# A Survey on Auto-Parallelism of Neural Networks Training



SUBMISSION DATE / POSTED DATE

06-04-2022 / 08-04-2022

CITATION

Liang, Peng; Tang, Yu; Zhang, Xiaoda; Bai, Youhui; Su, Teng; qiao, linbo; et al. (2022): A Survey on Auto-Parallelism of Neural Networks Training. TechRxiv. Preprint. https://doi.org/10.36227/techrxiv.19522414.v1

DOI

10.36227/techrxiv.19522414.v1

# A Survey on Auto-Parallelism of Neural Networks Training

Peng Liang, Yu Tang, Xiaoda Zhang, Youhui Bai, Teng Su, Zhiquan Lai†, Linbo Qiao†, 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. It is necessary to train DL models in clusters with novel parallel and distributed training algorithms. However, traditional training algorithms owns the inability to train large-scale neural networks in heterogeneous computing clusters. Nowadays, auto-parallelism is promising to handle the above issue. Auto-parallelism makes large-scale DL model training efficient and practical 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. Firstly, 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 neural networks, training technique, parallel and distributed training**

## I. Introduction

DEEP learning [1] has drawn a lot of attention for its superior performance in tasks like speech recognition [2], machine translation [3], object detection [4], recommendation [5], and so on. The success of deep learning is highly related to the availability of large, labeled databases, and the ability to train these huge volumes networks. So far, the largest models have trillions of parameters [6]–[8]. Furthermore, the volume of training corpus have achieved tera-byte. For instance, Wudao 2.0 [6] is trained with 4.9TB high-quality Chinese and English corpus from WuDaoCorpora [9] and Pile [10] datasets.

While the number of model parameters increases exponentially, the storage capacity of a computing device only increases from a few GBs to 80 GBs (e.g., NVIDIA A100, H100) in the last decade, which results in a memory wall bottleneck. Thus, a moderate computing device can no longer hold the entire model. In order to train a large-scale model, we may need thousands of computing devices to work corporately, and to deliberately manage these devices to effectively and efficiently train large-scale models gains more and more attention from research community and industrial fields.

Distributed training [11] jointly make use of multiple devices to train the model and achieve reasonable training speedup. In 2012, [12] trained AlexNet with 2 GPUs in parallel, which is a initiate and successful attempt at training model with multiple devices. Then, Jeffrey Dean etc. proposes the first generation distributed deep learning system DistBelief [13], introducing the concept of distributed calculation in deep neural network model training. Moreover, they systematically design the policies of parallelism and way of synchronization so that the training process can apply to large-scale clusters. At present, the research on accelerating distributed training mainly focuses on the design of parallelism strategies and how to select them.

In this work, we draw conclusion on parallelism works available in publications and make a comprehensively analysis on these algorithms. Parallelism strategy can be divided into two categories: intra-operator parallelism [14] and inter-operator parallelism [15]. Intra-operator parallelism includes data parallelism (DP) and tensor parallelism (TP). TP is also known as intra-layer model parallelism and has some varieties, such as Row-TP, Column-TP, 2D-TP [16], and 3D-TP [17]. Inter-operator parallelism includes inter-layer model parallelism and pipeline parallelism (PP). All these strategies are helpful to accelerate the training of models, but it is unable to achieve the best performance by using only few of them. To gain better performance, researchers propose hybrid parallelism. Hybrid parallelism is the work that uses the combination of data, model, and pipeline parallelism to partition the model in a fine-grained way to increase throughput. Representative work includes Megatron-LM [18] and 3D-parallelism from DeepSpeed [19].

However, manually applying the intra-operator or inter-operator strategy of parallelism methods to a model is difficult since manual partitions require the engineer to be an expert at communication and computing. They are required to be aware of all the execution time and memory state in every sub-procedure of the training process. Moreover, strategies vary according to the device topology and the model structure. Once the devices or the model changes, experts may need to redesign a modified strategy from scratch. An excellent parallelism strategy can perform better than only using data or model parallelism, repeatedly manually designing is empirical, and often sub-optimal, sometimes even impractical in real-world applications. Thus, it is essential to search for the optimal hybrid parallel strategy automatically to save time, energy, and money.

Peng Liang, Yu Tang, Zhiquan Lai, Linbo Qiao and Dongsheng Li are with the College of Computer, National University of Defense Technology, Changsha, Hunan, P.R.China, 410073. Xiaoda Zhang, Youhui Bai and Teng Su are with the Huawei Technologies Co. Ltd. E-mail: dsli@nudt.edu.cn.

†: corresponding authors.

Manuscript received Mar. 31, 2022; revised xx xx, xxxx.

Auto-parallelism is a technique that automatically generates parallelism strategies for a given model on a specific cluster. It tries to obtain the optimal or decent parallelism strategy by, for example, optimizing a cost model built for a given neural network model and a cluster. Then it maps the strategy to devices using a well-designed runtime system. Auto-parallelism is the ultimate goal of distributed training, which could liberate engineers from manually designing strategies, and make industrial departments own the ability to efficiently train large-scale models on various computing infrastructures. Representative works include OptCNN [14], Flexflow [20], Pipedream [15], Dapple [21], Double-Recursive [22], etc [23]–[36]. However, currently, all practicable works consider only a few combinations of parallelism schemes or otherwise have weak scalability for its high arithmetical complexity [36].

What is more, researchers have proposed many emerging parallelism technologies, such as ZeRO [37], Sequence parallelism [38], Token-level parallelism [39], and so on [16], [17], [40], [41]. However, most of the existing auto-parallelism methods fail to involve all of them, while the optimal strategy needs to consider using them all. Therefore, auto-parallelism is still an insufficiently explored field for us to step in. On the one hand, the automatic parallelization search space can be further expanded, which may implicitly include better solutions. On the other hand, auto-parallelism should comprehensively consider heterogeneous computing devices, communication pace, and topology.

There are a few related surveys in the research field of distributed machine learning. Mayer and Jacobsen [11] published a detailed survey about scalable deep learning on distributed infrastructures. Verbraeken [42] discusses the techniques used for distributed machine learning. They gave a comprehensive understanding of the deep learning 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.

In this survey, we introduce the definition, challenges, basis, classification, and existing works of auto-parallelism. We first present a unified auto-parallelism definition in Sec.II that abstracts a wide range of traditional and current auto-parallelism methods. And then in Sec.III, we conclude the challenges of auto-parallelism, which are the problems that researchers should be concerned about. Thirdly, we comprehensively analyze the basis of a broad class of auto-parallelism methods in Sec.IV. Specifically, we analyze the communication consumption and memory consumption of data-parallelism, model-parallelism, and pipeline-parallelism. We also summarize some extended parallelism methods in this section. Fourthly, we give a precise classification of strategy searching methods for auto-parallelism and discuss their advantages and disadvantages in Sec.V. 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 the development of auto-parallelism in Sec.VI, including the drawbacks of existing auto-parallelism methods and possible solutions.

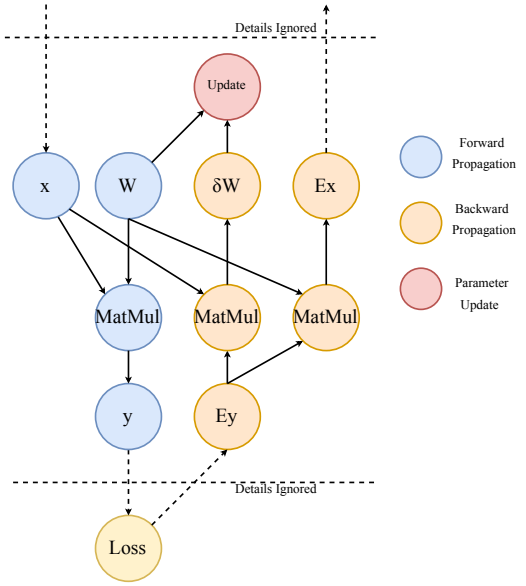



Fig. 1. A Part of a Training Computation Graph

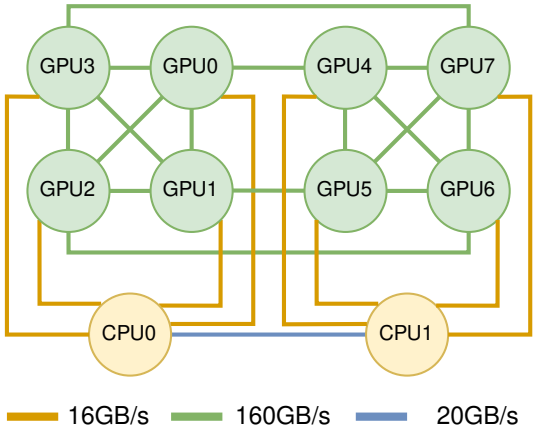



Fig. 2. Device Topology Graph of DGX-1

## II. Problem Definition

Auto-parallelism, also known as auto-parallelization or automatic parallelization, refers to automatically converting sequential code into multi-threaded or vectorized code in order to make use of available computing devices. Nowadays, auto-parallelism is frequently used in deep learning community for training and inference of deep neural networks. In the DL field, auto-parallelism refers to automatically generating computation tasks for computing devices by means of splitting, merging, or re-formalization on the network's computation graph. 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 neural network 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., a $n$-dimensional array).

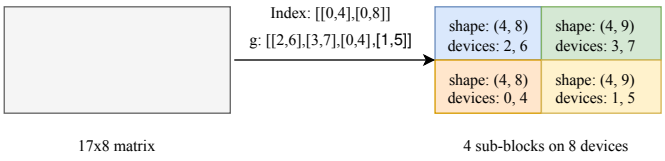



Fig. 3. Illustration of partitioning a 17x8 matrix into sub-blocks.

Tensor could be inputs, outputs of the model, intermediate 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 transport 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 parameter weight. Fig. 1 shows an extracted sub-computation graph, which shows the details of training a matrix multiplication operator, ignores its preceding and succeeding details, and uses an optimizer without optimizer states.

As for devices, device topology can be modeled as a undirected graph $\mathcal{D} = (V_\mathcal{D}, E_\mathcal{D})$, 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 a hardware connection (e.g., PCI-e, NVLink, InfiniBand, etc.) between devices $d_i$ and $d_j$. Auto-parallelism algorithm $\mathcal{A}$ takes $\mathcal{G}$ 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 all nodes in $\mathcal{G}$.

For example, $p_i = (\textbf{index}, \textbf{g}) \in \mathcal{P}$ is a partition of $v_i$, where **index** is a 2D-array recording split index of each axis that can infer the sizes of all sub-blocks after partitioning, **g** is a device group 2D-array in which $i$-th array holds ids of device that holds $i$-th 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 **G**, where $G^1_{d_0} \in \textbf{G}$ 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). Figure 3 illustrates a toy example of partition using $p = ([[0, 4], [0, 8]], [[2, 6], [3, 7], [0, 4], [1, 5]])$ on a $17 \times 8$ matrix. Taking the fist sub-block for illustration, the **index** array indicates the range of this sub-block, which is from 0-th row to 4-th row, and from 0-th column to 8-th column; the **g** indicates that this sub-block is held by device 2 and 6.

## III. Challenges of Auto-Parallelism

There are five main challenges in auto-parallelism.

- The first and the 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 a similar computation 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 analysis for 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 sec 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 resource on each device are usually the same. Tofu [43], Hypar [44] and D-Rec [22] utilize this property, and consider communication cost only to produce DP and 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 Vanilla DP and 1D-TP (Row-TP and Column-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 reduces the memory of intermediate results (e.g., $T_{in}, T_{out}$) to a sub-linear degree, which makes Row-TP and Column-TP have lower memory cost and so that we can increase batchsize and model size. Some researchers prefer to reduce memory cost by applying 1D-TP strategies and check-pointing instead of DP, although DP theoretically has smaller communication cost in some cases. Applying check-pointing requires us to analyze both communication cost and memory cost in order to improve throughput finally.

Applying PP into the training can also reduce a large amount of intra-operator communication cost because PP creates stages held by corresponding subsets of devices. Devices only need to do intra-operator communication within their communication group. Nevertheless, it may bring a little 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 strategy searching methods in Sec.V.

### C. Load-Balance in Heterogeneous Topology

Heterogeneous topology here represents a device graph with different types of computing devices in it (e.g., CPU and GPU of multiple types). Using heterogeneous clusters to train a model more environmentally is a good choice. Buying new devices does not mean the old ones cannot work anymore because they could also participate in some parts of the work. Different types of devices usually have different computing performances. The goal is to arrange the computation properly to achieve load-balance on each device. However, only a few works involve the load-balance analysis in the strategy searching task. DeepSpeed [19] heuristically uses CPU to execute the update of parameters because the updates are less complicated compared to forward and backward propagation and the computation of updates on CPU can overlap the computation on GPU in certain situations. Paddle-HeterPS [28] uses reinforcement learning to decide computing devices for every layer, but it only supports DP and PP. AccPar [23] introduces a method that solves partition ratios of each kind of device and then partitions model layer by layer, after which the computation time on each device is similar. However, it only supports DP and 1D-TP without check-pointing. Auto-parallelism on heterogeneous topology is still explorable, and it will save much money from buying more devices.

### D. Topology-aware Communication Optimization

Auto-parallelism algorithms need to consider topology-aware communication strategies to reduce communication time further and increase throughput. Due to the limited size of the motherboard of a node, a large number of computing devices are distributed to different nodes, which results in bandwidth differences between intra-node and inter-node communication. While intra-node bandwidth is usually faster than inter-node's, making full use of intra-node bandwidth can optimize communication and thus reduce overall execution time. [45], [46] 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, $P^2$ [47] can generate DP and 1D-TP partition strategies and utilizes the system hierarchy to synthesize the best reduction strategies that consist of sequences of common collective communication operations, which is proven to be faster than a single All-Reduce operation among all devices in many cases. These works on all-reduce optimize intra-layer communication we mentioned above. Another communication optimization work on tensor distribution reduces the inter-layer communication cost. [48] reduces the communication amount of tensor re-distribution by replacing 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 of Runtime and Strategy Performance in Finding Strategy

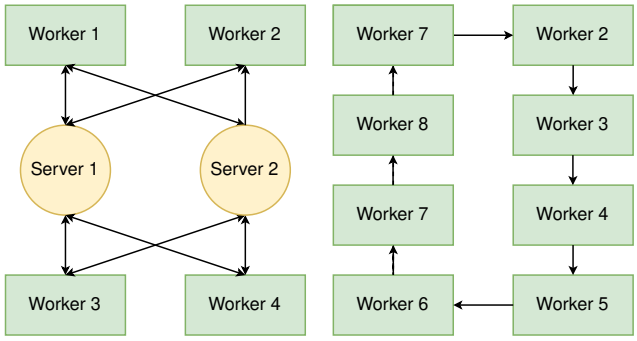
Strategy searching algorithms are time-consuming for two reasons. The first one is that partitioning a DAG for optimal performance is an NP-hard problem [49]–[52]. The second one is to evaluate every strategy that the algorithm finds.

To solve the NP-hard problem, researchers have tried to use machine-learning algorithms [1], [53] and classic algorithms like dynamic programming [54]. Some of the works [43], [55] additionally uses heuristic assumptions that help shorten searching runtime but may sacrifice the performance of the strategy. We will discuss more details of this in Sec.V.

To evaluate the performance of searched strategies, we can use a cost model that calculates the cost of a strategy or profile the execution time of strategy by deploying a strategy to the model and running it. Some auto-parallelism works [22], [56], [57] use a symbolic cost model to analyze the performance of strategies. However, most accelerators (e.g., GPU, NPU, FPGA) do the computation in parallel, while a symbolic cost model only reflects on the serial amount of computation. Meanwhile, different types of devices may have different performances and implementation on some specific tasks like convolution, which is hard for the system to be aware of and thus needs more artificial annotation work on tuning the cost model. Moreover, it is hard for symbolic cost models to be aware of the overlaps between computation and communication. These all make the symbolic cost model hard to accurately reflect the actual performance of found strategies, though it has shorter runtime than profiling. On the other hand, profiling every schedule that the algorithm generates is too time costly [15], [25], [58], although it can accurately tell us the difference between any two strategies. Using a profiling-based cost model [55], [59] seems to be a more reasonable decision, whose costs are the actual time obtained by running each operator in the computation graph. And then, we can find a near-optimal parallelism strategy by minimizing cost.

## IV. The Analysis on Different Parallelism Schemes

Detailed analysis on communication, computation, and memory cost 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 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 assume that we are using homogeneous clusters. Because in homogeneous clusters, devices have the same computation capacity, which means that partitions can be executed averagely, and thus each device has the same computation cost when given the partitioned task. This assumption helps us focus on evaluating communication costs in different strategies. It should be noted that communication amount is the most crucial factor that we need to consider in generating strategies, and computation is another critical factor for balancing works on every device.




(a) Parameter Sever architecture. (b) Ring-Allreduce architecture.

Fig. 4. Typical centralized architecture and decentralized architecture.

We divide parallelism schemes into two categories: intra-operator parallelism and inter-operator parallelism. Intra-operator parallelism shards the tensors (i.e., $v_i \in V$) along its axes while inter-operator divides a computation graph $\mathcal{G}$ into several sub-graphs $\mathbf{G}$ by nodes. Tab. I shows the intra-operator partitions of a matrix multiplication $T_{out} = T_{in}W$, where $T_{out}$ is an output of shape $(b, w_{out})$, $T_{in}$ is an input of shape $(b, w_{in})$ and $W$ is a weight matrix of shape $(w_{in}, w_{out})$. It shows that different partitions represent different intra-operator parallelism schemes, including DP, ZeRO-powered DP, row-wise TP (Row-TP), column-wise TP (Column-TP), and 2D, 2.5D, 3D-TP. As Tab. II shows some examples of different strategies under a specific number of devices, the partitions of intra-operator parallelism can be modeled by $p_i \in \mathcal{P}$ mentioned in Section II. Note that tensor parallelism is also known as intra-layer model parallelism since it partitions model weight. Inter-operator parallelism includes inter-layer model parallelism and pipeline parallelism (PP). They both partition models into several stages that consist of layers and do not change the partitions within any tensors, and we can use sub-graphs to describe this kind of partition.

Then, we move to analyze the parallelism schemes we mentioned above.

### A. Data Parallelism

Data Parallelism (DP) is the earliest parallelism scheme and is still one of the most commonly used schemes in distributed training. In DP, training data samples are split into several parts along the batchsize axis, and each worker computes the corresponding part. Each worker conducts an independent training process through stochastic gradient algorithms while using communication to sync the models.

*1) Vanilla Data Parallelism:* Vanilla data parallelism only partitions data-related tensors. Training samples are divided into several parts in the vanilla DP, and the entire model is duplicated in each worker. PyTorch DDP [60] is an auto-parallelism method that only supports vanilla DP. It automatically inserts all-reduce operators for gradients synchronization after backward propagation. Its usability attracts many researchers to use it to train networks. Vanilla data parallelism is functional when training a small neural network. However, it is very limited in training large-scale models because of the memory redundancy in storing replicas of the model.

*2) ZeRO-Powered Data Parallelism:* To address the above redundancy problem, DeepSpeed [19] team from Microsoft develop Zero Redundancy Optimizer (ZeRO) [37], which can partition model states including parameters, gradients, and optimizer states across all the computing devices (worker) averagely. Under the ZeRO-powered data parallelism (ZeRO-DP) strategy, each computing device also trains a different part of the input data and only maintains its owned partitioned model parameters. Therefore, ZeRO-DP is essentially a kind of data parallelism. ZeRO-DP requires a worker to gather a subset of the model from other workers only when needed, eliminating the redundancy of model states in vanilla DP. DeepSpeed has implemented an auto-parallelism runtime system that helps determine the stage of ZeRO-DP, batchsize, and other ZeRO optimization configurations. Users can use DeepSpeed to launch a ZeRO-DP training with only a few lines of change.

ZeRO-DP has three stages: stage 1 that only partitions optimizer states, stage 2 that partitions gradients and optimizer states, and stage 3 that partitions parameters additionally. When using ZeRO-DP stage 3 in a communication group with $N$ devices, each worker only needs to maintain $1/N$ of the model states; this means training a model with 1 billion parameters with Adam optimizer, each device only needs $16GB/N$ memory to store the model. However, ZeRO-DP stage 3 needs an extra all-gather communication operation in backward-propagation to collect parameters. ZeRO-Infinity [61] points out that using ZeRO and tricks like memory offloading, we can train a dense model with over 30 trillion parameters on 512 NVIDIA V100 Tensor Core GPUs, which is 107x larger than the current biggest singleton model Gopher [62] with 280 billion parameters. However, this requires an ultra-high communication bandwidth to train it in practical due to the highly increased communication volume.

*3) Communication of DP:* The communication of DP is the communication of model parameters. There are two topology architectures of synchronizing parameters: centralized architecture and decentralized architecture. Centralized architecture refers to the system that has one or more master workers, which send and receive parameters or gradients to/from each slave worker, such as Parameter Server [63]. Currently, the most representative work is BytePS [64], which claims to use more CPU worker parameter servers to increase bandwidth between master workers and slave workers. In parameter server, the main characteristic is that only Server devices connect to all other devices called Worker. Fig. 4a shows a parameter server architecture with 2 Servers and 4 Workers. Servers receive gradients from all the Workers after backward propagation. It updates model weights (parameter) with collected gradients and sends updated parameters back to other workers to finish an iteration step. This process results in a communication volume of two times of parameters. We can increase the bandwidth of the parameter server by adding more Servers. BytePS tries to use the same number of CPUs as GPUs to be the Servers and thus alleviate the burden of each Server device.

For decentralized architecture, the most representative work are PyTorch DDP [60], Horovod [65] and DeepSpeed [19],

TABLE I
PARTITIONS OF DIFFERENT INTRA-OPERATOR PARALLELISM SCHEMES

| Scheme Name | Input ($T_{in}$) | Weight ($W$) | Output ($T_{out}$) | Weight Gradient ($\delta W$) | 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-powered DP stage 1 [37] | $(b/p, w_{in})$ | $(w_{in}, w_{out})$ | $(b/p, w_{out})$ | $(w_{in}, w_{out})$ | $(w_{in}/p, w_{out})$ or $(w_{in}, w_{out}/p)$ |
| ZeRO-powered DP stage 2 [37] | $(b/p, w_{in})$ | $(w_{in}, w_{out})$ | $(b/p, w_{out})$ | $(w_{in}/p, w_{out})$ or $(w_{in}, w_{out}/p)$ | $(w_{in}/p, w_{out})$ or $(w_{in}, w_{out}/p)$ |
| ZeRO-powered DP stage 3 [37] | $(b/p, w_{in})$ | $(w_{in}/p, w_{out})$ or $(w_{in}, w_{out}/p)$ | $(b/p, w_{out})$ | $(w_{in}/p, w_{out})$ or $(w_{in}, w_{out}/p)$ | $(w_{in}/p, w_{out})$ or $(w_{in}, w_{out}/p)$ |
| Row-TP | $(b, w_{in}/p)$ | $(w_{in}/p, w_{out})$ | $(b, w_{out})$ | $(w_{in}/p, w_{out})$ | $(w_{in}/p, w_{out})$ |
| Column-TP | $(b, w_{in})$ | $(w_{in}, w_{out}/p)$ | $(b, w_{out}/p)$ | $(w_{in}, w_{out}/p)$ | $(w_{in}, w_{out}/p)$ |
| 2D-TP [16] | $(\frac{b}{\sqrt{p}}, \frac{w_{in}}{\sqrt{p}})$ | $(\frac{w_{in}}{\sqrt{p}}, \frac{w_{out}}{\sqrt{p}})$ | $(\frac{b}{\sqrt{p}}, \frac{w_{out}}{\sqrt{p}})$ | $(\frac{w_{in}}{\sqrt{p}}, \frac{w_{out}}{\sqrt{p}})$ | $(\frac{w_{in}}{\sqrt{p}}, \frac{w_{out}}{\sqrt{p}})$ |
| 2.5D-TP [41] | $(\frac{b}{\sqrt{pd}}, \frac{w_{in}}{\sqrt{p/d}})$ | $(\frac{w_{in}}{\sqrt{p/d}}, \frac{w_{out}}{\sqrt{p/d}})$ | $(\frac{b}{\sqrt{pd}}, \frac{w_{out}}{\sqrt{p/d}})$ | $(\frac{w_{in}}{\sqrt{p/d}}, \frac{w_{out}}{\sqrt{p/d}})$ | $(\frac{w_{in}}{\sqrt{p/d}}, \frac{w_{out}}{\sqrt{p/d}})$ |
| 3D-TP [17] | $(\frac{b}{p^{2/3}}, \frac{w_{in}}{p^{1/3}})$ | $(\frac{w_{in}}{p^{1/3}}, \frac{w_{out}}{p^{2/3}})$ | $(\frac{b}{p^{2/3}}, \frac{w_{out}}{p^{1/3}})$ | $(\frac{w_{in}}{p^{1/3}}, \frac{w_{out}}{p^{2/3}})$ | $(\frac{w_{in}}{p^{1/3}}, \frac{w_{out}}{p^{2/3}})$ |

TABLE II
EXAMPLES OF 2D-ARRAY **index** OF INTRA-OP PARALLELISM SCHEMES

| | $T_{in}$ | $W$ | $T_{out}$ | $\delta W$ | $O_s$ |
|---|---|---|---|---|---|
| Vanilla DP [1] | $[[0, b/2], [-1]]$ [4] | $[[-1], [-1]]$ | $[[0, b/2], [-1]]$ | $[[-1], [-1]]$ | $[[-1], [-1]]$ |
| ZeRO stage 1 [1] | $[[0, b/2], [-1]]$ | $[[-1], [-1]]$ | $[[0, b/2], [-1]]$ | $[[-1], [-1]]$ | $[[0, w_{in}/2], [-1]]$ |
| ZeRO stage 2 [1] | $[[0, b/2], [-1]]$ | $[[-1], [-1]]$ | $[[0, b/2], [-1]]$ | $[[0, w_{in}/2], [-1]]$ | $[[0, w_{in}/2], [-1]]$ |
| ZeRO 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] | $[[-1], [-1]]$ | $[[-1], [0, w_{out}/2]]$ | $[[-1], [0, w_{out}/2]]$ | $[[-1], [0, w_{out}/2]]$ | $[[-1], [0, w_{out}/2]]$ |
| 2D-TP [2] | $[[0, b/2],$ $[0, w_{in}/2]]$ | $[[0, w_{in}/2],$ $[0, w_{out}/2]]$ | $[[0, b/2],$ $[0, w_{out}/2]]$ | $[[0, w_{in}/2],$ $[0, w_{out}/2]]$ | $[[0, w_{in}/2],$ $[0, w_{out}/2]]$ |
| 3D-TP [3] | $[[0, b/4,$ $b/2, 3b/4],$ $[0, w_{in}/2]]$ | $[[0, w_{in}/2],$ $[0, w_{out}/4,$ $w_{out}/2, 3w_{out}/4]]$ | $[[0, b/4,$ $b/2, 3b/4],$ $[0, w_{out}/2]]$ | $[[0, w_{in}/2],$ $[0, w_{out}/4,$ $w_{out}/2, 3w_{out}/4]]$ | $[[0, w_{in}/2],$ $[0, w_{out}/4,$ $w_{out}/2, 3w_{out}/4]]$ |

[1] Partitioned on 2 devices, device group $g$ is $[0, 1]$ as an example.
[2] Partitioned on 4 devices, device group $g$ is $[0, 1, 2, 3]$ as an example.
[3] Partitioned on 8 devices, device group $g$ is $[0, 1, 2, 3, 4, 5, 6, 7]$ as an example.
[4] -1 represents for a non-partition along this axis.

which form DP group with ring topology to do collective operations [66] effectively. As Fig. 4b shows, there is no specific server worker in decentralized architecture as each worker serves as Server and Worker at the same time. Take Horovod as an example; the Worker in Horovod only communicates with its neighbor Worker using the ring-allreduce algorithm. Ring-based allreduce algorithm is proven to be bandwidth-optimal [67]. Decentralized architecture with ring-topology uses ring-allreduce to exchange parameters and gradients, and it has been widely used in large-scale model training. A ring-allreduce algorithm consists of two steps: a reduce-scatter and all-gather collective operations. In each step, each worker sends only $1/N$ of the data to its neighbor $N-1$ times. The total volume of sending data of each worker is, therefore, $(N-1)K/N$, where $K$ represents the volume of the data.

The communication of ZeRO-DP is based on ring-allreduce. However, there is a subtle difference between vanilla ring-allreduce DP and ZeRO-DP. Vanilla DP reduce-scatters and all-gathers only the gradients, after which all workers update parameters simultaneously, which results in redundant computation in the updates of parameters in vanilla DP. For stage 1 and stage 2, ZeRO-DP uses a reduce-scatter to get accumulated gradients for each worker and then updates its owned parameters. Finally, it uses an all-gather operation to synchronize updated parameters on all workers. This subtle change in communication enables ZeRO-DP to update parameters with no redundant computations while maintaining correctness. Based on ZeRO-DP stage 2, ZeRO-DP stage 3 needs an extra all-gather communication before executing backward propagation of the corresponding weight matrix. This all-gather operation results in 50% more communication volume.

Tab. III gives the comparison of the minimum communication volume (bandwidth cost) of vanilla allreduce algorithm in parameter server and ring-allreduce algorithm in decentralized architecture in an iteration, where $P$ represent the number of parameters, $N$ represents the number of Worker devices, and $n$ represents the number of Server devices. When the number of Worker devices increases, using the ring-allreduce algorithm has a less communication volume that converges to $2P$, indicating a high scalability potential. For parameter servers, by adding Server devices, the sending volume of Servers tends to converge to $P$, and hence all Server and Worker devices only need to communicate $P$ data; this is why BytePS claims that theoretically, it can be up to 2x faster than ring-allreduce. However, this may incur extra costs in maintaining clusters and designing device topology. In addition, the parameter server currently did not solve the redundancy problem that ZeRO solved. So researchers prefer to use ring-allreduce based DP to train large-scale models.

The biggest flaw of DP is the redundancy of storing replicas. Though ZeRO [37], [61] have tried to optimize the redundancy in DP, it introduces extra communication cost in the system. An effective way to alleviate redundancy and communication cost is to use model parallelism.

### B. Model Parallelism

Model parallelism occurs because device memory is insufficient to hold the entire replica of a model in vanilla DP. Since ZeRO-DP can address this flaw well, the goal of model parallelism now changes to reduce the amount of data that transfers between devices and the memory cost of temporary activation values. We divide MP into two categories: inter-layer and intra-layer models. Intra-layer model parallelism is also known as tensor parallelism (TP) since it partitions the weight tensor. To simplify our illustration, we note inter-layer model parallelism as MP and intra-layer model parallelism as TP in the following sections.

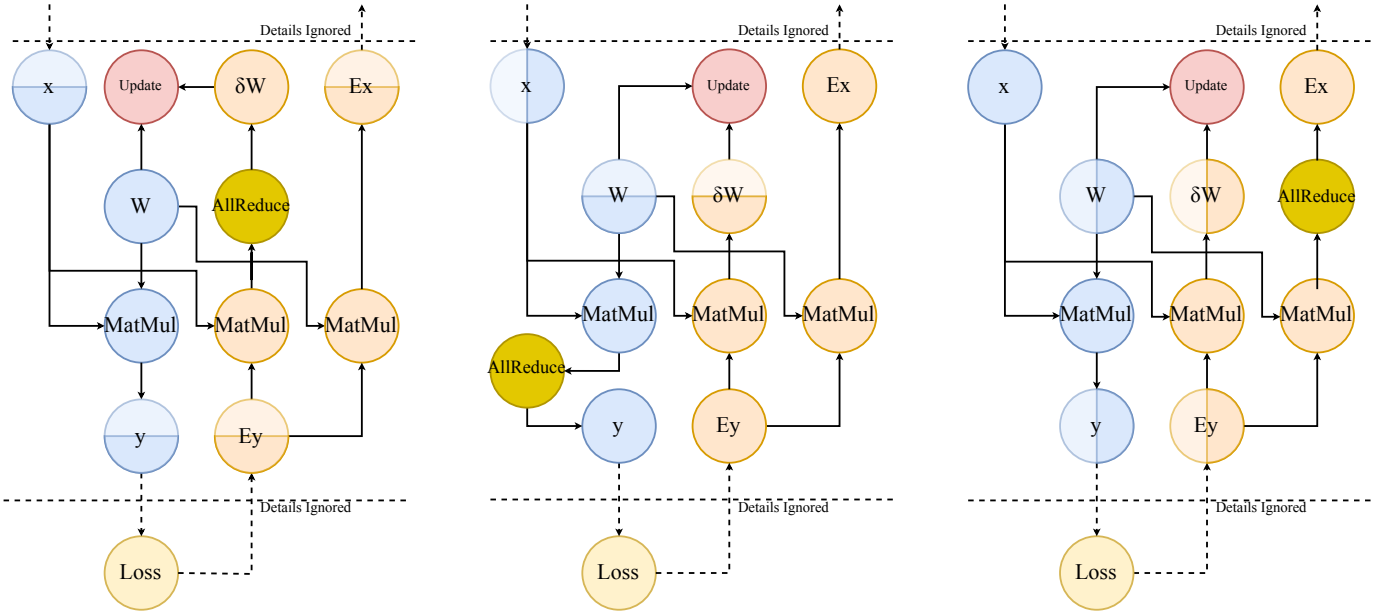
*1) Inter-Layer Wise Model Parallelism (MP) and Pipeline Parallelism:* In inter-layer model parallelism, the model is partitioned into several stages that usually consist of continuous layers. A stage will be executed only when the computation of the previous stage finishes. Devices only need to transfer intermediate activation values and gradients between devices. So the amount of transfer data is smaller than data parallelism when applied to a fully-connected (FC) network, which usually has a vast weight matrix but a small output tensor. Moreover, MP reduces the storage of temporary activation values on each device due to the model's partition. Thus the device can train the model with a larger batchsize. However, as shown in Fig.7.a, each computing device (worker) holds only one stage in MP, resulting in data relevance among computing devices. Due to data relevance, only one device is running at any time of training when using inter-layer model parallelism, which leads to a low utilization rate of computing devices.

*2) Intra-Layer Wise Model Parallelism:* Intra-layer model parallelism starts from partitioning a matrix that does multiplication computing. It splits the large weight matrix [18], [69] to execute it efficiently. A weight matrix often consists of two axes: row and column. Therefore, we have row-TP and column-TP, where the weight matrix is divided along the row dimension and column dimension, respectively. Intra-Layer model parallelism has an excellent performance in accelerating matrix multiplication but brings some communication overheads. We will further explore the overheads in section IV-B3.

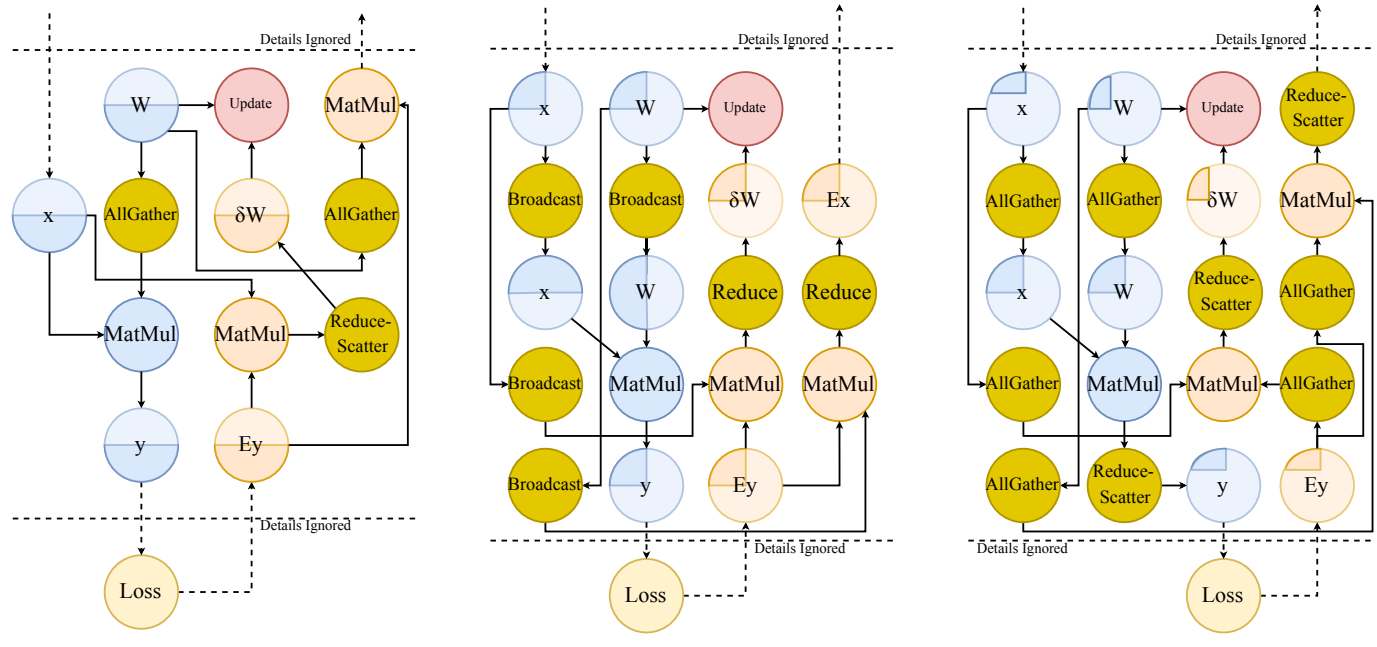
Intra-layer model parallelism has some varieties for different device topologies. Optimus [16], SUMMA2.5 [41] and 3D parallel transformer model [17] applies 2D, 2.5D and 3D tensor parallelism respectively to help further reduce activation memory and communication while increasing throughput. They split the weight matrix along both the row axis and column axis and prove that this helps reduce each worker's activation memory and communication volume. Suppose we have $p$ devices, 2D-TP partitions both input tensor and weight matrix into $p$ sub-blocks along both row and column axis (e.g., weight matrix is partitioned into sub-matrices with shape $[w_{in}/\sqrt{p}, w_{out}/\sqrt{p}]$ among $p$ devices). 2.5D-TP partitions input tensor into sub-blocks with shape $[b/\sqrt{pd}, w_{in}/\sqrt{p/d}]$, and weight matrix into sub-blocks with shape $[win/\sqrt{p/d}, w_{out}/\sqrt{p/d}]$, where $d$ here is the depth of the processor group. 3D-TP partitions input tensor into sub-blocks with shape $[b/p^{2/3}, w_{in}/p^{1/3}]$ and weight matrix

TABLE III
COMPARISON OF THE MINIMUM COMMUNICATION VOLUME OF DATA PARALLELISM

| Method | Volume on Servers | Volume on Workers | Total Volume |
|---|---|---|---|
| Parameter Server [63] | $(N-1)P$ | $P$ | $2(N-1)P$ |
| BytePS [64] | $(N-1)P/n$ | $P$ | $2(N-1)P$ |
| Ring-Allreduce [68] | None | $2(N-1)P/N$ | $2(N-1)P$ |
| ZeRO stage 1&2 [37] | None | $2(N-1)P/N$ | $2(N-1)P$ |
| ZeRO stage 3 [37] | None | $3(N-1)P/N$ | $3(N-1)P$ |




(a) DP (b) Row-TP (c) Column-TP




(d) ZeRO-DP stage 3 (e) 2D-TP (f) 3D-TP

Fig. 5. Computation Graph of Intra-operator Parallelsim

into sub-blocks with shape $[w_{in}/p^{1/3}, w_{out}/p^{2/3}]$. 2D-TP and 2.5D-TP uses broadcast and reduce communication operations to help calculate the output results, while 3D uses all-gather and reduce-scatter collective operations. It is emphasized that the communication amounts of these three methods are smaller than 1D-TP while having no redundancy in saving intermediate activation values. Currently 2D, 2.5D and 3D intra-layer model-parallelism has been integrated into deep learning framework Colossal-AI [70] from HPC-AI Tech.

*3) Communication of MP and TP:* Inter-layer model parallelism requires communication of intermediate results between two layers. We suppose that MP splits the model after this layer, then we will need to transfer $T_{out}$ to the device which holds the next layer to continue computation, resulting in communication volume of $bw_{out}$.

Now let us discuss intra-layer model parallelism. OWT [71] experimentally finds that using DP for CNN and TP for FC has a higher throughput than using only one of them in training AlexNet [12]. OWT inspires researchers to discover the reasons. It turns out that this is due to the communication difference between DP and TP: CNN has a lower weight matrix and huge output tensor, which results in less communication when applied in DP, and FC is the opposite. The communication of TP consists of intra-layer and inter-layer communication. Intra-layer communication happens when a tensor is partitioned and brings inconsistency during execution. They are often synchronous operations, such as all-reduce, broadcast, and reduce or combination of reduce-scatter and all-gather. Inter-layer communication is responsible for redistributing a tensor partitioned under strategy A to new partitions under strategy B. We can use an all-gather or all-to-all communication operator to handle the redistribution. AccPar [23] is an strategy searching method that makes good use of analysis of communication of intra-layer model parallelism to find the best heterogeneous parallelism strategies for clusters. In this survey, we extend their work in analyzing communication.

We first illustrate intra-layer communication. For row-TP on $p$ devices, we divide the weight matrix into $p$ sub-matrix whose shape is $(w_{in}/p, w_{out})$. In the meanwhile, we partition input tensor into $p$ part whose shape is $(b, w_{in}/p)$. Since all workers can generate an output tensor with shape $(b, w_{out})$ but may have different values, we must synchronize the output tensor using an all-reduce operation. Similarly, for column-TP on $p$ devices, we need to synchronize the error (i.e., gradient) of $T_{in}$ whose shape is also $(b, w_{in})$ which is generated in backward propagation. Obviously, the data that need to be synchronized here is also the redundant (or temporarily redundant) memory cost of each method. For multi-dimensional TP, both 2D-TP and 3D-TP need to transfer the intermediate activation values and model weights in every matrix multiplication. However, their communication volumes are smaller than 1D-TP in most cases. Tab. IV shows more details of the intra-layer communication in the training process, which contains replicated tensors, tensors that need communication, communication operators they use, and maximum communication cost on each worker. We ignore 2.5D-TP here because 2.5D-TP is actually a 2D-TP that replicates $d$ times.

Inter-layer communication is more complicated than intra-layer communication. Inter-layer communication is also known as tensor redistribution [48], [57] since it reorganizes the distribution of tensors. The combinations of DP (including ZeRO-DP), Row-TP, Column-TP, 2D-TP, and 3D-TP generate 25 kinds of tensor redistribution strategies. The communication of inter-layer happens in forward and backward propagation, where we calculate intermediate results $T_{out}$ and the error of input tensor $E_{in}$, respectively. Tab. V shows the communication volume of inter-layer communication. To simplify our illustration, we suppose that we have $p$ devices, and layer $l$ and layer $l+1$ both run on them. Layer $l$ generates $T_{out}$ and forward propagates it to layer $l+1$, and layer $l+1$ backward propagates error of $T_{out}$, $E_{out}$ for layer $l$ to calculate the $E_{in}$ We use all-gather or all-to-all to communicate for inter-layer TP communication and involve DP in the table since DP is also an intra-op partition strategy. All-gather happens in DP layer to Column-TP layer, Row-TP layer to DP, Row-TP, 2D-TP and 3D-TP layer, Column-TP to Column-TP layer, 2D-TP to Column-TP layer and 3D-TP to Column-TP layer. No communication happens in the DP layer to the DP layer, Row-TP layer to Column-TP layer, Column-TP layer to Row-TP layer, 2D-TP layer to 2D-TP layer, and 3D layer to 3D layer. All-to-all happens in the remaining circumstances.

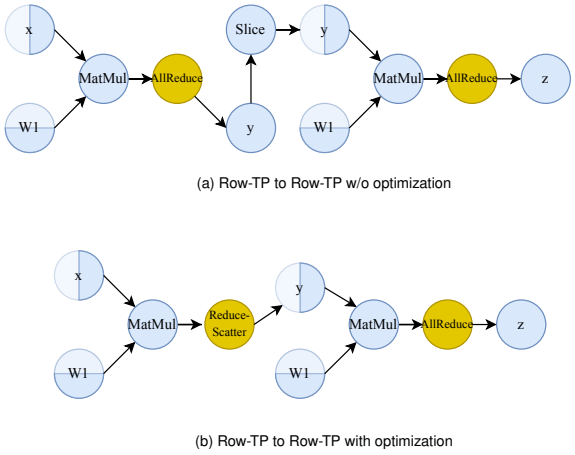



Fig. 6. Optimization on Row-TP to Row-TP

We find a possible optimization strategy under several circumstances when considering intra-layer communication and inter-layer communication together. We can replace the all-reduce operation in intra-layer communication of Row-TP with a reduce-scatter inter-layer communication operation when its next layer is a DP or Row-TP partition; replace the one of Column-TP when its preceding layer is a DP or Column-TP partition. This replacement saves a communication cost of $(p-1)bw_{out}/p$. Fig. 6 shows an example of this optimization of a Row-TP layer to Row-TP layer.

MP and TP both offer ideas to reduce communication pressure and activation memory. However, with an unavoidable data-relevance problem, the efficiency of MP is not high enough. Therefore, MP usually works with other parallelism strategies like DP or PP. Like DP, there is some redundancy and communication in TP, so TP is more like a trade-off choice for us. It is evident that using TP for CNN, which has a

TABLE IV
INTRA-OP PARALLELISM SCHEMES: INTRA-LAYER COMMUNICATION COST OF MATMUL $T_{out} = T_{in}W$ IN A TRAINING STEP

| Method | Replicated Tensor | Communication Tensor | Communication Pattern | Communication Cost |
|---|---|---|---|---|
| Vanilla DP | $W, \delta W, O_s$ | $\delta W$ | All-Reduce | $\frac{2(p-1)}{p} w_{in} w_{out}$ |
| ZeRO-DP 1 [37] | $W, \delta W$ | $W, \delta W$ | Reduce-Scatter, All-Gather | $\frac{2(p-1)}{p} w_{in} w_{out}$ |
| ZeRO-DP 2 [37] | $W$ | $W, \delta W$ | Reduce-Scatter, All-Gather | $\frac{2(p-1)}{p} w_{in} w_{out}$ |
| ZeRO-DP 3 [37] | None | $W, \delta W$ | Reduce-Scatter, All-Gather | $\frac{3(p-1)}{p} w_{in} w_{out}$ |
| Row-TP | $T_{out}, E_{out}$ | $T_{out}$ | All-Reduce | $\frac{2(p-1)}{p} b w_{out}$ |
| Column-TP | $T_{in}, E_{in}$ | $E_{in}$ | All-Reduce | $\frac{2(p-1)}{p} b w_{in}$ |
| 2D-TP [16] | None | $W, \delta W, T_{in}, E_{in}$ | Broadcast, Reduce | $\frac{3 \log p}{2\sqrt{p}} (b w_{in} + w_{in} w_{out})$ |
| 3D-TP [17] | None | $W, \delta W, T_{in}, E_{in}, T_{out}, E_{out}$ | Reduce-Scatter, All-Gather | $\frac{3(p^{1/3}-1)}{p} (b w_{in} + w_{in} w_{out} + b w_{out})$ |

TABLE V
TENSOR REDISTRIBUTION: INTER-LAYER COMMUNICATION VOLUME

| | | Layer $l+1$ | | | | |
|---|---|---|---|---|---|---|
| | | DP | Row-TP | Column-TP | 2D-TP | 3D-TP |
| Layer $l$ | DP | 0 | $\frac{2(p-1)bw_{out}}{p^2}$ | $(p-1)bw_{out}/p$ | $2\frac{1-p^{-1/2}}{p} bw_{out}$ | $2\frac{1-p^{-1/3}}{p} bw_{out}$ |
| | Row-TP | $(p-1)bw_{out}/p$ | $(p-1)bw_{out}/p$ | 0 | $(p-1)bw_{out}/p$ | $(p-1)bw_{out}/p$ |
| | Column-TP | $\frac{2(p-1)bw_{out}}{p^2}$ | 0 | $(p-1)bw_{out}/p$ | $2\frac{1-p^{-1/2}}{p} bw_{out}$ | $2\frac{1-p^{-1/3}}{p} bw_{out}$ |
| | 2D-TP | $2\frac{1-p^{-1/2}}{p} bw_{out}$ | $2\frac{1-p^{-1/2}}{p} bw_{out}$ | $(p-1)bw_{out}/p$ | 0 | $2bw_{out}/p$ |
| | 3D-TP | $2\frac{1-p^{-1/3}}{p} bw_{out}$ | $2\frac{1-p^{-1/3}}{p} bw_{out}$ | $(p-1)bw_{out}/p$ | $2bw_{out}/p$ | 0 |

small weight matrix but tremendous intermediate results, is not acceptable. TP is more suitable for a model with big weight matrixes in order to gain performance improvement [71].

### C. Pipeline Parallelism

MP is relatively more complicated than DP to design and reproduce because MP often requires a good balance of model scaling capacity, flexibility, and efficiency among devices. Most of the pipeline parallelism (PP) methods can automatically partition models to balanced stages on top of MP [15], [25], [72]. The partition pattern of PP is the same as that of MP; this is why some emerging popular frameworks like MindSpore [57] and OneFlow [31], and researchers name inter-layer model parallelism as pipeline parallelism directly. The only difference between PP and MP is that PP is the well-scheduled pipelined MP, which can overlap the computation of different batches. PP was proposed to solve the low-utility problem of MP. With the development of TP, researchers tend to replace MP/PP with TP in most cases. However, PP is still effective in training large-scale models because it introduces less communication than DP and TP, and it is beneficial to enlarge batch size.

Typical works on pipeline parallelism (PP) include asynchronous pipeline PipeDream [15] from Microsoft and synchronous pipeline GPipe [72] from Google. Synchronicity here represents the matches of weight versions between forward and backward propagation, guaranteeing convergence. From these two works, researchers have proposed lots of varieties. Fig.7 shows the scheduling details of the below-mentioned pipeline varieties.

*1) Asynchronous Pipeline Parallelism:* The most representative work of asynchronous pipeline parallelism is PipeDream. PipeDream pursues higher throughput and utilization of devices. It automatically partitions layers into load-balanced stages which have similar computation times. Each device on a different stage processes a different micro-batch of data simultaneously, avoiding the data relevance problem. PipeDream eliminates bubbles in the pipeline by storing multiple versions of the parameters. However, it brings extra memory costs and a staleness problem due to its asynchronous updates, resulting in a possible convergence problem. PipeDream-2BW [73] optimizes the memory usage in PipeDream, which needs only two buffers to store generated weights of different versions.

Although the experiments in PipeDream and PipeDream-2BW show that using asynchronous pipeline schedules does not hurt the convergence, we still need to be cautious about the convergence problem that may occur due to delayed updates in such schedules [74]–[77]. Currently, the PipeDream-based pipeline schedule has been widely used in Megatron-LM [78]

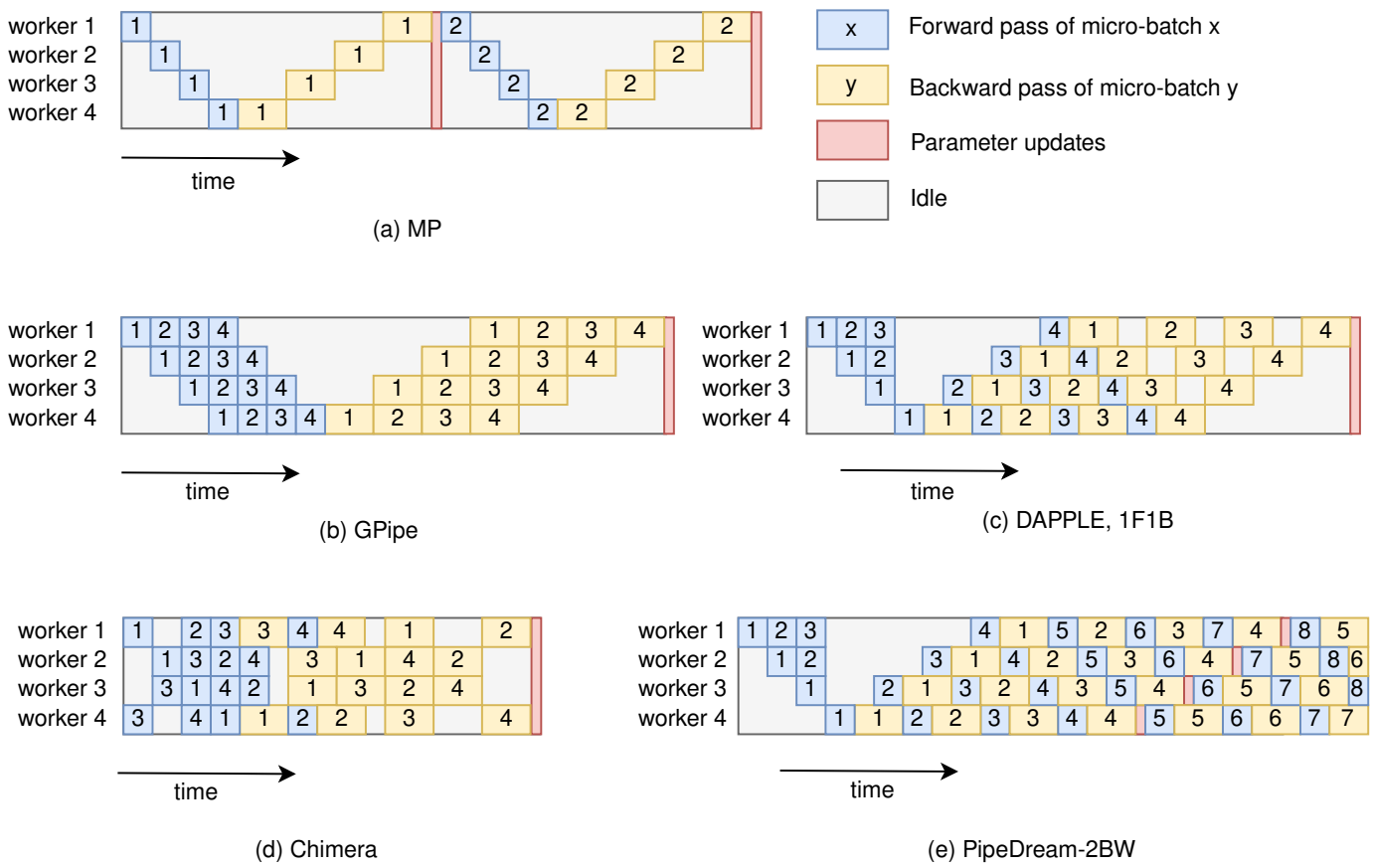



Fig. 7. Inter-layer wise MP.

(i.e., PipeDream-1F1B), and Microsoft Fiddle project's recent works [55], [58].

*2) Synchronous Pipeline Parallelism:* GPipe was first designed for inter-layer MP, as shown in Fig. 7.b. GPipe first partitions the model into $\mathcal{N}$ stages and puts the $n$-th stage on the $n$-th accelerator. Based on this partition, GPipe divides the training batch $\mathcal{B}$ into $\mathcal{M}$ micro-batches. These $\mathcal{M}$ micro-batches are executed in the $\mathcal{N}$ accelerators in a pipelined form. Unlike asynchronous PP methods, in GPipe, the gradients of each micro-batch are computed with the same model parameters in the forward pass during the back-propagation process. Finally, all of the gradients from all $\mathcal{M}$ micro-batches are accumulated and are used to update the model parameters in this mini-batch across all devices. The schedule of GPipe requires devices to store all the intermediate activation values before the execution of backward propagation. This storage could be a memory bottleneck in training big models. To address this problem, DAPPLE [21] modifies the order of backward propagation in GPipe to reduce the peak memory usage, and it is also faster than GPipe.

Absorbing the ideas from GPipe, DAPPLE, and GEMS [79], Chimera [35] proposes a bidirectional pipeline that further reduces the number of bubbles ratio in the pipeline. However, the memory cost of storing the weight matrix is twice that of GPipe and DAPPLE. Furthermore, the redundancy results in synchronization of the model weights, which increases the communication volume. However, if we treat Chimera as a PP method naturally with a DP-degree of 2 and compare it with DAPPLE with a DP-degree of 2, which also has a replica of model weights, we would find it has a lower bubble ratio, and thus more effective.

There exist bubbles in synchronous PP methods due to the synchronization before updating parameters; this may lead to insufficient utilization of devices. However, the convergence of synchronous PP is guaranteed since it is mathematically equivalent to the vanilla training process.

*3) Discussions and Comparisons of Pipeline Parallelism Methods:* In this subsection, we discuss the communication of PP and compare the intermediate activation memory, parameter weight memory, and bubble ratio of different PP methods.

*Communication volume.* All PP methods above except Chimera have the same communication volume when given specific batchsize, micro-batch size, and stage number. The communication form of PP is the same as MP. As mentioned in section IV-B, the communication amount of MP is $Bw_{out}$ for each stage (except for the last stage), where $w_{out}$ here is the column size of the last weight matrix of the stage. Suppose that batchsize of the data is $b$ In PP, and a batch is divided into $M$ micro-batch. A micro-batch, therefore, results in $Bw_{out}/M$ communication volume in a stage, and the total volume of a batch in this stage sums up to $Bw_{out}$. Suppose we now have $S$ stages and the output dimension of stage $i$ is $W_i$, the total communication volume of an iteration is $\sum_0^{S-2} BW_i$, which is the same as MP. In addition

to this communication volume, Chimera needs to all-reduces gradients between workers, which results in a communication volume equal to the size of model weights.

We have mentioned above that PP can reduce more communication volume than DP and TP since the communications in DP and TP happen in every layer while communications in PP happen only between stages. Moreover, synchronous PP methods have no redundancy tensor memory cost like DP and TP do.

*Comparison of PP methods.* We use the table in [35] to help illustrate the difference of PP methods, where $M_w$ represents for the memory of parameter weights of the model, $S$ represents for the number of stages, $N$ represents for the number of micro-batches. As Tab. VI shows, the asynchronous pipeline has the lowest bubble ratio that is approximate $0$ but may cost redundancy in weights memory, and it may harm the convergence. The synchronous pipeline has a higher bubble ratio than the asynchronous pipeline but guarantees convergence. DAPPLE highly optimizes the memory peak in GPipe, and this is very helpful in training large-scale models. Chimera further reduces the bubble ratio and the lower bound of activation memory and includes implicit data parallelism in the pipeline.

### D. Hybrid Parallelism

Hybrid parallelism uses a combination of data, model, or pipeline parallelism to partition the model in a fine-grained way. OWT [71] (one weird trick) is the most classic hybrid parallelism scheme. It is a heuristic convolution neural network (CNN) training method that uses data parallelism for CNN and model parallelism for a fully connected network (FC). Other representative work includes Megatron-LM [18] and 3D-parallelism [19] from DeepSpeed.

In Megatron-LM, domain experts manually partition the large-scale model like GPT-2 using Row-TP and Column-TP and apply a 1F1B PipeDream pipeline to improve throughput. Although Megatron-LM has an excellent training speed, it is hard for non-experts to reproduce the code or manually apply its thought to other models since it is designed specifically.

Like GPipe, DeepSpeed automatically partitions models into pipeline stages effortlessly and then uses ZeRO-DP on each stage to enlarge the throughput. However, this kind of partition is inter-layer-wise, which does not consider communication the computing of each stage well; and it can only handle sequential models. Moreover, DeepSpeed proposes 3D parallelism, a new hybrid scheme of large-scale training. It involves DP, TP, and PP at the same time. The details of 3D parallelism can be found in Fig. 8. Assuming there are three axes, $x$ axis, $y$, and $z$ axis, these three axes represent PP, TP, and DP, respectively.

### E. Other Methods

*1) Check-pointing:* Also known as recomputation, check-pointing [80], [81] 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

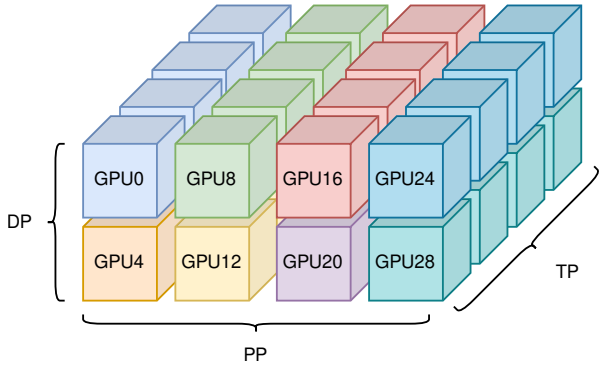



Fig. 8. The over view of 3D Parallelism, including Data Parallelism, Tensor Parallelism and Pipeline Parallelism, each of which lies in a independent axis.

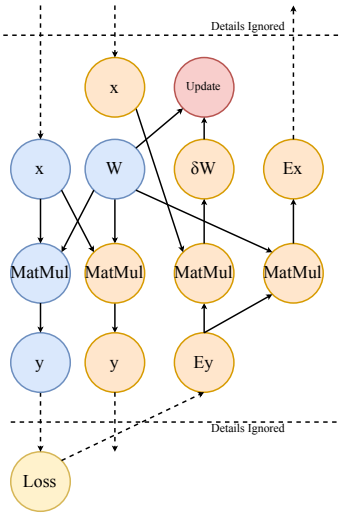



Fig. 9. Check-pointing Computation Graph

the batchsize in training a large-scale model. Check-pointing is a practical way to train neural networks, which has been adopted by frameworks like MindSpore, OneFlow, PyTorch, PaddlePaddle, and TensorFlow.

*2) Experts Parallelism:* Different from the above parallelism schemes, Expert Parallelism is specific to MoE-based models. There are several experts in the MoE layer, and each expert in the MoE layer is just a Feed-Forward Network that does matrix multiplication, which can be allocated into the same computing device. In other words, Experts Parallelism could also be reviewed as a variant of TP that partitions a weight matrix. Moreover, it could be utilized with DP and MP simultaneously.

*3) Token-level parallelism:* Token-level parallelism (TeraPipe) [39] is a variety of pipeline parallelism. Instead of feeding data in the unit of micro-batch to the pipeline, TeraPipe splits data by tokens axis (i.e., length axis) unevenly and then feeds them in the pipeline. It makes good use of the properties of Transformers [82] that longer sequences require a longer time to compute. TeraPipe is orthogonal to MP and TP, which may be helpful in training large-scale language models.

*4) Sequence parallelism:* Sequence parallelism [38] also uses the properties of Transformers. It is a ring-pipeline that

TABLE VI
COMPARISON OF DIFFERENT PP METHODS

| PP method | Bubble Ratio | Weights Memory | Activations Memory | Synchronous or not | Convergence |
|---|---|---|---|---|---|
| PipeDream [15] | $\approx 0$ | $[M_w, SM_w]$ | $[M_a, SM_a]$ | No | Unstable |
| PipeDream-2BW [73] | $\approx 0$ | $2M_w$ | $[M_a, SM_a]$ | | |
| GPipe [72] | $(S-1)/(N+S-1)$ | $M_w$ | $NM_a$ | Yes | Stable |
| DAPPLE [21] | $(S-1)/(N+S-1)$ | $M_w$ | $[M_a, SM_a]$ | | |
| Chimera [35] | $(S-2)/(2N+S-2)$ | $2M_w$ | $[(S/2+1)M_a, SM_a]$ | | |

each worker holds the same parameters and then computes different parts of the inputs. The inputs are chunked along the sequence-length axis. By transmitting the computing results of each device, it finally outputs the final complete results. Sequence parallelism can support a larger batchsize in training than TP and, in the meanwhile, has better throughput.

## V. STRATEGY SEARCHING METHODS FOR AUTO-PARALLELISM

As mentioned before, strategy searching is the key to auto-parallelism and, in the meanwhile, is an NP-hard problem. Researchers have proposed many methods [14], [15], [20]–[25], [25], [26], [30]–[32], [34]–[36], [43], [55]–[59], [73], [83]–[88] for auto-parallelism to find a near-optimal strategy. We divide existing strategy searching methods into two categories: classic-algorithm-based methods and machine-learning-based methods. Classic-algorithm based methods include recursive algorithm [89], dynamic programming algorithm [54], integer linear programming algorithm [90] as well as breath-first-search (BFS) algorithm [91]. We summarize the following analysis in Tab. VII. Machine-learning based methods include methods like Monte-Carlo Markov Chain (MCMC) [92], Monte-Carlo Tree Search (MCTS) [93] which help search strategies, and reinforcement learning [94] which help predict strategies for each operator and so on.

### A. Machine-Learning-Based Methods

*1) Reinforcement-Learning-Based Methods:* We first start with reinforcement-learning-based methods. ColocRL [24] is the first work that uses reinforcement learning to do auto-parallelism. It uses an attentional sequence-to-sequence model trained with Adam optimizer based on policy gradients computed via the REINFORCE equation [95] 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) [96] 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 outperforms an existing heuristic. Moreover, the standard policy gradient method is known to be inefficient, as it performs one gradient update for each data sample [97]. The author of ColocRL then proposes a long short-term memory (LSTM) [98] reinforcement-learning-based hierarchical device placement strategy (HDP) [84], which can support neural networks that have ten of thousands of operations. Spotlight [86] models the problem as a multi-stage Markov decision process (MDP) [99], 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 both 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 on capturing long-distance dependencies over large computation graphs. To alleviate this problem, Placeto [85] and GDP [83] use Graph Neural Network (GNN) [100] to make embedding information for nodes in computation graph $\mathcal{G}$. Placeto models problems as MDP and 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 [82] 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) [88] uses 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) [101] 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 lots 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 [28] 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 device type for each layer of Click Through Rate (CTR) 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 has higher throughput than homogeneous computing resources.

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

TAPP (Task allocation in pipeline parallelism) focuses on partitioning the model into several stages with reinforcement learning. It uses a feed-forward neural network (FFN), where the last layer is a Softmax operator that predicts the stage number for each layer. It then uses a reinforcement learning attention-based sequence-to-sequence model to predict which device a stage should be in. Auto-MAP [32] from Alibaba leverage Deep Q-Network (DQN) [102] with task-specific

pruning strategies to help efficiently explore the search space of DP, TP, or PP over XLA Higher Level Operations (HLO) with device and network interconnect topology specified. They choose HLO Intermediate Representation (IR) [103] that is produced by Accelerated Linear Algebra (XLA) [104] from TensorFlow [105] 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 graph, which fits our problem definition. Auto-MAP set rewards, states, and actions for all three DP, TP, PP to instruct DQN to search strategies. Given a cluster of 4 servers with 8 V100 GPUs, Auto-MAP can search TP strategies for 11-billion-parameter T5 [106] model within 1.5 hours, DP strategies within 17 minutes, and PP strategies within 280 seconds. However, Auto-MAP can currently automatically 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 [14]: SOAP, which represented 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 [12], FlexFlow can parallelize all kinds of 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 in the search space randomly, which results in an unacceptable time of solving the optimal solution for large-scale models. It requires 37 minutes to search strategies for NMT [107] model on 16 servers with 4 P100 GPUs. To support the partition of GNN models, the author of FlexFlow, Zhihao Jia, implements ROC [34] 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 between devices. Automap [33]from DeepMind is performed on MHLO, which is an MLIR [108] encoding of XLA HLO. It applies a Search and Learning method to annotate Megatron-like [18], [78] 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 [109] to compute 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 [63]. OptCNN can only handle TP partition. 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 [43] 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. What is more, 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 strategy for WResNet-152 [110] only takes 8.3 seconds.

Instead of coarsening, TensorOpt [26] extends the dynamic programming algorithm in OptCNN and name it as 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 needs 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 [15] 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 [73] 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 [59], which extends the dynamic programming algorithm in PipeDream to support partition for arbitrary DAG, and also proposes an integer programming solution to solve partition problem. However, like PipeDream and PipeDream-2BW, these methods do not consider tensor parallelism.

Also from project Fiddle, Piper [55] uses a two-level dynamic programming approach to search DP, TP, and PP strategies. The outer dynamic programming algorithm would generate hundreds of NP-hard knapsack sub-problems, which calculates 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 $O(|V|^2N|V_\mathcal{D}|^2)$, where $|V|$ is the number of vertices in 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 [111] on 2048 devices within only 2 hours, which is relatively a 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) [22] uses the observation that DP and TP have the same communication cost per worker, and thus only consider communication cost to do DP, TP partitions, and the combination of them. D-Rec builds its cost model statically, which asymptotically and statically analyzes communication cost based on the shape of tensor and type of operator using the formulations in Table V and IV. 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 algorithms due to its speed advantage. However, ignoring the computation analysis limits D-Rec from supporting heterogeneous clusters and PP in the future.

PaSE [56] also uses a static analysis-based cost model to generate DP, TP strategies, and a combination of them. It makes good use of the sparsity of computing graph to form a DP-based strategy searching algorithm, whose overall computational complexity of is $O(|V|^2K^{M+1})$, where $|V|$ is the total number of $v_i \in V$, $K$ is the maximum number configurations of an operator (vertex), and $M$ is the size of the largest dependent set (i.e., difference set of computing sub-graphs from two iterations). According to their experiment, PaSE can generate strategies for a Transformer NMT model on 16 devices and 64 devices in 2.2 minutes and 31.4 minutes, respectively. The throughput of generated strategies outperforms Mesh-Tensorflow [69]. PaSE can be applied on the heterogeneous cluster, but currently, it does not include heterogeneity in the cost model, which may result in unbalanced partitions. What is more, PaSE is not good at handling $\mathcal{G}$ that $|E|$ is tremendous, as the $M$ may be significantly large. The runtime overhead of solving strategies of models like DenseNET [112] is unacceptable since their computing graphs are uniformly dense. Both D-Rec and PaSE uses static analysis-based cost model to generate strategies. However, an asymptotic analysis may not be accurate enough as profiling does, which may result in some performance deterioration. Because static analysis usually can not capture some low-level details like cache effects and overlapping of computation and communication, which may be necessary for analyzing execution time.

AccPar [23] analyzes the intra-layer and inter-layer communication cost for all situations between DP and 1D-TP and does DP and 1D-TP partitions to the model. It simplifies the DAG partition problem by deciding 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 OWT and HyPar [44], which only do DP and Row-TP on homogeneous clusters.

DistIR [58] is an efficient IR for explicit representation of distributed DNN computation. It uses a linear regression model to simulate the cost of operators (e.g., MatMul and AllReduce), and 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 [36] uses a two-level hierarchical planning algorithm to search strategies and is the first auto-parallelism method that supports DP, 1D-TP, 1F1B-PP as well as ZeRO-DP. Alpa works on arbitrary DAG. It formalizes the intra-operator parallelism problem as integer linear programming (ILP) and formalizes the inter-operator parallelism as dynamic programming. The dynamic programming algorithm is built on top of that in TeraPipe [39], but additionally consider device mesh slicing. During the dynamic programming calculation for finding the best inter-operator parallelism strategy, Alpa uses ILP to find the best DP and TP 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 tries to use early pruning to reduce search space and use 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 strategies for GPT-39B on 2048 GPU devices may require thousands of hours, which shows poor scalability. But Alpa is fair enough to search 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 deep learning framework will automatically propagate strategies based on set rules. GSPMD [29] is the first work to do this. It proposes sharding propagation, and the corresponding algorithm has been integrated into TensorFlow's XLA compiler [105]. 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 [57], OneFlow [31] and PaddlePaddle [87] absorb sharding propagation and create their own 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. Conclusions and Discussions

Large-scale models are becoming increasingly important in industry and academia and also greatly improve the development of scalable distributed training systems that involves the auto-parallelism method. In this survey, we took a deeper look into distributed training from the perspective of auto-parallelism. We investigate the main challenges to make auto-parallelism methods more practical and have reviewed existing methods that tackle those challenges. We give a detailed analysis on the foundations of auto-parallelism, including the problem definition and parallelism strategies. Finally, we provide an overview and analyze the existing auto-parallelism methods.

Looking into the future, we suggest a few trends that may be important in the following years, which are acceleration of strategy searching, optimization of founded strategies, and combinations of more parallelism schemes.

### A. Accelerating Strategy Searching

*1) Grouping:* There are two ways in grouping to accelerate searching. The first way is to apply the same partitions on modules with the same architectures. The second way is to group some operators to form a layer and apply partitions to it.

The first way of grouping is based on the fact that many neural networks have regular structures. For example, ResNet consists of many residual blocks, and BERT consists of many transformer layers. Researchers have found that using the regularity of models can help us accelerate the auto partition of computation graphs because modules with the same architecture often have the same parallelism strategies. Thus, we only need to find a strategy for a layer and then broadcast it to the others, which can reduce the time to a sub-linear degree.

The second way is designed for models that do not have similar architecture, but it can also be used together with the first way. By grouping operators in the second way, we could transfer a non-linear computation graph into a linear one [43], and thus extend the usability of some algorithms like OptCNN. Moreover, we can accelerate searching by simultaneously deciding strategies for several operators (e.g., MatMul and succeeding ReLU).

We recommend using grouping as a heuristic method to help reducing the runtime of auto-parallelism methods. We could control the size of groups and the method to generate groups to explore the influence and effectiveness that grouping brings us.

*2) Profiling-based Cost Model:* As mentioned in section III, although using the symbolic cost model is very fast in evaluating strategies, it owns the inability of telling the difference between different devices, and it ignores many optimization strategies like cache and the overlap between computation and communication. Furthermore, profiling is too time-costly to evaluate every strategy for a large-scale model.

We recommend using a profiling-based cost model, which holds the actual runtime of an operation on a specific device and can be further fine-tuned to gain better performance (e.g., applying a linear regression model).

*3) Using Heuristics:* Heuristics help reduce the search space while keeping a well enough output. For example, Alpa uses early pruning to ignore strategies with costs over the threshold; Piper uses greedy heuristics to solve the knapsack problem.

### B. Optimizing Parallelism Strategies

Given a specific device topology, an auto-parallelism method should optimize the parallelism strategies by organizing computation among devices and designing good communication pace and pattern.

*1) Topology-aware Computation:* Only a few existing auto-parallelism methods handle topology-aware computation, especially on heterogeneous clusters. AccPar distributes computation tasks according to devices' computation capacity; DeepSpeed and PaddlePaddle let the CPU participate in part of the computation to alleviate the pressure of the GPU. Although many DL training is deployed in a homogeneous cluster, we suggest developing auto-parallelism methods that support heterogeneous partitions.

*2) Topology-aware Communication:* Auto-parallelism strategy searching methods need to consider topology-aware communication strategies to reduce communication time further and increase throughput. BytePS proposes that using more CPU as parameter servers can reduce the communication amount of synchronizing parameters. However, most of the current auto-parallelism methods fail to be aware of this possible option. [47] and [48] propose ways to reduce intra-node and inter-node communication. We suggest involving their work in generating new strategies.

### C. Supporting more Parallelism Schemes

Emerging methods including multidimensional TP [16], [17], [41], TeraPipe [39] and sequence-level parallelism [38] as well as ZeRO [37] can bring huge enhancement in training large-scale models. However, almost no auto-parallelism methods consider the above strategies in their implementation. We expect new algorithms that make the most of these emerging parallelism methods.

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

| Name | Supported Scheme | Detail | Evaluation Method | Scheduling Time |
|---|---|---|---|---|
| ColocRL [24] | MP | Training RNN RL | Profiling | NMT: 27 hours on 4 K80 GPUs |
| HDP [84] | | Training LSTM RL | | NMT: 3 hours on 8 K40 GPUs |
| GDP [83] | | Transformer RL. PreTrain and Finetune | | NMT: 7.35x faster than HDP |
| Spotlight [86] | DP+MP | Training LSTM+Attention RL | | CNN: 9 hours on 40 K80 GPUs |
| Placeto [85] | | MDP & Graph Embedding | Profiling-based cost model | NMT: 49 hours |
| REGAL [88] | MP | BRKGA & GNN & RL | | Graphs whose $\|V\| < 1000$: seconds |
| HeterPS [28] | DP+PP | LSTM RL | | CTR model: 20 Seconds on 8 V100 |
| FlexFlow [20] | | MCMC | | NMT: 0.6 hour on 64 K80 GPUs |
| Auto-MAP [32] | DP or TP or PP | DQN with pruning | | Bert-48: 262 seconds on 32*V100 |
| Automap [33] | DP+TP | MCTS & interaction Network | Cost Model | A few minutes |
| Pesto [113] | MP | ILP | Profiling-based cost model | NMT: 51 minutes on 2 V100 GPUs |
| vPipe [114] | PP | Dynamic Programming (KL) | Profiling | $O(\|V\|^2 \log \|V\|)$ |
| PipeDream [15] | DP+PP | Dynamic Programming | Profiling-based cost model | $\sum_{k=1}^{L} O(\|V\|^3 m_k^2)$ |
| RaNNC [25] | | Dynamic Programming | Profiling | Not Given |
| Chimera [35] | | Grid-Search | Profiling-based cost model | $\sum_{k=1}^{L} O(\|V\|^3 m_k^2)$ |
| DAPPLE [21] | | Dynamic Programming | Profiling-based cost model | Not Given |
| DNN-Partitioning [59] | | Dynamic Programming+ILP | | $O(\mathcal{I}^2(\|V_{\mathcal{D}}^{gpu}\|\|V_{\mathcal{D}}^{cpu}\| + \|V\| + \|E\|))$ |
| OptCNN [14] | DP+TP | Dynamic Programming (Graph Elimination and Regeneration) | | $O(\|V\|K^3)$ |
| Tofu [43] | | | Symbolic cost model | $O(\|V\|K^3)$ |
| TensorOpt [26] | | | | $O(\|V\|^2 K^3 log(K)(log(\|V\|) + log(K)))$ |
| D-Rec [22] | | Double Recursive Programming | | $O(\|V\|)$ |
| AccPar [23] | | Dynamic Programming | | $O(\|V\|)$ |
| PaSE [56] | | Dynamic Programming with GenerateSeq | | $O(\|V\|^2 K^{M+1})$ |
| GSPMD [29] | DP+TP+PP | Sharding Propagation | None | $O(\|V\|)$ |
| Neo [30] | | Greedy+Karmarker-karp algorithm | Symbolic cost model | Not Given |
| Alpa [36] | | ILP+Dynamic Programming | Profiling-based cost model | $O(\|V\|^5 \|V_{\mathcal{D}}\|(\|V_{\mathcal{D}}\|/d + \log d)^2)$ |
| DistIR [58] | | Grid-Search | | Not Given |
| Piper [55] | | 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.

## REFERENCES


[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz *et al.*, "The kaldi speech recognition toolkit," in *IEEE 2011 workshop on automatic speech recognition and understanding*, no. CONF. IEEE Signal Processing Society, 2011.

[3] M. Johnson, M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viégas, M. Wattenberg, and G. Corrado, "Google's multilingual neural machine translation system: Enabling zero-shot translation," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 339–351, 2017.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[5] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, Conference Proceedings, p. 173–182.

[6] BAAI, "Release of wudao2.0," 2021, https://2021.baai.ac.cn/schedule.

[7] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *arXiv preprint arXiv:2101.03961*, 2021.

[8] J. Lin, A. Yang, J. Bai, C. Zhou, L. Jiang, X. Jia, A. Wang, J. Zhang, Y. Li, W. Lin, J. Zhou, and H. Yang, "M6-10t: A sharing-delinking paradigm for efficient multi-trillion parameter pretraining," 2021. [Online]. Available: https://arxiv.org/abs/2110.03888

[9] S. Yuan, H. Zhao, Z. Du, M. Ding, X. Liu, and Z. Yang, "Wudaocorpora: A super large-scale chinese corpora for pre-training language models," *Preprint*, 2021.

[10] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, and N. Nabeshima, "The pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[11] R. Mayer and H.-A. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–37, 2020.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[13] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, and P. Tucker, "Large scale distributed deep networks," 2012.

[14] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in accelerating convolutional neural networks," in *International Conference on Machine Learning*. PMLR, Conference Proceedings, pp. 2274–2283.

[15] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, Conference Proceedings, pp. 1–15.

[16] Q. Xu, S. Li, C. Gong, and Y. You, "An efficient 2d method for training super-large deep learning models," *CoRR*, vol. abs/2104.05343, 2021. [Online]. Available: https://arxiv.org/abs/2104.05343

[17] Z. Bian, Q. Xu, B. Wang, and Y. You, "Maximizing parallelism in distributed training for huge neural networks," *CoRR*, vol. abs/2105.14450, 2021. [Online]. Available: https://arxiv.org/abs/2105.14450

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

[19] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Conference Proceedings, pp. 3505–3506.

[20] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Conference Proceedings, pp. 553–564.

[21] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, and L. Xia, "Dapple: A pipelined data parallel approach for training large models," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Conference Proceedings, pp. 431–445.

[22] H. Wang, "Freeing hybrid distributed ai training configuration," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1620–1624.

[23] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Accpar: Tensor partitioning for heterogeneous deep learning accelerators," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Conference Proceedings, pp. 342–355.

[24] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," *CoRR*, vol. abs/1706.04972, 2017. [Online]. Available: http://arxiv.org/abs/1706.04972

[25] M. Tanaka, K. Taura, T. Hanawa, and K. Torisawa, "Automatic graph partitioning for very large-scale deep learning," *arXiv preprint arXiv:2103.16063*, 2021.

[26] Z. Cai, X. Yan, K. Ma, Y. Wu, Y. Huang, J. Cheng, T. Su, and F. Yu, "Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[27] J. H. Park, G. Yun, C. M. Yi, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y. ri Choi, "HetPipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 307–321. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/park

[28] J. Liu, Z. Wu, D. Yu, Y. Ma, D. Feng, M. Zhang, X. Wu, X. Yao, and D. Dou, "Heterps: Distributed deep learning with reinforcement learning based scheduling in heterogeneous environments," *CoRR*, vol. abs/2111.10635, 2021. [Online]. Available: https://arxiv.org/abs/2111.10635

[29] Y. Xu, H. Lee, D. Chen, B. Hechtman, Y. Huang, R. Joshi, M. Krikun, D. Lepikhin, A. Ly, M. Maggioni *et al.*, "Gspmd: General and scalable parallelization for ml computation graphs," *arXiv preprint arXiv:2105.04663*, 2021.

[30] D. Mudigere, Y. Hao, J. Huang, A. Tulloch, and S. Sridharan, "High-performance, distributed training of large-scale deep learning recommendation models," *CoRR*, vol. abs/2104.05158, 2021. [Online]. Available: https://arxiv.org/abs/2104.05158

[31] J. Yuan, X. Li, C. Cheng, J. Liu, R. Guo, S. Cai, C. Yao, F. Yang, X. Yi, C. Wu, H. Zhang, and J. Zhao, "Oneflow: Redesign the distributed deep learning framework from scratch," *CoRR*, vol. abs/2110.15032, 2021. [Online]. Available: https://arxiv.org/abs/2110.15032

[32] S. Wang, Y. Rong, S. Fan, Z. Zheng, L. Diao, G. Long, J. Yang, X. Liu, and W. Lin, "Auto-map: A DQN framework for exploring distributed execution plans for DNN workloads," *CoRR*, vol. abs/2007.04069, 2020. [Online]. Available: https://arxiv.org/abs/2007.04069

[33] M. Schaarschmidt, D. Grewe, D. Vytiniotis, A. Paszke, G. S. Schmid, T. Norman, J. Molloy, J. Godwin, N. A. Rink, V. Nair, and D. Belov, "Automap: Towards ergonomic automated parallelism for ML models," *CoRR*, vol. abs/2112.02958, 2021. [Online]. Available: https://arxiv.org/abs/2112.02958

[34] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with roc," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 187–198, 2020.

[35] S. Li and T. Hoefler, "Chimera: Efficiently training large-scale neural networks with bidirectional pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476145

[36] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," 2022.

[37] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Conference Proceedings, pp. 1–16.

[38] S. Li, F. Xue, Y. Li, and Y. You, "Sequence parallelism: Making 4d parallelism possible," *arXiv preprint arXiv:2105.13120*, 2021.

[39] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica, "Terapipe: Token-level pipeline parallelism for training large-scale language models," *arXiv preprint arXiv:2102.07988*, 2021.

[40] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," *arXiv preprint arXiv:2006.16668*, 2020.

[41] B. Wang, Q. Xu, Z. Bian, and Y. You, "2.5-dimensional distributed model training," *CoRR*, vol. abs/2105.14500, 2021. [Online]. Available: https://arxiv.org/abs/2105.14500

[42] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–33, 2020.

[43] M. Wang, C.-c. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.

[44] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Hypar: Towards hybrid parallelism for deep learning accelerator array," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 56–68.

[45] Y. Ueno and R. Yokota, "Exhaustive study of hierarchical allreduce patterns for large messages between gpus," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2019, pp. 430–439.

[46] M. Cho, U. Finkler, D. Kung, and H. Hunter, "Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 241–251, 2019.

[47] N. Xie, T. Norman, D. Grewe, and D. Vytiniotis, "Synthesizing optimal parallelism placement and reduction strategies on hierarchical systems for deep learning," *CoRR*, vol. abs/2110.10548, 2021. [Online]. Available: https://arxiv.org/abs/2110.10548

[48] N. A. Rink, A. Paszke, D. Vytiniotis, and G. S. Schmid, "Memory-efficient array redistribution through portable collective communication," 2021.

[49] K. Kennedy and U. Kremer, "Automatic data layout for distributed-memory machines," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 4, pp. 869–916, 1998.

[50] J. L. M. Chen and J. Li, "Index domain alignment: Minimizing cost of cross-referencing between distributed arrays," 1989.

[51] U. Kremer, "Np-completeness of dynamic remapping," in *Proceedings of the Fourth Workshop on Compilers for Parallel Computers, Delft, The Netherlands*, 1993.

[52] J. Li and M. Chen, "The data alignment phase in compiling programs for distributed-memory machines," *Journal of parallel and distributed computing*, vol. 13, no. 2, pp. 213–221, 1991.

[53] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of machine learning*. MIT press, 2018.

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

[55] J. Tarnawski, D. Narayanan, and A. Phanishayee, "Piper: Multidimensional planner for dnn parallelization," in *NeurIPS 2021*, December 2021. [Online]. Available: https://www.microsoft.com/en-us/research/publication/piper-multidimensional-planner-for-dnn-parallelization/

[56] V. Elango, "Pase: Parallelization strategies for efficient dnn training," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 1025–1034.

[57] Huawei, "Mindspore," https://www.mindspore.cn/en, 2020.

[58] K. Santhanam, S. Krishna, R. Tomioka, T. Harris, and M. Zaharia, "Distir: An intermediate representation and simulator for efficient neural network distribution," *CoRR*, vol. abs/2111.05426, 2021. [Online]. Available: https://arxiv.org/abs/2111.05426

[59] J. Tarnawski, A. Phanishayee, N. R. Devanur, D. Mahajan, and F. N. Paravecino, "Efficient algorithms for device placement of DNN graph operators," *CoRR*, vol. abs/2006.16423, 2020. [Online]. Available: https://arxiv.org/abs/2006.16423

[60] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," *CoRR*, vol. abs/2006.15704, 2020. [Online]. Available: https://arxiv.org/abs/2006.15704

[61] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," *arXiv preprint arXiv:2104.07857*, 2021.

[62] J. W. Rae, S. Borgeaud, T. Cai, K. Millican, and J. H. and, "Scaling language models: Methods, analysis & insights from training gopher," *CoRR*, vol. abs/2112.11446, 2021. [Online]. Available: https://arxiv.org/abs/2112.11446

[63] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Conference Proceedings, pp. 583–598.

[64] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Conference Proceedings, pp. 463–479.

[65] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[66] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[67] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[68] A. Gibiansky, "Bringing hpc techniques to deep learning," *Baidu Research, Tech. Rep.*, 2017.

[69] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, and C. Young, "Mesh-tensorflow: Deep learning for supercomputers," *arXiv preprint arXiv:1811.02084*, 2018.

[70] Z. Bian, H. Liu, B. Wang, H. Huang, Y. Li, C. Wang, F. Cui, and Y. You, "Colossal-ai: A unified deep learning system for large-scale parallel training," *CoRR*, vol. abs/2110.14883, 2021. [Online]. Available: https://arxiv.org/abs/2110.14883

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

[72] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *arXiv preprint arXiv:1811.06965*, 2018.

[73] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," *arXiv preprint arXiv:2006.09503*, 2020.

[74] M. Assran, N. Loizou, N. Ballas, and M. G. Rabbat, "Stochastic gradient push for distributed deep learning," *CoRR*, vol. abs/1811.10792, 2018. [Online]. Available: http://arxiv.org/abs/1811.10792

[75] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," 2018.

[76] G. Nadiradze, A. Sabour, D. Alistarh, A. Sharma, I. Markov, and V. Aksenov, "Swarmsgd: Scalable decentralized sgd with local updates." *arXiv: Learning*, 2020.

[77] Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li, "Communication-efficient distributed deep learning: A comprehensive survey," 2020.

[78] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. A. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, and B. Catanzaro, "Efficient large-scale language model training on gpu clusters," *arXiv preprint arXiv:2104.04473*, 2021.

[79] A. Jain, A. A. Awan, A. M. Aljuhani, J. M. Hashmi, Q. G. Anthony, H. Subramoni, D. K. Panda, R. Machiraju, and A. Parwani, "Gems: Gpu-enabled memory-aware model-parallelism system for distributed dnn training," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[80] 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, p. 19–45, mar 2000. [Online]. Available: https://doi.org/10.1145/347837.347846

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

[82] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin.

[83] Y. Zhou, S. Roy, A. Abdolrashidi, D. L. Wong, P. C. Ma, Q. Xu, M. Zhong, H. Liu, A. Goldie, A. Mirhoseini, and J. Laudon, "GDP: generalized device placement for dataflow graphs," *CoRR*, vol. abs/1910.01578, 2019. [Online]. Available: http://arxiv.org/abs/1910.01578

[84] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *International Conference on Learning Representations*, 2018.

[85] R. Addanki, S. B. Venkatakrishnan, S. Gupta, H. Mao, and M. Alizadeh, "Placeto: Learning generalizable device placement algorithms

for distributed machine learning," *CoRR*, vol. abs/1906.08879, 2019. [Online]. Available: http://arxiv.org/abs/1906.08879

[86] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *International Conference on Machine Learning*. PMLR, 2018, pp. 1676–1684.

[87] Y. Ao, Z. Wu, D. Yu, W. Gong, Z. Kui, M. Zhang, Z. Ye, L. Shen, Y. Ma, T. Wu *et al.*, "End-to-end adaptive distributed training on paddlepaddle," *arXiv preprint arXiv:2112.02752*, 2021.

[88] A. Paliwal, F. Gimeno, V. Nair, Y. Li, M. Lubin, P. Kohli, and O. Vinyals, "Reinforced genetic algorithm learning for optimizing computation graphs," 2019.

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

[90] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[91] L. Luo, M. Wong, and W. Hwu, "An effective gpu implementation of breadth-first search," in *Design Automation Conference*, 2010.

[92] W. R. Gilks, S. Richardson, and D. Spiegelhalter, *Markov chain Monte Carlo in practice*. CRC press, 1995.

[93] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[94] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

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

[96] B. and Hammer, "Learning with recurrent neural networks," *Assembly Automation*, 1980.

[97] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[98] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 11 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[99] F. P. Miller, A. F. Vandome, and J. Mcbrewster, "Markov decision process," *Springer London*, 1985.

[100] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *CoRR*, vol. abs/1706.02216, 2017. [Online]. Available: http://arxiv.org/abs/1706.02216

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

[102] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[103] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," 2020.

[104] A. Sabne, "Xla : Compiling machine learning for peak performance," 2020.

[105] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[106] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *CoRR*, vol. abs/1910.10683, 2019. [Online]. Available: http://arxiv.org/abs/1910.10683

[107] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[108] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. A. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *CGO 2021*, 2021.

[109] P. W. Battaglia, R. Pascanu, M. Lai, D. Rezende, and K. Kavukcuoglu, "Interaction networks for learning about objects, relations and physics," 2016.

[110] S. Zagoruyko and N. Komodakis, "Wide residual networks," *CoRR*, vol. abs/1605.07146, 2016. [Online]. Available: http://arxiv.org/abs/1605.07146

[111] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://www.aclweb.org/anthology/N19-1423

[112] G. Huang, Z. Liu, and K. Q. Weinberger, "Densely connected convolutional networks," *CoRR*, vol. abs/1608.06993, 2016. [Online]. Available: http://arxiv.org/abs/1608.06993

[113] U. U. Hafeez, X. Sun, A. Gandhi, and Z. Liu, "Towards optimal placement and scheduling of dnn operations with pesto," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 39–51. [Online]. Available: https://doi.org/10.1145/3464298.3476132

[114] S. Zhao, F. Li, X. Chen, X. Guan, J. Jiang, D. Huang, Y. Qing, S. Wang, P. Wang, G. Zhang *et al.*, "v pipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel dnn training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 489–506, 2021.