Proteus: Simulating the Performance of Distributed DNN Training

Jiangfei Duan, Xiuhong Li, Ping Xu, Xingcheng Zhang, Shengen Yan, Yun Liang, *Senior Member, IEEE*, and Dahua Lin

***Abstract*—DNN models are becoming increasingly larger to achieve unprecedented accuracy, and the accompanying increased computation and memory requirements necessitate the employment of massive clusters and elaborate parallelization strategies to accelerate DNN training. In order to better optimize the performance and analyze the cost, it is indispensable to model the training throughput of distributed DNN training. However, complex parallelization strategies and the resulting complex runtime behaviors make it challenging to construct an accurate performance model. In this article, we present Proteus, the first standalone simulator to model the performance of complex parallelization strategies through simulation execution. Proteus first models complex parallelization strategies with a unified representation named *Strategy Tree*. Then, it compiles the strategy tree into a distributed execution graph and simulates the complex runtime behaviors, *comp-comm overlap* and *bandwidth sharing*, with a Hierarchical Topo-Aware Executor (*HTAE*). We finally evaluate Proteus across a wide variety of DNNs on three hardware configurations. Experimental results show that Proteus achieves 3.0% average prediction error and preserves order for training throughput of various parallelization strategies. Compared to state-of-the-art approaches, Proteus reduces prediction error by up to 133.8%.**

***Index Terms*—Deep neural networks (DNNs), distributed training, parallelism, performance modeling, simulation.**

I. Introduction

RECENT years, progressively larger DNN models continue to break predictive accuracy records [1], [2], [3], [4]. As these models grow, they are becoming computationally and memory expensive to train. To efficiently train DNN models, large GPU clusters and sophisticated parallelization strategies are employed to accelerate the training process [5], [6], [7], [8], [9], [10]. For example, NVIDIA trained an 8.3 billion parameters language model on 512 GPUs with expert-designed hybrid data and model parallelism [6].

Modeling the performance of a parallelization strategy is crucial for performance optimization and analysis, since the training performance (throughput) of a DNN highly depends on its parallelization strategy. *1)* Knowing the performance of a parallelization strategy can guide our optimization. Performance model can be leveraged to locate the bottleneck of a parallelization strategy in manual optimization and compare different parallelization strategies in automated parallelization systems [11], [12], [13], [14]. *2)* Because implementing a parallelization strategy on current deep learning frameworks [15], [16] is error-prone, labor-intensive and resource-costing, an accurate performance model can save lots of effort and resources in evaluating it. *3)* Predicting the performance of a parallelization strategy in advance can help us analyze cloud service budgets without requiring GPU resources, such as how many machine hours or nodes to buy, thereby saving computing resources.

Plenty of performance modeling approaches have been proposed to predict the performance of DNN models, but none of them scale beyond hybrid data and model parallelism. Most of recent efforts to model the performance of DNN models focus on the scenario of single GPU. For example, various analytical models [17], [18], [19], [20] that build with hardware metrics and learning-based models [21], [22] that learn from runtime statistics are presented to study the performance of GPU kernels. In multi-GPU scenario, prior works [23], [24], [25] build analytical or profiling-based performance models for different DNN layers and predict training performance by summing up the computation and communication time of each layer. These approaches focus on a small subset of parallelization strategies and are not applicable to emerging parallelization strategies.

Some automated parallelization approaches [11], [13], [26] also build performance models for distributed DNN training. For example, FlexFlow [11] customizes a simulator to evaluate parallelization strategies in SOAP space. However, these works aim at searching optimal parallelization strategy for a DNN model instead of accurate performance modeling for general parallelization strategies. The usability and scalability are greatly

Manuscript received 14 June 2023; revised 18 July 2024; accepted 9 August 2024. Date of publication 14 August 2024; date of current version 30 August 2024. This work was supported in part by the National Key R&D Program of China under Grant 2022ZD0160201, and in part by Shanghai AI Laboratory, CUHK Interdisciplinary AI Research Institute, and the Centre for Perceptual and Interactive Intelligence (CPII) Ltd. under the Innovation and Technology Commission (ITC)'s InnoHK. Recommended for acceptance by A. Bhatele. *(Corresponding author: Xiuhong Li.)*

Jiangfei Duan is with the Department of IE, The Chinese University of Hong Kong, HKSAR, China, and also with Shanghai AI Lab, Shanghai 200232, China (e-mail: dj021@ie.cuhk.edu.hk).

Xiuhong Li is with the National Engineering Laboratory for Big Data Analysis and Applications, Peking University, Beijing 100871, China (e-mail: lixiuhong@pku.edu.cn).

Ping Xu is with the SenseTime Research, Shanghai 200232, China (e-mail: xuping908@gmail.com).

Xingcheng Zhang is with the Shanghai AI Lab, Shanghai 200232, China (e-mail: zhangxingcheng@pjlab.org.cn).

Shengen Yan is with the Department of Electronic Engineering, Tsinghua University, Beijing 100084, China (e-mail: yansg@tsinghua.edu.cn).

Yun Liang is with the School of EECS, Peking University, Beijing 100871, China (e-mail: ericlyun@pku.edu.cn).

Dahua Lin is with the Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong, China, and also with Shanghai AI Lab, Shanghai 200232, China (e-mail: dhlin@ie.cuhk.edu.hk).

Digital Object Identifier 10.1109/TPDS.2024.3443255

TABLE I
COMPARISON OF PROTEUS AND EXISTING APPROACHES

Approach	*Complex* Parallelization Strategy				*Complex* Runtime Behavior
	Operator-Level		Subgraph-Level		
	Comp.	Mem.	Pipeline	Recomp.	
DAPPLE [13]	Data		✓		
FlexFlow [11]	SOAP				
Alpa [28]	Shard	✓	✓		
Yan *et al.* [24]	Hybrid				
Pei *et al.* [23]	Data				
Paleo [25]	Hybrid				
Proteus (ours)	**Shard**	✓	✓	✓	✓

Approaches in italics are automated parallelization frameworks, the others are performance modeling frameworks.

limited due to their non-programmability and limited strategy space.

We identify two primary challenges that impede the development of accurate performance models for distributed DNN training. The first challenge is **how to model *complex parallelization strategies***. The complexity here stems from the combination of various parallelization strategies, each with distinct computation and memory consumption characteristics, to accelerate DNN training [6], [7], [8], [9], [10]. For example, Megatron-LM combines recomputation [27] and hybrid data, model, and pipeline parallelism to train large transformer models [6]. The second challenge is **how to model *complex runtime behaviors***. Previous works [11], [24], [25], [26] often assume that *the cost of a single operator depends solely on its input and output tensor shapes*. However, this assumption breaks down when dealing with complex parallelization strategies. During runtime, communication operators may overlap with computation operators to hide the gradient synchronization costs, and different communication groups might compete for bandwidth resources. While these optimizations can improve efficiency, they also introduce additional overhead, potentially reducing the overall throughput of the DNN model. Consequently, an accurate performance model should explicitly account for both the optimizations inherent in complex parallelization strategies and the overhead arising from complex runtime behaviors.

To address these challenges, we present Proteus, a standalone simulation framework that aims at accurately modeling the training throughput for distributed DNN training. Table I highlights the advantages of Proteus against existing approaches.

First, we introduce a hierarchical tree structure, *Strategy Tree*, to model complex parallelization strategies. We find parallelization strategies can be classified into operator- and subgraph-level strategies as a DNN graph is often divided into disjoint subgraphs, each of which is assigned to a group of devices. The strategies at operator-level specify how the operators and tensors are split and mapped to devices, while the strategies at subgraph-level indicate how to schedule subgraphs (details refer to Section II). The hierarchical structure of strategy tree provides a unified representation for parallelization strategies at different levels and enables Proteus to efficiently program and model the huge and complex strategy space.

Second, we propose *HTAE (Hierarchical Topo-Aware Executor)* to simulate complex runtime behaviors, which have been overlooked in prior works. We identify two types of runtime behaviors that significantly impact performance: *comp-comm overlap* and *bandwidth sharing*. HTAE simulates the scheduling of subgraphs and operators to detect runtime behaviors during execution and adjusts operator cost according to the detailed cluster topology, thereby capturing complex runtime behaviors of different operators.

Given a DNN model and *Strategy Tree*, Proteus automatically compiles them into a distributed execution graph by splitting operators and tensors and inserting inferred collective communication operators. Subsequently, Proteus predicts the training throughput and OOM (Out-Of-Memory) error by mimicking the schedule and execution of the execution graph considering the cluster topology.

In summary, we propose Proteus, which to our knowledge is the first standalone simulator capable of simulating complex parallelization strategies through fine-grained scheduling and simulation execution. We make the following contributions in building Proteus:

1) We classify parallelization strategies into operator- and subgraph-level and formulate a unified parallelization space with *Strategy Tree* to model complex parallelization strategies.
2) We identify two types of runtime behaviors that affect performance: *comp-comm overlap* and *bandwidth sharing*, and introduce *Hierarchical Topo-Aware Executor* to dynamically detect and model such behaviors.
3) We evaluate Proteus across a wide variety of DNNs on 3 hardware configurations. Experiments show that Proteus achieves 3.0% average prediction error and preserves order for throughput among various parallelization strategies. Compared to state-of-the-art approaches, Proteus reduces prediction error by up to 133.8%.

II. BACKGROUND: DISTRIBUTED DNN TRAINING

DNNs are commonly represented as computation graphs in modern DL frameworks [15], [16], with nodes as operators and edges as tensors. Parallelizing a DNN involves parallelizing elements in the computation graph which can be categorized into two levels of strategies.

A. Operator-Level Strategy

Operator-level strategy partitions operators and tensors for parallel execution on multiple devices and can be further categorized into *computation parallelization* and *memory optimization*, respectively.

1) Computation Parallelization: Computation parallelization is achieved by partitioning the parallelizable dimensions of operators. Typically, we consider every unique dimension occurred in input or output tensors as parallelizable dimensions. We will describe different parallelization strategies taking the linear operator in Fig. 1(a) as an example: $output(b, s, o) = \sum_h input(b, s, h) \times weight(o, h)$. There are 4 unique dimensions: $b(batch)$, $s(sequence)$, $o(output\_channel)$, $h(hidden/reduction)$.

Data parallelism is the most widely used parallelization strategy, which splits batch dimension (b) and replicates $weight$ on all devices.

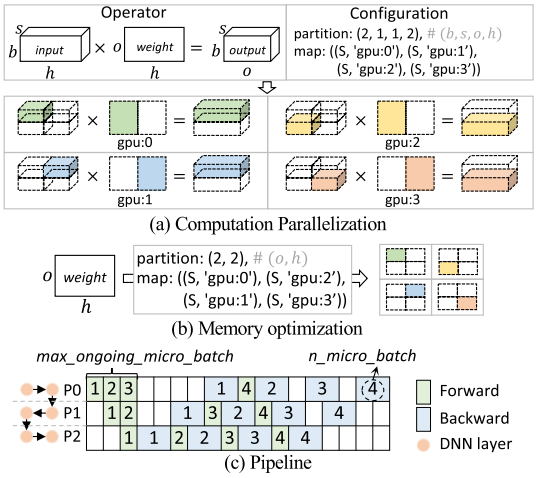


Fig. 1. Examples of parallelization strategies.

Model parallelism divides the operator in o or h dimension thus partitioning $weight$ into different parts and each part is trained on a dedicated device. *Hybrid parallelism* combines both *data* and *model parallelism* to partition operators.

Op shard is a general parallelization strategy that exploits the power of partitioning arbitrary dimensions of (b, s, o, h).

In this paper, Proteus targets on modeling the performance of *general op shard* unlike prior works that focus on *data* and *model parallelism* [23], [24], [25]. SOAP [11] partitions operators in b, s, o dimensions and is also a sub-space of op shard. Fig. 1(a) shows an example configuration to *shard* an operator in b and h dimensions. The *partition* describes how to parallelize different dimensions and *map* specifies how to place each partition. The operator is split into 4 ($|partition|$) parts, with each assigned to a GPU. As reduction dimension (h) is partitioned, the operator produces 4 *partial* output tensors, which should be aggregated to produce the final output tensor.

2) Memory Optimization: All dimensions of a tensor are parallelizable. Partitioning a tensor is achieved by splitting along its dimensions similar to partitioning operators.

ZeRO [9] and *Activation partitioning* [29] partitions tensors in the first dimension and maps each part to a device to reduce redundancy. They can be combined with parallelization in other dimensions. Fig. 1(b) shows an example that partitions o (ZeRO) and h dimensions.

Proteus explicitly defines a parallelization strategy for each tensor in a DNN model. Fig. 1(a) shows that splitting an operator also creates implicit parallelization strategy for its input and output tensors. The inconsistency between the implicit and explicit strategy will incur additional communication (e.g., *weight* need to transform from strategy of Fig. 1(b) to the implicit strategy of Fig. 1(a)).

B. Subgraph-Level Strategy

A subgraph is composed of operators and tensors with dependencies. Parallelization strategies that describe the schedule of subgraphs are called subgraph-level strategies, including *pipeline parallelism* and *recomputation*, which balance training throughput and memory footprint by parallelizing subgraph computations.

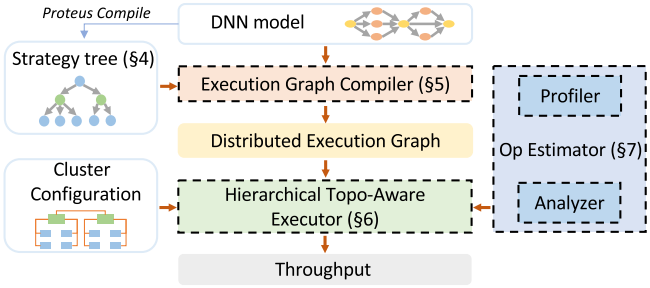


Fig. 2. An overview of Proteus.

Pipeline parallelism divides a computation graph into disjoint parts and assigns each part to a device group. It splits a batch input data into multiple micro-batches to exploit parallelism [30]. Fig. 1(c) shows a pipeline example with *n_micro_batch* micro-batches for each subgraph. To reduce memory consumption, forward and backward micro-batches are interleaved [14], and *max_ongoing_micro_batch* limits the number of forward micro-batches on the flight.

Recomputation (Activation Checkpointing) [27] is a schedule that trades computation for memory. It frees forward subgraph activations after execution and recomputes when intermediate activations are required in backward pass.

III. Proteus Overview

Fig. 2 shows an overview of Proteus, a simulation framework towards accurate performance modeling for distributed DNN training. Given that the performance of distributed DNN training is heavily influenced by the chosen parallelization strategy, explicitly modeling the strategy is crucial for an effective performance model. Leveraging the hierarchical property of operator- and subgraph-level strategies, Proteus introduces a unified representation, *strategy tree*, to model complex parallelization strategies (Section IV).

The *execution graph compiler* (Section V) serves as a bridge between high-level parallelization strategies and low-level execution. It takes both the DNN model and strategy tree as inputs, compiling DNN layers into tensors and operators. The compiler automatically inserts communication operators between tensors and generates a distributed execution graph.

In Section VI, we first discuss the characterization and impact of runtime behaviors, then introduce Proteus's *hierarchical topo-aware executor*, which simulates the schedule of the execution graph and predicts the training throughput. During simulation, it adapts operator cost, which is first obtained with the *op estimator* (Section VII), considering the cluster configuration and dynamic runtime behaviors.

IV. Strategy Tree

This section introduces *strategy tree*, which serves as a unified representation for modeling complex parallelization strategies. Moreover, the hierarchical tree structure also makes it easier for users to write parallelization strategies.

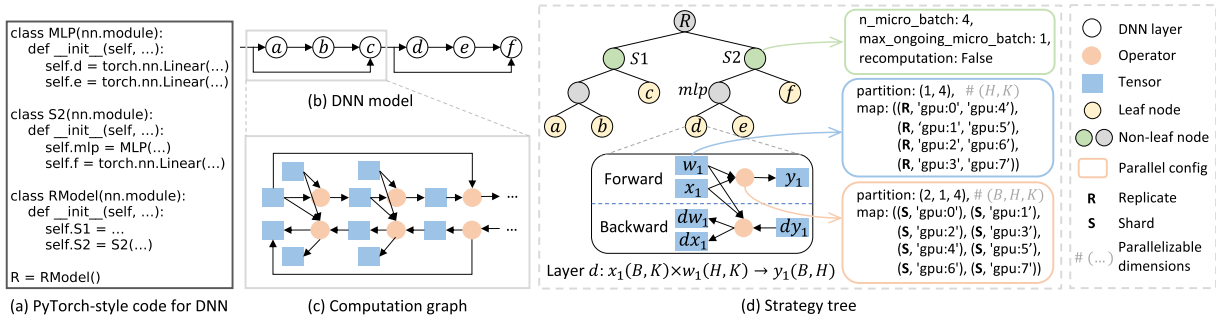


Fig. 3. (a) The PyTorch-style code for a DNN model. (b) The architecture of the DNN model with 6 layers (described by (a)). (b) The computation graph of (a), including forward and backward computations. (c) The strategy tree of (a). Parallel configurations are assigned to non-leaf nodes and leaf nodes.

A. Tree Representation and Construction

Fig. 3 shows a DNN model along with its PyTorch-style code, computation graph, and corresponding strategy tree. DNN model architectures (Fig. 3(b)) inherently possess a nested structure in their construction, as evident from their code (Fig. 3(a)). For instance, the linear layers d and e are encapsulated within the mlp module. This structured information can significantly simplify the definition of parallelization strategies. However, such structural details are often lost in conventional computation graph representations. To address this, we propose the strategy tree, a hierarchical tree representation that models complex parallelization strategies while preserving the nested structure of DNN models. We will further discuss and compare strategy tree with prior works in Section IV-D.

The strategy tree is automatically constructed based on the PyTorch-style model code. Proteus provides a PyTorch-style API to define DNN models, which can then be traced to build the corresponding strategy tree. As illustrated in Fig. 3(d), the strategy tree is a n-ary tree and the root node R represents the whole model.

A leaf node models the forward and backward computation graphs of individual DNN layers. It captures all forward and backward operators, along with the tensors they produce and consume. Proteus represents tensors by their shape, and operators by a set of unique parallelizable dimensions extracted from input and output tensors (Section II). As illustrated in Fig. 3(d), the leaf node d captures the tensors and operators of linear layer d.

A non-leaf node models a subgraph, which represents the forward and backward computation graphs of multiple DNN layers. Each non-leaf node corresponds to a specific DNN module, maintaining the hierarchical structure of the model. For example, the layers d and e form a non-leaf node corresponding to the mlp module, while the module mlp and layer f together constitute another non-leaf node representing the $S2$ module. This alignment between non-leaf nodes and DNN modules preserves the model's original structure, enabling more intuitive and accurate representation of complex parallelization strategies.

B. Parallel Configuration

A parallel configuration defines how different components are parallelized. Operator-level strategies are specified with *computation* and *memory config* in leaf nodes, subgraph-level strategies are assigned to non-leaf nodes with *schedule config*. The complete parallelization strategy is composed of parallel configurations for all tree nodes.

Computation/Memory Config: Computation (memory) configs are assigned to operators (tensors) in leaf nodes. It contains two aspects: *partition* and *map*. The *partition* ($\mathcal{P}$) defines the degree of parallelism in each dimension and splits the operator (tensor) into $|\mathcal{P}|$ disjoint parts. Each part will be mapped to one or more devices defined by *map*, namely shards on one device or replicates on a device group. In Fig. 3(d), the computation config partitions the B and K dimensions of the forward operator into 2 and 4 parts respectively, and shard each part on one GPU device.

Memory config defines the real placement of a tensor. With this separated memory config, Proteus is able to express the space of memory optimization.

Schedule Config: Schedule config specifies the subgraph-level strategy of a subgraph, with only one config needed for each non-leaf node due to the dual structure of the forward and backward subgraphs. The config has three aspects (Fig. 3(d)): *n_micro_batch* denotes the number of micro-batches consumed by the subgraph, Since executing forward micro-batches increases memory consumption, *max_ongoing_micro_batch* limits the maximum number of forward micro-batches executed before each corresponding backward micro-batch at any time, and *recomputation* indicates whether to use activation checkpointing.

Non-leaf nodes on the tree have a schedule config that is propagated from the parent node unless explicitly defined by the user. In particular, the schedule config on a non-leaf node is independent of the configs on leaf nodes. Strategy propagation will be discussed in Section VII.

C. Programming Primitive

Proteus provides several primitives to program the strategy tree efficiently. The following shows an example to program the parallelization strategy of Fig. 3(d):

```
R, ... = proteus.compile(dnn_model)
# define parallelization strategy
R.S2.mlp.d.split(dim=[0, 2], partition=[2, 4])
R.S2.mlp.e.split(dim=[0, 1], partition=[2, 4])
R.S2.mlp.d.map(dev_mesh)
R.S2.mlp.e.map(dev_mesh)
R.S2.schedule(n_micro_batch=4, max_ongoing_micro_batch=2)
R.S2.recompute(False)
```

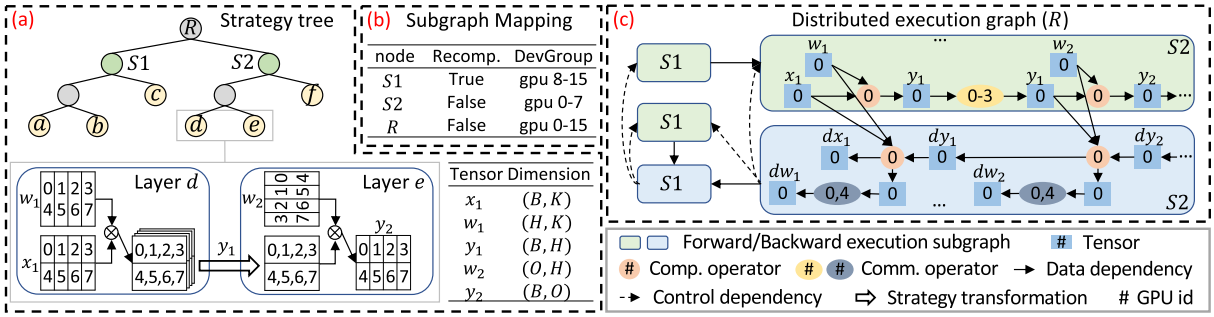


Fig. 4. The strategy tree and corresponding distributed execution graph of DNN model in Fig. 3.

Proteus first obtains strategy tree R with `.compile()`. The leaf and non-leaf nodes make it easier to specify operator- and subgraph-level strategies. Proteus then uses the primitive `.split(dim, partition[, item])` to partition an operator and `.map(dev_mesh[, item])` to map a node to a device mesh `dev_mesh`. Tensors and backward operators can be operated by specifying the optional `item`. With these two primitives, users can easily specify the parallel configurations for leaf nodes. Proteus provides `.schedule(n_micro_batch, max_ongoing_micro_batch)` and `.recompute()` to program the parallel configurations for non-leaf nodes.

D. Discussion and Comparison

Prior automated parallelization works [11], [28], [31] have proposed comprehensive parallelization space as listed in Table I. However, they focus on searching optimal strategy based on the computation graph, rather than enabling users to define custom parallelization strategies. This limitation becomes significant when considering performance modeling, where users need the flexibility to manually program and study specific strategies across a vast space of possibilities. The computation graph used in prior works is not easily adaptable for user-defined strategy specification due to two reasons. First, some layers are converted into multiple operators on the computation graph, making it challenging for users to directly manipulate and specify strategies. Second, it is difficult to annotate subgraph-level strategies for arbitrary subgraphs within the plain computation graph. In contrast, strategy tree provides a hierarchical representation that supports comprehensive complex strategies and preserves the original DNN structure for ease-of-programming. GSPMD [32] develops powerful programming APIs to specify parallelization strategies for different DNN layers, but the subgraph-level strategy is only supported for identical DNN blocks with tailored *vectorized_map* API. Furthermore, changing parallelization strategy also takes great efforts to rewrite the model. Our strategy tree unifies parallelization strategies at different levels and decouples parallelization strategy from model expression. This design allows Proteus to efficiently alter parallelization strategies for a given DNN by solely adjusting the strategy tree.

V. Execution Graph Compiler

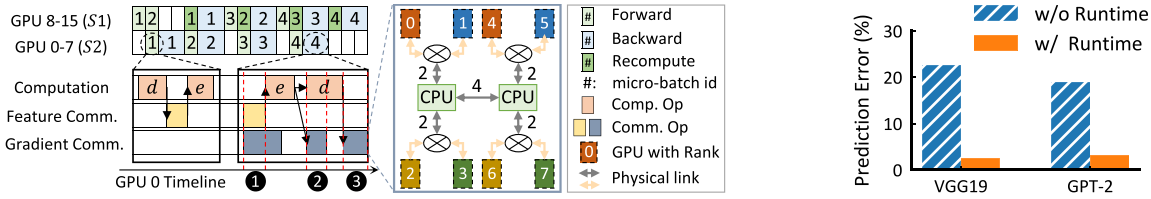
This section describes Proteus's *execution graph compiler*, which servers as a bridge between high level parallelization strategy and low level execution. Given a strategy tree, the compiler creates a distributed execution graph by splitting tensors and computation operators, and inserting communication operators and control dependencies. This graph is similar to execution graphs proposed in previous works [33], [34], [35], except that we represent operators and tensors across multiple devices in a single graph.

A. Graph Compilation

Fig. 4 illustrates the workflow of execution graph compiler. Proteus first divides the DNN model into disjoint subgraphs based on its *DevGroup*, which defines a set of devices, in order to parallelize the computations of different micro-batches. The DevGroup of a tree node is composed of all of its children nodes' DevGroups. Proteus splits all divisible nodes in breadth-first order from root node. A node cannot be divided unless all of its children nodes share some devices. Fig. 4(b) shows the DevGroups of three nodes. The DevGroup of node $S2$ is "gpu 0–7" because layer d and e are partitioned and mapped to these devices. The root node R is divided into 2 subgraphs since node $S1$ and $S2$ share no devices and they are not divisible.

Proteus then compiles each subgraph into a forward and backward execution subgraph as showed in Fig. 4(c). Tensors and operators are split into small partitions such that each partition resides on and is executed by one device. Communication operators, data and control dependencies are added to ensure the computational equivalence.

Data dependency: Each tensor and operator has a parallel configuration that defines the partition and mapping, as discussed in Section IV-B. Due to the data dependency between tensors and operators, Proteus also infers a parallel configuration for each input and output tensor of operators. Once the two parallel configurations of a tensor are inconsistent, Proteus automatically inserts communication operators via *strategy transformation* to adjust the parallel configuration, otherwise reusing original tensors. Fig. 4 shows the strategy transformation of tensor y_1. Layer e partitions y_1 into 2 parts and each part replicates on 4 GPUs, but layer d partitions y_1 into 8 partial tensors on 1



(a) Execution timeline. **(b)** Prediction error.

Fig. 5. (a) The execution timeline and runtime behaviors of the distributed execution graph in Fig. 4. The number on gray link denotes the number of groups that share the link. (b) The prediction errors for modeling runtime behaviors or not with Proteus.

GPU. Proteus adds communication operators between y_1 in the execution subgraph of $S2$ to handle this inconsistency.

For a subgraph with recomputation enabled, Proteus compiles it into two forward and one backward execution subgraphs and adjust the data dependency accordingly. The backward subgraph depends on one forward subgraph (i.e., recomputation subgraph) and the other subgraph can be immediately released after execution (e.g., S 1in Fig. 4(c)).

Control dependency: Control dependencies are inserted between execution subgraphs to follow the training schedule defined by the schedule config in non-leaf nodes. First, the forward subgraphs are control dependent on their corresponding backward subgraphs to limit peak memory consumption. Second, Proteus also adds control dependency for recomputation subgraphs such that they are executed immediately before the backward subgraphs. In Fig. 4(c), node $S1$ has two forward subgraphs and one of them is control dependent on the backward of node $S2$.

B. Strategy Transformation

Strategy transformation, a.k.a resharding [28], transforms tensors to desired parallel configurations with appropriate communication primitives. Proteus automatically infers collective communication primitives (e.g., All-Reduce [36]) and groups with communication pattern matching, failing back to point-to-point communication if necessary. Resharding has also been extensively discussed in recent works [28], [37].

VI. Hierarchical Topo-Aware Executor

This section describes Proteus's *Hierarchical Topo-Aware Executor* (HTAE), which simulates the schedule and runtime behaviors of a distributed execution graph and predicts the training throughput.

A. Performance Characterization

Before introducing the design of HTAE, we first characterize the performance of distributed DNN training using the example of Fig. 5(a), which illustrates the execution timeline and runtime behaviors of Fig. 4. The forward and backward execution subgraphs are interleaved and the execution of $S1$ and $S2$ is parallelized on different GPU groups. Operators in Fig. 4(c) consist of three types that can be executed simultaneously: computation, feature and gradient communication operators, and they are scheduled into three streams following data dependency. Modeling the training performance is to model the execution timeline, including schedule, computation and communication.

Runtime Behavior: Prior work [11], [23], [24], [25] assumes that the operator cost is fixed and focuses on modeling the performance of single operator. The training speed of a DNN is the summation of all the operators' costs. However, *runtime behavior*, which is ignored in prior work, has emerged as a critical aspect determining training performance under today's sophisticated parallelization strategies and optimizations. It is crucial to model runtime behaviors towards an accurate performance predictor since they can affect the execution cost of operators. Fig. 5(b) shows that ignoring runtime behaviors results in large prediction error on a cluster with 32 GPUs.

We find that major runtime behaviors can be categorized into two types. First, *bandwidth sharing* describes the scenarios that different communication operators compete for bandwidth (Fig. 5(a)❶,❸). Second, *comp-comm overlap* refers to the overlap of computation and communication operators (Fig. 5(a)❷). In addition, different computation operators could be overlapped on single GPU [38], Proteus does not model such scenario since it is rarely used in distributed DNN training. Fig. 5(a)❸ shows an example of bandwidth sharing by mapping gradient communication operators to a single node machine. The gradient communication includes 4 groups indicated by the GPU color: {{0, 4}, {1, 5}, {2, 6}, {3, 7}}, and their costs rise due to the competition for available bandwidth of scarce physical links.

Proteus is the first system to study and model runtime behaviors for distributed DNN training. Unlike prior analytical frameworks [24], [25], Proteus predicts training performance via simulation since runtime behaviors only occur during execution.

B. Simulator Design

Fig. 6 shows the design of Proteus's two level simulator, HTAE. The first level is *scheduler*, which consists of several second level *executors*. Different *schedulers* can time-share *executors*. To predict the performance, HTAE first gets single operator cost with *op estimator* (Section VII) and then simulates the schedule of subgraphs and operators to discover runtime behaviors. During simulation, operator cost are adapted to model runtime behaviors considering the cluster topology.

Cluster Configuration: Cluster Configuration describes the topology of training cluster. There are two types of configurable parameters in device topology. For intra-node topology, we can

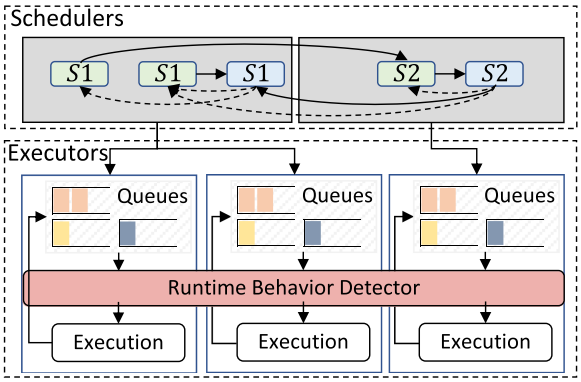


Fig. 6. The design of hierarchical topo-aware executor.

set device type, device memory, number of devices in a node and the intra-node connection, which describes the physical connections among devices (e.g., GPUs and CPUs). For inter-node topology, we can specify the number of nodes and inter-node connection bandwidth.

Scheduler: Each scheduler is assigned several forward and backward execution subgraphs, and it interleaves the execution of them based on data and control dependencies to balance micro-batch parallelism and peak memory consumption. The scheduler first selects current execution state (forward or backward), then it chooses one subgraph from available dependency-free execution subgraphs. It alternates different backward subgraphs and prefers forward subgraph that enables backward execution. After determining the subgraph to be executed, the scheduler dispatches initial tasks to executors and begin executing.

Executor: The *executor* schedules the execution of operators for a subgraph and records the peak memory consumption. Each executor contains a computation queue, a feature communication queue and a gradient communication queue (Fig. 6). Operators in different queues can be executed concurrently such that achieving *comp-comm overlap*. By separating feature and gradient communication queue, Proteus makes it possible to overlap feature and gradient communication and avoid feature communication blocked by gradient communication.

The executor executes computation and communication alternatively. It pops a computation operator from the queue for computation execution and pops one feature and one gradient communication operator at the same time for communication execution. These operators are first sent to the *runtime behavior detector* to check runtime behaviors and executed afterwards. During execution, the operator cost will be accumulated to count the time cost for each queue separately. The execution of operators will decrease the number of dependencies of their consumers and dependency-free operators will be put into the corresponding queue.

Memory Consumption: Proteus predicts whether a parallelization strategy will out-of-memory (OOM) by monitoring the memory consumption of executors. During execution, each operator reads and writes some tensors. HTAE monitors executor memory footprint by recording these tensor activities. When writing a new tensor, HTAE tracks its memory consumption and reference counter. The memory will be released when the reference counter decreases to zero.

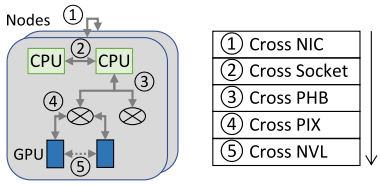


Fig. 7. The hierarchy of bandwidth sharing.

C. Modeling Runtime Behaviors

As previously discussed, the operator cost may change during execution due to complex runtime behaviors. The *runtime behavior detector* checks runtime behaviors for all operators and adapt operator cost accordingly. To enable efficient detection, it keeps execution history records of different execution streams.

Bandwidth Sharing: There are two types of bandwidth sharing. One is inside a group of gradient or feature communication operators (Fig. 5(a)❸), and the other is between a group of gradient and feature communication operators (Fig. 5(a)❶). These operators transfer datas within different device groups and compete for bandwidth of shared physical links. To model this behavior, Proteus assumes that concurrent operators fairly share the bandwidth of a physical link and detects how many communication groups share a link during execution. We find this assumption generally holds in practice.

Proteus first checks bandwidth sharing for feature and gradient communication operators separately by mapping communication groups to cluster topology. Fig. 7 shows the hierarchy of physical links in a cluster and Proteus detects bandwidth sharing following this hierarchy. Proteus starts from NIC bandwidth sharing. Each communication group is split into sub-groups such that each sub-group is composed of devices in the same node. The groups that consist of more than two sub-groups fairly share the bandwidth of NIC. Proteus checks all the physical links from top to bottom.

Proteus finally detects the intersection effects of feature and gradient communication groups. Since the communication volumes and operations of these groups may different, Proteus only adapt operator cost for the overlapped parts. The detection algorithm is the same as the first step, except for that the communication groups include both feature and gradient communications.

Comp-Comm Overlap: In distributed DNN training, computation and gradient communication operators may overlap, because gradient communication operators can be launched asynchronously and feature communication operators usually block the computation stream. To detect *comp-comm overlap*, Proteus keeps the start and end time of operators. When executing a computation (communication) operator, Proteus considers it *overlapped* if it finds a gradient communication (computation) operator in execution.

Proteus introduces an overlap factor γ to model the effect of *comp-comm overlap*. When finding an operator *overlapped*, its

cost will increase by γ. This is motivated by the observation that operator costs increase by about the same percentage on average during overlap.

To obtain γ, we profile the speeds of backward pass with and without overlapping in data parallel training and γ is set to the increase ratio. As γ is fixed for the type of machine and DNN model, we can get γ in advance with few cost. Prior works [39], [40] also try to model the effect of *comp-comm overlap*. For example, Pollux [39] introduces a learnable parameter η to model the data parallel training speed by combining computation time T_{grad} and gradient communication time T_{sync} with $({T_{grad}}^{\eta} + {T_{sync}}^{\eta})^{1/\eta}$. These works target on data parallel training, and we find our simple formulation works pretty well with complex parallelization strategies (Section VIII-D). In addition, the overlapping adjustment is implemented as a standalone function that takes overlapped operators as inputs. This modular design also allows for the integration of more advanced estimation techniques in the future.

VII. Implementation

Proteus is implemented as a standard python library ($\sim$ 9K LoC). Proteus follows PyTorch [15] API to build model, and is open-sourced at https://github.com/JF-D/Proteus.

Strategy Propagation: Proteus develops a strategy propagation algorithm to ease the programming difficulty of parallelization strategies. For a complete parallelization strategy, users are required to specify parallel configurations for critical leaf and non-leaf nodes. Proteus will propagate the parallel configurations to the other nodes. The identification of critical nodes relies on expert knowledge. We leave the automatic identification as future work.

Proteus first propagates parallel configurations from top to bottom following tree structure. The schedule config of a non-leaf node is inherited from its parent node unless explicitly defined. Proteus then propagates parallel configurations among leaf nodes following data dependency. The propagation proceeds in topological order and includes two steps: forward graph propagation and backward graph propagation. Proteus infers the memory config of a tensor according to its producer's computation config and infers the computation config of an operator according to its inputs' memory config.

Op Estimator: The *op estimator* predicts costs for all operators in the distributed execution graph using a profiler and analyzer. The profiler measures computation operator time costs on target hardware with negligible cost. Proteus can be also be extended to adopt existing performance models for single operator cost estimation [17], [18], [19]. The analyzer estimates communication costs using the α-β model [41]. It estimates the bandwidth of a communication group according to the detailed cluster topology. When estimating the time cost of a collective operation, a correction factor is applied to revise the bandwidth to reflect the characteristics of different collective operations. To simplify implementation, we utilize NCCL topo detection algorithm [36] to find all the communication channels of a communication group and its bandwidth is the summation of these channels.

TABLE II
Overview of Hardware Configurations Evaluated

Config	#Node	#GPU per Node	Intra-node	Inter-node
HC1	1	8×TitanXp	PCI-e	N/A
HC2	4	8×V100	NVLink	100 Gbps
HC3	2	8×A100	NVLink	200 Gbps

TABLE III
Overview of the Six Benchmark Models Evaluated

Task	Model	#Params	Dataset
Vision	ResNet50 [1]	25.6M	Synthetic
	Inception_V3 [42]	23.8M	
	VGG19 [43]	137M	
NLP	GPT-2 [44]	117M	
	GPT-1.5B [44]	1.5B	
Recommendation	DLRM [45]	516M	

VIII. Evaluation

A. Methodology

All the experiments are conducted with PyTorch 1.8 (CUDA 10.1, cuDNN 7.6.5 and NCCL 2.7.8).

Benchmarks: Table III summarizes the six representative DNN models that we used as benchmarks, they are widely used in prior works [11], [13], [19], [25]. These models cover widely used basic units such as Attention, MLP, Conv and Embedding, and can scale to different sizes and cover popular DNNs. We evaluate throughput with synthetic dataset, which ignores the data loading latency. Modeling real-world dataset is orthogonal to Proteus.

Hardware Configurations: Proteus is evaluated across three different hardware configurations. Table II summarizes the cluster type and size, intra- and inter-node connections.

B. Simulation Accuracy

To evaluate Proteus across a wide variety of parallelization strategies, we evaluate each model with 2 popular parallelization strategies. One is most commonly used parallelization strategy ($S1$), the other is the optimal expert-designed parallelization strategy ($S2$). These two strategies covers hybrid data, model, and pipeline parallelism and memory optimizations. Since Proteus is aimed at accurate performance modeling rather than discovering new parallelization strategies, and implementing entirely new parallelization strategies is difficult, we do not evaluate Proteus on less commonly used parallelization strategies. But the parallelization strategies we tested already cover both operator- and subgraph-level strategies.

Since the most commonly used parallelization strategy is data parallelism, Proteus uses data parallelism or its variants as $S1$ for six DNNs. To enable data parallelism training of large model, Proteus combines memory optimization (ZeRO [10]) and recomputation in $S1$ to evaluate GPT-1.5B. Expert-designed parallelization strategies ($S2$) exhibits more diverse patterns.

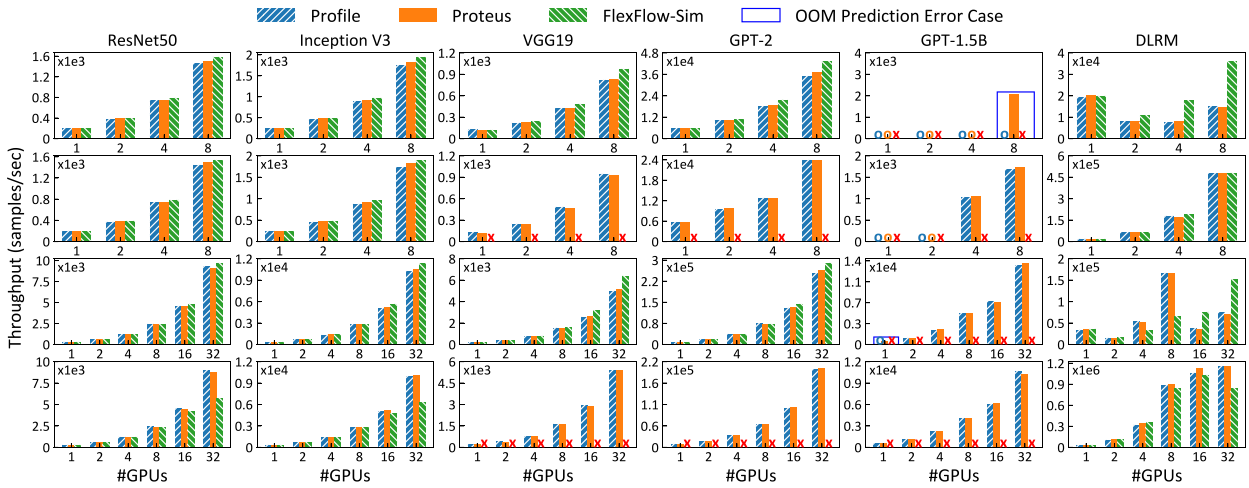


Fig. 8. Throughput of DNN models on different hardware configurations with distinct parallelization strategies. The parallelization strategies from top to bottom row are $S1$ ($HC1$), $S2$ ($HC1$), $S1$ ($HC2$) and $S2$ ($HC2$), respectively. **o**: out of memory, $\times$: unsupported strategy.

TABLE IV
COMPARISON OF AVERAGE AND MAXIMUM PREDICTION ERROR OF PROTEUS AND FLEXFLOW-SIM (FF-SIM)

Model	Strategy	Avg Error (%)		Max Error (%)	
		Proteus	FF-Sim	Proteus	FF-Sim
ResNet50	*S1*	2.09	3.59	6.00	8.69
	S2	2.30	5.98	5.77	35.65
Inception_V3	*S1*	3.24	5.53	7.52	11.71
	S2	3.19	6.57	7.97	36.73
VGG19	*S1*	1.97	8.11	4.97	28.17
	S2	1.68	✗	6.64	✗
GPT-2	*S1*	2.56	6.97	6.20	24.14
	S2	2.31	✗	11.38	✗
GPT-1.5B	*S1*	3.91	✗	8.09	✗
	S2	3.65	✗	9.92	✗
DLRM	*S1*	5.07	48.14	14.68	137.89
	S2	4.55	14.05	11.44	114.63

Each strategy contains 15 results on 3 hardware configurations.

ResNet50 and Inception_V3 partitions data and output channels, while VGG19 and GPT-2 partitions data, output channels and reduction dimensions for computation parallelization. The $S2$ of GPT-1.5B combines op shard, pipeline and recomputation. DLRM partitions huge embedding table in $S2$ to optimize memory footprint.

Fig. 8 shows the simulation results of various DNN models on two hardware configurations ($HC1$ and $HC2$) and Table IV displays the overall results on three hardware configurations. Proteus delivers an accurate performance model and achieves 3.0% average prediction error for training throughput. Out of 180 simulation results, Proteus's estimation of OOM is incorrect only under 2 cases (blue box in Fig. 8).

Proteus is the first standalone simulator that targets on simulating complex parallelization strategies. The most related and representative cost-model and simulator is Paleo [25] and FlexFlow [11], respectively. Paleo [25] is an analytical cost-model. It delivers high prediction error even on single GPU (ResNet50 (59.8%), Inception_V3 (40%)) and does not support GPT, DLRM models and complex parallelization strategies. Therefore, we did not dive into Paleo and show the results. FlexFlow [11] is an automated parallelization framework on SOAP space. To compare generated parallelization strategies, it internally tailors a simulator to simulate the training throughput.

To compare Proteus and FlexFlow, we re-implement its simulator as FlexFlow-Sim, since it is hard to run a manually specified parallelization strategy with FlexFlow. FlexFlow-Sim also uses profiled operator cost for fair comparison. To support realistic simulation, FlexFlow-Sim inserts collective communication operators for strategy transformation instead of point-to-point operators as described in FlexFlow paper. The comparison results are shown in Fig. 8 and Table IV. The average prediction error for FlexFlow is 12.4%, which is 9.4% higher than Proteus. Among all the test cases, the maximum error is 14.7% and 137.9% for Proteus and FlexFlow, respectively. For the total 180 training tasks, FlexFlow fails to estimate the performance of 1/3 of them. Fig. 8 also shows that the prediction error of FlexFlow-Sim becomes larger as the number of GPUs increases.

We find that Proteus outperforms FlexFlow mainly in three aspects. 1) Proteus can be applied to a much larger parallelization strategy space with the abstraction of strategy tree. 2) FlexFlow ignores complex runtime behaviors thus cannot accurately model the training throughput. 3) FlexFlow's communication bandwidth estimation ignores fine-grained cluster topology. For example, FlexFlow delivers high prediction error for DLRM model, where communication dominates.

C. Parallelization Strategy Comparison

Comparing the training throughput of various parallelization strategies is an important problem in designing and understanding high performance parallelization strategies. In this section, we use GPT-2 as benchmark because GPT model is the most popular and widely used model to study all kinds of parallelization strategies and these strategies can generalize to other models. In these experiments, we select 4 parallelizable dimensions across operator- and subgraph level strategies and represent the parallelization strategy as $DP \times MP \times PP(n\_micro\_batch)$ and DP, MP and PP is the degree of data, model and pipeline

TABLE V
THE PREDICTION ERROR AND RANK OF THROUGHPUT FOR GPT-2 WITH DIFFERENT PARALLELIZATION STRATEGIES ON *HC1* AND *HC2*

HC1			*HC2*		
Strategy	Error	Rank	Strategy	Error	Rank
8×1×1 (1)	2.34%	2 / 2	16×1×1 (1)	3.38%	1 / 1
4×2×1 (1)	2.92%	1 / 1	8×2×1 (1)	2.89%	2 / 2
2×4×1 (1)	2.43%	3 / 3	4×4×1 (1)	5.87%	3 / 3
1×8×1 (1)	1.77%	5 / 5	2×8×1 (1)	3.88%	4 / 4
2×2×2 (1)	2.54%	6 / 6	8×1×2 (4)	3.60%	6 / 6
2×2×2 (2)	1.60%	4 / 4	8×1×2 (8)	2.99%	5 / 5
			2×4×2 (4)	5.64%	7 / 7

Rank is denoted in the format of truth / predicted rank.

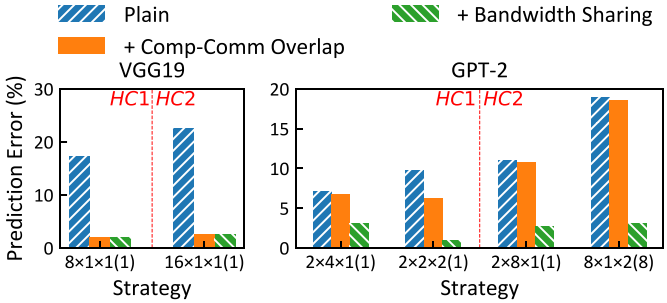


Fig. 9. The prediction error of VGG19 and GPT-2 with different components. Plain: Proteus without runtime behavior detector.

parallelism. The global batch size is 8 and 64 for *HC1* and *HC2* respectively.

Table V shows the simulation results of GPT-2 with various parallelization strategies. For these parallelization strategies, Proteus can accurately model the performance and achieves 3.2% average prediction error. Order preservation is an important feature in strategy comparison and Proteus maintains the rank of diverse parallelization strategies. Table V demonstrates that *HC2* prefers data parallelism training, since hybrid model parallelism shares the bandwidth of IB net, and pipeline parallelism introduces bubbles during training, which will decrease the training throughput. The simulation results also confirms that pipeline efficiency can be improved by injecting more micro batches. *HC1* consists of a single NUMA node, and the 2-way model parallelism can fully utilize the QPI links between two CPU sockets, thus achieving highest throughput.

D. Runtime Behavior Ablation Study

To study the effective of *runtime behavior detector*, we test the throughput of VGG19 and GPT-2 on *HC1* and *HC2* with different parallelization strategies. For VGG19, we use batch size 32 per GPU with data parallelism training. For GPT-2, the global batch size is 8 and 64 on *HC1* and *HC2* with hybrid op shard and pipeline parallelism. Fig. 9 shows that *runtime behavior detector* can greatly improve the simulation accuracy of throughput (average error: Plain (14.4%) versus Proteus (2.4%)). VGG19 is very sensitive to *comm-comp overlap*, hence introducing overlap factor can significantly improve the prediction accuracy. As there is no *bandwidth sharing* in data parallelism training, prediction error of VGG19 keeps after adding *bandwidth sharing*. In contrast, GPT-2 is more sensitive to *bandwidth sharing* which is especially common in complex parallelization strategies. Therefore, the throughput prediction error reduces remarkably after modeling *bandwidth sharing*.

TABLE VI
SIMULATION COST OF PROTEUS ON *HC2* IN *SECONDS*

#GPUs	VGG19			GPT-2		
	compile	exe.	total	compile	exe.	total
1	0.033	0.006	0.039	0.188	0.070	0.258
2	0.053	0.407	0.460	0.341	0.450	0.792
4	0.114	0.530	0.644	0.504	0.692	1.196
8	0.170	0.563	0.733	1.008	0.873	1.881
16	0.336	0.630	0.966	1.966	1.172	3.138
32	0.777	0.921	1.698	4.143	2.123	6.265

E. Simulation Cost

To evaluate the simulation cost of Proteus, we measure the time it takes to evaluate VGG19 and GPT-2 on *HC2* with data parallelism. Since the cost of computation operators can be profiled in advance, we only evaluate the time cost of *execution graph compiler* and *HTAE*. Table VI demonstrates that Proteus takes seconds to simulate the performance of DNNs with a large number of GPUs, and the time increases linearly in the number of GPUs. We believe this cost is acceptable to evaluate a specified parallelization strategy since Proteus provides a fine-grained simulation without requiring GPU resources. In contrast, profiling a general parallelization strategy takes lots of efforts and GPU resources.

IX. RELATED WORK

Handcrafted Parallelization Strategies are designed to optimize distributed DNN training. One wired trick [46] introduces model parallelism for linear layers to accelerate AlexNet. Megatron-LM [6] presents an expert-designed strategy to expedite transformer models combining data, model and pipeline parallelism. DeepSpeed [29] introduces ZeRO to reduce memory footprint by partitioning model states across data parallel processes. Recomputation [27] utilizes tensor rematerialization to decrease memory consumption. Proteus can model the performance of these manual designed strategies thus assisting their analysis and optimization.

Automatic Parallelization: FlexFlow [11] and Tofu [12] proposes SOAP and partition-n-reduce space to parallelize operators. GSPMD [32] introduces a more general parallelization space by partitioning all parallelizable dimensions of tensors. DAPPLE [13] and PipeDream [14] optimize parallelization strategies in data and pipeline parallelization space. Alpa [28] combines data, model and pipeline parallelism and proposes a inter-operator and intra-operator parallelization space. Existing automatic approaches focus on exploring the space of computation parallelization, while our work introduces a unified parallelization strategy space considering computation parallelization and memory optimization at operator level and schedule at subgraph level.

Performance Model: Previous works propose analytical performance models for DNN training on single GPU [17], [18], [19], [20], [47] or on multiple GPUs with data parallelism

or hybrid data and model parallelism [23], [24], [25]. These approaches are not applicable to increasingly complex training workload and strategies. FlexFlow [11] introduces a simulation model to estimate the cost of a SOAP strategy, but it is not designed to capture the cost of general strategies and runtime behaviors. Proteus aims to provide a general simulation performance model for various parallelization strategies.

X. Conclusion

In this work, we present Proteus to simulate the performance of distributed DNN training strategies on diverse clusters. Proteus features *strategy tree* to model unified parallelization strategy space and *hierarchical topo-aware executor* to model runtime behaviors of computation and communication operators accurately. We can leverage Proteus to analyze and optimize the performance of general parallelization strategies.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[2] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 6000–6010.

[3] A. Radford et al., "Improving language understanding by generative pre-training," Tech. Rep., 2018.

[4] T. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 1877–1901.

[5] J. Dean et al., "Large scale distributed deep networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1223–1231.

[6] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training multi-billion parameter language models using model parallelism," 2019, *arXiv: 1909.08053*.

[7] 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.

[8] Z. Li et al., "TeraPipe: Token-level pipeline parallelism for training large-scale language models," 2021, *arXiv:2102.07988*.

[9] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory optimizations toward training trillion parameter models," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–16.

[10] J. Ren et al., "ZeRO-offload: Democratizing billion-scale model training," 2021, *arXiv:2101.06840*.

[11] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks.," *Proc. Mach. Learn. Syst.*, vol. 1, pp. 1–13, 2019.

[12] M. Wang, C.-C. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–17.

[13] S. Fan et al., "DAPPLE: A pipelined data parallel approach for training large models," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Republic of Korea, 2021, pp. 431–445.

[14] D. Narayanan et al., "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 1–15.

[15] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, Art. no. 721.

[16] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[17] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. Narayanan, and K. Srinathan, "A performance prediction model for the CUDA GPGPU platform," in *Proc. Int. Conf. High Perform. Comput.*, 2009, pp. 463–472.

[18] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 382–393.

[19] G. Liu, S. Wang, and Y. Bao, "SEER: A time prediction model for CNNs from GPU kernel's view," in *Proc. 30th Int. Conf. Parallel Architectures Compilation Techn.*, 2021, pp. 173–185.

[20] H. Jia, Y. Zhang, G. Long, J. Xu, S. Yan, and Y. Li, "GPURoofline: A model for guiding performance optimizations on GPUs," in *Proc. 18th Int. Conf. Parallel Process.*, Rhodes Island, Greece, Springer, 2012, pp. 920–932.

[21] T. Chen et al., "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 578–594.

[22] R. Baghdadi et al., "A deep learning based cost model for automatic code optimization," in *Proc. Mach. Learn. Syst.*, 2021, vol. 3, pp. 181–193.

[23] Z. Pei, C. Li, X. Qin, X. Chen, and G. Wei, "Iteration time prediction for CNN in multi-GPU platform: Modeling and analysis," *IEEE Access*, vol. 7, pp. 64788–64797, 2019.

[24] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, "Performance modeling and scalability optimization of distributed deep learning systems," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2015, pp. 1355–1364.

[25] H. Qi, E. R. Sparks, and A. Talwalkar, "PALEO: A performance model for deep neural networks," in *Proc. 5th Int. Conf. Learn. Representations*, Toulon, France, 2017.

[26] V. Elango, "Pase: Parallelization strategies for efficient DNN training," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 1025–1034.

[27] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," 2016, *arXiv:1604.06174*.

[28] 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.

[29] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2020, pp. 3505–3506.

[30] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, Art. no. 10.

[31] C. Unger et al., "Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 267–284.

[32] Y. Xu et al., "GSPMD: General and scalable parallelization for ML computation graphs," 2021, *arXiv:2105.04663*.

[33] Z. Lin et al., "Building a performance model for deep learning recommendation model training on GPUs," in *Proc. IEEE 29th Int. Conf. High Perform. Comput., Data, Analytics*, 2022, pp. 48–58.

[34] M. Liang et al., "Mystique: Enabling accurate and scalable generation of production ai benchmarks," in *Proc. 50th Annu. Int. Symp. Comput. Archit.*, 2023, pp. 1–13.

[35] S. Sridharan et al., "Chakra: Advancing performance benchmarking and co-design using standardized execution traces," 2023, *arXiv:2305.14516*.

[36] NVIDIA NCCL, 2021. [Online]. Available: https://developer.nvidia.com/nccl

[37] Y. Zhuang et al., "On optimizing the communication of model parallelism," *Proc. Mach. Learn. Syst.*, vol. 5, 2023.

[38] Y. Ding, L. Zhu, Z. Jia, G. Pekhimenko, and S. Han, "IOS: Inter-operator scheduler for CNN acceleration," in *Proc. Mach. Learn. Syst.*, A. Smola, A. Dimakis, and I. Stoica, Eds., 2021.

[39] A. Qiao et al., "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *Proc. 15th USENIX Symp. Operating Syst. Des. Implementation*, 2021.

[40] C. Yang et al., "PerfEstimator: A generic and extensible performance estimator for data parallel DNN training," in *Proc. IEEE/ACM Int. Workshop Cloud Intell.*, 2021, pp. 13–18.

[41] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[42] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2818–2826.

[43] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[44] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI Blog, Tech. Rep., vol. 1, no. 8, p. 9, 2019.

[45] M. Naumov et al., "Deep learning recommendation model for personalization and recommendation systems," 2019, *arXiv: 1906.00091*.

[46] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," 2014, *arXiv:1404.5997*.

[47] L. Yuan and Y. Zhang, "A locality-based performance model for load-and-compute style computation," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2012, pp. 566–571.

Jiangfei Duan received the BS degree from the University of Chinese Academy of Sciences, Beijing, China, in 2020. He is currently working toward the PhD degree with the Department of Information Engineering, The Chinese University of Hong Kong. His research interests lie in machine learning systems, include efficient distributed DNN training and inference and performance modeling.

Xiuhong Li received the BS and PhD degrees from Peking University, China. He was a senior researcher with Deep Learning Platform Department, Sensetime, from 2019 to 2023. He is currently a research assistant professor with the National Engineering Laboratory for Big Data Analysis and Applications, Peking University. His research interests include deep learning compiler, distributed parallel computing, and high-performance computation on heterogeneous architectures.

Ping Xu received the BS and master's degrees from Tsinghua University, China. His research interests include deep learning compiler and high performance computing.

Xingcheng Zhang received the BEng degree from Tsinghua University, in 2015, and the MPhil degree from The Chinese University of Hong Kong, in 2018. He is currently the research director of the Center of AI Training and Computation, Shanghai AI Lab.

Shengen Yan received the BS degree from the Harbin Institute of Technology, Harbin, China, in 2009, and the PhD degree from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2014. He was a visiting student with NC State University, Raleigh, North Carolina from 2013 to 2014. He was a postdoctoral researcher with Multimedia Lab, Chinese University of Hong Kong, Hong Kong, from 2015 to 2017. He is currently a research associate professor with the Department of Electronic Engineering, Tsinghua University. His research interests include large scale deep learning and high performance computing.

Yun Liang (Senior Member, IEEE) is currently an Endowed Boya distinguished professor with the School of EECS, Peking University, China. His research interests include hardware-software interface with work spanning electronic design automation (EDA), hardware and software co-design, and computer architecture. He was the recipient of the best paper awards with FCCM 2011 and ICCAD 2017 and best paper nominations with PPoPP 2019, DAC 2017, ASPDAC 2016, DAC 2012, FPT 2011, and CODES+ISSS 2008 for his research. He is an associate editor for *ACM Transactions in Embedded Computing Systems (TECS)*, *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, and *Embedded System Letters (ESL)*. He is also with the program committees in the premier conferences in the related domain including MICRO, DAC, HPCA, FPGA, ICCAD, FCCM, ICS, etc.

Dahua Lin received the BEng degree from the University of Science and Technology of China (USTC), in 2004, the MPhil degree from The Chinese University of Hong Kong (CUHK), in 2006, and the PhD degree from the Massachusetts Institute of Technology (MIT), in 2012. He is an associate professor with the Department of Information Engineering, The Chinese University of Hong Kong. Prior to joining CUHK, he served as a research assistant professor with Toyota Technological Institute, Chicago, from 2012 to 2014. His research interest covers machine learning, computer vision, and Big Data analytics. He has published more than 120 papers on top conferences and journals, e.g., ICCV, CVPR, ECCV, NIPS, and *IEEE Transactions on Pattern Analysis and Machine Intelligence*. His work on a new construction of Bayesian nonparametric models has won the best student paper award in NIPS 2010. He also received the outstanding reviewer award in ICCV 2009 and ICCV 2011.