A Survey on Auto-Parallelism of Large-Scale Deep Learning Training

Peng Liang, Yu Tang, Xiaoda Zhang, Youhui Bai, Teng Su, Zhiquan Lai, Linbo Qiao, and Dongsheng Li

***Abstract*— Deep learning (DL) has gained great success in recent years, leading to state-of-the-art performance in research community and industrial fields like computer vision and natural language processing. One of the reasons for this success is the huge amount parameters adopted in DL models. However, it is impractical to train a moderately large model with a large number of parameters on a typical single device. Thus, It is necessary to train DL models in clusters with distributed training algorithms. However, traditional distributed training algorithms are usually sub-optimal and highly customized, which owns the drawbacks to train large-scale DL models in varying computing clusters. To handle the above problem, researchers propose auto-parallelism, which is promising to train large-scale DL models efficiently and practically in various computing clusters. In this survey, we perform a broad and thorough investigation on challenges, basis, and strategy searching methods of auto-parallelism in DL training. First, we abstract basic parallelism schemes with their communication cost and memory consumption in DL training. Further, we analyze and compare a series of current auto-parallelism works and investigate strategies and searching methods which are commonly used in practice. At last, we discuss several trends in auto-parallelism which are promising in further research.**

***Index Terms*—Auto-parallelism, large-scale deep learning model, training technique, parallel and distributed training.**

I. Introduction

Large-scale deep learning (DL) [1] models like ChatGPT[1] has recently drawn a lot of attention for their superior performance in natural language tasks like dialogue, text summarization, translation, and so on. Training large models is hard due to two reasons. On the one hand, their volumes of model parameters exceed the storage capacity of a typical computing device. On the other hand, these models are trained with terabyte (TB) degree datasets, which require several or more GPU years to finish the training process [2]. Thus, research and industrial communities apply distributed training [3] to address this problem. They manually design decent parallelism strategies that make the best efforts to utilize the aggregated computing power of all the available devices. These strategies may consist of schemes such as data parallelism (DP) [4], tensor parallelism (TP) [5], and pipeline parallelism (PP) [6]. However, with the increasing diversity of model types and sizes and the rapid development of deep learning infrastructure, manual strategies designed for specific models and hardware may become inadequate and require redesign. Such redesigns can be time-consuming and require expert engineering experience in deep learning, distributed training, and infrastructure.

Derived from distributed training, auto-parallelism was developed to automatically design strategies. Auto-parallelism, also known as auto-parallelization or automatic parallelization, refers to the process of automatically converting sequential code into multi-threaded or vectorized code to utilize available computing devices. Nowadays, auto-parallelism is frequently used in the DL community for the training and inference of deep neural networks. In the field of deep learning, auto-parallelism refers to automatically generating computation tasks for computing devices using splitting, merging, or re-formalization on the network's computation graph. Auto-parallelism typically generates parallelism strategies by automatically determining partitions of each tensor in the computation graph, inserting communication operations, and scheduling the entire computation process. Auto-parallelism is the ultimate goal of distributed training, as it liberates engineers from the manual design of strategies and empowers industrial departments to train large-scale models efficiently on various computing infrastructures.

In this survey, we provide the problem definition, challenges, basis, classification, and existing works of auto-parallelism. We first present a unified definition of the problem that auto-parallelism solves, which abstracts a wide range of traditional and current auto-parallelism methods. Then, we analyze the challenges of auto-parallelism, including detailed analyses of parallelism schemes and trade-offs between them, load balance problems in heterogeneous topology, topology-aware communication optimization, and the trade-off between runtime and strategy performance in finding decent parallelism strategy. Alpa [7] divides parallelism schemes into two categories: intra-operator parallelism (Intra-P) and inter-operator parallelism (Inter-P). This intuitive classification helps us make a detailed analysis of parallelism schemes. Based on this classification, we comprehensively analyze the basis of a huge body of work on auto-parallelism use. Specifically, we analyze the communication and memory consumption of different schemes. We also

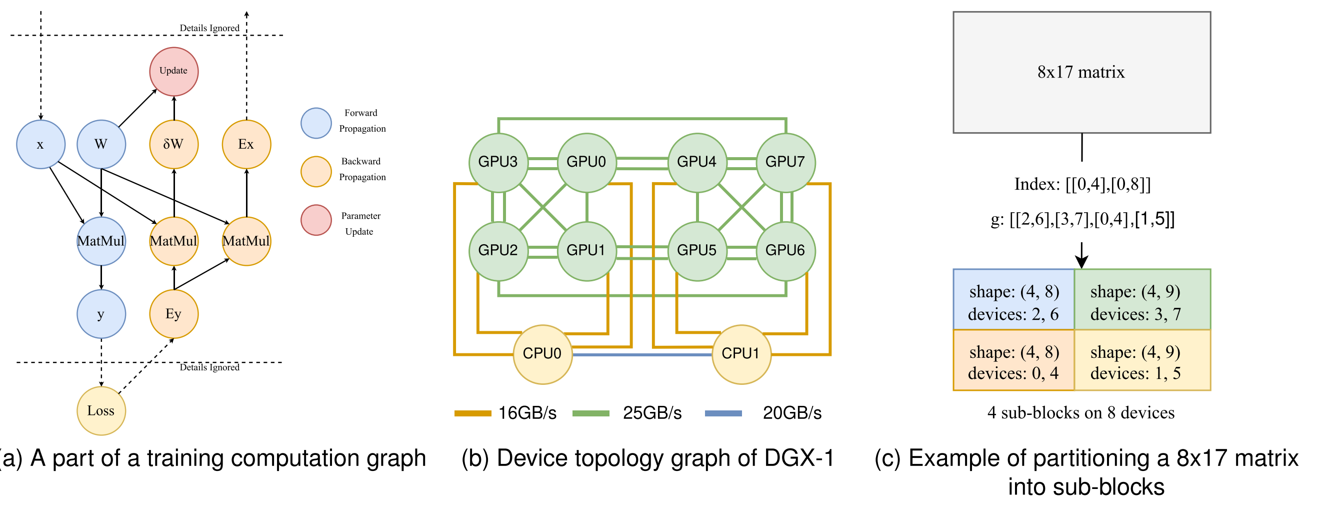
Manuscript received 7 October 2022; revised 21 April 2023; accepted 29 May 2023. Date of publication 26 June 2023; date of current version 5 July 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 62025208. Recommended for acceptance by T. Estrada. *(Corresponding author: Dongsheng Li.)*

Peng Liang, Yu Tang, Zhiquan Lai, Linbo Qiao, and Dongsheng Li are with the State Key Laboratory of Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, China (e-mail: peng_leung@nudt.edu.cn; tangyu14@nudt.edu.cn; zqlai@nudt.edu.cn; qiao.linbo@nudt.edu.cn; dsli@nudt.edu.cn).

Xiaoda Zhang, Youhui Bai, and Teng Su are with Huawei Technologies Co. Ltd., Shenzhen 518100, China (e-mail: zhangxiaoda@huawei.com; baiyouhui@huawei.com; suteng@huawei.com).

Digital Object Identifier 10.1109/TPDS.2023.3281931

[1]https://openai.com/blog/chatgpt



(a) A part of a training computation graph (b) Device topology graph of DGX-1 (c) Example of partitioning a 8x17 matrix into sub-blocks

Fig. 1. Auto-parallelism illustration.

additionally summarize some extended parallelism schemes. Then, we provide an overview and comparison of the collected auto-parallelism works. We divide the strategy searching methods into two categories: machine-learning-based and classic algorithm-based methods. Lastly, we discuss and suggest some research hotspots for developing auto-parallelism, including the limitations of current auto-parallelism techniques and potential solutions.

There are a few related surveys in the research field of distributed machine learning. Matthias et al.[3] give a taxonomic perspective of distributed training of DL. Mayer and Jacobsen[8] published a detailed survey about scalable DL on distributed infrastructures. Verbraeken et al.[9] discuss the techniques used for distributed machine learning. They give a comprehensive understanding of the DL system and related machine-learning algorithms but did not involve clear illustrations on selecting strategies. Unlike these two surveys, we explore more basis and details about auto-parallelism and how they are used to accelerate a model's training in this survey.

We structure our survey as follows. In Section II, we present a unified definition of the problem that auto-parallelism solves. In Section III, we list the challenges of auto-parallelism, covering five important aspects: detailed analysis of parallelism schemes, trade-offs between different parallelism schemes, load balance in heterogeneous topology, topology-aware communication optimization and trade-offs between runtime and strategy performance in finding strategy. In Section IV, we comprehensively analyze the basis of a broad class of auto-parallelism methods. In Section V, we provide a precise classification, overview, and comparison of strategy searching methods for auto-parallelism. In Section VI, we provide an outlook on future trends and open problems in the field that deserve further research. Finally, In Section VII, we conclude our survey.

II. Problem Definition

Auto-parallelism usually generates parallelism strategies by automatically determining partitions of each tensor in the computation graph, inserting communication operations, and scheduling the whole computation process.

We abstract the computing of DL model's training or inference as a directed acyclic graph (DAG) (i.e., computation graph). Suppose there is a computation graph $\mathcal{G} = (V, E)$, where each node $v_i \in V$ is an operator (e.g., matrix multiplication, Softmax, etc.) or a tensor (i.e., an n-dimensional array). Tensors could be inputs, outputs, intermediate values, or model states (parameter weight, gradients, or optimizer states). Each edge $e_{ij}(v_i, v_j) \in E$ in the DAG indicates that there is a data transfer between v_i and v_j. For example, if v_j is an operator, then v_i is one of the inputs of this operator, which could be a global input, an output generated by operator v_i, or a model state like a parameter weight. Fig. 1(a) is an extracted computation sub-graph, which shows the details of training a matrix multiplication operator, ignores its preceding and succeeding details, and uses an optimizer without optimizer states (e.g., momentum or variance).

As for devices, device topology can be modeled as an undirected graph $\mathcal{D} = (V_{\mathcal{D}}, E_{\mathcal{D}})$ like Fig. 1(b), where each node $d_i \in V_{\mathcal{D}}$ is a device (e.g., CPU, GPU, etc.), and each edge $b_{ij}(d_i, d_j) \in E_{\mathcal{D}}$ is labeled with bandwidth, and represents for interconnection (e.g., PCI-e, NVLink, InfiniBand, etc.) between devices d_i and d_j.

Auto-parallelism algorithms $\mathcal{A}$ take $\mathcal{G}$ and $\mathcal{D}$ as inputs, and then outputs a partition-set $\mathcal{P}$ for all $v_i \in \mathcal{G}$, a sub-graph set $\mathcal{G}_d$ for all $d_i \in \mathcal{D}$, and pipeline schedules. $\mathcal{P}$ records the partitions of each node in $\mathcal{G}$. For example, $p_i = (\mathbf{index}, \mathbf{g}) \in \mathcal{P}$ is a partition of v_i, where $\mathbf{index}$ is a 2D-array recording split index of each axis that can infer the sizes of all sub-blocks after partitioning, $\mathbf{g}$ is a device group 2D-array in which ith array holds IDs of device that holds ith sub-block. By applying $\mathcal{P}$ to $\mathcal{G}$ and inserting corresponding communication and tensor redistribution operators, the auto-parallelism algorithm generates a sub-graph set $\mathbf{G}$, where $G^1_{d_0} \in \mathbf{G}$ represents the sub-graph that would be deployed on d_0, and its pipeline stage number is 1. Finally, the auto-parallelism algorithm inserts corresponding control flows to arrange the execution of the pipeline (i.e., the schedule of executing sub-graphs). Fig. 1(c) illustrates a toy example of a partition using $p = ([[0,4],[0,8]],[[2,6],[3,7],[0,4],[1,5]])$ on node "x" in Fig. 1(a), assuming to be a 8×17 matrix. Let us take the first sub-block for illustration; the *index* array indicates the range of this sub-block, which is from the 0th row to 4th row and from the 0th column to 8th column; the **g** indicates that this sub-block is held by device 2 and 6 (i.e, GPU2 and GPU6 in Fig. 1(b)).

III. Challenges of Auto-parallelism

There are five main challenges in auto-parallelism.

- The first and most important one is the detailed analysis of different parallelism schemes, which is the foundation of auto-parallelism.
- The second challenge is considering trade-offs between different parallelism schemes, on which most of the auto-parallelism methods work.
- The third one is the load balance problem across heterogeneous devices. The goal is to organize the program well so that each device in a heterogeneous cluster has an approximate execution time.
- The fourth one is the optimization of network communication on specific device topology. A good communication arrangement often brings less communication time and thus increases the computation/communication ratio.
- The last is the trade-off between runtime and strategy performance in finding strategy. Profiling every strategy is time-consuming, and thus many works try to use a cost-model-based method to reduce runtime.

In the following subsections, we present detailed analyses of each challenge.

A. Detailed Analysis on Parallelism Schemes

Auto-parallelism needs to consider the computation, communication, and memory cost of different parallelism schemes. A good auto-parallelism strategy set S often has minor computation and communication costs while having an acceptable memory cost. Based on the analysis, auto-parallelism searching methods decide the appropriate parallelism strategy for each operator. We give our thorough analysis in Section IV.

B. Trade-Offs Between Different Parallelism Schemes

Different parallelism schemes may bring different computation, communication, and memory cost. In a homogeneous cluster, the computation resources on each device are usually the same. Tofu [10], Hypar [11] and D-Rec[12] utilize this property, and consider communication cost only to produce 1D-TP (Vanilla DP [4], Row-TP and Column-TP) strategies for each $v_i \in V$. Intuitively, we tend to select strategies with less communication cost and less replication to improve throughput and scalability. As analyzed in Section IV, most of the communication happens in the redundant part of the model, especially for 1D-TP. It seems to be enough to choose the strategy with the least communication amount for each node, as it, in the meanwhile, has the least memory cost. However, the technique of check-pointing[13] reduces the memory of intermediate results (e.g., X, Y) to a sub-linear degree, which makes Row-TP and Column-TP have lower memory costs so that we can increase batchsize and model size. Some researchers prefer to reduce memory costs by applying Row-TP, Column-TP strategies, and check-pointing instead of DP, although DP theoretically has smaller communication costs in some cases. Applying check-pointing requires us to analyze both communication cost and memory cost in order to improve throughput finally.

Applying Inter-P to the training can also reduce a large amount of Intra-P communication costs because Inter-P creates stages held by corresponding subsets of devices, and thus, devices only need to do Intra-P communication within their stage.

Nevertheless, it may bring some performance degradation due to unavoidable bubbles in the pipelines.

To achieve the highest throughput, we need to choose appropriate parallelism strategies for each $v_i \in V$, which is usually decided after deep consideration by humans or long time searching by algorithms. Many auto-parallelism methods apply algorithms to search for trade-off strategies automatically. We will further discuss these methods for searching for parallelism strategies in Section V.

C. Load Balance in Heterogeneous Topology

Heterogeneous topology refers to a device graph with different types of computing devices, such as multiple types of CPUs and GPUs. Using heterogeneous clusters for environmentally-friendly training of models is a good choice because older devices can still participate in some parts of the work. However, because different types of devices have varying computing performances, researchers need to properly arrange the computation to achieve load balance on each device. Unfortunately, only a few works have involved load balance analysis in the strategy searching task. BPT-CNN[14] can partition datasets into batches and distribute them to devices according to their computing capacity. DeepSpeed heuristically uses CPUs to execute the parameter updates because they are less complicated than forward and backward propagation, and the computation of updates on CPUs can overlap the computation on GPUs in certain situations. Paddle-HeterPS [15] uses reinforcement learning to select computing devices for every layer, but it only supports DP and PP. AccPar [16] introduces a method that solves partition ratios of each kind of device and then partitions the model layer by layer, after which the computation time on each device is similar, but it only generates TP strategies. Merak[17] searches PP strategy for inter-node heterogeneous topology and applies TP within nodes to achieve load balance. Auto-parallelism on heterogeneous topology is still an area of exploration and it has the potential to save a lot of money by avoiding the need to buy more devices.

D. Topology-Aware Communication Optimization

Auto-parallelism algorithms need to consider topology-aware communication strategies to further reduce communication time and increase throughput. Due to the limited size of a node's motherboard, many computing devices are distributed to different nodes, resulting in differences in bandwidth between intra-node and inter-node communication. While intra-node bandwidth is usually faster than inter-node bandwidth, making full use of intra-node bandwidth can optimize communication and reduce overall execution time. Works such as [18], [19] divide all-reduce operations among all devices in a cluster into several all-reduce operations among subgroups of devices to achieve better performance. Inspired by this, P^2 [20] can generate TP partition strategies and utilize the system hierarchy to synthesize the best reduction strategies that consist of sequences of common collective communication operations, which have been proven to be faster than a single All-Reduce operation among all devices in many cases. These works on all-reduce optimize Intra-P communication. Another communication optimization work on tensor redistribution reduces the communication cost between operators. [21] reduces the communication amount of tensor redistribution by replacing the original All-to-All operations with sequences of portable collective communication operations, including All-Gather, Dynamic-Slice, All-Permute, and All-to-All.

E. Trade-Off Between Runtime and Strategy Performance in Finding Strategy

Strategy searching algorithms are time-consuming for two reasons. The first reason is that partitioning a DAG for optimal performance is an NP-hard problem [22], [23], [24], [25]. The second reason is that we need to evaluate every strategy that the algorithm finds.

To solve the NP-hard problem, researchers have tried to use machine-learning algorithms [1], [26] and classic algorithms like dynamic programming[27]. Some works [10], [28] additionally use heuristic assumptions that help shorten searching runtime but may sacrifice the strategy's performance. We will discuss more details of this in Section V.

Using a cost model to evaluate the performance of searched strategies is a good choice since it predicts the runtime within a short time [29]. Some auto-parallelism works [10], [12], [16], [30], [31] use a symbolic cost model to analyze the performance of strategies by evaluating the communication volume and FLOPs of operators. However, most accelerators (e.g., GPU, NPU, FPGA) perform computations in parallel, which differs from the serial amount of computation reflected in a symbolic cost model. Additionally, different types of devices may have varying performances and implementations for specific tasks, such as convolution, making it necessary to artificially annotate the cost model to accurately tune it. Furthermore, symbolic cost models struggle to account for overlaps between computation and communication. While symbolic cost models have a shorter runtime than profiling, these limitations make it challenging for them to accurately reflect the actual performance of discovered strategies. Profiling the execution time of a strategy by deploying and running it is a more accurate approach to compare the performance of different strategies, but profiling every schedule that the algorithm generates is too time-consuming [32], [33]. Therefore, using a profiling-based cost model [28], [34], [35] is a more practical approach since its costs represent the actual time taken to run each operator in the computation graph. By minimizing cost, we can find a near-optimal parallelism strategy.

IV. The Analysis on Different Parallelism Schemes

A detailed analysis of the communication, computation, and memory costs of every parallelism scheme is the basis of auto-parallelism since different partition strategies bring different amounts of cost. Auto-parallelism methods try every combination of parallelism schemes they can handle and select the one with the minimum cost as the final decision. This section discusses the partition and communication in every parallelism scheme. To simplify the illustration in this section, we analyze based on a homogeneous environment where devices have the same computation capacity. Thus, each device has the same computation cost when given the average partitioned task. This assumption helps us focus on evaluating communication costs in different strategies. Note that communication is the most crucial factor we need to consider in generating strategies, and computation is another critical factor for balancing work on every device.

Alpa[7] divides parallelism schemes into two categories: intra-operator parallelism (Intra-P) and inter-operator parallelism (Inter-P). Intra-P shards the tensors (i.e., $v_i \in V$) along their axes while Inter-P divides a computation graph $\mathcal{G}$ into several sub-graphs $\mathbf{G}$ by nodes. We follow this division in our survey because it is intuitive and helps us make a detailed analysis.

A. Intra-Operator Parallelism

Intra-operator Parallelism (Intra-P) can also be named Tensor Parallelism (TP), as it partitions an operator's input and output tensors. All the Intra-P schemes contain specific tensor partitions, and most of them contain collective communication. Taking matrix multiplication (MatMul) as an example, its forward computation is shown as (1), and its backward computation is shown as (2) and (3).

$$Y = XW \tag{1}$$

$$\delta W = X^T E_y \tag{2}$$

$$E_x = E_y W^T. \tag{3}$$

To determine the parallelism scheme of an operator, we first need to find all its related tensors and operators in a backward propagation and group them together because they share the same partitions. In MatMul, E_x and X, E_y and Y, W and δW has the same partitions respectively. Tables I and II show the specific partitions and 2D-array indexes mentioned in Section II of different Intra-P schemes of MatMul. In Table I, b, w_{in} and w_{out} represent batchsize, input dimension number, and output dimension number, respectively. Fig. 2 shows the communication patterns used in different parallelism schemes. DP needs to sum up the δW on different ranks using an All-Reduce primitive; Row-TP partitions weight along the row-axis, and thus the output tensor Y becomes a partial sum tensor, which needs an All-Reduce to form the correct tensor. The communication pattern of

TABLE I
SHAPES OF MATMUL RELATED TENSORS GIVEN DIFFERENT INTRA-P SCHEMES ON p DEVICES

Scheme Name	Input (X)	Weight (W)	Output (Y)	Weight Gradient (δW)	Optimizer States (O_s)
Vanilla DP	$(b/p, w_{in})$	(w_{in}, w_{out})	$(b/p, w_{out})$	(w_{in}, w_{out})	(w_{in}, w_{out})
ZeRO-DP stage 1 [36]	$(b/p, w_{in})$	(w_{in}, w_{out})	$(b/p, w_{out})$	(w_{in}, w_{out})	$(w_{in}/p, w_{out})$
ZeRO-DP stage 2 [36]	$(b/p, w_{in})$	(w_{in}, w_{out})	$(b/p, w_{out})$	$(w_{in}/p, w_{out})$	$(w_{in}/p, w_{out})$
ZeRO-DP stage 3 [36]	$(b/p, w_{in})$	$(w_{in}/p, w_{out})$	$(b/p, w_{out})$	$(w_{in}/p, w_{out})$	$(w_{in}/p, w_{out})$
Row-TP [5]	$(b, w_{in}/p)$	$(w_{in}/p, w_{out})$	(b, w_{out})	$(w_{in}/p, w_{out})$	$(w_{in}/p, w_{out})$
Column-TP [5]	(b, w_{in})	$(w_{in}, w_{out}/p)$	$(b, w_{out}/p)$	$(w_{in}, w_{out}/p)$	$(w_{in}, w_{out}/p)$

TABLE II
EXAMPLES OF 2D-ARRAY **index** OF INTRA-P OF MATMUL

Scheme Name	X	W	Y	δW	O_s
Vanilla DP [1]	$[[0, b/2], [-1]]$ [2]	$[[-1], [-1]]$	$[[0, b/2], [-1]]$	$[[-1], [-1]]$	$[[-1], [-1]]$
ZeRO-DP stage 1 [1]	$[[0, b/2], [-1]]$	$[[-1], [-1]]$	$[[0, b/2], [-1]]$	$[[-1], [-1]]$	$[[0, w_{in}/2], [-1]]$
ZeRO-DP stage 2 [1]	$[[0, b/2], [-1]]$	$[[-1], [-1]]$	$[[0, b/2], [-1]]$	$[[0, w_{in}/2], [-1]]$	$[[0, w_{in}/2], [-1]]$
ZeRO-DP stage 3 [1]	$[[0, b/2], [-1]]$	$[[0, w_{in}/2], [-1]]$	$[[0, b/2], [-1]]$	$[[0, w_{in}/2], [-1]]$	$[[0, w_{in}/2], [-1]]$
Row-TP[1]	$[[-1], [0, w_{in}/2]]$	$[[0, w_{in}/2], [-1]]$	$[[-1], [-1]]$	$[[0, w_{in}/2], [-1]]$	$[[0, w_{in}/2], [-1]]$
Column-TP [1]	$[[-1], [-1]]$	$[[-1], [0, w_{out}/2]]$	$[[-1], [0, w_{out}/2]]$	$[[-1], [0, w_{out}/2]]$	$[[-1], [0, w_{out}/2]]$

[1] Partitioned on 2 devices, device group g is $[0, 1]$ as an example.
[2] -1 represents for a non-partition along this axis.

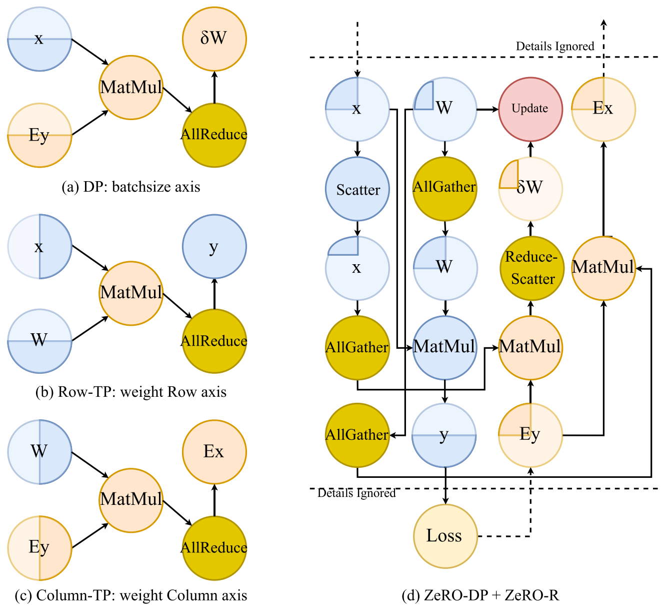


Fig. 2. Some Intra-P schemes of a MatMul operator.

Column-TP happens after (3), which is responsible for summing up the error E_x of tensor X. ZeRO-DP is a variant of vanilla DP, which replaces the All-Reduce primitive with a Reduce-Scatter and an All-Gather primitive. After Reduce-Scatter, ZeRO-DP distributes δW to all ranks in the DP group, and each worker uses its own δW to update its own W slices. Finally, ZeRO-DP [36] uses an All-Gather to collect weight from all ranks in the DP group. ZeRO-DP has three stages: stage 1 partitions the optimizer states, stage 2 additionally partitions δW, and stage 3 partitions W as well by introducing another All-Gather to guarantee mathematical equivalence. To save more memory, ZeRO-R [36] in Fig. 2(d) partitions tensor X after (1), and uses an All-Gather to collect it before its backward propagation.

Most communication costs and memory redundancy in distributed training are from Intra-P. Table III shows the replicated tensors and communication cost of each parallelism scheme of MatMul.

Simultaneously considering Intra-P strategies of several continuous operators, sometimes help discover better strategies. Megatron-LM replaces the All-Reduce in TP with a Reduce-Scatter and an All-Gather to reduce the computation and memory cost of LayerNorm and Dropout operators while maintaining the same communication cost[37].

The key of Intra-P is to partition tensors of an operator and insert communication patterns to make it mathematically equivalent. Thus, we can extrapolate the Intra-P schemes of other operators. Taking a convolution operator as another example, DP partitions along the batchsize axis, Row-TP partitions along the channel-in dimension axis, and Column-TP partitions along the channel-out dimension axis.

B. Inter-Operator Parallelism

Inter-operator Parallelism (Inter-P) splits the computation graph into several stages and allows each rank to hold one

TABLE III
INTRA-P SCHEMES: INTRA-OPERATOR COMMUNICATION COST OF MATMUL $Y = XW$ IN A TRAINING STEP

Scheme Name	Replicated Tensor	Communication Tensor	Communication Pattern	Communication Volume	Memory Redundancy
Vanilla DP	$W, \delta W, O_s$	δW	All-Reduce	$\frac{2(p-1)}{p} w_{in} w_{out}$	$3(p-1) w_{in} w_{out}$
ZeRO-DP stage 1	$W, \delta W$	$W, \delta W$	Reduce-Scatter, All-Gather	$\frac{2(p-1)}{p} w_{in} w_{out}$	$2(p-1) w_{in} w_{out}$
ZeRO-DP stage 2	W	$W, \delta W$	Reduce-Scatter, All-Gather	$\frac{2(p-1)}{p} w_{in} w_{out}$	$(p-1) w_{in} w_{out}$
ZeRO-DP stage 3	None	$W, \delta W$	Reduce-Scatter, All-Gather	$\frac{3(p-1)}{p} w_{in} w_{out}$	0
Row-TP	Y, E_y	Y	All-Reduce	$\frac{2(p-1)}{p} b w_{out}$	$(p-1) b w_{out}$
Column-TP	X, E_x	E_x	All-Reduce	$\frac{2(p-1)}{p} b w_{in}$	$(p-1) b w_{in}$

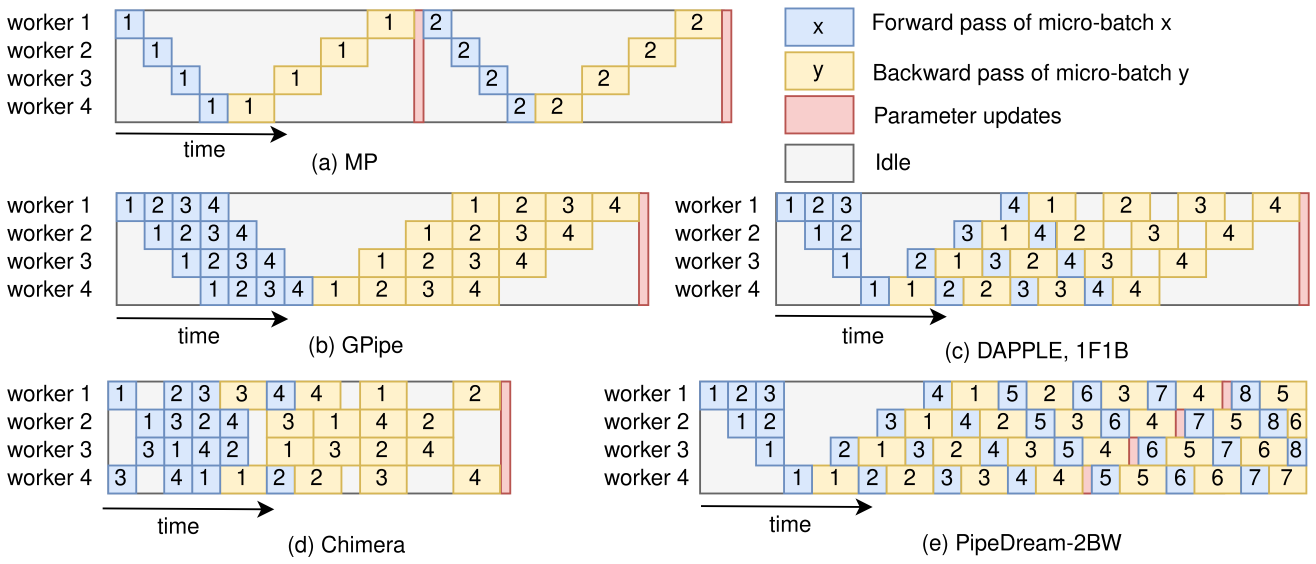


Fig. 3. Examples of Inter-P schemes.

or some of them, thus forming a pipeline. A model can be easily scaled up using Inter-P. The most naive way to do Inter-P is through model parallelism (MP). As shown in Fig. 3(a), only one worker is activated at any time during training. To improve the pipeline's throughput, researchers have proposed Inter-P schemes like GPipe [38] and PipeDream[34], which can overlap the computation of different micro-batches. These well-designed schemes are all called Pipeline Parallelism (PP). Referring to Chimera [39], we make Fig. 3 to help illustrate the differences of different PP schemes, including GPipe, DAPPLE, PipeDream-1F1B, Chimera, and PipeDream-2BW. Following Chimera, we divide PP into two categories: asynchronous-PP (APP) and synchronous-PP (SPP). Synchronicity here represents the matching of weight versions between forward and backward propagation. APP schemes like PipeDream-2BW[6] reduce pipeline bubbles by maintaining two versions of model parameters during training. However, APP schemes cannot guarantee the convergence of the model, as it is not mathematically equivalent to naive training procedures. Thus, large-scale model training instances like Pangu-α[40] and Megatron [41] do not use APP schemes. Instead, they use SPP schemes like DAPPLE[42] and PipeDream-1F1B [6] as they reschedule the pipeline to reduce peak memory during training. Absorbing the advantages of DAPPLE, GPipe, and GEMS[43], Chimera [39] forms a bi-directional schedule and thus reduces bubbles in the pipeline, which we think would be widely adopted in the future. Considering the fact that Chimera is naturally an Inter-P scheme whose DP-degree is 2, we might ignore the redundant memory it brings by maintaining two stages within a device.

The communication of Inter-P happens when a stage needs to send its output to its next stage. The communication volume of a stage in Inter-P equals the size of the tensor it sends/receives.

C. Hybrid Parallelism

Hybrid parallelism combines several Intra-P and Inter-P schemes to partition the model in a fine-grained way. Representative work includes 3D CNN hybrid parallelism[44], Megatron-LM [5], 3D-parallelism [45] from DeepSpeed, and 3D-TP[46] from Colossal-AI[47]. 3D CNN hybrid parallelism extends DP with spatial parallelism and achieves good scaling for large-scale 3D CNN models. In Megatron-LM, domain experts manually partition the large-scale model like GPT-3 [48] using Row-TP and Column-TP and apply a PipeDream-1F1B Inter-P scheme to improve throughput. DeepSpeed automatically partitions models evenly into pipeline stages and then uses ZeRO-DP on each stage to enlarge the throughput. 3D-TP uses ZeRO-DP, Row-TP, Column-TP, and ZeRO-R simultaneously to further reduce memory redundancy and communication volume. However, the least communication volume does not mean the least communication cost in most cases because Inter-node communication might become the bottleneck.

D. Other Methods

1) Check-Pointing: Also known as recomputation, check-pointing [13], [49] drops the activation values generated by forward propagation and recomputes them in backward propagation, which reduces the memory of activation values to a sub-linear degree. Using check-pointing enables us to enlarge the batchsize during training or scale up the model size. Check-pointing is a practical way to train DL models, which has been adopted by frameworks like MindSpore[50], OneFlow[51], PyTorch[52], PaddlePaddle[53], and TensorFlow[54].

2) Expert Parallelism: Unlike the parallelism schemes mentioned above, Expert Parallelism[55] is specific to Mixture-of-Experts (MoE)-based models. The MoE layer consists of several expert networks and a gating network that selects expert networks to execute. Expert Parallelism distributes the expert networks to different devices and uses All-to-All communication to guarantee mathematical equivalence. Moreover, it can be used with DP and PP simultaneously.

3) Token-Level Parallelism: Token-level parallelism (TeraPipe) [56] is a variety of pipeline parallelism. It makes good use of the property of the Transformer[57] that longer sequences require a longer time to compute. Instead of feeding data in the unit of micro-batch to the pipeline, TeraPipe splits sequence data along the token axis (i.e., length axis) unevenly and then feeds them in the pipeline, where each split has a similar execution time. TeraPipe is orthogonal to TP, which may be helpful in training large-scale language models.

4) Sequence Parallelism: Sequence parallelism[58] also utilizes the property mentioned above of the Transformer. It is implemented as a ring-pipeline, where each worker holds the same set of parameters and computes different parts of the inputs chunked along the sequence-length axis. By exchanging the computed results between devices, the pipeline produces the final complete results. Sequence parallelism supports larger batch sizes during training than TP while achieving better throughput.

V. Strategy Searching Methods for Auto-Parallelism

As mentioned above, strategy searching is the key to auto-parallelism and, in the meanwhile, is an NP-hard problem. Researchers have proposed many methods [6], [7], [10], [12], [16], [28], [30], [31], [32], [33], [34], [35], [39], [42], [53], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69], [70] for auto-parallelism to find a near-optimal strategy. We divide the existing strategy searching methods into two categories: classic algorithm-based methods and machine learning-based methods. Classic algorithm-based methods include recursive algorithm [71], dynamic programming algorithm [27], integer linear programming algorithm [72], as well as breadth-first search (BFS) algorithm [73]. We summarize the following analysis in Table IV. Machine learning-based methods include methods such as Monte-Carlo Markov Chain (MCMC) [74], Monte-Carlo Tree Search (MCTS) [75], which help search strategies, and reinforcement learning [76], which helps predict strategies for each operator, among others.

A. Machine-Learning-Based Methods

1) Reinforcement-Learning-Based Methods: We first start with reinforcement-learning-based methods. ColocRL [63] is the first work that uses reinforcement learning to do auto-parallelism. It uses an attentional sequence-to-sequence model trained with an Adam optimizer based on policy gradients computed via the REINFORCE equation [77] to predict the placements of operators. However, it is a coarse-grained method that only does model parallelism, and it is too expensive for the recurrent neural network (RNN) [78] policy to learn when the number of operations is enormous. It took 27 hours over a cluster of 160 workers to find a placement that outperformed an existing heuristic. Moreover, the standard policy gradient method is inefficient, as it performs one gradient update for each data sample [66]. The author of ColocRL then proposes a long short-term memory (LSTM) [79] reinforcement-learning-based hierarchical device placement strategy (HDP) [66], which can support neural networks that have ten of thousands of operations. Spotlight[68] models the problem as a multi-stage Markov decision process (MDP) [80] and applies proximal policy optimization on a two-layer sequence-to-sequence RNN with LSTM cells and a content-based attention mechanism. However, HDP and Spotlight rely too much on LSTM controllers that are hard to train(i.e., Spotlight takes 9 hours on five worker machines to find better placement than ColocRL). Moreover, LSTM performs poorly in capturing long-distance dependencies over large computation graphs To alleviate this problem, Placeto [67] and GDP[64] use Graph Neural Network (GNN) [81] to make embedding information for nodes in computation graph $\mathcal{G}$. Placeto models problems as MDP, relies on hierarchical grouping and only generates placement for one operator at each time step. Instead, GDP pre-trains and fine-tunes a Transformer-based [57] attentive network to generate whole graph operator placements at once and is 16.7x faster than HDP when finding strategies for an 8-layer Transformer model. In addition, GDP can support partitions for large hold-out graphs with over 50k nodes. REGAL (Reinforced Genetic Algorithm Learning) [69] uses a GNN policy to predict node-specific non-uniform proposal distribution choices, which are parameterized as beta distributions over [0,1]. REGAL then uses a biased random key genetic algorithm (BRKGA) [82] to run with those choices and outputs the best solution found by its iteration limit. REGAL can generalize to a broad set of previously unseen computation graphs, which saves a lot of training time, and it can produce an MP strategy for a graph with 1k nodes in only a few seconds. However, REGAL only considers peak memory minimization, while GDP focuses on model throughput and scalability. HeterPS [15] applies parameter server architecture on CPU and ring-allreduce architecture on GPU/XPU to exploit heterogeneous computing devices fully. It uses a reinforcement-learning-based LSTM model to predict the device type for each layer of the models. Their experiment shows that HeterPS is exponentially faster (i.e., 10 seconds to find the best strategy) than Brute Force Search when the number of device types and the generated schedule plan on heterogeneous computing resources have higher throughput than homogeneous computing resources.

TABLE IV
COMPARISON OF DIFFERENT STRATEGIES SEARCHING METHODS FOR AUTO-PARALLELISM

Name	Supported Scheme	Detail	Evaluation Method	Scheduling Time
ColocRL [63]	MP	Training RNN RL	Profiling	NMT: 27 hours on 4 K80 GPUs
HDP [66]		Training LSTM RL		NMT: 3 hours on 8 K40 GPUs
GDP [64]		Transformer RL. PreTrain and Finetune		NMT: 7.35x faster than HDP
Spotlight [68]	DP+MP	Training LSTM+Attention RL		CNN: 9 hours on 40 K80 GPUs
Placeto [67]		MDP & Graph Embedding	Profiling-based cost model	NMT: 49 hours
REGAL [69]	MP	BRKGA & GNN & RL		Graphs whose $\|V\| < 1000$: seconds
HeterPS [15]	DP+PP	LSTM RL		CTR model: 20 Seconds on 8 V100
FlexFlow [61]	TP	MCMC		NMT: 0.6 hour on 64 K80 GPUs
Auto-MAP [62]	TP or PP	DQN with pruning		Bert-48: 262 seconds on 32*V100
Automap [60]	TP	MCTS & interaction Network	Cost Model	A few minutes
Pesto [91]	MP	ILP	Profiling-based cost model	NMT: 51 minutes on 2 V100 GPUs
vPipe [92]	PP	Dynamic Programming (KL)	Profiling	$O(\|V\|^2 \log \|V\|)$
PipeDream [34]	DP+PP	Dynamic Programming	Profiling-based cost model	$\sum_{k=1}^{L} O(\|V\|^3 m_k^2)$
RaNNC [33]			Profiling	Not Given
DAPPLE [42]			Profiling-based cost model	Not Given
DNN-Partitioning [35]		Dynamic Programming+ILP		$O(\mathcal{I}^2(\|V_{\mathcal{D}}^{gpu}\|\|V_{\mathcal{D}}^{cpu}\| + \|V\| + \|E\|))$
OptCNN [59]	TP	Dynamic Programming (Graph Elimination and Regeneration)		$O(\|E\|K^3)$
Tofu [10]			Symbolic cost model	$O(\|E\|K^3)$
TensorOpt [31]				$O(\|V\|^2 K^3 log(K)(log(\|V\|) + log(K)))$
D-Rec [12]		Double Recursive Programming		$O(\|V\|)$
AccPar [16]		Dynamic Programming		$O(\|V\|)$
PaSE [30]		Dynamic Programming with GenerateSeq		$O(\|V\|^2 K^{M+1})$
GSPMD [93]	TP+PP	Sharding Propagation	None	$O(\|V\|)$
Neo [70]		Greedy+Karmarker-karp algorithm	Symbolic cost model	Not Given
Alpa [7]		ILP+Dynamic Programming	Profiling-based cost model	$O(\|V\|^5 \|V_{\mathcal{D}}\|(\|V_{\mathcal{D}}\|/d + \log d)^2)$
DistIR [32]		Grid-Search		Not Given
Piper [28]		2-level Dynamic Programming		$O(\|V\|^2 N \|V_{\mathcal{D}}\|^2)$

1 m_k: the device number of k-th hierarchy in device topology.
2 $\mathcal{I}$: number of already-partitioned region.
3 K: the number of configurable strategies.
4 M: the size of largest dependent set.
5 d: the number of device nodes (i.e, depth).
6 N: the maximum sum of DP degrees.

MP: model parallelism.
DP: data parallelism.
PP: pipeline parallelism.
TP: tensor parallelism (including DP, Row-TP and Column-TP).

The above reinforcement learning methods all focus on DP and MP. We will discuss methods related to TP or PP below.

TAPP is a method that focuses on partitioning a model into stages using reinforcement learning. It predicts the stage number for each layer using a feed-forward neural network (FFN) and then predicts which device a stage should be on using a reinforcement learning attention-based sequence-to-sequence model. Auto-MAP [62] from Alibaba leverage Deep Q-Network (DQN)[83] with task-specific pruning strategies to help efficiently explore the search space of either TP or PP over XLA Higher Level Operations (HLO) with device and network interconnection topology specified. They choose HLO Intermediate Representation (IR) [84] that is produced by Accelerated Linear Algebra (XLA)[85] from TensorFlow as the operational level of Auto-MAP. Because exploring distributed plans on HLO IR can achieve better performance benefits from its finer granularity than operators. Moreover, IR is a kind of expression of computation graphs, which fits our problem definition. Auto-MAP set rewards, states, and actions for all three DP, TP, and PP to instruct DQN to search strategies. Given a cluster of 4 servers with 8 V100 GPUs, Auto-MAP can search TP strategies for the 11-billion-parameter T5 [86] model within 1.5 hours, DP strategies within 17 minutes, and PP strategies within 280 seconds. The text mentions that Auto-MAP can currently only give a single parallelism strategy, which may result in sub-optimal runtime performance in large-scale distributed training. The authors are considering supporting a hybrid of these strategies in the future.

2) Other Methods: FlexFlow proposed four possible parallelizable dimensions based on OptCNN[59]: SOAP, representing sample, operation, attribute, and parameter, respectively. Among SOAP, sample-dimension division corresponds to DP, operation-dimension division corresponds to MP, attribute-dimension division corresponds to each attribute dimension of input Tensor (such as height and width), and parameter-dimension division corresponds to TP. The partitioning of attributes and parameters corresponds to model parallelism. Unlike OptCNN, which only supports linear models like AlexNet[87], FlexFlow can parallelize arbitrary computation graphs. They use a random MCMC algorithm to find the optimal partition configuration and determine the appropriate parallelism strategy for each operator in a neural network. However, the MCMC tries to enumerate strategies randomly in the search space, which results in an unacceptable time of solving the optimal solution for large-scale models. It requires 37 minutes to search strategies for NMT [88] model on 16 servers with 4 P100 GPUs. To support the partition of GNN models, the author of FlexFlow, Zhihao Jia, implements ROC [65] on top of FlexFlow. They design a cost model, which could predict the execution time of GNN on an arbitrary graph, and then uses an online linear regression model to learn the cost model. The learned cost model enables the graph partitioner to discover balanced and effective partitioning for GNN training and inference. In addition, ROC uses a dynamic programming algorithm to minimize communication costs. Automap [60] from DeepMind is performed on MHLO, which is an MLIR [89] encoding of XLA HLO. It applies a Search and Learning method to annotate Megatron-like [5] strategies for all operators. They implement MCTS to help search and propagate strategies when traversing the program. To reduce the search space and improve the quality of strategies propagation, Automap uses a learned interaction network [90] to compute the per-node relevance score, and the top-k will be considered first in the search space. Automap shows that using MCTS with a learned filter can find strategies similar to Megatron.

B. Classic Algorithm-Based Methods

OptCNN proposes the layer-wise parallelism strategy, which is an auto-parallelism solution under the parameter server architecture[94]. OptCNN can only handle TP partitions. Taking the computer vision task as an example, OptCNN considers the input dimensions of every layer in the model, including batchsize, width, height, and the number of channels. All dimensions can be divided into various devices. They first build a computation graph $\mathcal{G}$ of the model and a device graph $\mathcal{D}$ of the cluster and then build a cost model to estimate cost under any TP strategies. Using a dynamic programming graph search algorithm, OptCNN can determine combinations of partition dimensions for every layer. However, OptCNN can only solve computation graphs with linear structure. To address this problem, Tofu [10] coarsens the computation graph $\mathcal{G}$ by grouping some nodes in V to make it linear, after which they use the dynamic programming algorithm in OptCNN to produce strategies. Moreover, ToFu considers only communication cost to reduce the search space under the observation that different strategies of an operator, like matrix multiplication, have the same arithmetic complexity. Tofu uses the dynamic programming algorithm in OptCNN and applies some techniques to make it more practical. In addition to the coarsening technique, Tofu accelerates the dynamic programming algorithm by applying recursively. Compared to dynamic programming with coarsening, which takes 8 hours to get the best strategy set $\mathcal{P}$ for 8 workers, using recursion to search the optimal strategy for WResNet-152 [95] only takes 8.3 seconds. Instead of coarsening, TensorOpt [31] extends the dynamic programming algorithm in OptCNN and names it FT-Elimination (Frontier Tracking Elimination) to make it executable on a computation graph with a non-linear structure. However, the runtime is not efficient enough. So TensorOpt also tries to group operators and applies an FT-LDP (Frontier Tracking Linear Dynamic Programming) algorithm to help reduce the time complexity. In addition, the FT-LDP algorithm can be parallelized by multi-threading to generate the computation of different parallelization strategies efficiently. For WReset, FT-LDP with multi-threading can find the best strategy in 22 minutes, while FT-Elimination takes 5.5 hours. Though the search time is longer than Tofu, the throughput of TensorOpt's generated strategy is much higher than Tofu since Tofu tries to search strategies that use less memory. However, these memory-saved strategies may have smaller throughput.

PipeDream [34] provides auto-parallelism solutions that support asynchronous pipeline training. In order to accurately obtain the execution time of each layer, PipeDream first profiles the model that needs to be partitioned to obtain the execution time, activation size, and model parameter size of each layer. Then, according to the obtained results, they create a profiling-based cost model and design a dynamic programming algorithm that divides pipeline stages and determines the DP degree of each stage to meet the load balance need. Based on PipeDream, PipeDream-2BW [6] optimizes the memory consumption by applying activation recomputation and reducing the number of parameter weight buffers that store different versions of computed gradients to 2. To accelerate the partition, PipeDream-2BW exploits the repetitive structure of models (e.g., transformer layers in BERT) by grouping them and only considering configurations where all model stages replicate an equal number of times. However, PipeDream only supports linear graphs. To address this problem, researchers from Fiddle propose dnn-partitioning[35], which extends the dynamic programming algorithm in PipeDream to support partition for arbitrary DAG and also proposes an integer programming solution to solve the partition problem. However, like PipeDream and PipeDream-2BW, these methods do not consider tensor parallelism.

Also from project Fiddle, Piper [28] uses a two-level dynamic programming approach to search TP and PP strategies. The outer dynamic programming algorithm would generate hundreds of NP-hard knapsack sub-problems, which calculate the throughput of a sub-graph under given hyper-parameters. Piper uses a bang-per-buck heuristic to accelerate the solving procedure of generated knapsack sub-problems, reducing the computation complexity. The computation complexity of the Piper algorithm is $O(|V|^2 N |V_{\mathcal{D}}|^2)$, where $|V|$ is the number of vertices in the computing graph, $|V_{\mathcal{D}}|$ is the number of devices, $N \leq |V_{\mathcal{D}}|$ is the maximum sum of DP degrees. Piper can partition a 64-layer BERT [96] on 2,048 devices within only 2 hours, which is a relatively short time compared to its training time. However, the current implementation of the algorithm is serial and inefficient. A potential advantage of Piper is that some procedures in Piper can be executed in parallel, which can scale the runtime for this algorithm linearly on a multi-core CPU server and further reduce searching time.

Double Recursion Algorithm (D-Rec)[12] uses the observation that TP has the same communication cost per worker and thus only considers communication cost to do TP partitions and the combination of them. D-Rec builds its cost model statically, which asymptotically and statically analyzes communication cost based on the tensor's shape and type of operators. Based on the analysis, D-Rec automatically determines strategies for each operator within a linear complexity short time (28 seconds to search strategies for 24-layer BERT on 8 devices). MindSpore implements D-Rec as a choice of strategy searching algorithm due to its speed advantage. However, since D-Rec ignores the computation analysis, it may be limited in its ability to support heterogeneous clusters and PP in the future.

PaSE [30] also uses a static analysis-based cost model to generate TP strategies and combinations of them. It makes good use of the sparsity of the computing graph to form a dynamic-programming-based strategy searching algorithm, whose overall computational complexity is $O(|V|^2K^{M+1})$, where $|V|$ is the total number of vertices $v_i \in V$, K is the maximum number of configurations of an operator (vertex), and M is the size of the most extensive dependent set (i.e., the difference set of computing sub-graphs from two iterations). Their experiment shows that PaSE can generate strategies for a Transformer NMT model on 16 and 64 devices in 2.2 minutes and 31.4 minutes, respectively. The throughput of the generated strategies outperforms that of Mesh-TensorFlow [97]. PaSE can be applied to heterogeneous clusters, but currently, it does not include heterogeneity in the cost model, which may result in unbalanced partitions. Moreover, PaSE is not good at handling $\mathcal{G}$ where $|E|$ is tremendous, as M may be significantly large. The runtime overhead of solving strategies for models like DenseNet [98] is unacceptable since their computing graphs are uniformly dense. Both D-Rec and PaSE use static analysis-based cost models to generate strategies. However, the symbolic analysis may not be as accurate as profiling, which may result in some performance deterioration. Because static analysis usually cannot capture some low-level details like cache effects and the overlapping of computation and communication, which may be necessary for analyzing execution time.

AccPar [16] analyzes the intra-operator and inter-operator communication cost for all situations between DP, Row-TP, and Column-TP, and uses these partitions to partition the model. It simplifies the DAG partition problem by deciding on strategies layer by layer using dynamic programming, whose arithmetic complexity is $O(|V|)$. By introducing a partition ratio, AccPar can support heterogeneous clusters. Its experiments show that the performance of AccPar outperforms HyPar [11], which only does DP and Row-TP on homogeneous clusters.

DistIR [32] is an efficient IR for the explicit representation of distributed DNN computation. It uses a linear regression model to simulate the cost of operators (e.g., MatMul and All-Reduce), and the simulated throughput has a strong correlation for both MLP training and GPT-2 inference for all model sizes. DistIR uses a simple grid search to find the minimum cost strategy that consists of DP, Row-TP, and 1F1B-PP. Although DistIR is very efficient in finding the best strategy in their search space, the optimal strategy may be too coarse to use compared to others.

Alpa [7] uses a two-level hierarchical planning algorithm to search for strategies and is the first auto-parallelism method that supports TP, 1F1B-PP, and ZeRO-DP. Alpa works on an arbitrary DAG. It formalizes the Intra-P problem as integer linear programming (ILP) and the Inter-P as dynamic programming. The dynamic programming algorithm is built on top of that in TeraPipe [56], but additionally considers device mesh slicing. During the dynamic programming calculation for finding the best Inter-P strategy, Alpa uses ILP to find the best Intra-P strategy for each stage (i.e., sub-graph). However, the overall complexity of this algorithm is $O(|V|^5|V_\mathcal{D}|(|V_\mathcal{D}|/d + \log d)^2)$, where d is the number of device nodes in the cluster. To reduce the complexity, Alpa uses early pruning to reduce the search space and uses another dynamic programming algorithm to group operators whose computation complexity is $O(|V|^2L)$. Here, L represents the number of layers after grouping. Alpa can find the best strategy within 40 minutes for GPT-39B on 64 GPU devices. However, considering the complexity cost of this method, searching for strategies for GPT-39B on 2,048 GPU devices may require thousands of hours, showing poor scalability. Nonetheless, Alpa is capable of searching for near-optimal strategies for small models.

Some works use the breadth-first search (BFS) algorithm to propagate strategies. BFS algorithm-based algorithm requires users to annotate some parallelism strategies of tensors or operators, after which the DL framework will automatically propagate strategies based on set rules. GSPMD[93] is the first work to do this. It proposes sharding propagation and has integrated the corresponding algorithm into TensorFlow's XLA compiler. It uses a priority-queue-based heuristic method to arrange the parallelism strategies of rest operators in the compute graph. More specifically, it gives the element-wise operators top priority when propagating strategies.

Inspired by GSPMD, frameworks like MindSpore, OneFlow, and PaddlePaddle absorb sharding propagation and create their semi-auto-parallelism method. Currently, they all use the BFS algorithm in propagating annotations. Among them, OneFlow shows that using split-broadcast-partial (SBP) parallelism and actor-based runtime can further accelerate the training of large-scale models.

VI. Discussions on Future Trends

Looking towards the future, we suggest that several trends will be of significant importance. These include the need to improve cost models to achieve more accurate predictions, accelerate strategy searching, support heterogeneous computing to make the most of available resources and incorporate operator fusion to gain better performance.

A. *Making Cost Model Accurate*

Auto parallelism is facing difficulties in accurately measuring the cost of strategies, which results in non-optimal solutions. The main reasons are the lack of analysis of the communication paces and patterns on different interconnections, the implementation efficiency of operators on specific accelerator architecture, and the performance degradation of overlapping. To address this problem, we provide three possible directions to make the cost model more accurate.

1) Analysis on Communication: Previous work [99] has shown that the performance of communication can vary with changes in the transferred data size. Moreover, different interconnections and network topologies can result in different optimal communication paces and patterns. For instance, BytePS[100] involves Reduce and Broadcast communication between GPU and CPU nodes and Ring-AllReduce across GPU nodes. Future works should focus on a detailed analysis of communication to make the cost model more accurate.

2) Analysis on Computing: The efficiency of computation operators can vary depending on the size of the input data. Furthermore, to fully utilize the computing power of hardware resources, frameworks like PyTorch provide multiple implementations optimized for different input sizes and shapes. In the future, researchers may need to take into account these subtle differences in computing efficiency when developing cost models.

3) Analysis on Overlapping: The overlapping of computation and communication may bring performance degradation that future cost models should be concerned about. A representative work is PyTorch-DDP, which overlaps gradient synchronization with backward propagation. The degradation occurs because communication operators require streaming processors on GPUs to handle data transfers, resulting in resource competition that slows down both computation and communication. Additionally, emerging works [99], [101], [102] provide novel overlapping methods to increase training performance, which brings challenges to improve future cost models.

B. Accelerating Strategy Searching

1) Paralleling Dynamic Programming Algorithms: We investigated the implementations of some dynamic programming based auto-parallelism works and found that they only use a single CPU thread to execute their algorithms. We believe that there is potential for acceleration through the use of multiple threads, and we anticipate future works will explore parallel programming to speed up the searching.

2) Applying Greedy Algorithms: Since considering more parallelism schemes into the search space heavily increases the time complexity, we have observed a trend of applying greedy algorithms [60], [70] to help accelerate searching. For instance, we can search the strategies of the most compute-intensive operators first, which is only a small part of the DAGs, to alleviate the pressure of generating strategies of other operators. Another way to apply greedy algorithms is to assume that the same sub-graphs in a DAG share the same parallelism strategy. This assumption helps reduce problem complexity to a sub-linear degree.

3) Learning to Predict: Previous work[15] has demonstrated the potential of utilizing DL methods to rapidly generate strategies. As the number of model types and their corresponding well-designed strategies continues to increase, we believe there will be sufficient samples to train neural networks to help predict strategies in a reasonable amount of time.

C. Supporting Heterogeneous Computing

Given that many researchers have limited computing resources and renting high-performance clusters such as the DGX system is expensive, heterogeneous computing will play an important role in this area. Specifically, Specifically, there are different development trends for intra-node, inter-node, and inter-cluster parallelism in heterogeneous computing.

1) Intra-Node Parallelism: For intra-node parallelism, the future trend is towards the development of more sophisticated auto-parallelism algorithms to optimize the use of CPUs and accelerators such as GPUs, TPUs, NPUs, and FPGAs. For instance, swapping inactive model weights between GPU and CPU memory when GPU-CPU interconnections (e.g., PCIe) are idle can strengthen the ability to train a model with larger sizes. Meanwhile, we can offload some parameter-update operations to CPUs and thus alleviate the pressure of storing optimizer states for GPU [45], [53]. However, this approach can introduce extra gradient transmission between GPU and CPU memory, and future auto-parallelism works should consider these trade-offs.

2) Inter-Node Parallelism: Managing nodes with different types of GPUs in a cluster for collaborative training of models is a visible trend for the future. The main problem here is to partition computation workloads evenly according to the characteristics of heterogeneous nodes to achieve a more efficient performance. Presently, large-scale model training is promoting the development of more efficient communication protocols, such as the use of Remote Direct Memory Access (RDMA) and low-latency networking technologies. This may require future auto-parallelism works to integrate network topology-aware scheduling algorithms to optimize communication overhead.

3) Inter-Cluster Parallelism: For inter-cluster parallelism, the future trend is to leverage the advantages of cloud and edge computing to form distributed heterogeneous clusters and achieve collaborative computing across geographic locations. Cloud service providers can offer idle resources to users to perform non-urgent training tasks. Auto-parallelism works aiming at training models in inter-cluster environments should further consider the fault tolerance and elastic training problem.

D. Fusing Operators

Future auto-parallelism research may focus on integrating operator fusion techniques with strategy searching to achieve greater speedup in DL models training. Fusing multiple computation operators into a single operator can improve computation efficiency and reduce memory access frequency, which has significant advantages in accelerating training.

VII. Conclusion

In this survey, we provide a comprehensive analysis of distributed training from the perspective of auto-parallelism, which is increasingly important in both industrial and academic communities with the rise of large-scale models. We explore the main challenges that hinder the practical implementation of auto-parallelism methods and review existing approaches that have overcome these challenges. Our analysis covers the fundamental concepts of auto-parallelism, including problem definition and parallelism strategies, and delves into an overview and evaluation of existing auto-parallelism methods. Additionally, we conclude by discussing potential future trends for the continued development of auto-parallelism.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.
[2] T. B. Brown et al., “Language models are few-shot learners,” May 2020, *arXiv:2005.14165*.

[3] M. Langer, Z. He, W. Rahayu, and Y. Xue, "Distributed training of deep learning models: A taxonomic perspective," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 12, pp. 2802–2818, Dec. 2020.

[4] S. Li et al., "PyTorch distributed: Experiences on accelerating data parallel training," in *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3005–3018, Aug. 2020.

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

[6] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel DNN training," in *Proc. 38th Int. Conf. Mach. Learn.*, 2021, pp. 7937–7947.

[7] L. Zheng et al., "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," in *Proc. 16th USENIX Symp. Oper. Syst. Des. Implementation*, 2022, pp. 559–578.

[8] R. Mayer and H.-A. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–37, Jan. 2021.

[9] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Comput. Surv.*, vol. 53, no. 2, pp. 1–33, Mar. 2021.

[10] 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.

[11] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "HyPar: Towards hybrid parallelism for deep learning accelerator array," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2019, pp. 56–68.

[12] H. Wang et al., "Efficient and systematic partitioning of large and deep neural networks for parallelization," in *Proc. Eur. Conf. Parallel Process.*, 2021, pp. 201–216.

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

[14] J. Chen, K. Li, K. Bilal, X. Zhou, K. Li, and P. S. Yu, "A bi-layered parallel training architecture for large-scale convolutional neural networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 5, pp. 965–976, May 2019.

[15] J. Liu et al., "HeterPS: Distributed deep learning with reinforcement learning based scheduling in heterogeneous environments," Nov. 2021, *arXiv:2111.10635*.

[16] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "AccPar: Tensor partitioning for heterogeneous deep learning accelerators," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2020, pp. 342–355.

[17] Z. Lai et al., "Merak: An efficient distributed DNN training framework with automated 3D parallelism for giant foundation models," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 5, pp. 1466–1478, May 2023.

[18] Y. Ueno and R. Yokota, "Exhaustive study of hierarchical allreduce patterns for large messages between GPUs," in *Proc. IEEE/ACM 19th Int. Symp. Cluster Cloud Grid Comput.*, 2019, pp. 430–439.

[19] M. Cho, U. Finkler, D. Kung, and H. Hunter, "BlueConnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy," in *Proc. Mach. Learn. Syst.*, 2019, pp. 241–251.

[20] N. Xie, T. Norman, D. Grewe, and D. Vytiniotis, "Synthesizing optimal parallelism placement and reduction strategies on hierarchical systems for deep learning," in *Proc. Mach. Learn. Syst.*, 2022, pp. 548–566.

[21] N. A. Rink, A. Paszke, D. Vytiniotis, and G. S. Schmid, "Memory-efficient array redistribution through portable collective communication," Dec. 2021, *arXiv:2112.01075*.

[22] K. Kennedy and U. Kremer, "Automatic data layout for distributed-memory machines," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 4, pp. 869–916, Jul. 1998.

[23] J. Li and M. Chen, "Index domain alignment: Minimizing cost of cross-referencing between distributed arrays," in *Proc. 3rd Symp. Front. Massively Parallel Comput.*, 1990, pp. 424–433.

[24] U. Kremer, "NP-completeness of dynamic remapping," in *Proc. 4th Workshop Compilers Parallel Comput.*, Delft, The Netherlands, 1993. [Online]. Available: https://people.cs.rutgers.edu/~uli/cs516/spring2020/readings/NP-completenessOfDynamicRemapping-TR1993.pdf

[25] J. Li and M. Chen, "The data alignment phase in compiling programs for distributed-memory machines," *J. Parallel Distrib. Comput.*, vol. 13, no. 2, pp. 213–221, 1991.

[26] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. Cambridge, MA, USA: MIT Press, 2018.

[27] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[28] J. M. Tarnawski, D. Narayanan, and A. Phanishayee, "Piper: Multidimensional planner for DNN parallelization," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 24829–24840.

[29] Z. Li, M. Paolieri, L. Golubchik, S.-H. Lin, and W. Yan, "Predicting throughput of distributed stochastic gradient descent," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2900–2912, Nov. 2022.

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

[31] Z. Cai et al., "TensorOpt: Exploring the tradeoffs in distributed DNN training with auto-parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 8, pp. 1967–1981, Aug. 2022.

[32] K. Santhanam, S. Krishna, R. Tomioka, A. W. Fitzgibbon, and T. Harris, "DistIR: An intermediate representation for optimizing distributed neural networks," in *Proc. 1st Workshop Mach. Learn. Syst.*, 2021, pp. 15–23.

[33] M. Tanaka, K. Taura, T. Hanawa, and K. Torisawa, "Automatic graph partitioning for very large-scale deep learning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 1004–1013.

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

[35] J. Tarnawski, A. Phanishayee, N. Devanur, D. Mahajan, and F. N. Paravecino, "Efficient algorithms for device placement of DNN graph operators," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 15451–15463.

[36] 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.

[37] V. Korthikanti et al., "Reducing activation recomputation in large transformer models," May 2022, *arXiv:2205:05198*.

[38] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 103–112.

[39] S. Li and T. Hoefler, "Chimera: Efficiently training large-scale neural networks with bidirectional pipelines," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, pp. 1–14.

[40] W. Zeng et al., "PanGu-α: Large-scale autoregressive pretrained Chinese language models with auto-parallel computation," Apr. 2021, *arXiv:2104.12369*.

[41] 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.*, 2021, pp. 1–15.

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

[43] A. Jain et al., "GEMS: GPU-enabled memory-aware model-parallelism system for distributed DNN training," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–15.

[44] Y. Oyama et al., "The case for strong scaling in deep learning: Training large 3D CNNs with hybrid parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1641–1652, Jul. 2021.

[45] 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.

[46] Z. Bian, Q. Xu, B. Wang, and Y. You, "Maximizing parallelism in distributed training for huge neural networks," May 2021, *arXiv:2105.14450*.

[47] Z. Bian et al., "Colossal-AI: A unified deep learning system for large-scale parallel training," Oct. 2021, *arXiv:2110.14883*.

[48] T. B. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, Art. no. 159.

[49] A. Griewank and A. Walther, "Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation," *ACM Trans. Math. Softw.*, vol. 26, no. 1, pp. 19–45, Mar. 2000.

[50] Mindspore, Huawei, Aug. 2019. Accessed: Aug. 31, 2022. [Online]. Available: https://www.mindspore.cn/en

[51] J. Yuan et al., "OneFlow: Redesign the distributed deep learning framework from scratch," Oct. 2021, *arXiv:2110.15032*.

[52] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035.

[53] Y. Ao et al., "End-to-end adaptive distributed training on PaddlePaddle," Dec. 2021, *arXiv:2112.02752*.

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

[55] D. Lepikhin et al., "GShard: Scaling giant models with conditional computation and automatic sharding," in *Proc. Int. Conf. Learn. Representations*, 2021.

[56] Z. Li et al., "TeraPipe: Token-level pipeline parallelism for training large-scale language models," in *Proc. 38th Int. Conf. Mach. Learn.*, 2021, pp. 6543–6552.

[57] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[58] S. Li, F. Xue, Y. Li, and Y. You, "Sequence parallelism: Making 4D parallelism possible," May 2021, *arXiv:2105.13120*.

[59] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in accelerating convolutional neural networks," in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 2274–2283.

[60] M. Schaarschmidt et al., "Automap: Towards ergonomic automated parallelism for ML models," Dec. 2021, *arXiv:2112.02958*.

[61] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2017, pp. 553–564.

[62] S. Wang et al., "Auto-MAP: A DQN framework for exploring distributed execution plans for DNN workloads," Jul. 2020, *arXiv:2007.04069*.

[63] A. Mirhoseini et al., "Device placement optimization with reinforcement learning," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 2430–2439.

[64] Y. Zhou et al., "GDP: Generalized device placement for dataflow graphs," Oct. 2019, *arXiv:1910.01578*.

[65] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with ROC," in *Proc. Mach. Learn. Syst.*, 2020, pp. 187–198.

[66] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *Proc. Int. Conf. Learn. Representations*, 2018.

[67] R. Addanki, S. B. Venkatakrishnan, S. Gupta, H. Mao, and M. Alizadeh, "Placeto: Learning generalizable device placement algorithms for distributed machine learning," Jun. 2019, *arXiv1906.08879*.

[68] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1676–1684.

[69] A. Paliwal et al., "Reinforced genetic algorithm learning for optimizing computation graphs," in *Proc. Int. Conf. Learn. Representations*, 2020.

[70] D. Mudigere et al., "High-performance, distributed training of large-scale deep learning recommendation models," Apr. 2021, *arXiv:2104.05158*.

[71] E. W. Dijkstra, "Recursive programming," *Numerische Mathematik*, vol. 2, no. 1, pp. 312–318, 1960.

[72] A. Schrijver, *Theory of Linear and Integer Programming*. Hoboken, NJ, USA: Wiley, 1998.

[73] L. Luo, M. Wong, and W. Hwu, "An effective GPU implementation of breadth-first search," in *Proc. 47th Des. Automat. Conf.*, 2010, pp. 52–55.

[74] W. R. Gilks, S. Richardson, and D. Spiegelhalter, *Markov Chain Monte Carlo in Practice*. Boca Raton, FL, USA: CRC Press, 1995.

[75] C. B. Browne et al., "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.

[76] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *J. Artif. Intell. Res.*, vol. 4, no. 1, pp. 237–285, Jan. 1996.

[77] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3, pp. 229–256, May 1992.

[78] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 1310–1318.

[79] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Proc. Annu. Conf. Int. Speech Commun. Assoc.*, 2014, pp. 338–342.

[80] F. P. Miller, A. F. Vandome, and J. Mcbrewster, *Markov Decision Process*. London: Springer, 1985.

[81] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1025–1035.

[82] J. Gonçalves and M. Resende, "Biased random-key genetic algorithms for combinatorial optimization," *J. Heuristics*, vol. 17, no. 5, pp. 487–525, 2011.

[83] T. Zahavy, N. Ben-Zrihem, and S. Mannor, "Graying the black box: Understanding DQNs," in Proc. 33rd Int. Conf. Mach. Learn., 2016, pp. 1899–1908.

[84] M. Li et al., "The deep learning compiler: A. comprehensive survey," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 708–727, Mar. 2021.

[85] A. Sabne, "XLA: Compiling machine learning for peak performance," 2020. [Online]. Available: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/compiler/xla

[86] C. Raffel et al., "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[87] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, Jun. 2017.

[88] M. Johnson et al., "Google's multilingual neural machine translation system: Enabling zero-shot translation," *Trans. Assoc. Comput. Linguistics*, vol. 5, pp. 339–351, 2017.

[89] C. Lattner et al., "MLIR: Scaling compiler infrastructure for domain specific computation," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2021, pp. 2–14.

[90] P. W. Battaglia, R. Pascanu, M. Lai, D. J. Rezende, and K. Kavukcuoglu, "Interaction networks for learning about objects, relations and physics," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 4502–4510.

[91] U. U. Hafeez, X. Sun, A. Gandhi, and Z. Liu, "Towards optimal placement and scheduling of DNN operations with pesto," in *Proc. 22nd Int. Middleware Conf.*, 2021, pp. 39–51.

[92] S. Zhao et al., "vPipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel DNN training," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 3, pp. 489–506, Mar. 2022.

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

[94] M. Li et al., "Scaling distributed machine learning with the parameter server," in *Proc. 11th USENIX Symp. Oper. Syst. Des. Implementation*, 2014, pp. 583–598.

[95] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *Proc. Brit. Mach. Vis. Conf.*, 2016, pp. 87.1–87.12.

[96] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2019, pp. 4171–4186.

[97] N. Shazeer et al., "Mesh-TensorFlow: Deep learning for supercomputers," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 10435–10444.

[98] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 2261–2269.

[99] A. Jangda et al., "Breaking the computation and communication abstraction barrier in distributed machine learning workloads," in *Proc. 27th ACM Int. Conf. Architectural Support Programm. Lang. Oper. Syst.*, 2022, pp. 402–416.

[100] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation*, 2020, pp. 463–479.

[101] S. Wang et al., "Overlap communication with dependent computation via decomposition in large deep learning models," in *Proc. Int. Conf. Architectural Support Programm. Lang. Oper. Syst.*, 2023, pp. 93–106.

[102] S. Wang, A. Pi, X. Zhou, J. Wang, and C. -Z. Xu, "Overlapping communication with computation in parameter server for scalable DL training," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2144–2159, Sep. 2021.

Peng Liang received the BS degree in network engineering from the National University of Defense Technology (NUDT), Changsha, China, in 2019. He is currently working toward the PhD degree in computer science and technology with NUDT. His current research interests include deep learning and distributed machine learning.

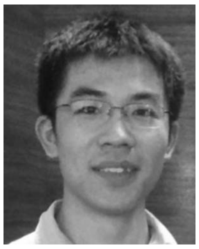
Yu Tang received the BS and MS degrees in computer science from the National University of Defense Technology (NUDT), in 2018 and 2020, respectively, where he is currently working toward the doctor's degree. His current research interests include distributed machine learning, alternating direction method of multipliers (ADMM).

Xiaoda Zhang received the PhD degree from Nanjing University, Nanjing, China, in 2019. He is currently a software engineer with Huawei. He works on developing efficient software for distributed systems and mathematical solvers.

Youhui Bai received the PhD degree from the Department of Computer Science, University of Science and Technology of China (USTC), Hefei, China, in 2021. He is currently a senior engineer of Huawei at 2012 Laboratories-Central Software Institute-Distributed and Parallel Software Laboratory-Parallel Distributed Computing Laboratory. His research interests include distributed machine learning systems, graph processing and storage systems.

Teng Su received the PhD degree from Zhejiang University, Hangzhou, China, in 2012. He joined Huawei, in 2013, and is currently an architect with Huawei AI framework MindSpore. He has lead the development of Huawei task based distributed computinng framework, resource management framework, and distributed reliable key-value stroe, from 2013 to 2019.

Zhiquan Lai received the BS, MS, and PhD degrees in computer science from the National University of Defense Technology (NUDT), in 2008, 2010, and 2015 respectively. He is currently an assistant researcher with the National Key Laboratory for Parallel and Distributed Processing of NUDT. He worked as a research assistant with the Department of Computer Science, The University of Hong Kong during October 2012 to October 2013. His current research interests include high-performance system software, distributed machine learning, and power-aware computing.



Linbo Qiao received the BS, MS, and PhD degrees in computer science and technology from the National University of Defense Technology (NUDT), Changsha, China, in 2010, 2012, and 2017, respectively. Now, he is a assistant research fellow with the National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, China. He worked as a research assistant with the Chinese University of Hong Kong, from May 2014 to October 2014. His research interests include structured sparse learning, online and distributed optimization, and deep learning for graph and graphical models.

Dongsheng Li received the PhD degree in computer science and technology from the National University of Defense Technology (NUDT), in 2005. He is currently a professor and doctoral supervisor with the College of Computer, NUDT. He was awarded the Chinese National Excellent Doctoral Dissertation, in 2008. His research interests include distributed systems, cloud computing and big data processing.