UMPIPE: Unequal Microbatches-Based Pipeline Parallelism for Deep Neural Network Training

Guangyao Zhou, Wenhong Tian, *Member, IEEE*, Rajkumar Buyya, *Fellow, IEEE*, and Kui Wu, *Member, IEEE*

***Abstract*—The increasing need for large-scale deep neural networks (DNN) has made parallel training an area of intensive focus. One effective method, microbatch-based pipeline parallelism (notably GPipe), accelerates parallel training in various architectures. However, existing parallel training architectures normally use equal data partitioning (EDP), where each layer's process maintains identical microbatch-sizes. EDP may hinder training speed because different processes often require varying optimal microbatch-sizes. To address this, we introduce UMPIPE, a novel framework for unequal microbatches-based pipeline parallelism. UMPIPE enables unequal data partitions (UEDP) across processes to optimize resource utilization. We develop a recurrence formula to calculate the time cost in UMPIPE by considering both computation and communication processes. To further enhance UMPIPE's efficiency, we propose the Dual-Chromosome Genetic Algorithm for UMPIPE (DGAP) that accounts for the independent time costs of forward and backward propagation. Furthermore, we present TiDGAP, a two-level improvement on DGAP. TiDGAP accelerates the process by simultaneously calculating the end time for multiple individuals and microbatches using matrix operations. Our extensive experiments validate the dual-chromosome strategy's optimization benefits and TiDGAP's acceleration capabilities. TiDGAP can achieve better training schemes than baselines, such as the local greedy algorithm and the global greedy-based dynamic programming. Compared to (GPipe, PipeDream), UMPIPE achieves increases in training speed: $(13.89, 11.09)\%$ for GPT1-14, $(17.11, 7.96)\%$ for VGG16 and $\geq (170, 100)\%$ for simulation networks.**

***Index Terms*—Parallel training, data parallelism, unequal data partitions, pipeline parallelism, genetic algorithm.**

I. Introduction

THE advancement of intelligent technology has significantly increased the application of deep neural networks (DNNs) in various domains, such as image processing [1], [2] and natural language processing (NLP) [3]. DNNs come in diverse structures, such as Convolutional Neural Networks (CNN) [4], transformer layer-based networks [5], [6], and Graph Neural Networks [7], [8]. The advent of large language models (LLMs) has led to an exponential growth in parameter scale. Consequently, parallel training on distributed systems, such as GPU clusters, has emerged as a crucial approach and a focal point of research to overcome these challenges in large-scale DNNs [4], [9], [10]

Data Parallelism (DP), Tensor Model Parallelism (TMP), and Pipeline Model Parallelism (PMP) are three foundational parallel modes for training DNNs on distributed systems [6], [11], [12], [13]. PMP and TMP, in particular, often leverage a combination with DP to enhance their architectures. Renowned PMP architectures include GPipe [14], PipeDream [15], Dapple [16], and Hpipe [11], while TMP is represented by Megatron-LM [5] and Tofu [17]. Further, hybrid 3D parallelism models integrating DP, TMP, and PMP, like FOLD3D [18] and Merak [13], have significantly boosted parallel training performance. These existing architectures all rely on microbatch-based data parallelism. In the parallel training, forward propagation (FP) and backward propagation (BP) comprise computation and communication processes. A notable challenge is the "bubbles" or idle times each device experiences while waiting for preceding processes on other devices to complete [11], [16]. Reducing bubbles and increasing the overlap between computation and communication is a key focus in optimizing parallel training [18]. Techniques such as dynamic programming [19], linear programming [20], and recurrence methods [21] are developed to mitigate bubbles.

Despite the above progress, Equal Data Partitioning (EDP) remains a significant limitation across most parallelism. EDP means that all processes of computations and communications, must handle the same number of microbatches within a mini-batch [22]. The limitation caused by EDP is prevalent in almost all existing parallel architectures that rely on microbatch-based data parallelism (e.g., GPipe [14], PipeDream [15], Dapple [16]. In the dynamic landscape of DNNs, layer heterogeneity significantly impacts computation and communication times, which are not necessarily proportional to the microbatch-size [23], [24], [25]. For example in Fig. 1, the layers of GPT-1 with transformer and VGG16 with CNN have variations in processing the same 512 pieces of data under different microbatch-sizes, especially when the microbatch-size is small (if computation time is proportional to data size, the curves should be straight

Received 13 January 2024; revised 22 November 2024; accepted 7 December 2024. Date of publication 11 December 2024; date of current version 30 December 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2018AAA0103203 and in part by Project of Key Research and Development Program of Sichuan Province under Grant 2021YFG0325. Recommended for acceptance by J. Zola. *(Corresponding author: Guangyao Zhou.)*

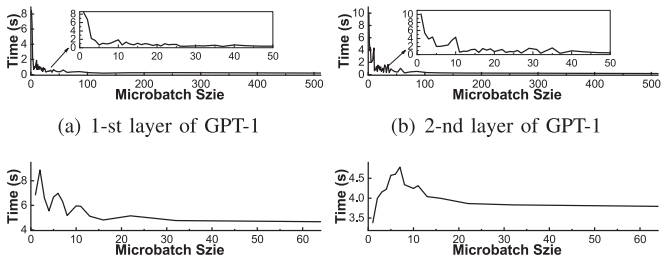
Guangyao Zhou is with the School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu, Sichuan 610032, China (e-mail: guangyao_zhou@swjtu.edu.cn).

Wenhong Tian is with the School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu, Sichuan 610056, China (e-mail: tian_wenhong@uestc.edu.cn).

Rajkumar Buyya is with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, University of Melbourne, Melbourne, VIC 3052, Australia (e-mail: rbuyya@unimelb.edu.au).

Kui Wu is with the Department of Computer Science, University of Victoria, Victoria, BC V8P 5C2, Canada (e-mail: wkui@uvic.ca).

Digital Object Identifier 10.1109/TPDS.2024.3515804



(a) 1-st layer of GPT-1 (b) 2-nd layer of GPT-1

(c) 1-st layer of VGG16 (d) 2-nd layer of VGG16

Fig. 1. Computation time to process 512 pieces data in realistic GPU devices.

lines parallel to X-axis). Additionally, different layers have not only different time consumption but also different inflection points, which reflects heterogeneity between layers. The heterogeneity and nonlinearity suggests that optimal partition numbers for different processes might vary. Hence, EDP may result in inefficiencies.

Motivated by this observation, we propose UMPIPE, a novel pipeline parallel structure utilizing unequal microbatches. This approach allows for unequal data partitioning (UEDP) across both computation and communication processes in DNNs. UMPIPE represents the first exploration into unequal partitioning within parallel training and effectively harnesses the heterogeneous time consumption characteristics of DNN layers. We develop a general recurrence formula and conduct an in-depth analysis of UMPIPE's optimality. Our findings indicate that UMPIPE's optimal scheme is at least as efficient, if not more so, than existing methods like GPipe.

To obtain the optimization scheme of UMPIPE, we propose a dual-chromosome genetic algorithm (DGAP), leveraging the independence of data partitions of FP and BP. Since the microbatch-sizes for processes in UMPIPE can be different, the formula of estimating training time in GPipe [26], [27] does not apply to UMPIPE. To obtain explicit expressions of training time in UMPIPE is a challenge. It significantly complicates the computational process in determining UMPIPE's optimal scheme using recurrence formulas. To address this challenge, we propose a two-level improvement method based on matrix operations, which expedites calculations by simultaneously processing multiple individuals and microbatches. We integrate this method with DGAP to form Two-level Improved Dual-Chromosome Genetic Algorithm (TiDGAP).

Our contributions are summarized as follows:

1) *UMPIPE*: Our approach considers not just the computation and communication simultaneously but also their nonlinear time consumption relative to microbatch-size. We propose UMPIPE, a UEDP-based pipeline parallelism framework. This innovation allows different microbatch-sizes for processes within DNNs, thereby accelerating parallel training. We derive a recurrence formula and conduct a comprehensive theoretical analysis for UMPIPE.
2) *DGAP*: Recognizing that time costs for FP and BP are independent in UMPIPE, we develop Dual-Chromosome Genetic Algorithm (DGAP) to identify the optimal scheme. Our theoretical analysis of the dual-chromosome strategy highlights its statistical advantages in efficiently resolving UMPIPE's unique scheme requirements.
3) *TiDGAP*: Addressing the challenges in calculating UMPIPE's training time due to UEDP, we conduct theoretical derivation and propose TiDGAP, a matrix operation-based two-level improved method to accelerate DGAP. TiDGAP can simultaneously calculate the end time corresponding to multiple individuals and multiple microbatches, substantially boosting DGAP's search capability.
4) Extensive experiments demonstrate the fast speed and optimality of TiDGAP compared to baseline methods. Additionally, results confirm that UMPIPE achieves a faster training speed than baseline parallelism without compromising the training convergence of DNNs. Compared to (GPipe, PipeDream), UMPIPE achieves $(13.89, 11.09)\%$ improvement in GPT1-14, $(17.11, 7.96)\%$ in VGG16, and $\geq (170, 100)\%$ in simulation networks.

The rest of this paper is organized as follows. We review the related work in Section II. The UMPIPE parallelism is presented in Section III. The DGAP and TiDGAP are presented in Section IV. The experiment evaluations are presented in Section V. Finally, we conclude this paper in Section VI.

II. Related Work

A. Parallelism for Training DNNs

Data parallelism (DP), tensor model parallelism (TMP) and pipeline model parallelism (PMP) are three basic architectures to train large-scale DNNs [13], [18].

Data level parallelism generally divides the data into multiple parts for time-sharing or equipment-sharing training [28], [29]. Joshua Romero et al. [12] implemented a lightweight decentralized coordination strategy by utilizing a response cache to accelerate collective communication in data parallel training. Lei Guan et al. [30] proposed pdlADMM that splits the optimization problem into sub-problems to train the fully connected DNN in a data-parallel manner.

Tensor model parallelism divides layers of DNN into multiple parts from the dimension of tensor operations [11], [27]. Based on model parallelism, Deepak Narayanan et al. [5] proposed Megatron-LM to divide the self-attention layer into a multi-tensor operation model, which allowed the self-attention layer to be put on different devices.

Pipeline-related parallelism, usually combining DP and TMP, is an important method in parallel training. GPipe [14] split the minibatch into multiple microbatches and utilized the pipeline to train each part of the model on its corresponding distributed node. Terapipe [6] followed the pipeline of GPipe and improved the pipelining granularity to reduce the pipeline bubbles of the transformer-based NLP model by proposing a new dimension, i.e., token dimension. Based on GPipe, PipeDream [15] shifted the gradient backward-propagation (BP) of each microbatch earlier to the moment immediately after its last part of forward. Dapple [16] followed PipeDream and shifted the BP of other sub-model nodes to earlier besides that of the last sub-model node for the same minibatch [16].

Hybrid 3D parallelism, which integrates DP, PMP, and TMP, is currently an effective framework for high training efficiency [13], [18]. Fanxin Li et al. [18] proposed FOLD3D to slice a DNN into multiple segments, which allowed the computations in the same segment to be scheduled together. Zhiquan Lai et al. [13] proposed Merak that can automatically deploy 3D parallelism. Some other hybrid parallelism includes EffTra [26], ParaDL [31] and PipePar [32].

Other strategies to improve the performance of parallel training include momentum-driven adaptive synchronization model [33], NeoFlow (flexible framework for enabling efficient compilation) [34], PaSE parallelization [21], etc.

B. Optimization Methods

The parallel architecture needs optimization methods to determine the optimal parallel schemes. Because the cost model was generally non-analytical or recursive, dynamic programming is a suitable and widely used method to obtain the optimal partition of data or model parallelism [19]. Some examples using dynamic programming include PipeDream [16], Dapple [16], Terapipe [6], EffTra [26], PaSE [21], PipePar [32] et al. Linear programming is also a frequently-used method in parallel training [4], [20], [29]. Some examples include NetPlacer [20], HGP4CNN [4], DPDA [29]. Other partition methods include off-the-shelf graph partitioning algorithms [7], recurrence [21], grouping genetic algorithm [35], and near-optimal layer partition of local search method [36].

C. Difference of Our Solution

From the reviewed literature, how to construct a well-performed parallel training architecture to accelerate training speed by reducing pipeline bubbles and redundant communication is an urgent topic. The existing DP and TMP frameworks rely on equal partitioning. However, the optimal partitioning numbers of layers in DNN may differ, and equal partitioning will restrict the optimization boundary of parallelism.

The notable feature of our proposed UMPIPE is unequal data partitioning (UEDP). In UMPIPE, communication and computation time are non-negligible and are not necessarily proportional to the microbatch-size. These properties indicate that the targeted scenarios of our proposed UMPIPE are far more realistic. Our TiDGAP is also significantly different from existing algorithms due to UMPIPE. Although TiDGAP still relies on recurrence formulas, its calculation process has been improved through matrix operations based on theoretical derivation. Matrix operation-based formulas enable TiDGAP to obtain schemes of UMPIPE in the order of seconds.

III. UMPIPE Parallelism and Cost Model

A. Architecture of UMPIPE Parallelism

For the sake of presenting UMPIPE's architecture and cost model, we list notations in Table I. In one epoch of training DNNs, a dataset is divided into multiple minibatches and each minibatch needs to start after the previous minibatch ends. The process undergoing one FP and one BP using a minibatch as the input data is one training iteration of the DNN on the minibatch. Minimizing the total training time can be equivalently converted to minimizing the time for training one iteration of one minibatch. It can be set that the minibatch-size of training DNN is P, which means training P pieces of data simultaneously in one minibatch. These P data can be set as $\{d_1, d_2, \ldots, d_P\}$. To reduce parallel training time, the existing microbatch-based pipeline parallelism (e.g., GPipe) partitions a minibatch into multiple microbatches. As GPipe makes a constraint that each layer has the same number of microbatches (equal data partitioning, EDP), some time-consuming processes (computation or communication) may limit the optimization of the total training time.

TABLE I
Notations and Descriptions

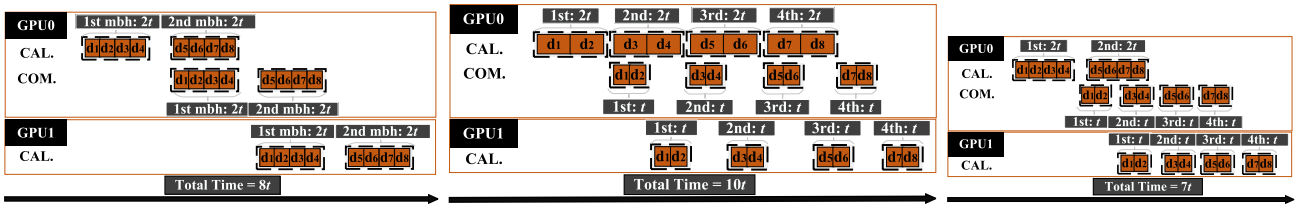
Notation	Description
N	Number of stages
p_k	Number of microbatch in the k-th forward process
q_k	Number of microbatch in the k-th backward process
$F_i^P(p)$	The forward compute time of one microbatch in the i-th stage when partitioning the minibatch into p microbatches
$F_i^M(p)$	The forward communication time of one microbatch
$B_i^P(p)$	The backward compute time of one microbatch
$B_i^M(p)$	The backward communication time of one microbatch
$F_k(p_k)$	The time cost of one microbatch in the k-th forward process
E_{kj}	The end time of the j-th data of the k-th forward process
T_{ki}	The begin time of the i-th microbatch in the k-th process
$B_k(q_k)$	The time cost of one microbatch in the k-th backward process
R_{kj}	The end time of the j-th data of the k-th backward process

To reduce the training time, UMPIPE introduces unequal data partitioning (UEDP) into microbatch-based pipeline parallelism. In UMPIPE, different processes in DNN can have different microbatch-sizes. UEDP is beneficial for layers' processes flexibly selecting appropriate numbers of data partitions. For the convenience of discussion, we regard all computations and communications in training DNN to belong to a set of processes. Based on the characteristics of training DNN, there are $4N-2$ processes for N stages, marked as $\kappa = \langle \kappa_1, \kappa_2, \ldots, \kappa_{4N-2} \rangle$ according to the order of execution. Therefore, we can obtain that for $k \leq 2N-1$, κ_k corresponds to the forward computation of the $(k+1)/2$-th stage if k is an odd number, else it is the forward communication process between the $k/2$-th and the $\left(\frac{k}{2}+1\right)$-th stages. It is similar to backward propagation. Assuming the number of microbatches in the k-th forward process is p_k and that in the k-th backward process is q_k, the relationships can be obtained as (1).

$$\begin{cases} F_k(p_k) = \begin{cases} F^P_{\frac{k+1}{2}}(p_k), & k \mod 2 = 1 \\ F^M_{\frac{k}{2}}(p_k), & \text{otherwise} \end{cases} \\ B_k(q_k) = \begin{cases} B^P_{N-\frac{k-1}{2}}(q_k), & k \mod 2 = 0 \\ B^M_{N-\frac{k}{2}}(q_k), & \text{otherwise} \end{cases} \end{cases} \tag{1}$$

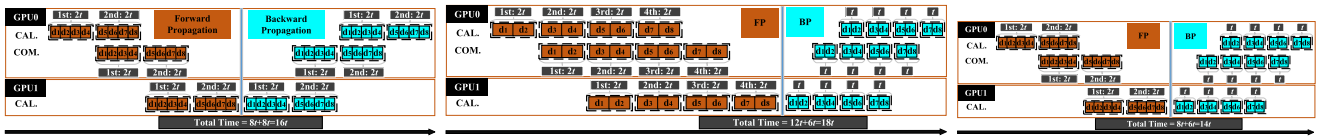
With p_k and q_k, we can give the mathematical definitions of GPipe and UMPIPE to highlight their differences.

- In GPipe, p_k and q_k must satisfy that for $0 \leq \forall i \leq \forall j \leq 2N-1$, $p_i = p_j = q_i = q_j = p$.
- In UMPIPE, it is allowed that $\exists i, j \; s.t. \; p_i \neq p_j$, or $q_i \neq q_j$, or $p_i \neq q_j$.



(a) GPipe (2 microbatches): $8t$ (b) GPipe (4 microbatches): $10t$ (c) UMPIPE (2-4-4 microbatches): $7t$

Fig. 2. FP timeline for an example of a network under GPipe and UMPIPE in two stages with time functions as: $F_1^P(2) = F_1^P(4) = 2t$ (non-linear time-consuming process), $F_1^M(2) = 2F_1^M(4) = 2t$, $F_2^P(2) = 2F_2^P(4) = 2t$, $F_1^P(1) = F_1^M(1) = F_2^P(1) = 4t$. Mbh means microbatch.



(a) GPipe (2 microbatches): $8t + 8t = 16t$ (b) GPipe (4 microbatches): $12t + 6t = 18t$ (c) UMPIPE (FP: 2 microbatches, BP: 4 microbatches): $8t + 6t = 14t$

Fig. 3. Timeline with FP and BP for an example of a network under GPipe and UMPIPE in two stages with time functions as:: $F_1^P(2) = F_1^P(4) = 2t$, $F_1^M(2) = F_1^M(4) = 2t$, $F_2^P(2) = F_2^P(4) = 2t$ and $B_1^P(2) = 2B_1^P(4) = 2t$, $B_1^M(2) = 2B_1^M(4) = 2t$, $B_2^P(2) = 2B_2^P(4) = 2t$.

As mentioned above, some time-consuming processes in GPipe slow down the entire training. It is mainly because F_k (or B_k) and p_k (or q_k) are not necessarily inversely proportional in actual training. In some cases, there may exist a number p $s.t.$ that $F_k(x) = F_k(y)$ for $\forall x > y \geq p$. A similar phenomenon appears in B_k. If different processes choose different partitions in this case (i.e., applying UMPIPE), it may further reduce the total training time, which will be better than all partition schemes of GPipe.

To illustrate the advantages of UEDP between different processes in forward propagation, Fig. 2 provides an example for forward propagation with two stages to present the differences between EDP-based pipeline (GPipe) and UMPIPE. Paying attention to the fact that $\exists p$ $s.t.$ $F_k(x) = F_k(y)$ for $\forall x > y \geq p$ in some cases, the example in Fig. 2 sets a non-linear time function for the first layer: when $p \geq 2$, the computation time of each microbatch in the first layer will no longer decrease correspondingly, i.e., $F_1^P(2) = F_1^P(4)$. In such cases, parallel training adhering to EDP will be slowed down by the non-linear time-consuming process that will become the bottleneck process of the entire EDP-based parallelism In Fig. 2(a) and (b), two schemes of GPipe, dividing one minibatch into two microbatches and four microbatches respectively, correspond to $8t$ and $10t$ training time. Due to the existence of non-linear time-consuming process, EDP-based parallelism cannot fully unleash the potential of parallel training. Using UEDP, UMPIPE in Fig. 2(c) can achieve less training time as $7t$, accelerating the speed by 14.29% compared to GPipe.

UEDP between forward propagation and backward propagation can also reduce training time. An example is shown in Fig. 3 which also takes nonlinear time-consuming processes into account by setting $F_1^P(2) = F_1^P(4)$, $F_1^M(2) = F_1^M(4)$, and $F_2^P(2) = F_2^P(4)$. It can be noted that the training time for executing the BP in Fig. 3(b) is $6t$ less than that in Fig. 3(a). Thus, if combining the scheme of forward propagation in Fig. 3(a) and the scheme of backward propagation in Fig. 3(b), a better scheme of UMPIPE can be obtained as Fig. 3(c). According to Fig. 3(c), the training time under UMPIPE parallelism is $14t$ better than the best scheme of GPipe.

These two examples demonstrate that unequal data partitioning of UMPIPE can introduce better solutions than EDP, which can further improve the speed of parallel training. Data parallelism is one of the foundations of most existing parallel architectures. Therefore, UEDP, which improves data parallelism, is significant for parallel training.

B. Formulas for UMPIPE

Due to the introduction of UEDP, the training time of DNN using UMPIPE is not as easy to derive as using GPipe. However, solving optimization solutions requires evaluation of the optimized solutions inevitably. Therefore, in this section, we present a recurrence formula and provide a time-consuming recurrence calculation algorithm.

Setting the begin time of the i-th microbatch in the k-th process is T_{ki}, a recurrence formula for UMPIPE is

$$T_{ki} = \max\left(T_{(k-1)x_i} + F_{k-1}, T_{k(i-1)} + F_k\right) \tag{2}$$

where $x_i = \lceil (ip_{k-1})/p_k \rceil$ where $\lceil \cdot \rceil$ is upward rounding function. It is complex to derive its analytical expression of the training time. In practice, we can calculate the end time of each data based on the dependencies between each data in each process of DNN. Denoting the end time of the j-th data in the k-th process as E_{kj}, the formula of dependencies is:

$$E_{kj} = \max\left(E_{(k-1)z_{kj}}, E_{ky_{kj}}\right) + F_k \tag{3}$$

Algorithm 1: Recurrence Algorithm for the Forward Training Time of One Minibatch Under UMPIPE.

Input : $F_k(p_k)$ and p_k for $\forall k$, minibatch-size P
Output: E_{kj} for $1 \le k \le 2N-1$ and $1 \le j \le P$
Initial $E_{kj} = 0$ for $0 \le k \le 2N-1$ and $0 \le j \le P$
for $k \in [1, 2N-1]$ **do**
  for $j \in [1, P]$ **do**
    Calculate z_{kj} and y_{kj}
    Calculate E_{kj} according to Eq. 3

where $y_{kj} = p_k \cdot (\lceil j/p_k \rceil - 1)$ and $z_{kj} = \min(p_k + y_{kj}, P)$. According to the (3), E_{kj} is only determined by $E_{(k-1)j}$ and $E_{ky_{kj}}$. Thus, we can use the recurrence formula with two layers of loops to calculate the training time under UMPIPE as Algorithm 1 assuming F_k is known.

As the backward propagation under UMPIPE parallelism has a similar process with forward propagation, Algorithm 1 also applies to backward propagation, which only needs to replace E_{kj} by R_{kj}, F_k by B_k and p_k by q_k. Although Algorithm 1 can be utilized to obtain the training time under UMPIPE parallelism, it will consume a plethora of computational time due to its two layers of loops, especially in the genetic algorithm. In the next section, when introducing the optimization algorithm for solving UMPIPE's scheme, we will detail the improvement methods for Algorithm 1 to accelerate the calculation of training time.

When F_k and B_k are given for $\forall k$ and $\forall p_k$, $\langle p_1, p_2, \ldots, p_{2N-1}, q_1, q_2, \ldots, q_{2N-1}\rangle$ will determine the final training time under UMPIPE parallelism. Therefore, $\langle p_1, p_2, \ldots, p_{2N-1}, q_1, q_2, \ldots, q_{2N-1}\rangle$ can be regarded as the solution of UMPIPE's scheme. Then, we can obtain the optimization problem of UMPIPE as

$$\min \omega = E_{(2N-1)P} + R_{(2N-1)P} \tag{4}$$

where $E_{(2N-1)P}$ is determined by $\langle p_1, p_2, \ldots, p_{2N-1}\rangle$ and $R_{(2N-1)P}$ by $\langle q_1, q_2, \ldots, q_{2N-1}\rangle$.

C. Analysis for Optimality of UMPIPE

According to properties of basic functions and GPipe, we can obtain a theorem about the optimal partition number.

Theorem 1: Assuming the set of optional partitions is $\langle \beta_1, \beta_2, \ldots, \beta_\eta\rangle$ where $\beta_1 = 1 < \beta_2 < \cdots < \beta_\eta = P$ and the optimal partition number of GPipe is β_α where $\alpha < \eta$, that means $p_k = \beta_\alpha$ for $\forall k$, there must exist γ $s.t.$ $F_\gamma(\beta_\alpha) = F_\gamma(\beta_{\alpha+1})$.

Theorem 1 can be proved by contradiction.

Proof 1: According to the property of basic function, the following relationship is tenable for $\forall \gamma$.

$$F_\gamma(\beta_\alpha) \ge F_\gamma(\beta_{\alpha+1})$$

If $F_\gamma(\beta_\alpha) > F_\gamma(\beta_{\alpha+1})$, it can be derived that the partition $\beta_{\alpha+1}$ is better than β_α. It contradicts that β_α is the optimal partition number. Thus, Theorem 1 is proved.

With Theorem 1, we can obtain a relationship between the optimal solutions of GPipe and UMPIPE.

Theorem 2: (1) If $F_\gamma(\beta_{\alpha-1}) > F_\gamma(\beta_\alpha) = F_\gamma(\beta_{\alpha+1})$ for $\forall \gamma$, then $p_k = \beta_\alpha$ is also the optimal solution of UMPIPE, and the best scheme of GPipe equals to that of UMPIPE. (2) If p_k is the optimal solution of UMPIPE and better than GPipe, then there must exist $i \ne j$ $s.t.$ $p_i \ne p_j$.

Theorem 2 reveals that the theoretical optimal training scheme of UMPIPE must be no worse than that of GPipe.

Proof 2: For the first property of Theorem 2, we use the inductive method to prove it. Assuming the number of processes in DNN is M, for $M = 1$, the first property of Theorem 2 is obviously tenable. For $M = 2$, it can be assumed that the optimal solution of UMPIPE is $\langle p_1 = a_1, p_2 = a_2\rangle$. As the GPipe belongs to UMPIPE, the solution of UMPIPE must be no worse than GPipe. When $a_1 = a_2$, the optimal solution of UMPIPE is also that of GPipe. In this case, Theorem 2 is tenable. For $a_1 \ne a_2$, when only one of a_1, a_2 equals to β_α, it can be set $a_1 = \beta_\alpha$ and $a_2 \ne \beta_\alpha$. As $F_\gamma(\beta_{\alpha-1}) > F_\gamma(\beta_\alpha) = F_\gamma(\beta_{\alpha+1})$, thus it can be derived that $\langle p_1 = a_1, p_2 = a_2\rangle$ is worse than $p_1 = \beta_\alpha = p_2$. When $a_1 \ne \beta_\alpha$ and $a_2 \ne \beta_\alpha$, it can be proved that $p_1 = a_1, p_2 = a_2$ must be worse than $(p_1 = \beta_\alpha, p_2 = a_2)$ or $(p_1 = a_1, p_2 = \beta_\alpha)$. $(p_1 = \beta_\alpha, p_2 = a_2)$ or $(p_1 = a_1, p_2 = \beta_\alpha)$ are both worse than $(p_1 = \beta_\alpha, p2 = \beta_\alpha)$. Thus, $(p_1 = a_1, p_2 = a_2)$ is worse than $(p_1 = \beta_\alpha, p_2 = \beta_\alpha)$. Therefore, the first property is tenable for $M = 2$.

Assuming the first property of Theorem 2 is tenable for $\forall M \le K-1$. For $M = K$, we can assume there exists a solution of UMPIPE better than $p_k = \beta_\alpha$. Thus, there must exist one piece of data at one process with an earlier ending time in the scheme of UMPIPE than that of GPipe and its previous microbatches all have the same ending time in UMPIPE as that of GPipe. It can be set that the index of this process is $l \le K$. Thus, for $\forall k \le l-1$, $p_k = \beta_\alpha$ and $p_l \ne \beta_\alpha$ according to the assumption that Theorem 2 is tenable for $\forall M \le K-1$. As $F_\gamma(\beta_{\alpha-1}) > F_\gamma(\beta_\alpha) = F_\gamma(\beta_{\alpha+1})$, thus $\forall p_l \ne \beta_\alpha$ the ending time of any piece data in the l-th process must be not earlier than that of $p_l = \beta_\alpha$. Thus, Theorem 2 is tenable for $M = K$. Thus, the first property is proved.

As the first property is tenable, the second property clearly holds, otherwise, it contradicts the condition that the solution of UMPIPE is better than that of GPipe.

In fact, all the training schemes of GPipe are special cases of UMPIPE, which means the schemes of UMPIPE include that of GPipe. Therefore, our algorithm considers using the optimal solution of GPipe as the initial solution, which ensures to obtained UMPIPE's scheme that is not inferior to GPipe.

IV. Improved Genetic Algorithm for UMPIPE

This paper focuses on the improvement of unequal data partitioning and optimization algorithm, therefore mainly considering optimization of data partitioning without considering the model partitioning optimization problem of PMP. It can be assumed that the minibatch-size P has Q possible cases for partitioning. Therefore, GPipe also has Q feasible solutions as it requires EDP. As UMPIPE allows UEDP for different processes, it has Q^{4N-2} feasible solutions, which are far more than that of GPipe and increase exponentially with the number of stages. In

the previous section, we analyze that as the solution set expands, UMPIPE has a better theoretical optimal solution. Leveraging dynamic programming, recurrence algorithm or enumerate algorithm requires large computational complexity. Thus, one of the keys is to find an algorithm with acceptable complexity that can find a better solution than the GPipe optimal solution.

As $\langle p_1, p_2, \ldots, p_{2N-1}, q_1, q_2, \ldots, q_{2N-1}\rangle$ can be regarded as the solution of UMPIPE, we consider using the genetic algorithm to search the optimal solution of (4). In genetic algorithms, the fitness of each individual is related to the training time corresponding to its solution. According to Algorithm 1, calculating the fitness of all individuals in each generation will require three layers of loops, which consume a large amount of time. Therefore, we also propose a method to eliminate loops through GPU-based matrix operations, significantly improving the search speed of genetic algorithms.

A. DGAP: Dual Chromosomes-Based Genetic Algorithm

Referring to the existing terms of genetic algorithms [37], [38], the base components of DGAP are set as:

1) *Gene, Chromosome and Individual*: we regard the partition number p_k or q_k of each process as a gene. We set each individual has two chromosomes: first is a vector $C = \langle p_1, p_2, \ldots, p_{2N-1}\rangle$ and the second is a vector $D = \langle q_1, q_2, \ldots, q_{2N-1}\rangle$ both with $2N-1$ genes corresponding to a data partition scheme of the UMPIPE.
2) *Fitness and Chromosomes selector*: An individual's fitness equals the training time of DNN corresponding to individual's genes-determined partition scheme under UMPIPE parallelism (called time corresponding to the individual). The fitness of the chromosome C is equal to its corresponding forward propagation time $E_{(2N-1)P}$ and that of D is to backward time $R_{(2N-1)P}$. DGAP sorts the two sets of chromosomes of all individuals separately and then selects better parts of each set of chromosomes in pairing and crossover respectively.
3) *Crossover*: In this paper, we set the crossover to occur between four chromosomes C_α, C_β, D_γ, D_η. Their crossover is defined as separately extracting a part of genes from them to gain two new chromosomes $C^{(new)}$ and $D^{(new)}$ to construct the children individual.
4) *Mutation*: Mutation is replacing some elements of a chromosome by randomly generated genes.
5) *Population regeneration mechanism*: The genetic algorithm for UMPIPE applies elitist strategy [38] to combine the parent individuals with their children individuals to jointly compete to produce the next generation.

Then, we can present the dual-chromosome genetic algorithm for UMPIPE in Algorithm 2.

Algorithm 2 is a version calling Algorithm 1 to calculate the fitness (i.e., training time) of each individual, it requires four layers of loops, including loop in generation index, loop in individual index i, loop in DNN's process index k and loop in input data index j. Due to too many loop layers, Algorithm 2 will take a long time to search for an optimal solution with time complexity $\mathrm{O}(N_g \cdot P \cdot (N_p + N_c) \cdot (4N-2))$, which is mainly consumed in the calculation of fitness. To enable genetic algorithms feasible to solve the optimal data partitioning in practical applications, it is necessary to reduce the time spent on each iteration of the genetic algorithm. We will introduce the improvement way to accelerate DGAP subsequently. Before that, we analyze the convergence of the dual chromosome strategy theoretically.

Algorithm 2: Dual-Chromosome Genetic Algorithm for UMPIPE (DGAP).

Input : $F_k(p)$ and $B_k(q)$ for $\forall k, p, q$. P, N_p, N_g and N_c. $W = [w_0, w_1, \ldots, w_{Q-1}]$ of optional partitions, i.e., $p \in W$ and $q \in W$.
Output: $\langle p_1, p_2, \ldots, p_{2N-1}, q_1, q_2, \ldots, q_{2N-1}\rangle$

```
Set Q individuals both with UEDP and randomly initial the rest N_p − Q individuals
for generation ∈ [1, N_g] do
    for i ∈ [1, N_p] do
        Call Algorithm 1 to obtain the fitness of the i-th individual
    for i ∈ [1, N_c] do
        Select and Pair chromosomes C_α, C_β, D_γ and D_η
        Execute crossover and mutation to generate children chromosomes C^(new) and D^(new)
        Call Algorithm 1 to obtain the fitness of the i-th children individual
    Sort children and parents, and retain the best N_p individuals as the next generation
Set the genes of the individuals with the best fitness as solution ⟨p_1, p_2, ..., p_{2N-1}, q_1, q_2, ..., q_{2N-1}⟩
```

B. Analysis of Convergence for Dual-Chromosomes Strategy

In one minibatch, all the backward propagation need to start after the ending of all forward propagation, so the optimization of ω in (4) can be divided into two independent objectives $\min E_{(2n-1)P}$ and $\min R_{(2n-1)P}$. Correspondingly, we set one individual to have two chromosomes C and D in DGAP.

Next, we can analyze the convergence probability in each generation to demonstrate the superiority of two chromosomes compared to using one chromosome.

It can be obtained that the number of feasible solutions for UMPIPE is Q^{4N-2}. We can set all genes of individuals in each generation to be randomly generated again without considering the evolutionary ability of the crossover and mutation in genetic algorithms. Then, the probability of obtaining the theoretically optimal individual in one generation is $\phi_1 = N_p/Q^{4N-2}$ assuming each generation has N_p different individuals and there is only one theoretical solution. Thus, for one chromosome, the probability of obtaining the global optimal solution at the g-th generation is $(1-\phi_1)^{g-1} \cdot \phi_1$. Therefore, the expectation of generations required to achieve theoretically optimal individuals is Q^{4N-2}/N_p.

For two chromosomes, the probabilities of obtaining the theoretically optimal C and D in one generation are both $\phi_2 = N_p/Q^{2N-1}$. The probability of obtaining the global optimal C at the g-th generation is $(1-\phi_2)^{g-1} \cdot \phi_2$, which is same to D. Therefore, the expectation of generations required to achieve theoretical optimal C chromosome is $1/\phi_2$. As C and D are

independent, the expectation of generations required to both achieve theoretical optimal individual for dual-chromosomes-based genetic algorithm (DGAP) is $2Q^{2N-1}/N_p$ which is far less than that of one-chromosome-based genetic algorithm.

This conclusion is also valid when considering the evolutionary ability of crossover and mutation in genetic algorithms. For the general situation considering crossover and mutation, it can be proved that the dual-chromosome-based genetic algorithm has a higher probability of reaching the optimal solution for each generation than the one-chromosome-based genetic algorithm. Therefore, the solution of DGAP in each generation will be statistically superior to GAP.

C. OiDGAP: One-Level Improved DGAP

To accelerate DGA for UMPIPE, we improve Algorithm 2 from two aspects, including eliminating the loop in the individuals' index (the "for loop" of line 3 and line 5 in Algorithm 2) and eliminating the loop in input data's index (the "for loop" of line 3 in Algorithm 1).

Eliminating the loop in the individual index indicates simultaneously calculating the training time corresponding to multiple individuals. It can be set that the available partition numbers construct a one-dimensional array $W = [w_0, w_1, \ldots, w_{Q-1}]$; the corresponding time functions of each process in DNN construct two arrays $F = \{f_{kj}\}_{(2N-1)\times Q}$ and $B = \{b_{kj}\}_{(2N-1)\times Q}$ with two-dimensions where $f_{kj} = F_{k+1}(w_j)$ and $b_{kj} = B_{k+1}(w_j)$ for $0 \le k < 2N-1$ and $0 \le j < Q$; the indexes for all individuals' genes in array W construct a two-dimensional array $G = \{g_{ik}\}_{N_p\times(4N-2)}$ which means the partition number of the $(k+1)$-th process determined by the $(i+1)$-th individual' genes is

$$W[g_{ik}] = w_{g_{ik}} = \begin{cases} p_{(i+1)(k+1)}, & \text{if} \quad k < 2N-1 \\ q_{(i+1)(k+2-2N)}, & \text{others} \end{cases} \tag{5}$$

Therefore, when W is given, the array G is equivalent to genes in the genetic algorithm. As the calculations for the training time of FP and BP have similarities, we only discuss the calculation for the training time of FP. The array, composed of the k-th gene of all individuals, can be obtained as $G[:,k]$ where $0 \le k < 2N-1$, thus the array of its corresponding time functions of all individuals are $F[k, G[:,k]]$ and that of the partition numbers are $W[G[:,k]]$. The microbatch-sizes of the k-th process for all individuals are $\left\lceil \frac{P}{W[G[:,k]]} \right\rceil$. It can be assumed that the end time of the j-th data in the k-th forward process for the $(i+1)$-th individual is e_{ijk} which can construct a three-dimensional array $E = \{e_{ijk}\}_{N_p\times(P+1)\times 2N}$. Thus, the recurrence relationship for calculating the fitness of multiple individuals simultaneously can be derived as (6) where the array $Z^{(b)}$ indicates the starting index of data whose initial value is $[1,1,\ldots,1]_{N_p}$ and $Z^{(e)}$ is end index.

$$\begin{cases} Z^{(e)} = \min\left(Z^{(b)} + \lceil P/W[G[:,k]] \rceil, P+1\right) \\ E\left[:, Z^{(b)}[:] : Z^{(e)}[:], k\right] = F[k, G[:,k]] \\ \qquad + \max\left(E\left[:, Z^{(e)}[:]-1, k-1\right], E\left[:, Z^{(b)}[:]-1, k\right]\right) \\ Z^{(b)} = Z^{(e)} \end{cases} \circlearrowleft \tag{6}$$

Algorithm 3: One-Level Improved DGA for UMPIPE (OiDGAP): Simultaneously Calculating the Training Time Corresponding to Multiple Individuals.

Input : $F = \{f_{kj}\}_{(2N-1)\times Q}$, $B = \{b_{kj}\}_{(2N-1)\times Q}$, P, $W = [w_0, w_1, \ldots, w_{Q-1}]$. N_p, N_g and N_c.
Output: $G\left[\arg\min\left(H^{(C)} + H^{(D)}\right)\right]$

1 Set Q individuals both with UEDP and randomly initial the rest $N_p - Q$ individuals
2 Set $T = \{t_{ij} = j\}_{N_p\times(P+1)}$, $E = \{e_{ijk} = 0\}_{N_p\times(P+1)\times 2N}$ and $R = \{r_{ijk} = 0\}_{N_p\times(P+1)\times 2N}$
3 **for** $k \in [0, 2N-2]$ **do**
4 Initial $Z^{(b)} = [1,1,\ldots,1]_{N_p}$
5 **while** $\min\left(Z^{(b)}\right) < P+1$ **do**
6 Calculate $Z^{(b)}$ and $Z^{(e)}$ according to Eq. 6
7 Use the similar way as line 3-6 to calculate R
8 Obtain the arrays of fitness of $H^{(C)} = E[:, P+1, 2N]$ and $H^{(D)} = R[:, P+1, 2N]$
9 **for** $generation \in [1, N_g]$ **do**
10 **for** $i \in [1, N_c]$ **do**
11 Execute crossover and mutation to obtain children $\bar{G}$
12 Obtain $\bar{H}^{(C)}$ and $\bar{H}^{(D)}$ of children with similar way as line 3-8 ; // Two-layer loops
13 Concatenate children and parent individuals as $G = \text{concat}(G, \bar{G})$, $H^{(C)} = \text{concat}\left(H^{(C)}, \bar{H}^{(C)}\right)$, $H^{(D)} = \text{concat}\left(H^{(D)}, \bar{H}^{(D)}\right)$
14 Update population as: $U = \text{argsort}\left(H^{(C)} + H^{(D)}\right)$, $G = G[Sort[:N_p]]$ with its fitness $H^{(C)} = H^{(C)}[U[:N_p]]$ and $H^{(D)} = H^{(D)}[U[:N_p]]$

Due to the possible differences in the number of partitions corresponding to the same process for multiple individuals, the number of columns required to be broadcast in each row of the second formula of (6) will be different. It can be achieved by using matrix multiplication if the programming with array operation does not support imbalanced broadcasting. Then, the genetic algorithm for UMPIPE after one level of improvement can be shown in Algorithm 3. Assuming executed on one or multiple GPUs with sufficient parallel capability, Algorithm 3 has the time complexity as $\mathrm{O}(N_g \cdot P \cdot (4N-2))$.

D. TiDGAP: Two-Level Improved DGAP

Eliminating the loop in input data indicates simultaneously calculating the end time of multiple microbatches (all data in one minibatch of the k-th process), i.e., obtaining $E[i,:,k]$ after one set of operations without loop.

As shown in (3), the start time of the j-th data in the k-th process mainly depends on the maximum end time between the z_{kj}-th data in the $(k-1)$-th process (called preprocess baseline) and the y_{kj}-th data in the k-th process (called preorder baseline). Therefore, we can divide the start time of data in the k-th process into p_k parts (corresponding to p_k microbatches), where all data in the same part has the same start time. Thus, we only need to quickly calculate the start time of one data in each part to know the start time of other data in the same part. Assuming the end time of each data in the $(k-1)$-th process is

known, we can obtain the preprocess baseline for each microbatch in the k-th process is $[E[i, \lceil \frac{P}{p_k} \rceil, k-1], E[i, 2\lceil \frac{P}{p_k} \rceil, k-1], \ldots, E[i, P, k-1]]$ that can be set as $[\eta_1, \eta_2, \ldots, \eta_{p_k}]$ for the sake of presentation. Therefore, the start time of the 1st microbatch in the k-th process is η_1, and that of the 2nd microbatch is $\max(\eta_1 + f, \eta_2)$ where f means the time function. By mathematical induction, we derive a provable property that the start time of the j-th microbatch in the k-th process is

$$\max_{l=1}^{j}(\eta_l + (j-l)f) = \max_{l=1}^{j}(\eta_l - lf) + jf \tag{7}$$

Thus, we only need to calculate $\max_{l=1}^{j}(\eta_l - lf)$ for each j. We can construct a array as $PB = [\eta_1 - f, \eta_2 - 2f, \ldots, \eta_j - jf, \ldots, \eta_{p_k} - p_k f]$. Significantly, $\max_{l=1}^{j}(\eta_l - lf)$ is exactly the maximum value of the first j items for the array PB, which can be quickly obtained by calling the 'cummax' function of array operation or using upper triangular matrix (if without 'cummax' function). Then, $\mathrm{cummax}(PB) + [1, 2, \ldots, p_k]f$ is the array composed of the start time of each microbatch in the k-th process.

This approach still applies to calculating the starting time of multiple data of multiple individuals simultaneously by combining with Algorithm 3, which can simultaneously eliminate two layers of loops in Algorithm 2 including the loop in the individuals' index (the "for loop" of line 3 and line 5 in Algorithm 2) and the loop in input data's index (the "for loop" of line 3 in Algorithm 1). Then, the two-level improved DGA for UMPIPE simultaneously calculating the starting time of multiple data and multiple individuals can be seen in Algorithm 4. The time complexity of Algorithm 4 is $\mathrm{O}(N_g \cdot (4N-2))$ which is approximately $\frac{1}{P}$ of Algorithm 3.

Our proposed TiDGAP has a much lower time complexity that provides a method to quickly calculate the training time corresponding to different data partitioning schemes and allows UMPIPE (unequal data partitions for microbatch-based pipeline parallelism) to apply in the practical optimization and acceleration of parallel training. With the increase of N_p and P, the execution time of TiDGAP in real GPU devices will also increase slowly, while it is still small enough for optimization of parallel training for large-scale DNN. In subsequent experiments, we will also evaluate the execution time of TiDGAP compared to DGAP and OiDGAP.

Algorithm 4: Two-Level Improved DGA for UMPIPE (TiDGAP): Simultaneously Calculating the Starting Time of Multiple Data and Multiple Individuals.

Input : F, B, P, W. N_p, N_g and N_c.
Output: $G\left[\arg\min\left(H^{(C)} + H^{(D)}\right)\right]$

1 Set Q individuals both with UEDP and randomly initialize the rest $N_p - Q$ individuals
2 Set $T = \{t_{ij} = j\}_{N_p \times (P+1)}$, $E = \{e_{ijk} = 0\}_{N_p \times (P+1) \times 2N}$ and $R = \{r_{ijk} = 0\}_{N_p \times (P+1) \times 2N}$, $Z^r = \{t_{ij} = i\}_{N_p \times (P+1)}$.reshape(shape $= (-1,)$)
3 **for** $k \in [0, 2N-2]$ **do**
4 Set $A = \{0\}_{N_p \times (P+1)}$ and $M = \lceil T/W\,[G[:,k]] \rceil$
5 Obtain the index of the reference end time in the previous process as $Z = M * (W\,[G[:,k]])$
6 Calculate the basic time for each individual of current p_k as $M_1 = M * (F[k, G[:,k]])$
7 Obtain the array of PB as $PB = E[k][Z^r, Z]$.reshape(shape $= (N_p, P+1)) - M_1$
8 $A[:, 1:] = \mathrm{cummax}\,(PB[:, 1:], dim = 1)$
9 $E\,[k+1] += A + (M+1) * F[k, G[:,k]]$
10 Use the similar way as line 3-9 to calculate R
11 Obtain the arrays of fitness of $H^{(C)} = E\,[:, P+1, 2N]$ and $H^{(D)} = R\,[:, P+1, 2N]$
12 **for** *generation* $\in [1, N_g]$ **do**
13 Use array operation to execute the crossover and mutation to generate children $\bar{G}$
14 Obtain fitness $\bar{H}^{(C)}$ and $\bar{H}^{(D)}$ of children with similar way as line 3-9 ; `// One loop`
15 Concatenate children and parents, update individuals ; `// Same as line 13-14 of Algorithm 3`

V. Performance Evaluation

A. Experiment Settings

For the sake of the comprehensive evaluations of the unequal data partitions-based parallelism (i.e., UMPIPE) and two-level improved dual-chromosome genetic algorithm (TiDGAP), we carry out four groups of experiments from various aspects including:

1) EX_1: Comparing TiDGAP with TiGAP to demonstrate the optimization effect of dual-chromosome strategy;
2) EX_2: Comparing TiDGAP with OiDGAP and DGAP to demonstrate the acceleration effect on the evolution of two-level improvement with array operation;
3) EX_3: Comparing TiDGAP with baselines local greedy algorithm and global greedy-based dynamic programming to demonstrate the optimality of DGAP for UMPIPE.
4) EX_4: Comparing UMPIPE with GPipe, UMPipeDream and PipeDream to demonstrate the superiority of UEDP.

EX_1 and EX_2 are conducted on a randomly generated simulation dataset, which is beneficial for executing sufficient experiments. EX_3 and EX_4 are conducted on real parallel training in multiple GPUs, which is conducive to demonstrating the advantages and feasibility simultaneously of our proposed UMPIPE architecture and TiDGAP algorithm. The baseline algorithms in EX_3 represent that:

1) Local Greedy Algorithm for UMPIPE (LG): For each process, select the number of partitions that make the current process have the earliest end time. The algorithm can be seen in Algorithm 5.
2) Global Greedy-based Dynamic Programming for UMPIPE (GG): For each process, selecting the number of partitions that make the last process have the earliest end time, whose algorithm can be seen in Algorithm 6. In Algorithm 6, the parameter 'rounds' can be flexibly set and can also be replaced by that $\nexists C'$ better than C which is a convergence condition of Algorithm 6.

Each group of experiments adopts the control variables to ensure the reliability of comparisons. In EX_1, the indicators are the optimization results over generations and the probability of finding the global optimal solution over generations

Algorithm 5: Local Greedy for UMPIPE (LG).

Input : $F_k(p)$, $B_k(q)$; P; $W = [w_0, w_1, \ldots, w_{Q-1}]$
Output: $\langle p_1, p_2, \ldots, p_{2N-1}, q_1, q_2, \ldots, q_{2N-1} \rangle$

```
for k ∈ [0, 4N − 3] do
    if k < 2N − 1 then
        Choose the p_k in W which makes E_kP to take the minimum value
    else
        Choose the q_k in W which makes R_kP to take the minimum value
```

Algorithm 6: Global Greedy-Based Dynamic Programming for UMPIPE (GG).

Input : $F_k(p)$, $B_k(q)$; P; $W = [w_0, w_1, \ldots, w_{Q-1}]$
Output: $\langle p_1, p_2, \ldots, p_{2N-1}, q_1, q_2, \ldots, q_{2N-1} \rangle$

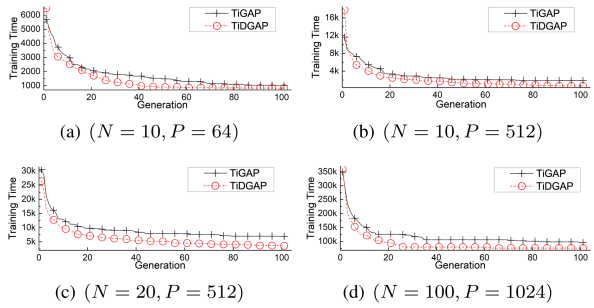
```
(Randomly or specifically) initial a partition scheme as C = ⟨p_1, ..., p_{2N−1}⟩ and D = ⟨q_1, ..., q_{2N−1}⟩
for i < rounds do
    for k ∈ [0, 4N − 3] do
        if k < 2N − 1 then
            Choose the p'_k in W s.t. E_{(2N−1)P}(C') = min_{l=0}^{Q−1} (E_{(2N−1)P}(C|_{p_k=w_l})) where C|_{p_k=w_l} means only changing the partition number of the k-th process to w_l
            Update p_k = p'_k, C = C'
        else
            Choose the q'_k in W to make R_{(2N−1)P}(D') take the minimum value
            Update q_k = q'_k, D = D'
```

to demonstrate the convergence of dual-chromosome strategy, where the optimization results present the parallel training time corresponding to the optimization solutions. EX_2 is mainly used to observe the execution time of different algorithms controlling the number of individuals (N_p) and generations (N_g) in different scales. EX_3 is to observe the stable optimization results of the different algorithms. EX_4 is to observe the optimal training time and convergence under UMPIPE parallelism, compared with other parallelism. We test a large number of instances in each group of experiments, and instances from the same group of experiments point to similar conclusions. Therefore, we only provide a subset of them in this paper. Then, the optimization algorithm is launched on a desktop and the realistic parallel training is launched on the servers. The configurations of them are as follows.

- Program version: Python 3.7 + Pytorch 1.13.1;
- Desktop: NVIDIA GeForce RTX 3060 Ti @ 8GB;
- Servers: NVIDIA TESLA V100 @ 32GB × 2.

B. EX_1: Evaluation of Dual-Chromosome Strategy of TiDGAP Compared With TiGAP

To observe the optimization effect of the dual-chromosome strategy, we compare TiDGAP with TiGAP in the simulation scenarios. In the simulation scenarios, we randomly generate $F = \{f_{kj}\}_{(2N-1)\times Q}$ and $B = \{b_{kj}\}_{(2N-1)\times Q}$. As the outcomes from different random distributions echo the consistent trends and results, we only present the results obtained from the uniform distribution $\mathrm{U} = [1, 100] \cap \mathbb{N}^*$.



(a) $(N = 10, P = 64)$ (b) $(N = 10, P = 512)$

(c) $(N = 20, P = 512)$ (d) $(N = 100, P = 1024)$

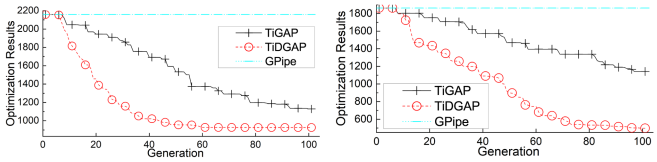
Fig. 4. The optimization results (corresponding to time for training one minibatch) over generations in randomly generated basic time arrays comparing TiDGAP with TiGAP, where: $N_p = 100$, $N_g = 100$, $F, B \sim \mathrm{U} = [1, 100]$, randomly initializing N_p individuals.

TABLE II
THE QUANTITATIVE OPTIMIZATION RESULTS OF TIDGAP AND TIGAP AT THE 100TH GENERATION IN THE EXPERIMENTS OF FIG. 4

Scenarios	TiDGAP	TiGAP	$\varepsilon_{\frac{\text{TiGAP}}{\text{TiDGAP}}}$
$(N = 10, P = 64)$	833	1020	18.33%
$(N = 10, P = 512)$	671	1865	64.02%
$(N = 20, P = 512)$	3601	6909	47.88%
$(N = 100, P = 1024)$	75765	95419	20.60%

First, we observe the trends of optimization results over generations in several scenarios, including $(N = 10, P = 64)$, $(N = 10, P = 512)$, $(N = 20, P = 512)$ and $(N = 100, P = 1024)$. In experiments, we set the number of individuals as $N_p = 100$, and the number of generations as $N_g = 100$, and the algorithms don't set equal partitions into initial states (i.e., random initialization). Then, we plot the results in Fig. 4.

From Fig. 4, the curves of TiDGAP with dual-chromosome strategy remain lower than that of TiGAP. The unique difference between TiDGAP and TiGAP is the number of chromosomes per individual in genetic algorithms. TiDGAP and TiGAP both have $4N - 2$ genes in one individual. The two chromosomes of the individual in TiDGAP are composed of $2N - 1$ genes respectively, while the individual in TiGAP only has one chromosome. The comparative results in Fig. 4 show that the usage of the dual-chromosome strategy is beneficial for improving convergence of GAP, which is consistent with the analysis in Section IV-B. The time costs of forward and backward propagation in UMPIPE are independent mutually. Therefore, setting two chromosomes to represent forward propagation and backward propagation respectively has lower expected generations than the single-chromosome to achieve the optimal solution of UMPIPE. The statistical indicator (lower expected generations) points to better optimization results (corresponding to lower training time under UMPIPE parallelism) over the generation as Fig. 4. For the sake of observation of the quantitative comparison between TiDGAP and TiGAP in Fig. 4, we have listed the optimization results of TiDGAP and TiGAP at the 100th generation in Table II where



(a) $(N = 10, P = 64)$ (b) $(N = 10, P = 512)$

Fig. 5. The optimization results (corresponding to training time for one minibatch) over generations in randomly generated basic time arrays comparing TiDGAP with TiGAP, where: $N_p = 100$, $N_g = 100$, $F, B \sim \mathrm{U} = [1, 100]$, using the optimal GPipe solution as the initial individuals.

TABLE III
THE QUANTITATIVE OPTIMIZATION RESULTS OF TIDGAP, TIGAP AT THE 100TH GENERATION AND THAT OF GPIPE IN THE EXPERIMENTS OF FIG. 5 FOR $N = 10$, SETTING EQUAL PARTITIONS INTO INITIAL STATES

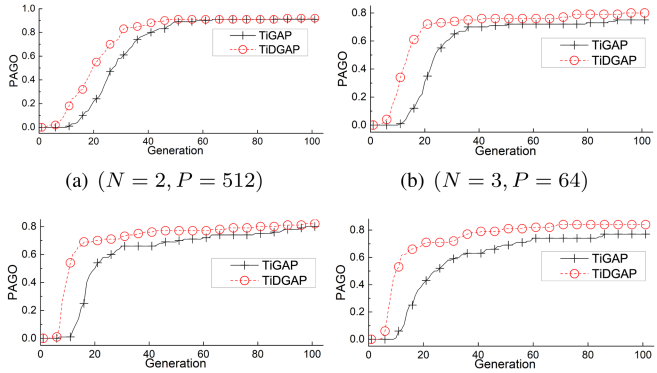
Scenarios	GPipe	TiDGAP	TiGAP	$\varepsilon^{\mathrm{TiGAP}}_{\mathrm{TiDGAP}}$	$\epsilon^{\mathrm{GPipe}}_{\mathrm{TiDGAP}}$
$P = 64$	2158	926	1129	17.98%	1.33×
$P = 512$	1863	498	1142	56.39%	2.74×

$\varepsilon^{\mathrm{TiGAP}}_{\mathrm{TiDGAP}} = \frac{\mathrm{TiGAP-TiDGAP}}{\mathrm{TiGAP}}$ means the reduction magnitude of TiDGAP in training time compared to TiGAP. From Table II, TiDGAP within 100 generations reduces the training time under UMPIPE by 18.33%, 64.02%, 47.88% and 20.60% compared to TiGAP.

The experiments in Fig. 4 do not set the solution of GPipe as one of the initial individuals. To verify the advantages of dual-chromosome strategy for UMPIPE over single chromosome have universality, we carry out experiments in two combinations of $(N = 10, P = 64)$ and $(N = 10, P = 512)$ by setting equal partitions into initial states as the line 1 in Algorithm 4. Then, we plot the optimization results over generations of TiDGAP and TiGAP in Fig. 5. In Fig. 5, we also draw a straight line paralleling to the horizontal axis to represent the optimal training time of DNN under GPipe parallelism. From Fig. 5, the convergence of TiDGAP to solve the scheme of UMPIPE is still better than that of TiGAP. This also confirms once again that the dual-chromosome strategy is more suitable for UMPIPE than the single-chromosome. Moreover, the solutions of TiDGAP and TiGAP are both better than those of GPipe, which proves that the UMPIPE architecture is superior to GPipe in the randomly generated basic time functions. This phenomenon can reflect the significance of UMPIPE. In order to evaluate the performance of TiDGAP and UMPIPE in this scenario, we list the optimization results at the 100th generation in Table III where $\epsilon^{\mathrm{GPipe}}_{\mathrm{TiDGAP}} = \frac{\mathrm{GPipe-TiDGAP}}{\mathrm{TiDGAP}}$ means the improvement ratio of TiDGAP (also representing UMPIPE) in training speed compared to GPipe.

From Table III, the parallel training scheme of UMPIPE solved by the TiDGAP algorithm has a speed increase of $1.33\times$ and $2.74\times$ respectively in $(N = 10, P = 64)$ and $(N = 10, P = 512)$ compared to GPipe, which is consistent with the analysis of UMPIPE's optimality in Section III-C.

To further evaluate the convergence of TiDGAP, we carry out experiments in small-scale scenarios including $(N = 2, P = 512)$, $(N = 3, P = 64)$, $(N = 5, P = 8)$, $(N = 12, P = 2)$. We use an enumerated algorithm to obtain the theoretical optimal solution of UMPIPE. In each scenario, we execute 100 instances,



(a) $(N = 2, P = 512)$ (b) $(N = 3, P = 64)$

(c) $(N = 5, P = 8)$ (d) $(N = 12, P = 2)$

Fig. 6. The probabilities of achieving global optimization (PAGO) over generations in randomly generated basic time arrays comparing TiDGAP with TiGAP, where: $N_p = 100$, $N_g = 100$, $F, B \sim \mathrm{U} = [1, 100]$, randomly initializing N_p individuals.

record the generations when the algorithm reaches the theoretical optimal for each instance, and calculate the probability of achieving global optimization over generations. The genetic algorithm in GAP preserves the current best individual to the next generation, so if the algorithm reaches theoretical optimal in a certain generation, it will remain theoretically optimal in all subsequent generations, and hence the count of achieving theoretical optimal in subsequent generations will be increased by 1. Then, we plot the results in Fig. 6. Obviously, TiDGAP has higher probabilities than TiGAP to obtain the theoretical optimal solution within the same generations in Fig. 6, which also proves the superiority of the dual-chromosome strategy.

C. EX_2: Evaluation of Two-Level Improvement of TiDGAP Compared With OiDGAP and DGAP

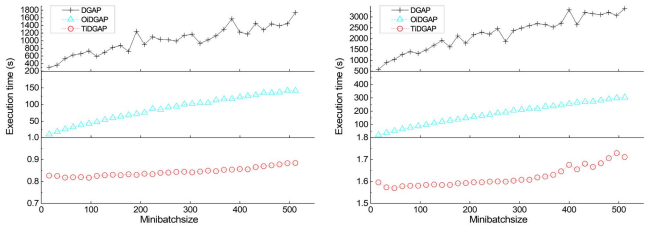
To observe the acceleration effect of the two-level improvement, we compare TiDGAP with OiDGAP and DGAP in the simulation scenarios with random basic time functions. As the two-level improvement of TiDGAP mainly eliminates two layers of loops, including the loop in the individuals' index and the loop in the input data's index, we execute experiments in four configurations with varying minibatch-size or number of individuals as follows:

- $(N = 10, P \in [1, 32] \times 16)$, $(N_p = 100, N_g = 100)$;
- $(N = 20, P \in [1, 32] \times 16)$, $(N_p = 100, N_g = 100)$;
- $(N = 10, P = 512, (N_p \in [1, 10] \times 10, N_g = 100)$;
- $(N = 20, P = 512, (N_p \in [1, 10] \times 10, N_g = 100)$.

These algorithms are executed on the Desktop.

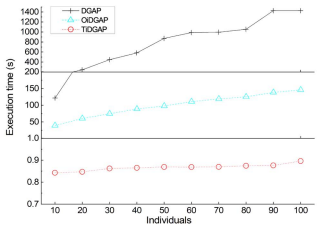
Using the changed parameters as the abscissa and the algorithm execution time as the ordinate, we plot the execution time of TiDGAP, OiDGAP and DGAP in Fig. 7.

First, the execution time of TiDGAP in Fig. 7 is significantly smaller in magnitude than OiDGAP and DGAP. Concretely, for $N = 10$ and $N_g = 100$ of Fig. 7(a) and (c), The execution time of TiDGAP is about $[0.8\mathrm{s}, 0.9\mathrm{s}]$ less than 1s; and that of OiDGAP and DGAP are respectively $[1\mathrm{s}, 150\mathrm{s}]$ and $[100\mathrm{s}, 1800\mathrm{s}]$. For $N = 20$ and $N_g = 100$ of Fig. 7(b) and (d), that of TiDGAP is about $[1.5\mathrm{s}, 1.8\mathrm{s}]$ less than 2s; and that of OiDGAP and

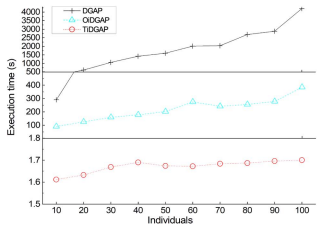


(a) $(N = 10, P \in [1, 32] \times 16)$, $(N_p = 100, N_g = 100)$

(b) $(N = 20, P \in [1, 32] \times 16)$, $(N_p = 100, N_g = 100)$



(c) $(N = 10, P = 512, (N_p \in [1, 10] \times 10, N_g = 100)$

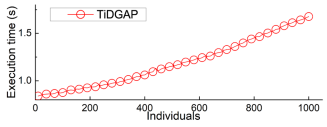


(d) $(N = 20, P = 512, (N_p \in [1, 10] \times 10, N_g = 100)$

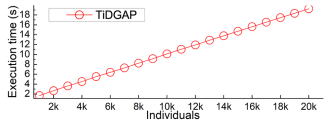
Fig. 7. The execution time of TiDGAP, OiDGAP and DGAP for solving UMPIPE in simulated scenarios launched on GeForce RTX 3060 Ti.

DGAP are respectively $[2\text{s}, 400\text{s}]$ and $[300\text{s}, 4000\text{s}]$. Second, comparing OiDGAP to DGAP can demonstrate the effectiveness of improvement to simultaneously calculate the end time corresponding to multiple individuals, as well as comparing TiDGAP to OiDGAP can demonstrate the effectiveness of improvement to simultaneously calculate the end time of multiple microbatches. Lastly, the execution time of OiDGAP is approximately proportional to both minibatch-size P and the number of individuals N_p, i.e., linearly increasing with the increase of P and N_p. DGAP also has the same phenomenon. Unlike OiDGAP and DGAP, in the scenarios of Fig. 7, the execution time of TiDGAP remains relatively stable without increasing with P and N_p. These are generally consistent with the theoretical time complexities. This indicates that within a certain range of parameters, the computing speed of TiDGAP can be maintained unaffected by P and N_p beneficial from the parallel computing ability of GPU. In detail, according to the results of Fig. 7(c) and (d), the execution speeds of TiDGAP are $(162.86\times, 226.36\times)$ of OiDGAP, and $(1590.23\times, 2473.02\times)$ of DGAP in $(N = 10, N = 20)$ for $P = 512$ and $N_p = 100$.

However, the execution time of OiDGAP in Fig. 7(c) and (d) is directly proportional to the number of individuals N_p which seems to contradict the theoretical complexity of OiDGAP but actually not. Theoretical complexity assumes that the ideal GPU has sufficient parallel capability, while the parallel capability of GPUs is not infinite in reality. When the computational complexity reaches a certain level that exceeds the maximum value that the GPU's parallel cores can carry, its computational complexity will also increase with the N_p. This property will also apply to TiDGAP. To further evaluate the execution time of TiDGAP with respect to the number of individuals, we conduct experiments in simulated scenarios of $N_p \in [10, 1000]$ and $N_p \in [1000, 20000]$



(a) $N_p \in [10, 1000]$



(b) $N_p \in [1000, 20000]$

Fig. 8. The execution time of TiDGAP for solving UMPIPE in simulated scenarios where $(N = 10, P = 512, N_g = 100)$.

TABLE IV
DETAIL OF SELF-DESIGNED CNNS

Layer	Types	Input Channel	Output Channel	Kernel
L_1	CNN	$In_1 = 1$	$Ou_1 = \text{U}(20, 50)$	
L_x	CNN	$In_x = Ou_{x-1}$	$Ou_x = \text{U}(20, 50)$	3×3
L_K	FC	$In_K = Ou_{K-1}$	$Ou_K = 10$	

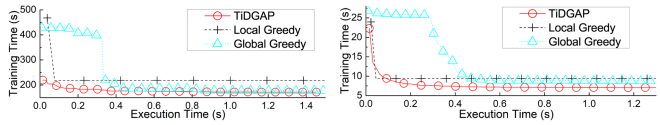
both with $(N = 10, P = 512, N_g = 100)$. Then, we plot the execution time of TiDGAP in Fig. 8.

In Fig. 8, we can observe that when the number of individuals is large enough, the execution time of TiDGAP will also linearly increase with the number of individuals N_p. However, the slope of TiDGAP is still much smaller than that of OiDGAP and DGAP. It is worth emphasizing that, the execution time of TiDGAP in $N_p = 1000$ is only 1.68s for $N = 10$ and that in $N_p = 20000$ is only 19.24s. In the experiments of Figs. 4 and 5, 100 individuals are enough to obtain competitive optimal solutions. This strongly validates the rapidity of TiDGAP with the two-level improvement based on matrix operations on GPU. Similarly, when minibatch-size P is large enough, the execution time of TiDGAP will also linearly increase with P, while with a far smaller slope than that of OiDGAP and DGAP. The speed of TiDGAP is adequate to meet the requirements of current realistic large-scale DNNs.

D. EX_3 : Evaluation of TiDGAP for UMPIPE Compared With Local Greedy Algorithm and Dynamic Programming

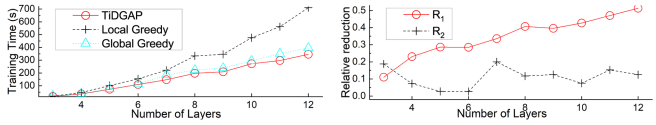
To additionally demonstrate the superiority and feasibility of our proposed TiDGAP for UMPIPE, we choose two typical neural networks (VGG16 and GPT-1) and some self-designed networks, and then execute the experiments in realistic environments. VGG16 (trained in MNist dataset) and GPT-1 (in WikiText-2 dataset) are respectively typical CNN-based DNN in CV and transformer-based DNN in NLP. Then self-designed DNNs are based on CNN, whose configuration can be seen in Table IV. In order to compare algorithms at the same computational speed level, we apply our proposed two-level improvements to the baseline algorithms to greatly improve their computational speed. As the local greedy algorithm does not have continuous search capability, we use its convergence solution to supplement the curve of subsequent time.

For VGG16 and GPT-1, we carry out the training on the servers with multiple Tesla V100 GPUs and obtain the basic time functions. We do not use the solution of GPipe as the initial solution of these algorithms, i.e., the initial states of the algorithms participating in the comparison are all randomly



(a) GPT-1, minibatch-size is 512 (b) VGG16, minibatch-size is 64

Fig. 9. The optimization results (i.e., time for training one minibatch) over times in realistic environments for GPT-1 and VGG16 comparing TiDGAP with local greedy and global greedy algorithms with randomly generated initial solutions, where: $N_p = 100$, $N_g = 100$, under distributed systems with Tesla V100 GPUs, randomly initializing N_p individuals.

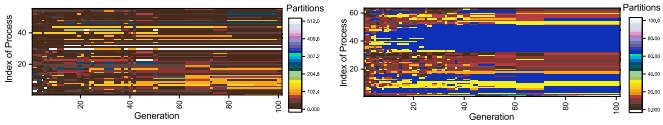


(a) Optimization training time (b) Relative reduction in training time

Fig. 11. The results of self-designed CNN-based networks for 100 minibatches with different numbers of layers in realistic environments comparing TiDGAP with LG and GG algorithms, where $N_p = 100$, $N_g = 100$, randomly initializing N_p individuals. $R_1 = \varepsilon^{\mathrm{LG}}_{\mathrm{TiDGAP}}$, $R_2 = \varepsilon^{\mathrm{GG}}_{\mathrm{TiDGAP}}$.

TABLE V
THE QUANTITATIVE OPTIMIZATION RESULTS OF TIDGAP, LOCAL GREEDY (LG) AND GLOBAL GREEDY (GG) IN THE EXPERIMENTS OF FIG. 9 WITH RANDOMLY GENERATED INITIAL SOLUTIONS

Scenarios	TiDGAP	LG	GG	$\varepsilon^{\mathrm{LG}}_{\mathrm{TiDGAP}}$	$\varepsilon^{\mathrm{GG}}_{\mathrm{TiDGAP}}$
GPT-1	171.71	218.56	177.47	21.44%	3.25%
VGG-16	7.06	9.37	8.80	24.68%	17.78%



(a) GPT-1, minibatch-size is 512 (b) VGG16, minibatch-size is 64

Fig. 10. Waterfall charts of optimal partitions of TiDGAP in Fig. 9.

generated. Then, the optimal results over the execution time of TiDGAP and baselines (local greedy and global greedy) are plotted in Fig. 9. From the overall trends in Fig. 9, it can be seen that solutions of TiDGAP are better than baselines over time. When the time is long enough, TiDGAP achieves the best convergence solutions in both VGG16 and GPT-1 followed by global greedy and local greedy. Although the global greedy algorithm also has search-ability, each search step in it requires calculating the overall training time corresponding to the current optimization solution, which consequently consumes redundant computational complexities. Moreover, the global greedy algorithm can only search for one solution at a time. TiDGAP, based on the individual evolution strategy of the genetic algorithm, can solve multiple solutions simultaneously, which makes it less likely to fall into local optima. The optimal solutions of TiDGAP at the 100th generation and the solutions of baselines at the corresponding time point are listed in Table V. Concretely, the optimization results of TiDGAP are 171.71s and 7.06s respectively for GPT-1 and VGG16, reducing the training time by $(3.25, 17.78)\%$ compared to global greedy and by $(21.44, 24.68)\%$ compared to local greedy algorithm.

To show the role of UEDP in the solution process of UMPIPE's training scheme, Fig. 10 provides waterfall charts of the current generation's optimal partition number for TiDGAP in solving the optimization scheme. It can be seen that the optimal number of partitions for each process of DNN throughout the entire solving process has always been unequal, revealing the advantage of UEDP. In fact, GG and TiDGAP can be combined to establish a growable genetic algorithm using GG as the growth route of TiDGAP, which will have improved performances in terms of convergence and optimality. The growable genetic algorithm is an innovative framework, allowing different algorithms to serve as growth routes to solve optimization problems [39]. As this paper focuses on proposing the novel UMPIPE parallel architecture and its corresponding optimization algorithm with two-level improvement to accelerate DGAP, we do not delve into the discussion of the growable genetic algorithm for UMPIPE.

In order to increase the comprehensiveness of the validation scenario, we perform training of self-designed DNNs on RTX 3060Ti GPUs to obtain the basic time functions. In self-designed DNNs, we change the number of layers to observe the trend of algorithms. We use the stable convergence solution as the ordinate. Then, the optimal results and relative reduction with different numbers of layers are plotted in Fig. 11.

In Fig. 11(a), the curves of TiDGAP remain the lowest followed by global greedy and local greedy. The performance ranking is consistent with that of Fig. 9. Combined with Fig. 11(a), the results validate the advantages of our proposed TiDGAP are universal. From Fig. 11(a), it can be seen that as the number of layers increases, the absolute differences between our proposed TiDGAP and the comparison baselines become increasingly larger. In Fig. 11(b), $\varepsilon^{\mathrm{LG}}_{\mathrm{TiDGAP}}$ shows an increasing trend from 0.1 to 0.5 with the number of layers. This is because the feasible solution space of UMPIPE increases exponentially with the number of layers increases, which leads to a decrease in the algorithm's solving ability. LG does not have search capability and decreases faster than TiDGAP. Additionally, $\varepsilon^{\mathrm{GG}}_{\mathrm{TiDGAP}}$ fluctuates between 0 and 0.2. TiDGAP can maintain its advantage of around 10%.

E. EX_4 : Evaluation of UEDP Compared UMPIPE With State-of-the-Art Parallelism

To demonstrate the superiority of our proposed parallelism UMPIPE and UEDP, we compare it to GPipe and PipeDream in this group of experiments.

To further verify the potential of UEDP for various parallel architectures based on the control variable method, we also include the architecture that combines UEDP with pipedream (called UMPipeDream referring to the naming convention of UMPIPE where $\mathrm{UMPipeDream} = \mathrm{UMPIPE} + \mathrm{1F1B} = \mathrm{PipeDream} + \mathrm{UEDP}$) in the comparison. Due to the lack of fast calculation formulas for PipeDream and UMPipeDream, our experiments use recursive algorithms to calculate the training time corresponding to the data partitioning scheme of PipeDream or UMPipeDream to support the optimization. In experiments, the optimal data partitioning schemes of GPipe and PipeDream

TABLE VI
THE COMPARISON OF VARIOUS PARALLEL ARCHITECTURES (GPIPE, UMPIPE, PIPEDREAM, UMPIPEDREAM) IN DIFFERENT SCENARIOS

Scenarios	Networks	Training Time (s) for One Minibatch				$\epsilon^{\text{GPipe}}_{\text{UMPIPE}}$	$\epsilon^{\text{PipeDream}}_{\text{UMPIPE}}$	$\epsilon^{\text{PipeDream}}_{\text{UM-PipeDream}}$
		UMPIPE	GPipe	PipeDream	UMPipeDream			
Typical Networks	GPT-1 $(N = 14, P = 512)$	167.939	191.264	186.568	164.389	**13.89%**	**11.09%**	13.49%
	VGG16 $(N = 16, P = 64)$	7.060	8.268	7.622	6.462	**17.11%**	**7.96%**	17.95%
Simulation Networks	$(N = 2, P = 512)$	97	271	203	67	178.38%	109.28%	202.98%
	$(N = 2, P = 1024)$	103	595	456	81	477.67%	342.72%	462.96%
	$(N = 5, P = 512)$	207	1141	957	170	451.21%	362.32%	462.94%
	$(N = 5, P = 1024)$	254	1058	894	211	316.54%	251.97%	323.70%
	$(N = 8, P = 512)$	353	1694	1610	303	379.89%	356.09%	431.35%
	$(N = 8, P = 1024)$	409	1735	1647	394	324.21%	302.69%	318.02%
	$(N = 10, P = 512)$	616	2007	1911	482	225.81%	210.22%	296.47%
	$(N = 10, P = 1024)$	478	2220	2024	398	364.44%	323.43%	408.54%
	$(N = 20, P = 512)$	786	3855	3759	722	390.46%	378.24%	420.64%
	$(N = 20, P = 1024)$	1000	4055	3961	895	305.50%	296.10%	342.57%
	$(N = 50, P = 512)$	2297	9963	9865	2150	333.74%	329.47%	358.84%
	$(N = 50, P = 1024)$	2102	9853	9645	1987	368.74%	358.85%	385.41%

Note: $\epsilon^{\text{GPipe}}_{\text{UMPIPE}} = \frac{\text{GPipe}_{\text{training time}} - \text{UMPIPE}_{\text{training time}}}{\text{UMPIPE}_{\text{training time}}}$ means the ratio of improvement in training speed of UMPIPE compared to GPipe. Using recursive algorithms to obtain optimized 1F1B schemes for PipeDream and UMPipeDream; using the enumeration to obtain optimized EDP schemes for GPipe and PipeDream; using genetic algorithm (through 4000 generations for the sake of sufficient optimization) to obtain the optimized UEDP schemes for UMPIPE and UMPipeDream; using the training time of optimized scheme solved as the representative of the corresponding architecture's performance.

are obtained through the enumeration method, as EDP allows enumeration; that of UMPIPE and UMPipeDream are solved by the genetic algorithm with 4000 generations (for the sake of obtaining sufficiently optimized results to reflect the inherent performance of the architectures themselves). Experiments include two representative networks VGG16 (with CNN layers, minibatch-size is 64) and GPT-14 (with transformer layers, minibatch-size is 512) in realistic environments executed with multiple Tesla V100 GPUs, as well as multiple simulation networks generated by realistic devices-based random simulation environments. The optimization results (corresponding to training time) for each architecture in different scenarios are shown in Table VI.

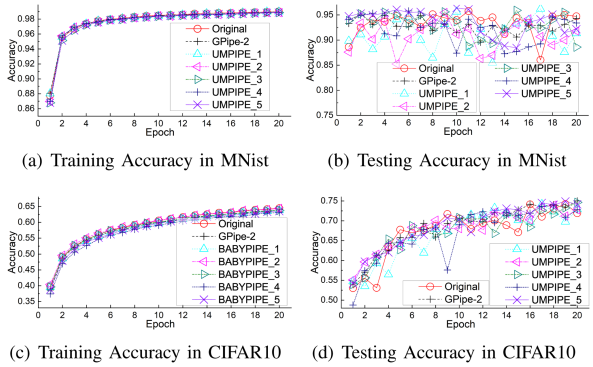
From Table VI, UMPIPE is generally superior to GPipe and PipeDream, indicating UMPIPE has certain advantages as a new parallel architecture. The comparison between UMPIPE and GPipe demonstrates that UEDP can improve the speed of AFAB (all forward all backward) architectures. Compared with PipeDream, UMPipeDream achieves faster training speed, indicating that UEDP also has a positive effect on accelerating 1F1B (one forward one backward) architectures such as PipeDream. Moreover, UMPipeDream overall achieved the best results, indicating the potential of extending UEDP to other parallel architectures. In GPT-1 and VGG16, UMPIPE accelerates the training speed by $(13.89, 11.09)\%$ and $(17.11, 7.96)\%$ respectively compared with (GPipe, PipeDrea); UMPipeDream accelerates the training speed by 13.49% and 17.95% respectively compared with PipeDream. In the simulation scenarios, the heterogeneity of the network layer is more apparent. Thus, UEDP-based parallel architectures (UMPIPE and UMPipeDream) have a greater improvement on the basis of EDP-based parallel architectures (GPipe, PipeDream), with an improvement $\geq (170\%, 100\%)$ in terms of training speed.

When evaluating the optimization scheme under the 1F1B architecture in experiments, we calculated the optimal 1F1B scheme under the given data partition as its evaluation value. Therefore, the optimization results of PipeDream in the experiment represent the performance of architectures such as PipeDream and Dapple [15], [16]. Therefore, the above results can not only demonstrate the significance of UMPIPE in reducing the training time under microbatch-based pipeline parallelism in AFAB, but also reveal the potential of UEDP in improving various architectures in 1F1B such as PipeDream and Dapple. It is notable that the UMPIPE architecture can be further improved, because adding 1F1B strategy on the basis of UMPIPE (i.e., UMPipeDream) can enhance the training speed on the basis of UMPIPE according to the experimental results in Table VI. This paper addresses a challenge: for UMPIPE with the addition of UEDP on the basis of GPipe, we propose the matrix operations-based fast calculation formulas (i.e., (6) and (7)) to simultaneously evaluate multiple training schemes of UMPIPE, which allows optimization algorithms to complete the search for optimization solutions of data partitioning in a short period of time. However, for UMPipeDream ($\text{UMPipeDream} = \text{UMPIPE} + \text{1F1B} = \text{PipeDream} + \text{UEDP} = \text{GPipe} + \text{UEDP} + \text{1F1B}$), the combination of UEDP and 1F1B strategies cause a more complex calculation process for the training time corresponding to the scheme. The recursive algorithm chosen in experiments for UMPipeDream consumes a significant amount of computation time, which also inspires future work to focus on deriving fast calculation formulas for UMPipeDream (i.e., the further extension of UMPIPE). In addition, UEDP will bring additional programming work and data-switching processes in parallel architecture, which require further research on generalization.

In order to supplement the practical application significance of UMPIPE, we need to verify that the UEDP of UMPIPE does not worsen the DNNs' accuracy over epochs during the training process. We choose two self-designed CNN-based networks with 5 layers and carry out the training respectively in MNist dataset and CIFAR10 dataset. The configures of networks and data partitioning schemes of UMPIPE are listed in Table VII, where "U_1" means UMPIPE_1 (the scheme of UMPIPE with index of 1), the column "Or". means the original structure without data partitions, GPipe-2 means dividing the minibatch into two microbatches for all layers in GPipe, and type of layer $C(1, 20)$ means CNN layer with the input channel as 1 and output channels as 20, and $FC(160, 10)$ means full connection layer with input neurons as 160 and output neurons

TABLE VII
THE CONFIGURES OF NETWORKS FOR MNIST AND CIFAR10 DATASET WITH THEIR DATA PARTITIONING SCHEMES OF PARALLELISM

Dataset	Layers	Type	Data Partitions Schemes						
			Or.	GPipe-2	U_1	U_2	U_3	U_4	U_5
MNist	1	$C(1,20)$	1	2	2	2	2	2	4
	2	$C(20,40)$	1	2	2	2	2	8	2
	3	$C(40,80)$	1	2	2	1	2	4	4
	4	$C(80,160)$	1	2	1	2	2	1	8
	5	$FC(160,10)$	1	2	2	1	4	1	1
CIFAR10	1	$C(3,16)$	1	2	2	2	2	8	2
	2	$C(16,32)$	1	2	2	2	1	2	2
	3	$C(32,64)$	1	2	1	1	2	1	4
	4	$C(64,128)$	1	2	2	1	2	1	2
	5	$FC(128,10)$	1	2	2	2	1	1	2



(a) Training Accuracy in MNist (b) Testing Accuracy in MNist

(c) Training Accuracy in CIFAR10 (d) Testing Accuracy in CIFAR10

Fig. 12. The accuracy over epochs in self-designed CNN-based networks in MNist and CIFAR10 dataset comparing UMPIPE with GPipe, where the data partitioning schemes of UMPIPE are listed in Table VII, minibatch-size is 64.

as 10. The minibatch-sizes of Table VII are both 64, and the convolution kernels are all 3×3. For the sake of presentation, we use the power of 2 as the number of microbatches for each layer. UMPIPEs with different subscripts correspond to different UEDP schemes. As communication doesn't affect the accuracy of over epochs, we don't consider the data partitions of communication processes. In this set of experiments, forward propagation and backward propagation in the same layer have the same number of data partitions, which does not affect the experimental conclusion, because the main process that determines the convergence of network training is the gradient descent in backward propagation. Then, we plot the training accuracy and testing accuracy within 20 training epochs in Fig. 12. The results of Fig. 12 show that the accuracy trends of UMPIPE with UEDP do not lag behind the original or GPipe-2. This demonstrates that UMPIPE will not worsen the DNNs' accuracy over epochs. With less time per epoch, UMPIPE will have better convergence over time than GPipe.

VI. CONCLUSION AND FUTURE WORK

Based on the microbatch-based pipeline parallelism, this paper proposes unequal microbatches-based (i.e., unequal data partitions-based) pipeline parallelism (UMPIPE), not only considering computation time and communication time simultaneously, but also considering them probably nonlinear with data size. This paper derives the recurrence formula for the training time of DNN under UMPIPE parallelism and proves the optimality of UMPIPE in theory.

To obtain the optimization scheme of UMPIPE, this paper proposes the dual-chromosome genetic algorithm (DGAP). Dual-chromosome is proved with better convergence than the single chromosome for solving training scheme of UMPIPE. Aiming at accelerating DGAP algorithm, we further delve into theoretical derivations of the recurrence formula for UMPIPE and propose the two-level improved DGAP (TiDGAP). TiDGAP can simultaneously calculate the end time of multi-schemes and multi-microbatches for UMPIPE.

The experimental results comprehensively get corroboration to our theoretical analysis, demonstrating the advantages of dual-chromosome strategy and matrix operation-based two-level improvement method of TiDGAP. Compared with baseline optimization methods (local greedy and global greedy), TiDGAP has better convergence and optimality. Compared with baseline parallelism (GPipe and PipeDream), UMPIPE achieves less training time. Compared to (GPipe, PipeDream), UMPIPE improves training speed by $(13.89, 11.09)\%$ in GPT1-14, $(17.11, 7.96)\%$ in VGG16, and $\geq (170\%, 100\%)$ in other simulation networks.

As part of future work, we will extend UEDP to other parallel architectures, as well as study the matrix-based fast calculation formulas for UMPipeDream and other UEDP-based parallelism. In this paper, UMPIPE mainly considers the optimization of training speed, without specifically considering memory. Introducing multi-objective optimization algorithms to simultaneously optimize the speed and memory of UEDP-based parallelism is also a potential new direction.

REFERENCES

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[2] F. Xue, Q. Wang, and G. Guo, "TransFER: Learning relation-aware facial expression representations with transformers," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, Montreal, QC, Canada, 2021, pp. 3581–3590.
[3] J. E. Zini and M. Awad, "On the explainability of natural language processing deep models," *ACM Comput. Surv.*, vol. 55, no. 5, pp. 103:1–103:31, 2023.
[4] H. Fu et al., "HGP4CNN: An efficient parallelization framework for training convolutional neural networks on modern GPUs," *J. Supercomput.*, vol. 77, no. 11, pp. 12 741–12 770, 2021.
[5] D. Narayanan et al., "Efficient large-scale language model training on GPU clusters using Megatron-LM," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, St. Louis, Missouri, USA, 2021, pp. 58:1–58:15.
[6] Z. Li et al., "TeraPipe: Token-level pipeline parallelism for training large-scale language models," in *Proc. 38th Int. Conf. Mach. Learn.*, 2021, pp. 6543–6552.
[7] Y. Lee, J. Chung, and M. Rhu, "SmartSAGE: Training large-scale graph neural networks using in-storage processing architectures," in *Proc. 49th Annu. Int. Symp. Comput. Architecture*, 2022, pp. 932–945.
[8] T. Rao, J. Li, X. Wang, Y. Sun, and H. Chen, "Facial expression recognition with multiscale graph convolutional networks," *IEEE Multimedia*, vol. 28, no. 2, pp. 11–19, Second Quarter, 2021.
[9] H. Wang et al., "A comprehensive survey on training acceleration for large machine learning models in IoT," *IEEE Internet of Things J.*, vol. 9, no. 2, pp. 939–963, Jan. 2022.
[10] Z. Li et al., "Optimizing makespan and resource utilization for multi-DNN training in GPU cluster," *Future Gener. Comput. Syst.*, vol. 125, pp. 206–220, 2021.
[11] 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. 71:1–71:10.
[12] J. Romero et al., "Accelerating collective communication in data parallel training across deep learning frameworks," in *Proc. 19th USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 1027–1040.

[13] 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.
[14] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 103–112.
[15] D. Narayanan et al., "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, T. Brecht and C. Williamson, Eds., 2019, pp. 1–15.
[16] 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.
[17] M. Wang, C. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proc. 14th EuroSys Conf.*, 2019, pp. 26:1–26:17.
[18] F. Li et al., "Fold3D: Rethinking and parallelizing computational and communicational tasks in the training of large DNN models," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 5, pp. 1432–1449, May 2023.
[19] S. Ouyang et al., "Communication optimization strategies for distributed deep neural network training: A survey," *J. Parallel Distrib. Comput.*, vol. 149, pp. 52–65, 2021.
[20] Z. Zhang, J. Chen, and B. Hu, "The optimization of model parallelization strategies for multi-GPU training," in *Proc. IEEE Glob. Commun. Conf.*, 2021, pp. 1–6.
[21] V. Elango, "Pase: Parallelization strategies for efficient DNN training," in *Proc. 35th IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 1025–1034.
[22] L. Cui et al., "A bidirectional DNN partition mechanism for efficient pipeline parallel training in cloud," *J. Cloud Comput.*, vol. 12, no. 1, 2023, Art. no. 22.
[23] J. Xu et al., "Effective scheduler for distributed DNN training based on MapReduce and GPU cluster," *J. Grid Comput.*, vol. 19, no. 1, 2021, Art. no. 8.
[24] J. Dong et al., "EFLOPS: Algorithm and system co-design for a high performance distributed training platform," in *Proc. Int. Symp. High Perform. Comput. Architecture*, 2020, pp. 610–622.
[25] G. Zhou et al., "CSIMD: Cross-search algorithm with improved multi-dimensional dichotomy for micro-batch-based pipeline parallel training in DNN," in *Proc. 30th Eur. Conf. Parallel Distrib. Process.*, Madrid, Spain, 2024, pp. 288–301.
[26] Z. Zeng, C. Liu, Z. Tang, W. Chang, and K. Li, "Training acceleration for deep neural networks: A hybrid parallelization strategy," in *Proc. 58th ACM/IEEE Des. Automat. Conf.*, 2021, pp. 1165–1170.
[27] 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.
[28] Z. Han et al., "Exploit the data level parallelism and schedule dependent tasks on the multi-core processors," *Inf. Sci.*, vol. 585, pp. 382–394, 2022.
[29] Y. Li, Z. Zeng, J. Li, B. Yan, Y. Zhao, and J. Zhang, "Distributed model training based on data parallelism in edge computing-enabled elastic optical networks," *IEEE Commun. Lett.*, vol. 25, no. 4, pp. 1241–1244, Apr. 2021.
[30] L. Guan, Z. Yang, D. Li, and X. Lu, "pdlADMM: An ADMM-based framework for parallel deep learning training with efficiency," *Neurocomputing*, vol. 435, pp. 264–272, 2021.
[31] A. N. Kahira et al., "An oracle for guiding large-scale model/hybrid parallel training of convolutional neural networks," in *Proc. 30th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2021, pp. 161–173.
[32] J. Zhang et al., "PipePar: Enabling fast DNN pipeline parallel training in heterogeneous GPU clusters," *Neurocomputing*, vol. 555, 2023, Art. no. 126661.
[33] Z. Zhang, Z. Ji, and C. Wang, "Momentum-driven adaptive synchronization model for distributed DNN training on HPC clusters," *J. Parallel Distrib. Comput.*, vol. 159, pp. 65–84, 2022.
[34] S. Zheng et al., "NeoFlow: A flexible framework for enabling efficient compilation for high performance DNN training," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 3220–3232, Nov. 2022.
[35] R. Gu et al., "Liquid: Intelligent resource estimation and network-efficient scheduling for deep learning jobs on distributed GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2808–2820, Nov. 2022.
[36] 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.
[37] K. S. Pal and P. P. Wang, *Genetic Algorithms for Pattern Recognition*. Boca Raton, FL, USA: CRC Press, 1996.
[38] K. Deb, A. Pratap, S. Agrawal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.
[39] G. Zhou, W. Tian, R. Buyya, and K. Wu, "Growable genetic algorithm with heuristic-based local search for multi-dimensional resources scheduling of cloud computing," *Appl. Soft Comput.*, vol. 136, 2023, Art. no. 110027.

Guangyao Zhou received the bachelor's and master's degrees from Tianjin University, China, and the PhD degree from the University of Electronic Science and Technology of China, China. He is now an assistant professor with the Southwest Jiaotong University, China. His research interests include scheduling algorithms in cloud computing or edge computing, image recognition especially facial expression recognition, parallel training of large-scale model, and evolution algorithms.

Wenhong Tian (Member, IEEE) received the PhD degree from the Department of Computer Science, North Carolina State University, Raleigh, NC, USA. He is now a professor with the University of Electronic Science and Technology of China, China. His research interests include scheduling in cloud computing and Bigdata platforms, image recognition by deep learning, algorithmic theory of machine learning, parallel training of large-scale model, and evolution algorithms.

Rajkumar Buyya (Fellow, IEEE) is a Redmond Barry distinguished professor and director with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovations in cloud computing. He served as a future fellow of the Australian Research Council during 2012–2016. He has authored more than 750 publications and seven text books. He is one of the highly cited authors in computer science and software engineering worldwide (h-index=170, gindex=374, 155100+ citations). He is recognized as a "Web of Science Highly Cited researcher" for six consecutive years since 2016, and Scopus researcher of the Year 2017 with Excellence in Innovative Research Award by Elsevier for his outstanding contributions to cloud computing and distributed systems.

Kui Wu (Member, IEEE) received the BSc and MSc degrees in computer science from Wuhan University, Wuhan, China, in 1990 and 1993, respectively, and the PhD degree in computing science from the University of Alberta, Edmonton, AB, Canada, in 2002. In 2002, he joined the Department of Computer Science, University of Victoria, Victoria, BC, Canada, where he is currently a professor. His current research interests include network performance analysis, online social networks, Internet of Things, and parallel and distributed algorithms.