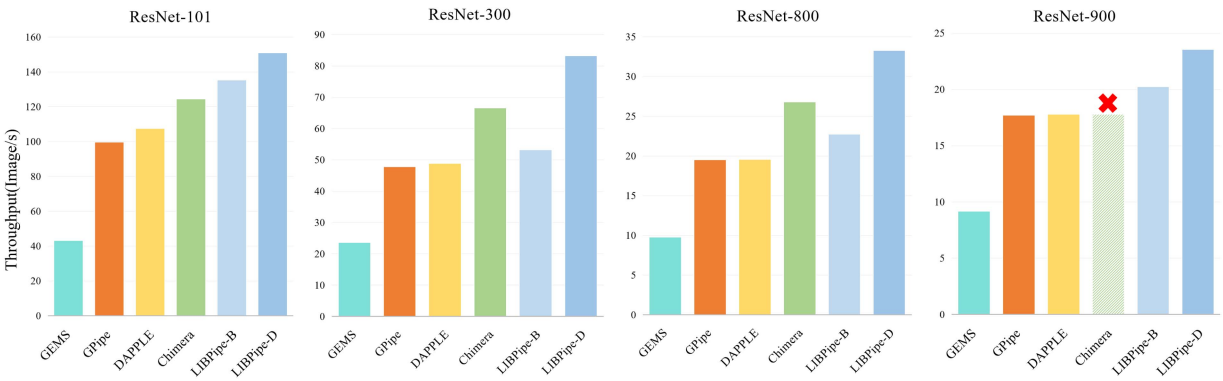

Fig. 11. Comparison of LIBPipe with other PMP methods on ResNet series models with 4 GPUs and 4 mini-Batches.

due to extensive cross-communication, greatly reducing its efficiency. Notably, Chimera encountered an OOM error with the large BERT-64 model. This indicates that the bidirectional pipeline Chimera requires higher hardware specifications, particularly in terms of memory, compared to LIBPipe. In scenarios where hardware upgrades are challenging, training large models with Chimera becomes impractical. In contrast, LIBPipe, with the aid of memory estimation as mentioned earlier, is able to identify the optimal partitioning strategy within memory constraints, ensuring that the model does not exceed memory limits while accelerating the training process.

2) Throughput Comparison of CNNs: As shown in Fig. 11, on 4 GPUs, LIBPipe-B outperforms GEMS by an average of 148.2%, with a maximum improvement of 213.2% across all unidirectional pipelines. It also leads GPipe by an average of 19.5%, with a maximum of 36.0%, and DAPPLE by an average of 16.2%, with a maximum of 26.0%. LIBPipe-D outperforms LIBPipe-B by an average of 32.6%, with a maximum improvement of 56.2%. For the bidirectional pipeline Chimera, an OOM error occurred on the ResNet-900 model, so no data is available. LIBPipe-B only surpasses Chimera by 8.7% on ResNet-101, while it falls behind Chimera for the other models. LIBPipe-D,

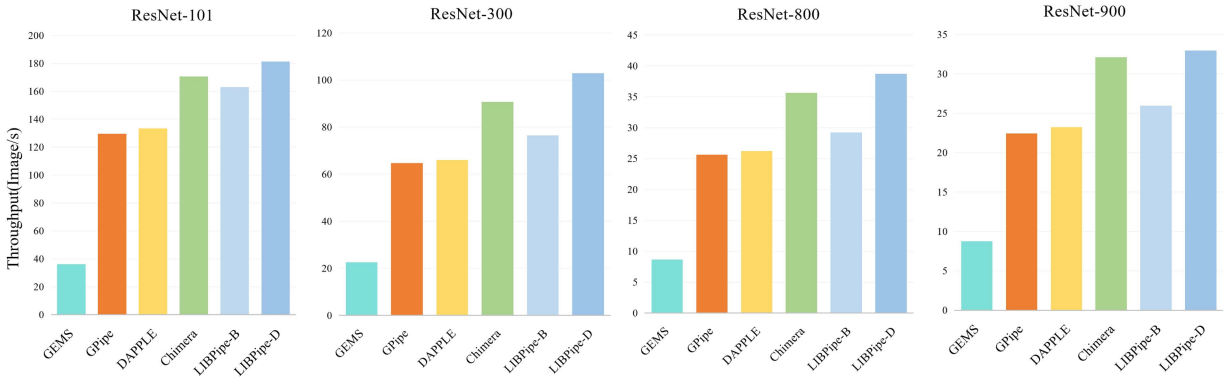

Fig. 12. Comparison of LIBPipe with other PMP methods on ResNet series models with 6 GPUs and 6 mini-batches.

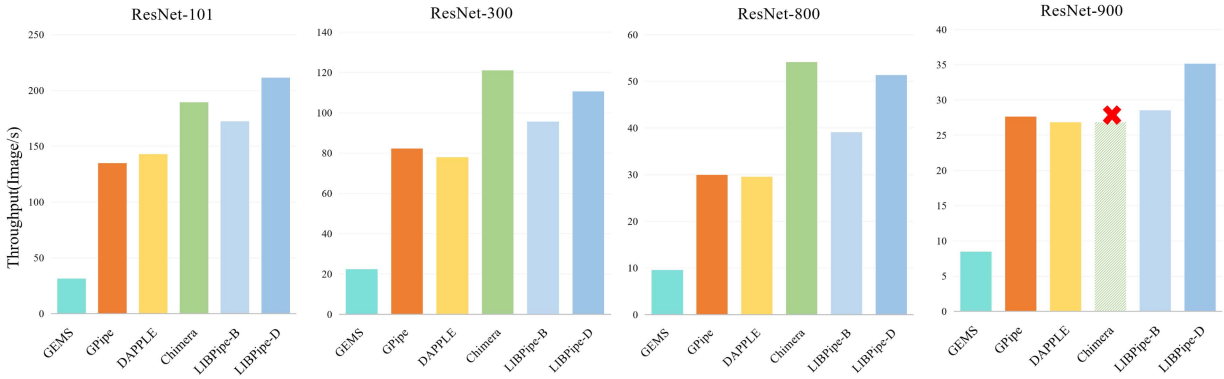

Fig. 13. Comparison of LIBPipe with other PMP methods on ResNet series models with 8 GPUs and 8 mini-batches.

however, outperforms Chimera by an average of 23.5%, with a maximum of 25.0%.

As shown in Fig. 12, On 6 GPUs, LIBPipe-B leads GEMS by an average of 254.4%, with a maximum improvement of 349.3%. It also surpasses GPipe by an average of 18.4%, with a maximum of 24.0%, and DAPPLE by an average of 15.2%, with a maximum of 22.0%. LIBPipe-D exceeds LIBPipe-B by an average of 26.3%, with a maximum improvement of 34.6%. For Chimera, LIBPipe-B consistently lags behind Chimera. LIBPipe-D outperforms Chimera by an average of 7.7%, with a maximum improvement of 13.4%.

As shown in Fig. 13, on 8 GPUs, LIBPipe-B outperforms GEMS by an average of 328.4%, with a maximum improvement of 445.9%. It also leads GPipe by an average of 19.5%, with a maximum improvement of 30.4%, and surpasses DAPPLE by an average of 20.5%, with a maximum of 32.4%. LIBPipe-D outperforms LIBPipe-B by an average of 23.21%, with a maximum improvement of 31.3%. For the bidirectional pipeline Chimera, it encountered an OOM error on ResNet-900. LIBPipe-B consistently lags behind Chimera. As for LIBPipe-D, it outperforms Chimera by 11.6% only on ResNet-101, but falls behind Chimera on other models, with the largest gap being 8.5%.

The experimental results also demonstrate that both LIBPipe-B and LIBPipe-D are effective on DCNNs. The trends observed in language models are also present in DCNN models. Theoretically, Chimera should consistently outperform LIBPipe-B. However, on 4 GPUs, the results for ResNet-101 indicate otherwise, as the computational workload of ResNet-101 is too small, and Chimera is significantly impacted by communication overhead. As for LIBPipe-D, it outperforms Chimera on 4 GPUs and 6 GPUs. However, on 8 GPUs, LIBPipe-D only outperformed Chimera on ResNet-101, while it lagged behind Chimera on ResNet-300 and ResNet-800. Similar to the BERT-64 case, Chimera encountered an OOM error on ResNet-900, whereas LIBPipe-D did not. This indicates that LIBPipe demonstrates strong performance across both language models and DCNNs.

C. Memory Analysis

In this section, we provide a detailed analysis of memory usage across 8 GPUs for the BERT and ResNet series models under different methods. As shown in Fig. 14, for the BERT series, the memory usage of LIBPipe-B on GPU 0 significantly exceeds that of other traditional methods. This is due to the load imbalance strategy, where LIBPipe-B tends to assign more computational work to GPU 0, leading to higher memory utilization. For LIBPipe-D, the highest memory usage is observed on GPU 0 and GPU 1, as these GPUs are used for data parallelism, and they are allocated a larger portion of the model layers. Consequently, the memory usage on the remaining GPUs is relatively lower. For BERT-48 and BERT-64, the memory usage of LIBPipe-D on GPU 0 and GPU 1 approaches the maximum GPU capacity.

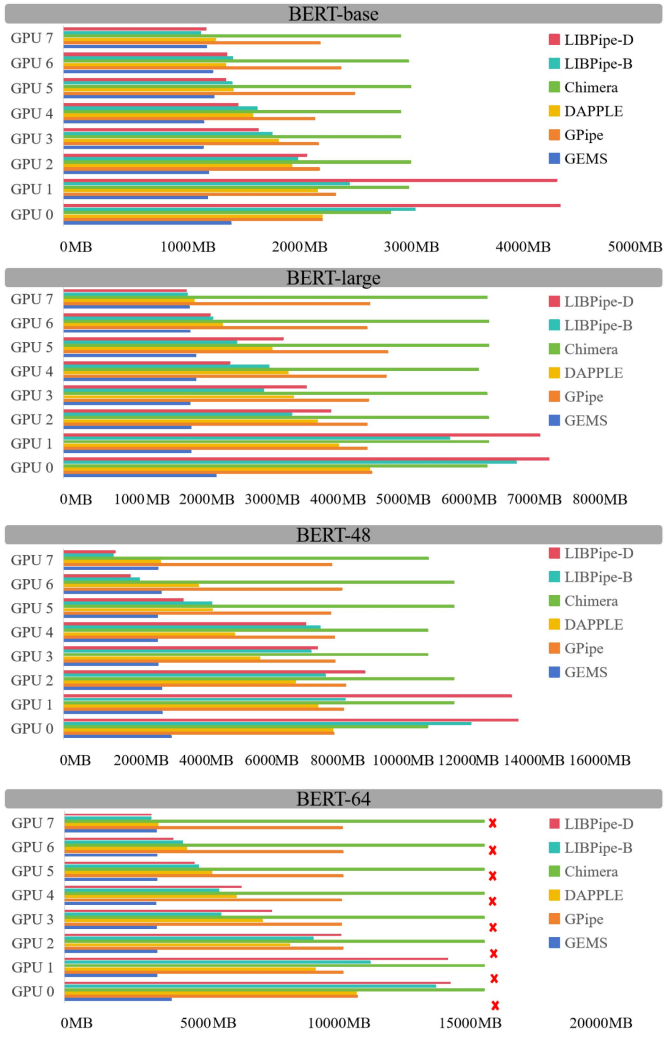

Fig. 14. GPU memory usage for BERT series models on 8 GPUs.

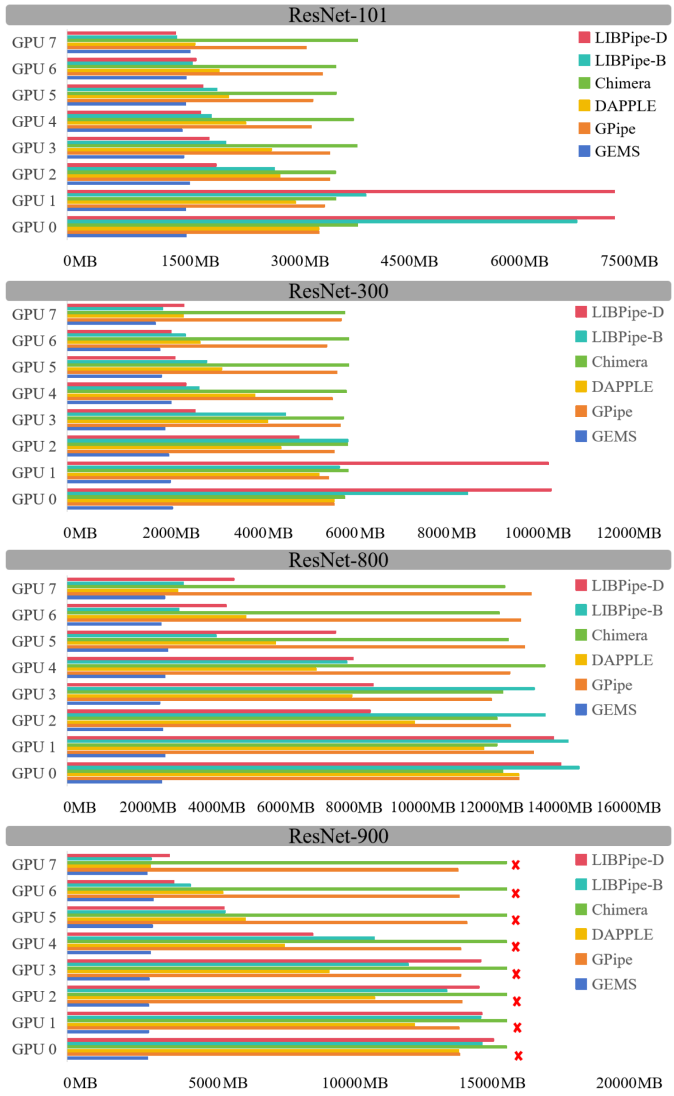

Fig. 15. GPU memory usage for ResNet series models on 8 GPUs.

However, with the help of the memory estimator, it can find the optimal partitioning within the GPU memory limits, avoiding exceeding the physical capacity. In contrast, Chimera encountered an OOM error when handling BERT-64, preventing it from completing the training task. This further highlights Chimera's limitations in memory management when dealing with complex models, especially under hardware constraints. As shown in Fig. 15, the memory usage pattern for the ResNet series is similar to that of the BERT series. Chimera also encountered an OOM error when handling ResNet-900, whereas LIBPipe was able to execute normally.

It is worth noting that although LIBPipe's load imbalance strategy uses more memory on GPU 0 and GPU 1, with the support of the memory estimator, it ensures that memory usage throughout the training process remains within the limit. Compared to Chimera, LIBPipe demonstrates a clear advantage in memory management, especially when training large-scale models, as it avoids OOM error caused by insufficient memory. This optimization in memory utilization not only improves the training speed but also ensures that large model training can be completed stably and efficiently in memory-constrained environments.

D. *Unequal Partitioning Algorithm Efficiency Comparison*

In this section, to validate the efficiency of the unequal partitioning algorithm, we designed a set of comparative experiments, applying the brute force solution and GA-NNP to the BERT series models and ResNet series models. Tests were conducted on 2 to 8 GPUs. We recorded the time taken by both methods to achieve the approximate optimal solution and checked whether the results obtained were consistent.

Table III displays the time taken to solve eight models using the two methods, measured in seconds. The "×" symbol in the Brute force column indicates that a solution could not be obtained within 1 h. Consistency represents whether the search results of Brute force and GA-NNP are identical, with "✓" indicating consistency and "×" indicating otherwise. For ease of viewing, we bold the shorter time obtained by the two methods.

TABLE III
SEARCH TIME COMPARISON OF BRUTE FORCE SOLUTION AND GA-NNP (SEC)

number of GPUs	BERT-base			BERT-large			BERT-48			BERT-64		
	Brute force	GA-NNP	consistency	Brute force	GA-NNP	consistency	Brute force	GA-NNP	consistency	Brute force	GA-NNP	consistency
2	**0.027**	4.717	✓	**0.038**	4.631	✓	**0.070**	5.828	✓	**0.099**	6.014	✓
3	**0.031**	6.786	✓	**0.068**	12.697	✓	**7.324**	13.936	✓	8.334	**23.005**	✓
4	**0.725**	21.562	✓	86.043	**32.324**	✓	×	**39.241**	×	×	**44.196**	×
5	×	**28.685**	×	×	**52.448**	×	×	**64.863**	×	×	**80.472**	×
6	×	**44.975**	×	×	**68.213**	×	×	**92.927**	×	×	**116.617**	×
7	×	**64.892**	×	×	**84.571**	×	×	**116.520**	×	×	**151.684**	×
8	×	**91.248**	×	×	**193.37**	×	×	**235.283**	×	×	**273.239**	×
number of GPUs	ResNet-101			ResNet-300			ResNet-800			ResNet-900		
	Brute force	GA-NNP	consistency	Brute force	GA-NNP	consistency	Brute force	GA-NNP	consistency	Brute force	GA-NNP	consistency
2	**0.052**	5.461	✓	**0.118**	7.274	✓	**0.468**	10.523	✓	**0.575**	12.639	✓
3	**0.882**	10.570	✓	21.425	**15.260**	✓	247.962	**18.502**	✓	351.578	**20.039**	✓
4	70.229	**28.698**	✓	×	**25.992**	×	×	**37.202**	×	×	**44.525**	×
5	×	**33.538**	×	×	**48.947**	×	×	**76.491**	×	×	**90.664**	×
6	×	**58.446**	×	×	**90.824**	×	×	**102.201**	×	×	**139.645**	×
7	×	**78.801**	×	×	**120.620**	×	×	**140.401**	×	×	**184.462**	×
8	×	**117.05**	×	×	**290.093**	×	×	**345.756**	×	×	**382.131**	×

In terms of search time, when the number of GPUs is limited to two or three, the experimental data shows that, except for BERT-64, ResNet-300, ResNet-800, and ResNet-900, the brute force solution outperforms GA-NNP on all models. However, the search time of GA-NNP remains within an acceptable range. When the number of GPUs increases to four or more, the situation changes. For all models except BERT-base, GA-NNP surpasses the brute force solution. This is because, in the specific implementation, BERT-base can only be split into 101 layers, resulting in a significantly smaller search space compared to other models. It is worth noting that as the number of model layers and GPUs increases, the gap between the brute force solution and GA-NNP widens dramatically. The brute force solution becomes significantly less efficient than GA-NNP for all models, and its search time becomes unacceptable.

In terms of search accuracy, GA-NNP can achieve the same accuracy as the brute force solution. However, due to the excessively long search time of the brute force solution on large models with multiple GPUs, it is unclear whether the GA-NNP results remain consistent beyond 4 GPUs. Nevertheless, based on the throughput experiments, GA-NNP's search results still deliver an acceleration effect.

By analyzing the data in Table III, it is evident that the time required to find the optimal solution using the brute force method increases exponentially with the number of GPUs and the depth of the model layers. In contrast, GA-NNP keeps the search time within an acceptable range while maintaining consistent results with the brute force method, highlighting the necessity of using GA-NNP. In the experiments, LIBPipe adopts the brute force method for scenarios involving a small number of GPUs and shallow models, while GA-NNP is used for scenarios involving a large number of GPUs and deeper models.

E. Discussion

1) Heterogeneous Environment Expansion: Heterogeneous environments naturally introduce load imbalance because GPUs differ in computational capability. As illustrated in Fig. 16, assume that each GPU has different computational capabilities. Existing heterogeneous-training systems typically employ unequal model partitioning to compensate for this hardware imbalance, aiming to make the execution time of each stage approximately equal. This type of unequal division is fundamentally different from the load imbalance proposed in this paper. Traditional unequal partitioning is used to achieve load balancing, whereas our method deliberately introduces computational load imbalance within a homogeneous pipeline to reshape the scheduling timeline and reduce bubble time. In other words, although unequal partitioning appears in heterogeneous methods, its purpose and mechanism differ completely from ours. Our imbalance is not a side effect of heterogeneity but a deliberate design that enables performance gains unobtainable with balancing-based approaches.

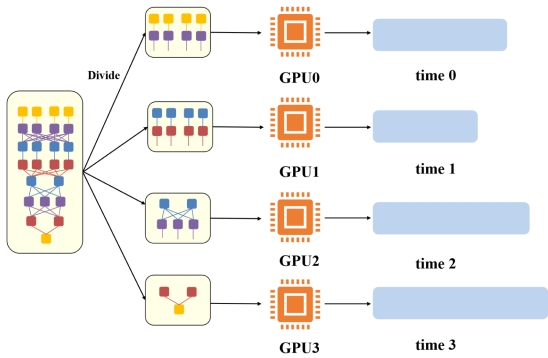

Fig. 16. PMP in heterogeneous environments.

2) Combination of LIBPipe and Freeze Algorithm: The Freeze algorithm is a strategy used to accelerate neural network model training by progressively freezing the parameters of certain layers to reduce the computational load. Its core idea is that as training progresses, some layers converge earlier and no longer require updates. By freezing these layers, the computational cost is reduced, allowing resources to be focused on training other unfrozen layers. When training large models such as Transformer models, the earlier layers often converge faster and can be frozen sooner. The frozen layers no longer require parameter updates or gradient storage, alleviating memory usage. Combining this algorithm with LIBPipe can help reduce memory usage on lower-level GPUs in LIBPipe. Additionally, once certain layers are frozen, LIBPipe can reallocate the model partitioning, further speeding up the training process. In the future, we plan to conduct further research on this combination.

Due to space constraints, the unified hyperparameter settings, the MixPipe experiments, and the extended Chimera memory comparison have been relocated to the supplementary file, available online.

V. Conclusion

This paper presents LIBPipe, an innovative method that effectively implements a load imbalance strategy to substantially reduce bubble time during the training process. This objective is accomplished through unequal model partitioning based on a training time evaluation model, with theoretical proofs ensuring the method's reliability. In the experiments, LIBPipe was benchmarked against existing methods (including GPipe, DAPPLE, GEMS, and Chimera), demonstrating superior performance. Specifically, the throughput of the BERT series increased by up to 60.3%, and the ResNet series saw a maximum improvement of 74.1%, without encountering any OOM errors. These results highlight LIBPipe's advantages in improving the training efficiency of deep neural networks. In the future, we will focus on exploring the significant potential of LIBPipe in heterogeneous environments, combining it with the Freeze algorithm, and further enhancing its applications in this field. Overall, LIBPipe demonstrates a robust capability in handling large datasets for deep learning models, offering innovative approaches to distributed training. This method not only paves the way for new possibilities in future research but also holds significant implications for applications in the field of distributed training.

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] J. Zhuang and M. Al Hasan, "Defending graph convolutional networks against dynamic graph perturbations via Bayesian self-supervision," in *Proc. AAAI Conf. Artif. Intell.*, 2022, pp. 4405–4413.
[2] B. Liu, H. Ren, J. Li, N. Duan, A. Yuan, and H. Zhang, "RE-RCNN: A novel representation-enhanced RCNN model for early apple leaf disease detection," *ACM Trans. Sensor Netw.*, 2023.
[3] C. Zheng et al., "Deep learning-based human pose estimation: A survey," *ACM Comput. Surv.*, vol. 56, no. 1, pp. 1–37, 2023.
[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, 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] H. Huang et al., "ChatGPT for shaping the future of dentistry: The potential of multi-modal large language model," *Int. J. Oral Sci.*, vol. 15, 2023, Art. no. 29.
[7] 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, 2022.
[8] Y. Xu and H. Zhang, "Convergence of deep convolutional neural networks," *Neural Netw.*, vol. 153, pp. 553–563, 2022.
[9] 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.
[10] M. Langer, Z. He, W. Rahayu, and Y. Xue, "Distributed training of deep learning models: A taxonomic perspective," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 12, pp. 2802–2818, Dec. 2020.
[11] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2020, pp. 3505–3506.
[12] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 103–112.
[13] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, and C. Guo, "Elastic parameter server load distribution in deep learning clusters," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 507–521.
[14] 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.
[15] 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.
[16] 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.
[17] 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, 2021.
[18] D. Narayanan et al., "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 1–15.
[19] 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.
[20] 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.
[21] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," 2014, *arXiv:1404.5997*.
[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] C. He, S. Li, M. Soltanolkotabi, and S. Avestimehr, "PipeTransformer: Automated elastic pipelining for distributed training of transformers," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 4150–4159.
[24] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel DNN training," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 7937–7947.
[25] T. Kim, H. Kim, G.-I. Yu, and B.-G. Chun, "BPipe: Memory-balanced pipeline parallelism for training large language models," in *Proc. 40th Int. Conf. Mach. Learn.*, 2023, pp. 16639–16 653.
[26] H. Dreuning, H. E. Bal, and R. V. v. Nieuwpoort, "MCAP: Memory-centric partitioning for large-scale pipeline-parallel DNN training," in *Proc. Eur. Conf. Parallel Process.*, 2022, pp. 155–170.
[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] L. Zhao et al., "BaPipe: Balanced pipeline parallelism for DNN training," *Parallel Process. Lett.*, vol. 32, 2022, Art. no. 2250005.
[29] Z. Li et al., "TeraPipe: Token-level pipeline parallelism for training large-scale language models," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 6543–6552.
[30] W. Zhang, B. Zhou, X. Tang, Z. Wang, and S. Hu, "MixPipe: Efficient bidirectional pipeline parallelism for training large-scale models," in *Proc. ACM/IEEE Des. Automat. Conf.*, 2023, pp. 1–6.
[31] P. Qi, X. Wan, G. Huang, and M. Lin, "Zero bubble (almost) pipeline parallelism," in *Proc. 12th Int. Conf. Learn. Representations*, 2024, pp. 47619–47635.

Bin Liu (Member IEEE) received the PhD degree in electronic and information engineering from Xi'an Jiaotong University, China, in 2014. Since 2018, he has been an associate professor and doctoral supervisor with the College of Information Engineering, Northwest A&F University, China, where he has obtained 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.

Zhonghao Zhang received the BS degree in computer science and technology from Taiyuan University of Technology, in 2022. He is currently majoring in computer science and technology with Northwest A&F University. His research interests include parallel computing, computer vision, and distributed computing.

Hengzhao Li received the BS degree in mechatronic engineering from Xi'an Shiyou University, in 2023. He is currently working toward the master's degree in electronic information with Northwest A&F University. His research interests include parallel computing, computer vision, and distributed computing.

Zeyu Ji received the BS degree from the School of Information Engineering, Zhengzhou University, the MS degree from the Polytechnic University of Tours, France, and the PhD degree from Xi'an Jiaotong University. He is currently an assistant professor with the College of Information Engineering, Northwest A&F University, Yangling, China. His main research interests include computer architecture, high-performance computing, and deep learning.

Hongming Zhang (Member IEEE) received the BS degree in computer science and technology, the MS degree in cartography and geographic information systems, and the PhD degree in Land Resources and Spatial Information Technology from Northwest A&F University, China, in 2003, 2008, and 2012, respectively. He is currently a professor in the College of Information Engineering, Northwest A&F University, Yangling, China. His research interests include computer vision and remote sensing image analysis and understanding.

Keqin Li (Fellow, IEEE) received the BS degree in computer science from Tsinghua University, in 1985 and the PhD degree in computer science from the University of Houston, in 1990. He is currently 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 950 journal articles, book chapters, and refereed conference papers. He received several best paper awards from international conferences including *PDPTA-1996*, *NAECON-1997*, *IPDPS-2000*, *ISPA-2016*, *NPC-2019*, *ISPA-2019*, and *CPSCom-2022*. He holds nearly 70 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top five most influential scientists in parallel and distributed computing in terms of single-year and career-long impacts based on a composite indicator of the Scopus citation database. He was a 2017 recipient of the Albert Nelson Marquis Lifetime Achievement Award for being listed in Marquis Who's Who in Science and Engineering, Who's Who in America, Who's Who in the World, and Who's Who in American Education for more than twenty consecutive years. He received the Distinguished Alumnus Award from the Computer Science Department at the University of Houston in 2018. 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 is a member of the SUNY Distinguished Academy. He is an AAAS Fellow, and an AAIA Fellow. He is a member of Academia Europaea (Academician of the Academy of Europe).