ArrayPipe: Introducing Job-Array Pipeline Parallelism for High Throughput Model Exploration

Hairui Zhao[1], Hongliang Li[1,2,*], Qi Tian[1], Jie Wu[3], Meng Zhang[1], Xiang Li[1], Haixiao Xu[4]
[1] College of Computer Science and Technology, Jilin University, China
[2] Key Laboratory of Symbolic Computation and Knowledge Engineering of the Ministry of Education, China
[3] Department of Computer and Information Sciences, Temple University, USA
[4] High Performance Computing Center, Jilin University, China
zhaohr21@mails.jlu.edu.cn, lihongliang@jlu.edu.cn, qitian23@mails.jlu.edu.cn, jiewu@temple.edu,
mzhang22@mails.jlu.edu.cn, {lxiang, haixiao}@jlu.edu.cn

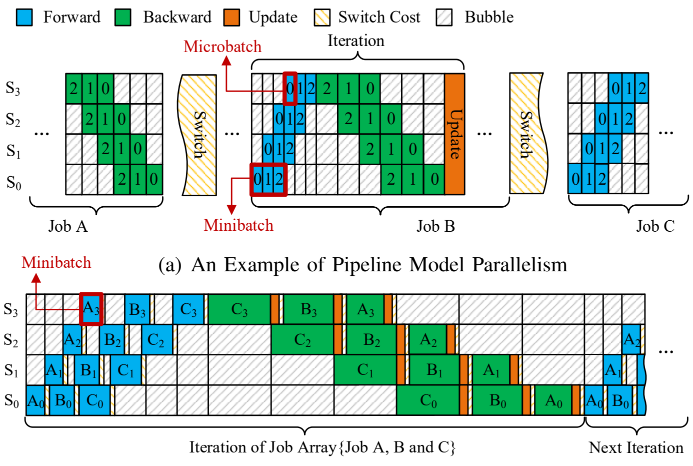
***Abstract*—Deep Learning (DL) applications have experienced exponential growth in data volume and model complexity, spurring various parallel approaches. Existing solutions mostly focus on accelerating individual training jobs. However, jobs submitted to a cluster may not always be independent. This is due to the distinctive characteristic of DL training that it is an *exploratory* process. Model developers often launch multiple training instances in a batch with the same model structure but different settings to tune hyper-parameters, which provides an opportunity to regard these jobs as *job-arrays*. With further support of low-cost job context switching, sharing resources among these jobs is not just feasible but also beneficial to the resource utilization and the throughput of a DL cluster. This paper introduces *Job-Array Pipeline Parallelism (JAP)* that assembles a batch of sibling DL training jobs into a concurrent *job-array*. We design ArrayPipe, a framework that supports high throughput model exploration with *JAP*. A novel scheduling problem in *JAP* is proposed that seeks to minimize the per-iteration training time for a *job-array*, along with two scheduling algorithms for different scales of *job-arrays*. Extensive testbed experiments and trace-driven simulations show that ArrayPipe achieves 1.46× training throughput on average compared with state-of-the-art related works.**

***Index Terms*—Deep Learning, Distributed Computing, Exploratory Jobs, Pipeline Model Parallelism, Scheduling**

I. Introduction

Recently, Deep Learning (DL) has shown tremendous success across a range of applications [1], [2]. Over the years, in pursuit of unprecedented accuracy, Deep Neural Network (DNN) training has grown exponentially in both data volume and model complexity, e.g., training GPT-4 (1.8 trillion parameters) model took approximately 2 years with 1024 H100 NVIDIA GPUs [3]. Thus, training DNN in parallel (e.g., GPUs) is an inevitable trend [4]–[7]. Various parallelism techniques have been proposed for both large dataset (Data Parallelism (DP) [4], [8]) and large model size (Tensor Model Parallelism (TMP) [9], [10], Pipeline Model Parallelism (PMP) [11]–[14], and hybrid parallelism [15], [16]).

PMP is widely used for large model training. As exampled in Fig.1(a), a 4-stage PMP utilizes 4 devices to accelerate the model training process. Each job occupies a set of resources



(a) An Example of Pipeline Model Parallelism

(b) An Example of Job-Array Pipeline Parallelism

Fig. 1. Model Exploration with PMP and *JAP* on 4 stages, S_1, S_2, S_3, and S_4. 0, 1, 2, and 3 in Fig. 1(a) denote the index of micro-batches in a job, while $\mathrm{A}_0, \mathrm{A}_1, \mathrm{A}_2$, and A_3 denote the stages of job A in a job-array.

exclusively, and after the completion of one job, the GPU running environment (e.g., CUDA context in NVIDIA) needs to be cleaned and initialized for the next one. Switching job context can cause a noticeable delay (usually more than 100× the duration of an iteration [17]. PMPs face under-utilization from two aspects, training efficiency (i.e., poor GPU ALU utilization caused by small mini-batches) and pipeline efficiency (i.e., high bubble ratio in pipelines) [11], [18]. Current parallelism studies have been focusing on accelerating individual DL training jobs [19], [20].

One key characteristic of DL training is that it is not a one-time effort and often involves a significant amount of trial-and-error jobs, known as *exploratory* jobs [21], [22]. Such *exploratory* jobs consume a large amount of time and resources in DL clusters (e.g., 37.69% GPUs time overall in Microsoft production clusters [23]). To select an optimal set of hyper-parameters (e.g., data sample batch size, learning rate, and so on), data scientists can submit tens to hundreds of instances of DL training jobs with the same model structure but varying model configurations in a batch [24].

Therefore, in this paper, a novel *Job-Array Pipeline Paral-*

* Corresponding author: lihongliang@jlu.edu.cn

lelism (JAP) is introduced to pursue high throughput for large model *exploration*. A key observation motivating us is that a batch of *exploratory* jobs are siblings that share the same core (e.g., model structures). This provides an opportunity to regard these jobs as a *job-array*, allowing resource sharing among them to improve the resource utilization and overall throughput of a DL cluster. *JAP* can be regarded as an extension of PMP that each job still executes in a pipeline fashion, only that multiple sibling jobs in a *job-array* share a set of resources and execute concurrently. As exampled in Fig. 1(b), the training *blocks* (i.e., mini-batch training stages) of sibling jobs A, B, and C can execute in parallel for better training throughput.

Specifically, there are two key challenges that need to be addressed. (a) To support resource sharing on the same device, the CUDA contexts need to be switched between consecutive training *block* of different jobs. How to enable low-cost switching between sibling jobs in a *job-array* is a challenge. (b) Unlike in traditional PMP, the execution time of each training segment is not identical for different jobs in *JAP*, as exampled in Fig.1, which invalidates existing PMP scheduling approaches. Moreover, stage dependencies within a job increase the difficulty of the already challenging problem. Sophisticated scheduling algorithms are needed to ensure high throughput *JAP* training.

To address the above issues, this paper makes the following contributions:

1) A novel parallel scheme, *Job-Array Pipeline Parallelism (JAP)* is introduced to enable a batch of sibling jobs to form a concurrent *job-array* and to execute concurrently, aiming for high throughput model *exploration*.
2) We design ArrayPipe, a framework to support *JAP* with low-cost job context switching mechanism within a *job-array* and a GPU-Host memory manager for fast job state loading.
3) We propose a novel scheduling problem *Job-Array Pipeline Scheduling Problem (JAPSP)* that seeks to minimize the per-iteration time of the *job-array*, along with two algorithms for different scales of *job-arrays*.
4) Extensive experiments both testbed and trace-driven simulations are conducted to evaluate the efficiency of ArrayPipe. The results show that ArrayPipe achieves an average 1.46× training throughput compared with state-of-the-art PMP approaches.

II. Background and Related Work

A. DNN Training

DL training is an iterative process that seeks a set of optimal model parameters i.e., the probability-weight associations between input and predicted output. DL jobs are typically trained over a large dataset which is divided into mini-batches. These mini-batches are fed to the DNN model to minimize a loss function until the model converges. In each training iteration, one mini-batch is processed in three steps: (1) Forward Propagation (FP): the mini-batch is computed by applying the model function to derive a loss value. (2) Backward Propagation (BP): gradients of model parameters are computed based on the loss value in the reverse order of FP. (3) Parameter update: the model parameters are updated by using a gradient with an optimization algorithm, e.g., adaptive moment estimation (Adam) [25].

B. Parallelism Techniques

To deal with large volumes of training data and complex models, researchers have put forward a range of parallel techniques. In DP [4], [8], a DL job starts multiple model instances (i.e., workers), each worker trains an identical DNN model with a part of the dataset. With TMP [15], layers (operations) of the model are split into different devices, and each matrix-multiplication requires two synchronous communications resulting in significant communication overhead. PMP [5], [20], [26] mitigates this issue by partitioning the layers of a DNN model into multiple stages, where each stage consists of a consecutive set of layers. Only the border layers of neighbor stages communicate in each iteration. This paper mainly focuses on PMP, which is widely applied parallelism for large DNN models in both commercial and academic DL clusters [15], [19], [26].

GPipe [11] splits a mini-batch into smaller micro-batches and synchronously updates parameter at mini-batch barriers. PipeDream [26] injects multiple mini-batches and updates model parameters *asynchronously* for different stages. However, PipeDream sacrifices the training accuracy and keeps multiple copies of the parameters. Similarly, Chimera [20] achieves a low bubble ratio by maintaining an extra model replica on each device to form a bidirectional pipeline. Hanayo [19] is an improvement of Chimera which mitigates the memory pressure by splitting more stages to achieve a more efficient bidirectional pipeline. DAPPLE [27] adopts one-forward-one-backward (1F1B) schedule to reduce the memory footprint of activations but introduces memory imbalance between devices. BPipe [5] leverages high-speed interconnects to transfer intermediate data between GPUs during training, enabling all GPUs to utilize comparable amounts of memory. Existing studies have been focusing on accelerating individual DL training jobs. The possibility of sharing resources among jobs is overlooked.

C. Accelerate Exploratory DNN Training

Despite the extensive efforts on PMPs, few had taken the *exploratory* feature of DL training jobs into consideration. One key characteristic of DL training is that it usually involves a significant amount of trial-and-error jobs [28]–[30]. Generally, once a DNN model structure has been identified, users launch multiple job instances and perform the *exploratory* process by using some searching tools. For example, HyperBand [30] can launch 128 DL jobs in a batch. Large DL model *exploration* would inevitably put great pressure on the already busy model training infrastructures.

Related work [22], [31] has taken advantage of the *exploratory* characteristic of DL jobs. Gandiva [21] is proposed as a scheduling framework to address the *exploratory* jobs

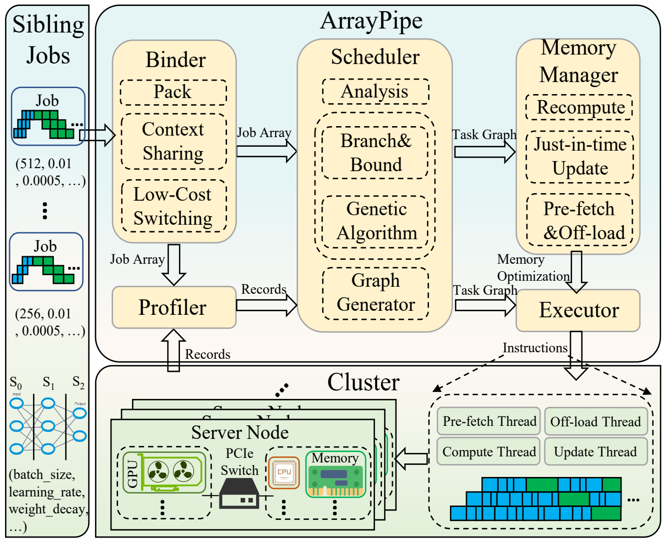
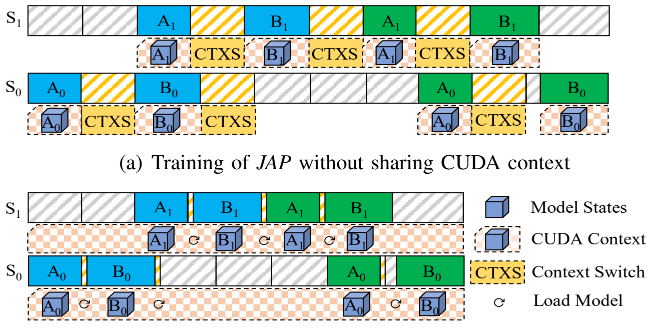


Fig. 2. Overview of ArrayPipe System Architecture



(b) Training of *JAP* with sharing the CUDA context

Fig. 3. Example of *JAP* training with and without shared CUDA context.

by allocating GPUs to a new job immediately and using the suspend-resume mechanism to provide early results. ExplSched [22] makes resource allocation decisions online to maximize the training progress of jobs. The above researches take into account the characteristics of early termination in *exploratory* jobs but do not address the characteristic associated with batch job submissions. They consider each exploratory job independently, while this paper proposes a new perspective of considering a batch of *exploratory* jobs as a whole. Existing studies (e.g., Hyperband, ExplSched) are all orthogonal to ArrayPipe and can be seamlessly integrated with it.

III. ArrayPipe Design

A. Overview

To accelerate the *exploratory* process, a collection of sibling jobs (i.e., DL training jobs with an identical structure and dataset but varying hyper-parameters) are submitted to a DL cluster in a batch [30]. For the given sibling job set, previous parallelisms consider each training job individually, which leads to both low training efficiency and pipeline efficiency. Thus, we introduce *Job-array Pipeline Parallelism (JAP)* as a novel parallelism by regarding the sibling jobs as a concurrent *job-array* and share a set of resources. *JAP* is inspired by the abstraction of array job in distributed computing which combines multiple similar jobs from one batch submission to reduce the scheduling and management overhead [32].

Definition 1. ***Job-Array Pipeline Parallelism (JAP).*** *Given a job-array* $J = \{j_1, j_2, ..., j_{|J|}\}$ *and a cluster of servers* H $(h_1, h_2, ..., h_{|H|})$*, JAP is a pipeline parallelism that supports the stages of different jobs in* J *execute concurrently rather than consecutively. This is enabled by low-cost context switching between sibling jobs that share the same core, e.g., model structure and CUDA context, and so on.*

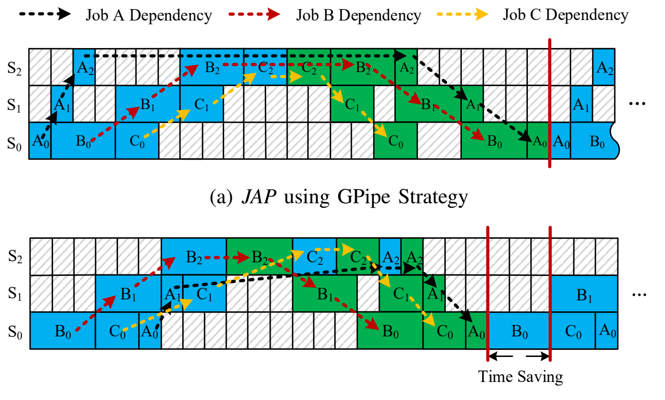
We design ArrayPipe, an efficient framework supporting *JAP* to accelerate the training process of model *exploration*. Fig. 2 shows the overview of ArrayPipe system architecture. To avoid sacrificing the training accuracy that is a non-trivial goal for *exploratory* jobs [22], we focus on the synchronization semantics. ArrayPipe achieves high throughput for model *exploration* by three components. Specifically, for submitted sibling jobs with identical model structure but varying hyper-parameters, *Binder* packs them into a *job-array*. Meanwhile, *Binder* implements CUDA context sharing among the sibling jobs for low-cost context switching (Sec. III-B). Then, *Scheduler* is utilized to generate a task graph of the *job-array*, aiming at minimizing the *job-array's* per-iteration training time of a job-array. Two scheduling algorithms are integrated into the *Scheduler* for different scales of *job-arrays* (Sec. III-C). Finally, ArrayPipe ships a *Memory Manager* that maintains a buffer on GPU memory to prepare for the upcoming training *blocks* and designs several approaches dedicated for *JAP* to reduce the memory pressure (Sec. III-D).

B. Binder

Despite the existing parallelisms releasing the memory pressure by dividing the model into multiple devices and improving the training throughput, there is still room for expediting them. As shown in Fig. 1(a), sibling jobs are submitted to a cluster and scheduled individually, which leaves a considerable amount of bubbles. However, the sibling jobs have identical model structures but varying hyper-parameters, which provides the potential to share resources among them.

For a given batch of sibling jobs, *Binder* regards them as a whole and packs them as a *job-array* by maintaining a public model structure. Fig. 3 shows an example of training a *job-array* with two jobs on two GPUs. While the CUDA context is required to be switched between the boundary of each *block* of different jobs, the training states of each job are saved and swapped from GPU memory to the host memory of the CPU. The CUDA context switching can be costly and even can deteriorate the training throughput, as illustrated in Fig. 3(a).

The key to enabling low-cost context switching is leveraging the unique characteristics of sibling jobs that share the same core, e.g., model structures, operation graphs, and so on. Specifically, the same core of jobs in *job-array* helps ArrayPipe maintain a shared CUDA context among them. As shown in Fig. 3(b), facilitated by the shared CUDA context, only model states (i.e., model parameters and optimizer) need to be switched between the training boundary of different jobs



(a) *JAP* using GPipe Strategy

(b) ArrayPipe Scheduling Strategy

Fig. 4. *JAP* Scheduling Examples.

in *job-array*. A model state is usually much smaller in size than the entire job context, e.g. the model states of BERT and GPT-2 take 1GB and 12GB space, respectively. Furthermore, this makes it possible for ArrayPipe to prepare model states for upcoming blocks in GPU memory, instead of host memory, to support low-cost switching within a *job-array*.

C. Scheduler

Though *job-array* generated by *Binder* has low-cost switching, an improper execution order of the *blocks* leads to a large number of bubbles that degrade the training throughput. As shown in Fig. 4, three jobs (A, B, and C) form a *job-array*, traditional PMPs (e.g., GPipe in Fig. 4(a)) can result in inefficient pipeline schedules for two reasons.

Firstly, previous studies assume equal training time for all micro-batches in a particular stage. Nonetheless, *JAP* processes jobs with varying hyper-parameters, and the training time of *blocks* (i.e., mini-batch training stages) can be different, as shown in Fig. 1(b). Secondly, with *JAP*, jobs do not have to be executed in the same order in each stage, as long as the stage dependencies within a job are satisfied, as shown in Fig. 4(b). The scheduling of *JAP* is more flexible compared with traditional PMP, and more sophisticated algorithms are needed to ensure high throughput *JAP* training.

Scheduler is integrated into ArrayPipe to generate the task graph, scheduling of *blocks*, for a *job-array*. Two scheduling algorithms are proposed for different scales of *job-arrays*, the algorithms will be discussed in Sec. IV.

D. Memory Manager

Multiple model states need to be stored on the GPU, which can cause over-subscribing of GPU memory. ArrayPipe ships with a *Memory Manager* to mitigate this issue. Firstly, ArrayPipe employs re-materialization [11] to reduce the GPU memory footprint of activations approximately to the square root of the total activations [33], incurring a 33% overhead. Secondly, there is no need for synchronization at the end of each block benefiting from job-level parallelism. Thus, just-in-time updates can release the footprint of gradients and further improve the throughput of training. Thirdly, as shown in Fig. 5, ArrayPipe maintains a small number of rather than all *block* states (i.e., model states of the corresponding *blocks*)

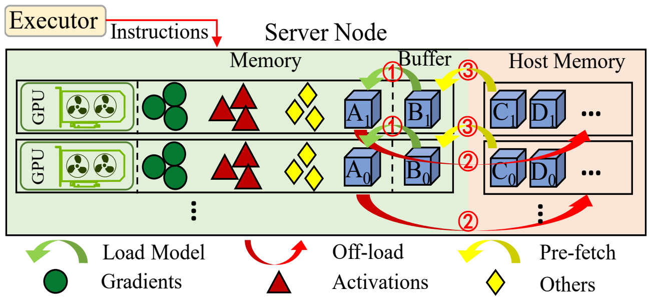


Fig. 5. Low-cost switching supported by memory manager.

buffers in each GPU memory, making sure that the state of an upcoming *block* is always available before *block* context switching. ArrayPipe constantly pre-fetches *block* states to GPU memory for the upcoming training *blocks*. The states are pre-fetched from the host memory or other GPUs in the backend through PCIe or NVLink, according to the task graph generated by *Scheduler*. Outdated *block* states will be offloaded to host memory, utilizing GPU virtualization [34], [35], to avoid GPU memory pressure. The number of buffered states can be configured according to GPU memory size and *block* state size.

IV. JAP Scheduling

For each job j in a *job-array* J, the model structure consists of L $(l_1, l_2, ..., l_{|L|})$ layers. L is partitioned into stages S $(s_1, s_2, ..., s_{|S|})$. Each stage consists of a consecutive set of layers (at least one layer), and the last layer of stage s_n is the predecessor of the first layer of stage s_{n+1}.

A DL cluster consists of a set of server nodes H $(h_1, h_2, ..., h_m)$, and each server h has G_h $(g_1, g_2, ..., g_{|G_h|})$ devices (i.e., GPUs). The bandwidth of inter-servers (intra-servers) communication is denoted by b_{inter} (b_{intra}). Note that the focus of our study is not the stage partitioning or resource allocation for jobs, but the scheduling of *blocks* of jobs in a *job-array* on a given set of devices.

A. Problem Formulation

As shown in Fig. 1(b), on server h, in each training iteration, a mini-batch of job j is fed into the device hosting the first stage (s_1) for FP. Let e^j_{shf} denote the start time of an FP *block* for stage s. During FP, each stage (except s_1 and $s_{|S|}$) gets the input activations from the previous stage and generates activations for the next stage. The training time of an FP *block* on stage s is represented by r^j_{shf}. Let $a^j_{s,s'}$ denote the size of activations (gradients) transferred between stage s and its adjacent stages. The communication time between s_n and $s_{n'}$ can be represented by $\eta^j_{s,s'}$,

$$\eta^j_{s,s'} = \begin{cases} \frac{a^j_{s,s'}}{b_{inter}}, & s, s' \text{ across servers} \\ \frac{a^j_{s,s'}}{b_{intra}}, & s, s' \text{ within a server.} \end{cases} \quad (1)$$

Upon completing an FP *block* for job j_i, the training state switches the model states (i.e., model parameters and optimizer) to the subsequent job j_{i+1}, and continues execution. Let w^j_{sh} denote the switch time (preparing context) for stage

TABLE I
NOTATION

J	A collection of sibling DL training jobs ($j \in J$).
L	Layer set of the model structure of job j.
S	L is partitioned into stage set S ($s \in S$).
H	DL cluster host severs set H ($h \in H$).
G_h	Server h has device set G_h.
b_{inter}	Inter-server communication bandwidth.
b_{intra}	Intra-server communication bandwidth.
$e^j_{shf}(e^j_{shb})$	The start time of an FP (BP) block of job j for stage s on server h.
$r^j_{shf}(r^j_{shb})$	The training time of an FP (BP) block of job j for stage s on server h.
w^j_{sh}	The time for preparing CUDA context for stage s of job j on server h.
re^j_{sh}	The re-computation time of job j for stage s of job j on server h.
u^j_{sh}	The update time of job j for stage s on server h.
$a^j_{s,s'}$	The size of activations (gradients) of job j transferred between s and its adjacent stage s'.
m^j_{sh}	The size of model states of job j on server h.
$\eta^j_{s,s'}$	Adjacent stage communication time of job j.

s of job j. Furthermore, to support low-cost switching, the model states of the upcoming *block* should be pre-fetched before using, and Eq. (2) gives the constraint of pre-fetch delay, where m^j_{sh} denotes the size of model states of job j on server h, and S_h denotes the stages on server h.

$$\sum_{\forall s \in S_h} m^j_{sh}/b_{intra} \leqslant \min_{j \in \{1,2,\ldots,|J|\}} \{r^j_{shf} - \eta^j_{sf}\} \qquad (2)$$

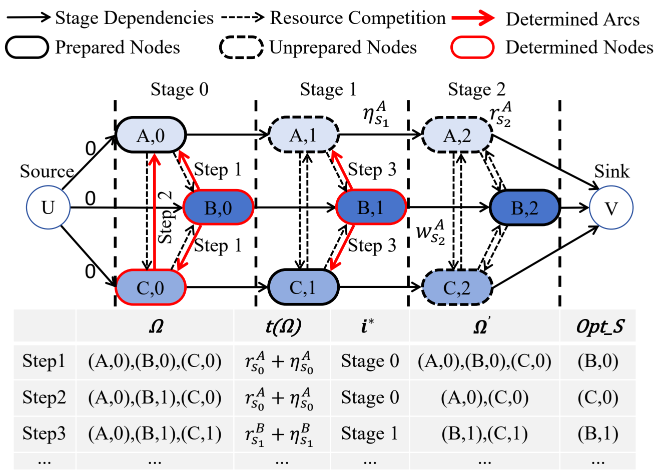
Note that the left-hand side of Eq. (2) gives the worst case for *block* state pre-fetch delay, when all the *block* states on a server are swapped at the same time. Here, we use b_{intra} to denote the bandwidth between GPU and host memory, which is shared among all devices The right-hand side of Eq. (2) gives the shortest training time of *blocks*. ArrayPipe enables low-cost switching by carefully arranging pre-fetch and offloading operations on a server, according to Eq. (2), that no extra swapping delay is introduced.

Then BP transfers gradients following a similar process to FP but in the reverse order. Let e^j_{shb} and r^j_{shb} represent the start time and training time of a BP *block* on stage s, respectively. The activations are not released until the completion of BP because they are generated during BP based on the activations in FP. To avoid GPU memory pressure, *JAP* utilizes rematerialization [33] that introduces extra computation overhead (denoted by re^j_{sh}). Finally, the model of each stage is updated once a BP *block* finishes (i.e., just-in-time update). The update time of stage s for job j is denoted by u^j_{sh}.

We use the overall training time of an iteration of all jobs in J as our performance metric, which can be calculated by the end time of the last BP *block*, denoted by T,

$$T = \max_{j \in \{1,2,\ldots,|J|\}} \{e^j_{s_1hb} + r^j_{s_1hb} + u^j_{s_1h} + w^j_{s_1h} + re^j_{s_1hb}\}. \qquad (3)$$

The optimization objective is to minimize T with the consideration of all solution spaces by scheduling the execution order of *blocks* of jobs in J.



	Ω	$t(\Omega)$	i^*	Ω'	*Opt_S*
Step1	(A,0),(B,0),(C,0)	$r^A_{s_0} + \eta^A_{s_0}$	Stage 0	(A,0),(B,0),(C,0)	(B,0)
Step2	(A,0),(B,1),(C,0)	$r^A_{s_0} + \eta^A_{s_0}$	Stage 0	(A,0),(C,0)	(C,0)
Step3	(A,0),(B,1),(C,1)	$r^B_{s_1} + \eta^B_{s_1}$	Stage 1	(B,1),(C,1)	(B,1)
...	...	...	...	...	...

Fig. 6. An Example of Disjuctive Graph for *JAPSP*.

Definition 2. ***Job-Array Pipeline Scheduling Problem (JAPSP).*** *Given a job-array of DL training jobs $J = \{j_1, j_2, \ldots, j_{|J|}\}$, and a cluster of servers $H = \{h_1, h_2, \ldots, h_{|H|}\}$, the JAPSP seeks a pipeline schedule of all blocks for J on H, with minimum T, as*

$$minimize \quad T, \qquad (4)$$

subject to :

$$e^j_{s_if} + r^j_{s_ihf} + \eta^j_{s_if} \leqslant e^j_{s_{i+1}f}, \quad i \in \{1, \ldots, |S|-1\}, \qquad (5)$$

$$e^j_{s_ib} + r^j_{s_ib} + \eta^j_{s_ib} + re^j_{s_ihb} \leqslant e^j_{s_{i-1}b}, \quad i \in \{2, \ldots, |S|\}, \qquad (6)$$

$$e^j_{shf} + \eta^j_{sf} + w^j_{sh} \leqslant e^j_{shb}, \quad \forall s \in S, \qquad (7)$$

$$e^j_{shf} + r^j_{shf} + w^j_{sh} \leqslant e^{j'}_{shf}, \quad \forall s \in S, \qquad (8)$$

$$e^j_{shb} + r^j_{shb} + w^j_{sh} + u^j_{sh} + re^j_{shb} \leqslant e^{j'}_{shb}, \quad \forall s \in S. \qquad (9)$$

Constraints (5) and (6) are the stage dependencies, ensuring that the execution of stage s_{n+1} in FP (BP) for job j must wait for the activations (gradients) transferred from the previous stage, respectively. Constraint (7) ensures FP is executed before the BP for the same job at any stage. Constraints (8) and (9) represent the resource competition of jobs on the same machine, $e^{j'}_{shb}$ denotes the next *block* on the same device.

JAPSP is a variant of the Job Shop Scheduling Problem (JSSP) [32]. It is straightforward that *JAPSP* can be reduced from JSSP, which proves that *JAPSP* is NP-hard.

B. Algorithms

To achieve the goal of high throughput *job-array* training, two scheduling algorithms are integrated into ArrayPipe for different scales of *job-arrays*:

Branch and Bound (B&B). To minimize T in Eq. 4, we consider FP and BP separately and reformulate *JAPSP* as a disjunctive graph form, as shown in Fig. 6. Consider a graph G with a set of nodes N and two sets of arcs. Each node in N corresponds to a *block* besides two dummy nodes (source node U and sink node V). The conjunctive (solid) arcs represent the stage dependencies within a job, and the disjunctive (broken) arcs represent the resource competition of jobs on the same device. U (V) has $|J|$ solid arcs emanating to (coming from) the first (last) *block* of $|J|$ jobs. Both the nodes and arcs in

G have weights, which is different from the traditional JSSP. The weight of nodes and arcs are shown in Fig. 6, (e.g., $r^A_{s_2}$ for (A, 2), $\eta^A_{s_1}$ for solid arc from (A, 1) to (A, 2), and $w^C_{s_2}$ for broken arc from (A, 2) to (C, 2)).

This problem can be reduced to minimize the length of the critical path in graph G. We use B&B to find an optimal selection of disjunctive arcs as shown in Alg. 1. Let Ω denote the set of all *blocks* that have already been *prepared*, and the initial Ω contains the first *block* of all jobs. Let Rel_Ω denote the earliest starting time of *prepared blocks* (s_n, j) in Ω. Let e^J_S, r^J_S, η^J_S, and w^J_S denote the start, training, communication and switching time of a *block* in the job set J for all stages s in S. Fig. 6 shows part of the steps to execute Alg. 1.

To select an optimal arc from G, we first need to determine which stage (denoted by i^*) has the optimal arc. Stage i^* can be obtained by calculating $\{Rel^j_{s_n} + r^j_{s_n}\}$ among all jobs (denoted by $t(\Omega)$ in line 5). The stage of the job with the minimum $t(\Omega)$ is i^* (line 6). Then, *blocks* on stage i^* in Ω are merged into Ω' (line 9). Here, if the $Rel^j_{s_n}$ is greater than $t(\Omega)$, it will be not merged into Ω' (lines 7-11). The reason for selecting i^* and merging is clear, selecting stage or job j should be prepared as early as possible and postpone other schedules as little as possible.

Then, for each node in Ω', solid arcs from this node to the nodes on stage i^* are added to a temp graph G' (lines 12-14). In G', the makespan can be updated, and the L_{max} of all stages can be calculated by using $1|Rel_j|L_{max}$ function, which is a classic problem as a sub-problem in numerous scheduling algorithms (lines 16-17). Due to space limitations, we have refrained from expanding $1|Rel_j|L_{max}$, please refer to [32] for more details. The lower bound of this extended graph ($LB(G')$) can be obtained by adding the makespan and L_{max}. By enumerating nodes in Ω', the minimum lower bound and corresponding node *Opt_S* can be found (lines 12-21). Then, *Opt_S* is eliminated from Ω and its follower node is added to Ω (line 21). Let *Opt_S* be the extended node (i.e., added to Sched) and update G (lines 23-24). Alg. 1 finishes till Ω is empty and returns task graph *Sched*.

However, the overhead of the B&B algorithm is innegligible for large-scale *job-arrays*. Furthermore, *exploratory* jobs can often experience early terminations. Thus, prompt scheduling results are needed for frequent reschedulings.

Genetic Algorithm (GA). ArrayPipe also employs a GA-based scheduling algorithm for computing prompt scheduling results for large-scale *job-arrays*. Let a one-dimensional vector of length $|J|\times|S|$ represent a feasible schedule (i.e., a chromosome or individual in GA). For each job, it appears $|S|$ times in an individual, and the number of occurrences n denotes the mini-batch of job training on stage s_n. GA initializes the population by randomly generating multiple individuals based on the specified population size. In each generation, the number of individuals designated for crossover is random. GA calculates the Fitness (i.e., the makespan of this individual) of all individuals in the population and obtains the best individual. To avoid a local optimum, GA provides mutations to increase the diversity of a population. Specifically, GA

Algorithm 1 Branch and Bound Algorithm for JSSP

```
1:  Input: Initial graph $G$, Job set $J$
2:  Output: Schedule for each block of jobs $Sched$
3:  Initial: $LB_{min}(G')$=$+\infty$, $\Omega'$=$\{\}$, makespan=0, $Sched$=$\{\}$
    , $\Omega$=$\{(s_1, j)\}$, $Rel_\Omega = \{Rel^j_{s_1} = 0\}, \forall j \in J$
4:  while $\Omega \neq \varnothing$ do
5:      $t(\Omega)$=$\min_{j\in J}\{Rel^j_{s_n} + r^j_{s_n}\}$
6:      $(i^*, \_) = \arg\min_{j\in J}\{Rel^j_{s_n} + r^j_{s_n}\}$
7:      for $(s_n, j)$ in $\Omega$ do
8:          if $s_n$ == $i^*$ and $Rel^j_{s_n} < t(\Omega)$ then
9:              $\Omega' = (s_n, j) \cup \Omega'$
10:     for $(i^*, j)$ in $\Omega'$ do
11:         $G' = G$
12:         add arcs to other jobs in stage $i^*$ in $G'$
13:         Update makespan
14:         $L_{max} = 1|Rel_j|Lmax(S)$
15:         $LB(G')$ = makespan + $L_{max}$
16:         if $LB(G') \leq LB_{min}(G')$ then
17:             $Opt_{Sched} = (i^*, j), LB_{min}(G') = LB(G')$
18:     $\Omega = \Omega - Opt\_S \cup (Opt\_S$.next follower)
19:     Sched = Sched $\cup$ $Opt\_S$
20:     $G = G'$
21: return $Sched$
```

decides individuals to mutate in the population, and swaps any two bits for them. Then, the mutating individuals are added to the new population.

Crossover is the most important operation of GA, which determines the global search ability of the genetic algorithm. GA calculates the probability of crossover for all individuals according to the Roulette Wheel rule. The rationale is that individuals with high fitness will have more chances to crossover with others. Two individuals are selected from the population as parents. To guarantee the children (individuals in the next generation) can inherit the good characteristics of parents (individuals in this generation), precedence operation crossover is utilized to generate two new individuals. Finally, a tournament rule is used for the remaining individuals to select individuals with higher fitness from the old population, ensuring a competitive and fitness-driven selection mechanism.

V. Implementation and Evaluation

ArrayPipe[1] is implemented by using Pytorch and CUDA. Specifically, ArrayPipe achieves a flexible PMP with communication among *blocks* in ArrayPipe handled via mpi4py based on NCCL. It uses the Pytorch profiler to collect information for *Scheduler* to generate a task graph. The job context switching mechanism is implemented by leveraging CUDA's memory management functions.

Binder. ArrayPipe's *Binder* supports sharing CUDA context among sibling jobs in the *job-array* by maintaining a public model structure to load model states of them. The main

[1]https://github.com/libai-master/ArrayPipe

TABLE II
DEEP NEURAL NETWORKS FOR EVALUATION

Model	Dataset	Optimizer	# of Params
VGG19 [36]	ImageNet	SGD	144M
ResNet152 [37]	ImageNet	SGD	60M
BERT-large [38]	GLUE	BertAdam	340M
GPT2-xl [39]	WikiText	AdamW	1.5B

challenge lies in achieving low-cost switching. Instead of swapping model states to the host memory of the CPU, ArrayPipe saves model states directly on GPU memory for quick access and loading.

Scheduler. ArrayPipe decomposes the sibling jobs so that each layer can be executed individually. Then, ArrayPipe uses the Pytorch profiler to record profiles (e.g., forward/backward computation time, the size of activations) for each layer in parallel. The scheduling algorithms take the layer-granularity profiles as input and analyze them to generate a graph for tasks of jobs in *job-array*, the *Scheduler* then executes the task graph on a set of GPUs in the deployment.

Memory Manager. ArrayPipe includes a *Memory Manager* to mitigate the memory over-subscribing issue. It achieves re-materialization by recording the checkpoint of input of each stage during the FP and recomputing according to these checkpoints in the BP. ArrayPipe executes the operation of updating at the end of each block in BP, then discards the gradients from GPU memory. To overlap the overhead of switching job context with the computation, the model states of the upcoming *blocks* are swapped asynchronously by leveraging dedicated CUDA streams. Furthermore, ArrayPipe can effectively pre-fetches and offload the model states between the GPU memory buffer and host memory according to the task graph generated by *Scheduler*.

A. Experimental Setup

We conduct comprehensive experiments including both testbed and trace-driven simulation studies to evaluate the efficiency of ArrayPipe. The results show that ArrayPipe achieves 1.46× training throughput over State-Of-The-Art (SOTA) PMPs on average.

1) Baseline: ArrayPipe is compared with three representative PMPs. (1) GPipe [11], one of the pioneering PMPs proposed by Google, splits a mini-batch into multiple micro-batches and utilizes recompute to reduce the memory footprint of activations. (2) BPipe [5], a memory-optimized PMP, adopts 1F1B to reduce the peak memory and mitigates the memory imbalance by transferring intermediate activations between pairs of GPUs. (3) Hanayo [19], is currently the most effective PMP, achieving a low bubble ratio without introducing an extra memory footprint.

2) DNN Models: Four representative DNN models are used in the experiments, as shown in Table II. The DNN models are divided into 8 stages with balanced execution time between stages, as in [19], [40].

3) Workload: For each DNN model, an *exploratory* workload is generated, which consists of 64 sibling jobs. The hyper-

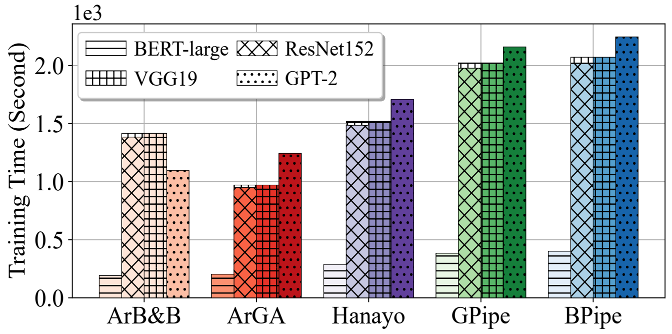


Fig. 7. Training Time Comparisons.

parameters of sibling jobs are set from the following combinations: for computer vision and nature language processing jobs, we set the batch_size to [128, 256, 512, 768] and [8, 16, 24, 32], respectively. The initial learning_rate is set in [0.01, 0.001, 0.0001, 0.00001], and the weight_decay within [0.0001, 0.00001], and the momentum is in [0.9, 0.99]. The number of micro-batches with the highest throughput is chosen in Hanayo, GPipe, and BPipe for different job sizes. The number of waves in Hanayo is set as 2 due to the limitation of layers of DNN models.

B. Testbed Experiments

The environment consists of one NVIDIA SXM4 server with 8 Ampere-100 GPUs, each with 80GB on-chip memory, and the communication bandwidth between each GPU is NVLink (300GB/s). The bandwidth between GPUs and CPU is PCIe4.0 with the 64GB/s. Running on 64-bit Ubuntu Ubuntu 20.04.5 LTS with CUDA toolkit V11.7.99 and PyTorch 2.0.1.

1) Training Time: Training time is the key metric representing the duration required to train all jobs in the workload for a single epoch. Fig. 7 shows the training time comparisons with different PMPs, the results of ArrayPipe with different algorithms are shown separately: Ar_B&B (ArrayPipe using B&B) and Ar_GA (ArrayPipe using GA). The size of *job-array* is set as large as possible based on the memory capacity, and multiple *job-arrays* are executed sequentially.

Fig. 7 indicates that Ar_GA reduces 32.1%, 48.4%, and 50.1% training time of different DNN models on average, compared with Hanayo, GPipe, and BPipe, respectively. For GPT-2 and BERT, Ar_B&B has 8.8% shorter training time than Ar_GA, while 45.8% longer time than Ar_GA for VGG19 and ResNet152. As the scale of *job-array* increases (i.e., more jobs are packed into a *job-array* for smaller DNN models), the overhead to obtain the optimal schedule by B&B is not tolerable. For example, 30 jobs with eight devices take Ar_B&B about 2 minutes to get the optimal schedule, and 3 seconds is needed for Ar_GA. Thus, we set 30 as the maximum *job-array* size for Ar_B&B in our experiment. In practice, the user can set the threshold parameter and guild ArrayPipe to choose the proper algorithm for different scales of *job-arrays*. ArrayPipe achieves an overall 1.46× training throughput compared with SOTA PMPs on average.

2) Utilization: Fig. 8 further illustrates the advantages of ArrayPipe by showing the averaged GPU utilization of different PMPs. ArrayPipe achieves the highest utilization

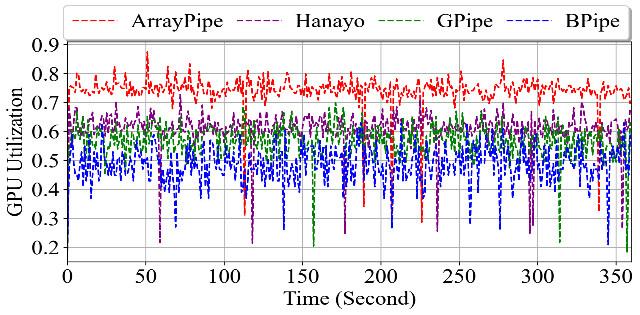


Fig. 8. GPU Utilization.

compared with the baselines (higher than Hanayo, GPipe, and BPipe 20%, 36.8%, and 41.8%, respectively). This is because ArrayPipe forms a pipeline with mini-batches of different jobs instead of partitioned micro-batches of the same job, leading to higher utilization. Hanayo has a lower bubble ratio which also leads to a higher utilization compared with GPipe and BPipe, however, the double stages reduce the utilization of Hanayo. Additionally, we could see that ArrayPipe has a stable utilization compared with other PMPs. According to the analysis, the GPU usage of training DNN usually fluctuates and follows a cyclic pattern [41], and ArrayPipe has longer iterations which prolongs the cyclic interval by consolidating multiple jobs into a *job-array*.

3) Low-Cost Switching and Memory Manager: To evaluate the effectiveness of Low-Cost Switching (LCS) and Memory Manager (MM), we compare ArrayPipe with the degraded ArrayPipe (i.e., ArrayPipe w/o MM, ArrayPipe w/o LCS). Fig. 9 shows the speedup of ArrayPipe and degraded ArrayPipe over vanilla model parallelism (i.e., only one device is active at each time when a mini-batch is trained with PMP [40]). ArrayPipe obviously has the best performance for all DNN models, since ArrayPipe switches the model states between training jobs with LCS and utilizes MM to reduce the memory pressure for packing more jobs into a *job-array*. Note that although ArrayPipe achieves LCS and mitigates memory pressure, the memory footprint still increases with the number of jobs for the reason of storing activation checkpoints.

ArrayPipe w/o MM stores as many as model states on GPU to support LCS, however, it only packs 7 BERT jobs into a *job-array* and even causes Out Of Memory (OOM) by storing two copies of GPT-2. While ArrayPipe packs 30 BERT jobs and 19 GPT-2 jobs into *job-arrays*, respectively, resulting in high throughput *job-array* training.

On the other hand, ArrayPipe w/o LCS packs the same number of jobs into a *job-array* as ArrayPipe due to the reduced memory footprint by MM. However, ArrayPipe w/o LCS has the worst performance because it frequently loads the model states from host memory instead of GPU with low-cost switching. For GPT-2, without LCS, the CUDA context switching between different jobs is based on the stop-resume, which costs 6.7s. With LCS, the training jobs can be switched by loading their model states through a shared public model structure. The loading model states from host memory takes 0.47s, where most of the loading time is consumed by the swapping operation. Furthermore, the model states can be

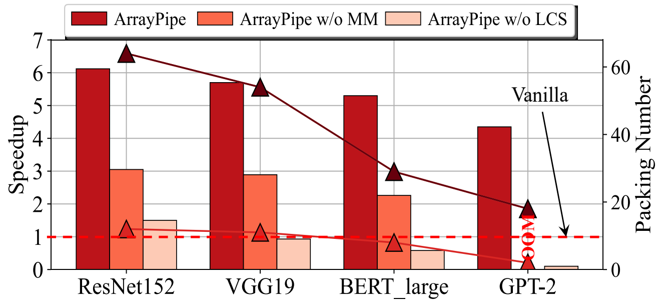


Fig. 9. Comparison between whether to use LCS and MM in ArrayPipe.

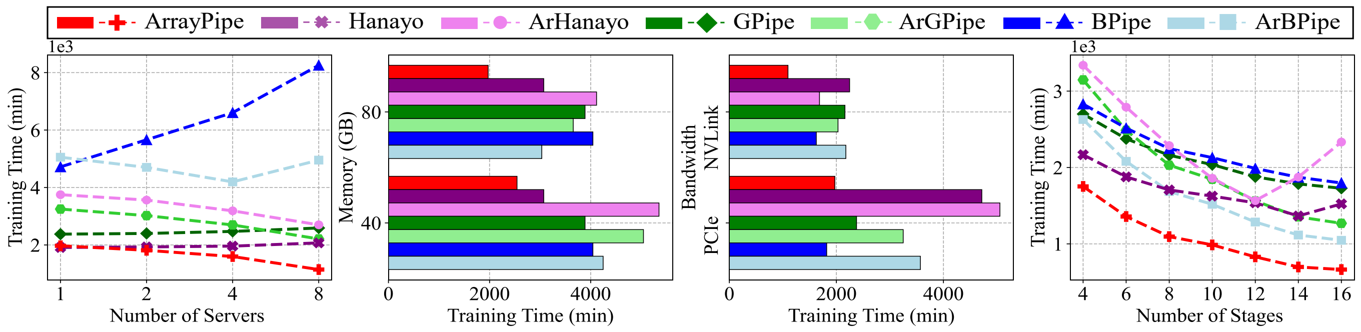
loaded directly from the GPU using pre-fetching, reducing the loading time to just 0.08s.

C. Simulation Studies

We further evaluate ArrayPipe in various environments by simulation studies. The simulated DL cluster consists of NVIDIA servers, including PCIe (64GB/s) versions with 40GB and 80GB memory, and NVLINK (300GB/s) versions with 80GB memory, each server has 16 GPUs, the communication between servers is connected by InfiniBand (50GBps). The workload is the same as in the testbed. We drive our simulation using profiled data collected from real-world measurements by running GPT-2 on one NVIDIA SXM4 server, including training time, model states memory, and context switching overhead. For a fair comparison, the variants of ArrayPipe by using other PMP scheduling strategies are added as baselines, including ArHanayo, ArGPipe, and ArBPipe.

1) Number of Servers: To evaluate the performance with the bandwidth between GPUs and host memory, we distribute 8 stages into 1, 2, 4, and 8 PCIe server(s) with a capacity of 80GB memory, as shown in Fig. 10(a). ArrayPipe reduces the training time by 39.1%, 43.7%, 48.9%, and 62.9% than other baselines on average with 1, 2, 4, and 8 servers, respectively. ArrayPipe and ArGPipe achieve higher performance when more servers are involved. This is because more bandwidth is being used to swap the model states between GPUs and the host memory (i.e., easier to satisfy the Eq. (2)). The results demonstrate that ArrayPipe is suitable in the scenario of using multiple servers, which is consistent with the common practice of hybrid parallelisms. Other baselines perform worse with the increasing number of servers because of the low inter-node bandwidth (i.e., InfiniBand) between stages. This issue is particularly pronounced for BPipe and ArBPipe, which require high-speed bandwidth between stages to support frequent communications. Though Hanayo introduces an additional reverse pipeline to reduce the bubble ratio, Hanayo is not suitable for ArrayPipe due to its complexity of bi-direction. The unequal execution time for each *block* seriously deteriorates the pipeline efficiency of ArHanayo.

2) GPU Memory: To illustrate the relationship between the performance and GPUs' memory, we use PCIe servers with different (40GB and 80GB) memory capacities. As shown in Fig. 10(b), ArrayPipe and its variants benefit from GPUs with large memory by packing more jobs into a *job-array*. For example, ArrayPipe can pack 19 GPT jobs into a *job-array* with an 80GB GPU but only 4 with a 40GB GPU,



(a) Performance with Different Number of Servers
(b) Performance with different Memory of GPUs
(c) Performance with Different Bandwidth of GPUs
(d) Performance with Different Number of Stages

Fig. 10. Performance Comparison in Different Resource Settings.

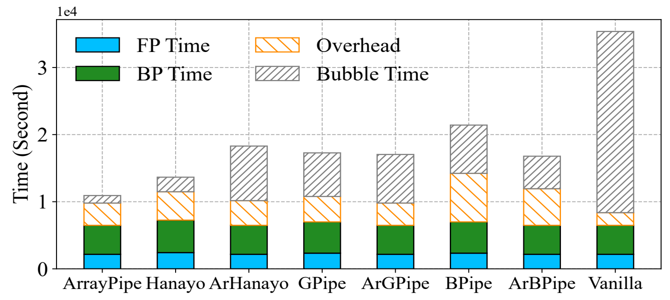


Fig. 11. Wall-clock time (accumulated actual time) of forward, backward time, overhead (e.g., switching context, communications), and bubble time.

which leads to a 29% performance degradation. In contrast, GPipe, BPipe, and Hanayo do not show performance changes with larger GPU memory due to the sequential execution. The performance of ArBPipe and BPipe is worse than other baselines because PCIe is not enough to overlap their extra activation communication between stages in the training.

3) Bandwidth: We further explore the performance of ArrayPipe with different bandwidths by using 80GB servers with PCIe and NVLink. As shown in Fig. 10(c), the performance of ArrayPipe and its variants when using a PCIe server degrades by 49.9% compared with using the server with NVLink. This is because by using a server with NVLink, the upcoming *blocks'* model states can not only be pre-fetched from the host memory but also from other GPUs, which reduces the significant overhead of pre-fetching. Furthermore, BPipe leverages NVLink to balance the memory usage and packs more jobs into a *job-array*, thus, ArBPipe has higher performance improvement compared with ArHanayo and ArGPipe. The performance of Hanayo and GPipe is stable at different bandwidth levels, due to their low dependence on the bandwidth.

4) Number of Stages: Fig. 10(d) shows the training time with a different number of stages by splitting GPT-2 into 4 to 16 stages in an NVLink server. With the number of stages increasing, the performance of ArrayPipe and its variants has obvious improvements, e.g., with 4 stages, ArrayPipe has 19.2% shorter training time than Hanayo, and with 16 stages, it is 57.5% shorter. Because more stages mean less memory footprint on each GPU, which makes more jobs could be packed into a *job-array*. Moreover, the large number of stages helps ArrayPipe have a fine-grained schedule. For other baselines, the training time first decreases and then increases as the increasing of stages, since more stages lead to fewer bubbles, while reducing the training efficiency.

5) Time Breakdown: Fig. 11 shows the breakdown of accumulating time of different PMPs, the overhead contains extra operations, such as context switching, communication between GPUs, and so on. The vanilla PMP is used to demonstrate the relationship between overhead and bubble time. The FP and BP time of existing PMPs (i.e., Hanayo, GPipe, and BPipe) are higher than ArrayPipe and their corresponding variants, due to a larger number of micro-batches and stages decreasing the training efficiency of GPUs. However, ArrayPipe and its variants introduce higher overhead compared with other PMPs. This is because ArrayPipe brings overhead of switching model states and memory optimization, but ArrayPipe overlaps almost 90% overhead with computation according to a carefully designed schedule. BPipe also incurs noticeable extra time due to transferring activations. One can see that Vanilla has the lowest overhead, but Vanilla has the highest bubble time due to the absence of any optimization strategies. In summary, ArrayPipe improves the efficiency of PMP training for model *exploration*, with a modest overhead.

VI. Conclusion

This paper introduces *Job-array Pipeline Parallelism (JAP)* for high throughput large model *exploration*. *JAP* enables a batch of *exploratory* DL training jobs to form a *job-array* and execute concurrently on a shared set of resources. ArrayPipe is designed as a framework to support *JAP* with low-cost job context switching in a *job-array*. A novel *Job-Array Pipeline Scheduling Problem (JAPSP)* is proposed to seek to minimize the per-iteration training time of a *job-array*, along with two algorithms for different scales of *job-arrays*. Extensive testbed experiments and trace-driven simulations are conducted to evaluate the efficiency of ArrayPipe. The results show that ArrayPipe achieves an average 1.46× training throughput compared with state-of-the-art approaches.

Acknowledgment

This work is supported by the Natural Science Foundation of Jilin Province (Grant 20230101062JC), the National Key Research and Development Plan of China (Grant 2017YFC1502306), and the National Natural Science Foundation of China (Grant 62272190).

References

[1] Y. Xiao, L. Wu, J. Guo, J. Li, M. Zhang, T. Qin, and T.-Y. Liu, "A survey on non-autoregressive generation for neural machine translation and beyond," *IEEE Transactions on Pattern Analysis and MachineIntelligence (TPAMI)*, 2022.

[2] H. Zhao, X. Li, and H. Li, "Visage: Visual-aware generation of adversarial examples in black-box for text classification," in *CCF International Conference on Natural Language Processing and Chinese Computing (NLPCC)*, 2024.

[3] O. J. Achiam, S. Adler, S. Agarwal, L. Ahmad, and I. Akkaya, "Gpt-4 technical report," 2023.

[4] Y. Wu, K. Ma, X. Yan, Z. Liu, Z. Cai, Y. Huang, and J. Cheng, "Elastic deep learning in multi-tenant gpu clusters," *(TPDS)*, 2022.

[5] T. Kim, H. Kim, G.-I. Yu, and B.-G. Chun, "Bpipe: Memory-balanced pipeline parallelism for training large language models," in *ACM International Conference on Machine Learning (ICML)*, 2023.

[6] B. Du, J. Liu, Z. Luo, C. Wu, Q. Zhang, and H. Jin, "Expediting distributed gnn training with feature-only partition and optimized communication planning."

[7] Z. L. J. L. S. Z. W. J. Yijun Li, Jiawei Huang and J. Wang, "Reducing staleness and communication waiting via grouping-based synchronization for distributed deep learning," *IEEE International Conference on Computer Communications (INFOCOM)*, 2024.

[8] Y. Liu, B. Jiang, S.-M. Zhao, T. Lin, X. Wang, and C. Zhou, "Libra: Contention-aware gpu thread allocation for data parallel training in high speed networks," *IEEE Conference on Computer Communications (INFOCOM)*, 2023.

[9] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *(ArXiv)*, 2019.

[10] G. Liu, Y. Miao, Z. Lin, X. Shi, S. Maleki, F. Yang, Y. Bao, and S. Wang, "Aceso: Efficient parallel dnn training through iterative bottleneck alleviation," *European Conference on Computer Systems (EuroSys)*, 2024.

[11] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *MIT Press Conference on Neural Information Processing Systems (NeuralPS)*, 2018.

[12] N. Niknami, A. Sawwan, and J. Wu, "Smartpipe: Intelligently freezing layers in pipeline parallelism for distributed dnn training," *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2023.

[13] Y. Duan and J. Wu, "Optimizing job offloading schedule for collaborative dnn inference," *IEEE Transactions on Mobile Computing (TMC)*, 2023.

[14] S. Shi, X. Pan, X. Chu, and B. Li, "Pipemoe: Accelerating mixture-of-experts through adaptive pipelining," *IEEE Conference on Computer Communications (INFOCOM)*, 2023.

[15] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. A. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. A. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-lm," *IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2021.

[16] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, Y. Jia, S. He, H. Chen, Z. Bai, Q. Hou, S. Yan, D. Zhou, Y. Sheng, Z. Jiang, H. Xu, H. Wei, Z. Zhang, P. Nie, L. Zou, S. Zhao, L. Xiang, Z. Liu, Z. Li, X. Jia, J. jun Ye, X. Jin, and X. Liu, "Megascale: Scaling large language model training to more than 10, 000 gpus," *Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[17] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "Pipeswitch: Fast pipelined context switching for deep learning applications," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[18] S. Zhao, F. Li, X. Chen, X. Guan, J. Jiang, D. Huang, Y. Qing, and S. Wang, "vpipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel dnn training," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2022.

[19] Z. Liu, S. Cheng, H. Zhou, and Y. You, "Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2023.

[20] S. Li and T. Hoefler, "Chimera: efficiently training large-scale neural networks with bidirectional pipelines," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[21] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[22] H. Li, H. Zhao, Z. Xu, X. Li, and H. Xu, "Explsched: Maximizing deep learning cluster efficiency for exploratory jobs," *IEEE International Conference on Cluster Computing (CLUSTER)*, 2023.

[23] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, and W. Xiao, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," *USENIX Annual Technical Conference (ATC)*, 2019.

[24] P. Mendes, M. Casimiro, and P. Romano, "Hyperjump: Accelerating hyperband via risk modelling," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

[25] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations (ICLR)*, 2015.

[26] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. A. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[27] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, L. Diao, X. Liu, and W. Lin, "Dapple: a pipelined data parallel approach for training large models," *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPopp)*, 2020.

[28] X. Li, Q. Guo, G. Zhang, S. Ye, G. He, Y. Yao, R. Zhang, Y. Hao, Z. Du, and W. Zheng, "Fasttuning: Enabling fast and efficient hyper-parameter tuning with partitioning and parallelism of search space," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2024.

[29] "Auto ML. retrieved 04/16/2023," https://www.ml4aad.org/automl/.

[30] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *MIT Press Journal of Machine Learning Research (JMLR)*, 2016.

[31] H. Wang, Z. Liu, and H. Shen, "Machine learning feature based job scheduling for distributed machine learning clusters," *IEEE/ACM Transactions on Networking (TON)*, 2023.

[32] P. Michael, "Scheduling. theory, algorithms and systems," 1995.

[33] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *ArXiv*, 2016.

[34] L. Liu, H. Wang, A. Wang, M. Xiao, Y. Cheng, and S. Chen, "vcpu as a container: towards accurate cpu allocation for vms," *ACM International Conference on Virtual Execution Environments (VEE)*, 2019.

[35] F. Strati, X. Ma, and A. Klimovic, "Orion: Interference-aware, fine-grained gpu sharing for ml applications," *European Conference on Computer Systems (EuroSys)*, 2024.

[36] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *International Conference on Learning Representations (ICLR)*, 2015.

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *IEEE/CVF Computer Vision and Pattern Recognition Conference (CVPR)*, 2016.

[38] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019.

[39] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[40] Z. Luo, X. Yi, G. Long, S. Fan, C. Wu, J. Yang, and W. Lin, "Efficient pipeline planning for expedited distributed dnn training," *IEEE International Conference on Computer Communications (INFOCOM)*, 2022.

[41] H. Li, H. Zhao, T. Sun, X. Li, H. Xu, and K. Li, "Interference-aware opportunistic job placement for shared distributed deep learning clusters," *Journal of Parallel and Distributed Computing (JPDC)*, 2023.