AMP: Automatically Finding Model Parallel Strategies with Heterogeneity Awareness

Anonymous Author(s)
Affiliation
Address
email

Abstract

Scaling up model sizes can lead to fundamentally new capabilities in many machine learning (ML) tasks. However, training big models requires strong distributed system expertise to carefully design model-parallel execution strategies that suit with the model architectures and cluster setups. In this paper, we develop AMP, a framework to automatically derive such strategies. AMP identifies a valid space of model parallelism strategies, and efficiently searches the space for high-performed strategies, by leveraging a cost model designed to capture the heterogeneity of the model and cluster specifications. Unlike existing methods, AMP is specifically tailored to support complex models composed of uneven layers and cluster setups with more heterogeneous accelerators and bandwidth. We evaluate AMP on popular models and cluster setups from public clouds and show that: AMP returns parallel strategies that match the expert-tuned strategies on typical cluster setups. On heterogeneous clusters or models with heterogeneous architectures, AMP demonstrates up to $1.37\times$ and $1.76\times$ higher throughput than state-of-the-art model-parallel systems, respectively.

1 Introduction

Recent progress in language understanding [4, 2], multimodal learning [1], and many other ML applications can largely credit to the use of extremely big models. For example, the largest language models [2, 12] with billions of parameters are found to have fundamentally new capabilities. Due to the increased model size, training such models, however, calls for *model parallelism* execution strategies in order to place the gigantic model computation across multiple, heterogeneous accelerator devices (e.g., TPUs and GPUs), and ensure efficient distributed execution.

Different from typical *data parallelism* strategies [23, 3, 13], model parallelism considers a much larger space of parallelization techniques. In particular, when a model is too large to fit in a single device memory, we can either place the computation of different layers across different devices (a.k.a. *pipeline parallelism*), or partition the computation of particular layers and dispatch parts onto parallel devices (i.e., *tensor model parallelism (TMP)* [21]). Due to their distinct communication and memory requirements, these model parallelism dimensions have different applicable scenarios, which may change with model architectures and cluster setups. For example, TMP results in high communication volumes, hence is favored on devices connected with high bandwidth, but may become less effective with limited communication bandwidth. Designing performant execution strategies requires combining these parallelism dimensions to trade-off among memory, communication, and computation, then tuning the configurations in each parallelism dimension to align with the heterogeneity of the model and the training cluster.

Existing model-parallel training strategies are either manually designed by domain experts for one or two specific models, or automatically generated with strong assumptions on the model architecture

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

or cluster topologies. For examples, the *3D parallelism* strategy adopted in Megatron-LM [21] and DeepSpeed [19] is manually specialized for transformer-based language models, assuming a fixed cluster setup and the model having the same layer repeated [21, 19, 17]. Therefore, they can hardly generalize to models with diverse layer compositions (i.e., *heterogeneous models*), or clusters with mixed types of accelerators and network switches (i.e., *heterogeneous clusters*). This motivates the core questions in this paper: *Is it possible to automatically find the best model-parallel strategies for more heterogeneous models and clusters?*

Given an arbitrary cluster and model setup, finding highly performed model-parallel strategies is difficult for the following reasons. First, when the model has diverse types of layers, existing heuristics on deciding the assignment of layers to pipeline stages (and balance stage workloads) can be invalidated, because different layers exhibit distinct execution and communication costs. Second, evaluating a model-parallel strategy on a large cluster can be expensive, and prevents search-based methods [9] that depend on extensive real evaluations. However, existing cost models do not capture the heterogeneity in the model or cluster (e.g. [22] assumes a uniform bandwidth in the cost model). In this paper, we present solutions to these challenges. To address the layer-stage assignment problem, we develop a new dynamic programming algorithm running in polynomial time. To evaluate a large number of strategies within a limited budget, we develop a new cost model which serves as a cheaper proxy to evaluate model-parallel strategies. We name our method **AMP**. In summary, AMP makes the following contributions:

- We identify factors that influence model-parallel performance in heterogeneous settings, and define strategies based on these factors.
- We present a new cost model that handles heterogeneity in both the model and the cluster to evaluate the quality of model parallel strategies with a minimal amount of real trials.
- We develop a dynamic programming approach to handle the load imbalance issue from heterogeneous models in the pipeline layer assignment problem.
- We empirically show that existing systems output sub-optimal strategies in heterogeneous settings. In contrast, AMP achieves $1.37\times$ speedup with heterogeneous clusters, and $1.76\times$ speedup on heterogeneous model architectures compared to state-of-the-art heuristics.

2 Background and Related Work

In this section, we discuss the trade-offs of existing model-parallel strategies, and the need for composing them with the awareness of model and cluster heterogeneity.

Data-parallel (DP) strategies. DP partitions the input data batch evenly among workers, and each worker holds an entire model replica [13, 20, 7]. At each iteration, each worker computes gradients over its assigned batch; the gradients are then synchronized among workers before the next iteration. DP requires each worker to hold an entire model replica, hence cannot be directly used to train models with massive parameters.

Tensor model-parallel (TMP) strategies. TMP, proposed by Megatron-LM [21], is a popular model-parallel approach for large transformer models. In TMP, the layer weights of each two consecutive layers are partitioned row-wise (*i.e.,* input dimension) first, then column-wise (*i.e.,* output dimension) [21]. TMP removes the need for synchronizing the intermediate output of the very first layer, but requires heavy cross-device communications afterwards. TMP is normally combined with data parallelism to increase the training throughput [16].

Pipeline-parallel (PP) strategies. In PP, different layers are placed across GPUs, the training mini-batch is split into smaller micro-batches. The forward and backward computation is then pipelined across micro-batches. PP requires less communication than DP and TMP, but suffers from device idle (*i.e.,* pipeline bubbles) [8, 16]. Despite synchronous pipelining schedules such as GPipe [8], PipeDream [16] proposes an 1F1B asynchronous schedule to reduce the pipeline bubble. TeraPipe develops token-level pipelining schedules in a single training sequence for auto-regressive models [14].

Automatic partition. Prior work, *e.g.,* FlexFlow, PipeDream, and DAPPLE explore automatic computation graph partitioning over a few typical cluster device setups [9, 16, 5], such as NVIDIA DGX workstations connected with Infiband switches. For partitioning the model over multiple

devices, the aforementioned work leverages carefully designed cost models to select the desired strategy over the search space.

3D parallelism. To trade off among the scalability, memory footprint, and device utilization, DP, MP, and PP have to be used together (which is also known as *3D parallelism*). DeepSpeed carefully designs a 3D parallelism strategy to train a massive scale language model with 17 billion parameters [19]. [17] manually tunes the degrees of DP, MP, PP to scale to 3072 GPUs – both of them are *not automatic*. The closest to our work is Piper [22], which leverages a simple cost model and a dynamic programming algorithm to search over the 3D strategy space. However, Piper works only with transformer-based models, and assumes that same type of devices in the cluster are homogeneously connected with equal bandwidth, ignoring heterogeneous setups. In contrast, AMP is designed to capture the heterogeneity of the given ML model and the deployed cluster, reflected in both the search space design and the searching algorithm (see section 3).

Heterogeneity in models and clusters. Newer generations of GPU devices and Ethernet switches emerge rapidly, thus it is common nowadays that multiple types of devices are deployed together in a training cluster. Conducting distributed training over such types of heterogeneous clusters introduces extra challenges. For instance, a poor load balancing strategy can cause straggler effects easily; heterogeneous memory capacity among GPUs introduces new constraints on model and data partitioning. Apart from hardware heterogeneity, DL models also introduce another dimension of heterogeneity. Many today's popular models are composed with diverse layers with different sizes and computing characteristics, e.g, *e.g.,* typical convolution networks have their layer width growing with depth [6], which would require distinct partitioning and placement strategies for each layer during model parallelism.

3 Methods

3.1 Problem Formulation

Formally, the input of AMP contains: (i) a given model, (ii) a given cluster, and (iii) the global batch size (gbs). A deep learning model $\mathcal{W}$ is represented as a graph, *i.e.,* $\mathcal{W} = (C, V)$. C is a set of vertices where each vertex $c_i \in C, i \in \{i, \cdots, |C|\}$ represents a layer, and each edge $v_{\mathcal{W},i} = (c_{\mathcal{W},i}; c_{\mathcal{W},i+1}) \in V_{\mathcal{W}}$ represents the tensor between layer $c_{\mathcal{W},i}$ and layer $c_{\mathcal{W},i+1}$. Additionally, we denote the *execution cost* of a layer $c_{\mathcal{W},i}$ as c_i, and the *communication volume* of an edge $v_{\mathcal{W},i}$ as v_i. A cluster is represented as $\mathcal{C} = \{\mathcal{D}, B\}$ where $\mathcal{D} = \{d_i | 1 \leq i \leq |\mathcal{D}|\}$ denotes a set of devices. $B \in \mathcal{R}^{|\mathcal{D}| \times |\mathcal{D}|}$ is a symmetric matrix where each matrix element b_{ij} represents bandwidth between device d_i and d_j. The goal of AMP is to output a high throughput parallel training strategy s given the hardware and model configuration. $\mathcal{S}$ denotes the entire space of candidate strategies. We formulate this as an optimization problem:

$$s^\star = \arg\min_{s \in \mathcal{S}} f(s; \mathcal{W}, \mathcal{C}, gbs) \tag{1}$$

where $f(\cdot;\cdot)$ denotes per iteration running time as a function of the strategy s, conditioned on the given ML model, cluster, and user-specified global batch size.

3.2 Method Overview

To solve the optimization problem in equation 1, we identify factors that influence the system performance (§3.3). However, there are two major problems in solving this optimization problem with these identified factors. First is how to evaluate $f(\cdot)$. One possible solution is to launch real trials, but this is prohibitively expensive given that we possibly need to evaluate a large number of strategies. Rather we develop a cost model (§ 3.4) that can estimate $f(\cdot)$ quickly. Second, the discrete nature of the optimization problem disallows us to leverage gradient-based optimization methods. However, enumerating all strategies is slow even with the cost model. One way to efficiently optimize the objective is to exploit structure in the strategy space $\mathcal{S}$, and prune areas with no promising candidates. Using this principle, we develop a dynamic programming approach (§3.5).

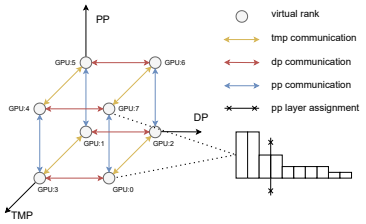

Figure 1: An example strategy with parallelism degrees = $(2, 2, 2)$. Note that there are $8!$ possible GPU assignments to these 8 virtual ranks. Assuming a uniform bandwidth, a uniform-layer assignment cuts at the 5th layer, while a more balanced assignment cuts at the 3th layer.

3.3 Representation of a Strategy

Intuitively, a strategy s defines how to train model $\{C_{\mathcal{W}}, V_{\mathcal{W}}\}$ on a given cluster $\{\mathcal{D}, B\}$ with the batch size gbs. A strategy s that composes DP, TMP, and PP strategies can be represented by a few key aspects, *e.g.,* , how many model replicas should be created (*i.e.,* how many pieces should a mini-batch be evenly split) in DP; how to partition model layers (for PP) or layer parameters (for TMP); and how shall the global batch size decompose into micro-batches in PP semantics. In AMP, we represent a strategy s with the following four aspects:

1. Parallelism degrees. Since PP, DP, and TMP have different characteristics, different degrees of each of them affect the system performance. Denote the degrees to be pp, dp, tmp. The tuple (pp, dp, tmp) defines a three-dimensional grid, which satisfies the constraint $pp \times dp \times tmp = |\mathcal{D}|; pp, dp, tmp \in \{1, \ldots, |\mathcal{D}|\}$. (*i.e.,* this tuple generates $pp \times dp \times tmp$ virtual ranks that will be bijectively mapped to available devices). Two ranks are in the same DP(TMP/PP) communication groups if they have different values in the DP(TMP/PP) dimension and the same values in others (*e.g.,* rank $(0, 0, 0)$ and rank $(0, 0, 1)$ forms a TMP group).

2. Device placement. Since physical devices may have different connection speeds, *i.e.,* bandwidth to each other, placing physical devices in different virtual ranks leads to different performance. For instance, if ranks in a communication-heavy group are all allocated with physical devices with high bandwidth, the per iteration time will be shorter Formally, We denote a device placement as a bijective function from virtual ranks to devices $p : \{1, \ldots, pp\} \times \{1, \ldots, dp\} \times \{1, \ldots, tmp\} \mapsto \mathcal{D}$.

3. Micro-batch size. Micro-batch size (mbs) is the batch size a device holds at a time. When mbs varies, devices generally have different utilization. This effect is empirically verified in [17], where the observation is that different micro-batch sizes lead to noticeably different in system throughput.

4. Pipeline layer assignment. How to assign layers to each device can also affect the system throughput. On the one hand, when activation volume v is large in the boundary, the next device needs to wait longer, resulting in a longer per iteration time. On the other hand, when a device is assigned a much higher load than other devices, it blocks all other devices and thus incurs a longer iteration time. We denote a feasible pipeline layer assignment as a function $a : C_{\mathcal{W}} \mapsto R$ that maps layers to the stage index $(1, \ldots, pp)$ in a pipeline.

In summary, a strategy s can be defined by a 6-dimensional space:

$$\mathcal{S} \triangleq (pp, dp, tmp, p, mbs, a) \tag{2}$$

3.4 Cost Model

In this section, we introduce our cost model that serves as a fast estimation of $f(\cdot)$.

Communication primitive. We consider all-reduce communication with ring topology in both DP and TMP [20]. Assume that the whole message size is M. In a cluster of n workers with uniform bandwidth B, the communication time cost of a single ring all-reduce operation can be modeled as

[11]:

$$T_{allreduce} = \frac{2(n-1)M}{nB} \tag{3}$$

In a cluster with non-uniform bandwidth, denote B' to be the slowest bandwidth in the all-reduce communicative group, we approximate the bandwidth with B'. For point-to-point communication operations between consecutive pipeline stages, we assume the communication operation can always enjoy the full bandwidth: $e = \frac{v}{b}$ where v is the activation volume and b is the bandwidth between these two GPUs.

Top-down run time decomposition. We now present our cost model designed from a *top-down* view. First, DP stands at the **highest** granularity, where there are dp pipelines running in parallel. It synchronizes gradients when all pipelines finish. At the DP level, the system runtime can be decomposed into two terms:

$$T = \max\{T_i^{pp} | 1 \leq i \leq dp\} + \max_{d \in \mathcal{D}} T_d^{dpsync} \tag{4}$$

T_i^{pp} is the time taken to complete a single pipeline i. T_d^{dpsync} denotes the time for a device d to communicate data parallelism gradient updates. The formula is given at equation 3 with message size equal to the number of model parameters held at device d. Since different pipelines may finish with different speeds, and different devices may finish its data parallelism synchronization with different speeds, we take maximum over these two terms respectively. At a **lower** granularity, each pipeline executes its own training schedule such as activation/gradients exchange at a layer level (*i.e.,* it does not need to consider how each layer performs computation, which is carried out by MP workers). Assume that a single pipeline consists of k stages. The communication time between the ith and $(i+1)^{th}$ is denoted as e_i. Execution cost for the ith stage t_i is the sum over all layer j execution time t_j^{layer} in the stage: $t_i = \sum_{a(j)=i} t_j^{layer}$.

Assuming that the P2P communication and computation can be *overlapped*, the runtime for a single pipeline can be modeled as:

$$T^{pp} = (gas - 1) \cdot \max\{t_i | 1 \leq i \leq pp\} + \sum_{i=1}^{pp-1} e_i + \sum_{i=1}^{pp} t_i \tag{5}$$

Intuitively, the first term models the straggler effect, which gets amplified by the number of gradient accumulation steps. When $gas = 1$, the schedule is straightforward to understand - the mini-batch simply performs a single forward pass and a single backward pass, sends or receives gradients/activations at the boundary. In this case, T^{pp} given at equation 5 reduces to $\sum_{i=1}^{pp-1} e_i + \sum_{i=1}^{pp} t_i$. Note that when all layers have the same execution cost and the cluster is equipped with the same bandwidth, the optimal solution is simply to balance the number of layers. This heuristics is often adopted as the default layer assignment in current systems[21].

At the **lowest** granularity, MP carries out results of individual layers: $t_j^{layer} = \frac{c_j}{tmp} + T_j^{mpsync}$. T_j^{mpsync} is the all-reduce communication timewhich can also be calculated by equation 3. One way to estimate the execution cost for a single layer c_j is by counting the number of floating-point operations (FLOPs) However, when matrix size is small, arithmetic intensity is low, and GPUs can hardly parallel efficiently. In such scenarios, matrix multiplication with few FLOPs can take a long time to execute. To handle general deep learning models, we simply obtain c_j by profiling with several tensor model parallelism settings. This profiling only needs to be run once, which is negligible compared to model training time.

3.5 Pipeline Layer Assignment

A key observation in the layer assignment problem is that only *consecutive* layers can be assigned to the same stage. Intuitively, computation up to layer i can be reused for that of layer $i+1$. To handle the max term in objective 5, an auxiliary variable tolerance **m** is introduced, where semantically m means a possible execution time for a single stage:

$$(gas - 1) \cdot \max\{0, \max\{t_i | 1 \leq i \leq pp\} - m\} + \sum_{i=1}^{pp-1} e_i + \sum_{i=1}^{pp} t_i \tag{6}$$

-	Megatron	AMP	SA
Min	1.32	**1.25**	1.57
Mean	2.41 $\pm$ 1.29	**1.55** $\pm$ **0.28**	1.82 $\pm$ 0.17

Table 1: Best and average strategies under homogeneous setup (scale: seconds)

Note that when $m = 0$, objective 6 reduces to the desired objective 5. Formally, we record the optimal assignments and the corresponding cost up to layer i with j stages and variable m in table dp[i][j][m]. Let L be the number of layers and k be the number of stages, then dp[L][k][0] stores the solution. We induct on the number of stages in the pipeline.

Base case $j = 1$**.** The only possible assignment assigns all i layers into a single stage. Denote the sum of execution time for these i layers as t_1. The associated optimal cost is thus

$$g(i) = (gas - 1) * max\{0, t_1 - m\} + t_1 \tag{7}$$

We store the table entry $dp[i][0][m]$ with $(i, g(i))$ for all possible i and m.

Recursive step. To compute $dp[i][j][m]$ when j $>$ 1, we enumerate all possible position for last cut i' for all $i' < i$. Denote the execution time from layer i' to layer i as $t_{2,i'}$, let the value stored at $dp[i'][j-1][\max(t_{2,i'}, m)]$ be $(a_{i'}, c_{i'})$. Denote the longest stage time in $a_{i'}$ as $t_{1,i'}$, the communication cost between the $(j-1)^{th}$ and j^{th} device at layer i' as $e_{i',j}$. The cost associated with the last cut at i' is: [1]

$$g(i') = c_{i'} + (gas - 1) * max\{0, t_{2,i'} - m\} + t_{2,i'} + e_{i',j} \tag{8}$$

Select the optimal last cut position $i'_{opt} = argmin_{i'<i} g(i')$. The dynamic programming table entry $dp[i][j][m]$ is updated as: $(a_{i'_{opt}} \cup (i - i'_{opt}), g(i'_{opt}))$.

Run time. There are $1 + 2 + \cdots + L = \mathcal{O}(L^2)$ possible values for m. Thus, our problem size is $\mathcal{O}(L * k * L^2)$, each subproblem takes $\mathcal{O}(L)$ time. Thus the total run time is $\mathcal{O}(kL^4)$, while a brute force solution takes O$\binom{L-1}{K-1}$.

3.6 Optimization Procedure

To find best strategies defined at § 3.3, we enumerate all possible parallelism degrees and micro-batch sizes. We deterministically solve for the optimal pipeline layer assignment using the dynamic programming approach developed at § 3.5. We adopt the heuristics at [17] to solve the device placement problem: prioritize placing model parallelism workers in the same node, over data parallelism workers, and pipeline parallelism workers. We also explore an alternative optimization for device placement at § 4.2.

4 Experiments

In this section, we evaluate our algorithm and several of its variants on different types of clusters and models. Specifically, we compare against the state-of-the-art Megatron heuristics [17].

4.1 Baseline

Megatron is a runtime system that supports 3D parallelism training on DNN models. It suggests several heuristics on how to train models with 3D parallelism faster. We summarize these heuristics as the following procedure.

Megatron heuristics loops over all possible micro-batch sizes. For each micro-batch size, it selects parallelism degrees with the smallest $tmp \times pp$, whose tmp is smaller than the number of devices in the smallest node. It solves the device placement problem by prioritizing tensor model parallelism

[1] proof can be found at the supplementary materials.

workers in the same node. If there are additional available intra-node connections, it places data parallelism workers next. It solves the pipeline layer assignment problem by either balancing the number of parameters or the number of layers in each stage.

Megatron heuristics assume a homogeneous cluster and model setup. The model is composed of simple stacks of identical layers. The cluster is equipped with the same type of devices with similar bandwidths to each other. We hypothesize that under this setup, it is (near) optimal. To validate this hypothesis, we conduct experiments on a homogeneous setup (section 4.3). Moreover, we are interested in how Megatron heuristics perform when the homogeneous assumption breaks. Specifically, we want to answer two questions. What will happen when the cluster is equipped with different types of devices and different bandwidths to each other? What is the effect of heterogeneity in models? Thus, we design two sets of experiments under heterogeneous cluster and heterogeneous model in section 4.3.

4.2 Alternative Device Placement Algorithm

Since the number of possible device placements is large and often difficult to optimize efficiently [15], we adopt Megatron heuristics in our method. To study the optimality of this heuristic, we use a Simulated Annealing(SA) algorithm to explore candidates with different placements. We name this method **SA**. Details can be found in the supplementary materials.

4.3 Experiment Setup

Homogeneous. We conduct experiments using GPT-2 ($L = 24, H = 1024$) [18] on 4 AWS EC2 g4dn.12xlarge nodes with a global batch size 32. Each instance is equipped with 4 T4 GPUs with 50 Gbps PCIe connection intra-node bandwidth, and 50 Gbps inter-node bandwidth.

Heterogeneous cluster. We ensemble a cluster using 3 AWS EC2 p3.8xlarge instances and 1 g4dn.12xlarge instance. Each p3.8xlarge instance is equipped with 4 V100 GPUs with approximately 170Gbps NVLink intra-node connection and 10 Gbps inter-node bandwidth. We evaluate performance on GPT-2 ($L = 24, H = 1024$) with a global batch size 32.

Heterogeneous model. We use a modified version of Transgan generator [10], which stacks transformer layers with gradually smaller hidden sizes. Specifically, we use 12 layers with dimensions 1024, and 12 layers with dimensions 16. We use 4 p3.8xlarge instances and a global batch size 64.

4.4 Evaluation Metric

We evaluate strategies by time per iteration. Specifically, we run each strategy by 40 iterations and take the average iteration time for the latest 20 iterations. The underlying system is Deepspeed (built on top of Megatron system) [19] with fp16 optimization enabled. [2]

4.5 Results and Discussion

Homogeneous setup. Under homogeneous setup as specified in 4.3, we hypothesize that Megatron heuristics is (near) optimal. We also expect that our method can find such optimal strategies. Additionally, we conjecture that our device placement algorithm can not propose better strategies than the default one from Megatron. In this experiment, we compare three methods:

- *Megatron*: Megatron Heuristics.
- *AMP*: our cost model that enumerates parallelism degrees and micro-batch size, and uses dynamic programming to solve the pipeline layer assignment.
- *SA*: additionally optimize device placement using the simulated Annealing algorithm.

In our experiment setting, Megatron proposes around 10 feasible strategies. To understand the difference in strategies between our proposal and the Megatron proposal, we evaluate the top 10 strategies from our method. The results are presented in Table 1. AMP finds slightly better strategies

[2] Codes are available in the supplementary materials.

-	Megatron	RS	AMP	SA
Min	1.97	1.71	**1.44**	1.46
Mean	2.62 ± 0.37	3.04 ± 0.82	2.49 ± 1.69	$\mathbf{2.29} \pm \mathbf{0.68}$

Table 2: Best and average strategies under heterogeneous cluster setup (scale: seconds).

than Megatron Heuristics. Since the model can be fit into a single GPU, Megatron heuristics selects an extreme configuration - $dp = 16$. Our method automates the strategies selection process and finds that $dp = 4$ is a better choice. By enabling our SA algorithm to explore device placement, SA finds slightly worse results. The reason is that in such a cluster setup where devices have the same connection speeds, device placements have limited influence on the system performance. For instance, whether the tensor model parallelism communication happens within a node or cross node shall not change the performance. However, it introduces more strategies that make optimization hard. This effect is illustrated in Figure 2.

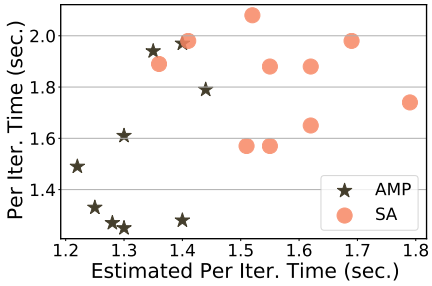

Figure 2: Predicted strategy cost versus actual cost under homogeneous setup. The predicted costs for top SA strategies are higher, leading to a higher actual cost.

Heterogeneity in cluster setup. In the setup at section 4.3, devices have different connectivity to each other, and different execution speeds. We hypothesize that some heuristics from Megatron can lead to bad performance, but our method should be able to automatically address the heterogeneity in (especially) bandwidth since we explicitly model it (3.4). With heterogeneity in the cluster, we are also interested in examining how much Megatron heuristics and our method improve compared to average strategies in the space. In this experiment, we have four methods to compare: Megatron, AMP, SA, and

- *RS*: Randomly selecting strategies without heuristics and a cost model.

We run RS for 50 iterations. Results are presented in Table 2. We found that average strategies have a high cost, but randomly evaluating them can find better strategies than Megatron. However, this comes with a much higher cost (50 vs 10 real trials), and cannot reliably produce good results. By using our cost model, we can find strategies up to $1.37\times$ speedup compared to Megatron. Specifically, the heuristics of Megatron that minimize $mp \times pp$ is sub-optimal. When using micro-batch size 1, the best strategy shall be dp=4 and pp=4, which results in a 1.44 second/iteration performance. However, since the model can be fit in a single GPU, Megatron selects dp=16, which results in 2.72 second/iteration performance. This strategy is much slower because it requires all GPUs to perform a data-parallel synchronization with large message size (the whole model parameter), while some GPUs have low bandwidth (10 Gbps). On the contrary, our cost model is aware of such a slow connection speed in the data-parallel communication group and punishes such strategies heavily.

Heterogeneity in model. When model is no longer homogeneous, layers have different execution time. Simply balancing the number of layers or parameters can not balance the workload. We expect our pipeline assignment is better than those in Megatron. In this experiment, We compare four methods:

- *Megatron*: Megatron heuristics with uniform layer assignment.

	Megatron	ours w/ parameter	ours w/ uniform	AMP	SA
Min	1.89	2.19	1.25	**1.07**	1.08
Mean	1.94 ± 0.05	2.19 ± 0.0	1.25 ± 0.0	1.42 ± 0.33	$\mathbf{1.37 \pm 0.30}$

Table 3: Best and average strategy under heterogeneous model setting (scale: seconds)

- *AMP*: our algorithm with dynamic programming layer assignment algorithm.
- *ours w/ uniform*: strategies by AMP replace with uniform layer assignment.
- *ours w/ parameter*. strategies by AMP replace with uniform parameter assignment.

Results are presented in Table 3. Our algorithm finds $1.5\times$ faster strategies compared to Megatron baseline because it does not enforce strategies to use large data parallelism degree. By enabling our pipeline layer assignment algorithm, the strategy gains an additional $1.17\times$ speedup, which gives an end-to-end $1.76\times$ speedup.

To study the relationship between heterogeneity in model and the system performance, we fix the parallelism degrees and vary only the model configuration. Results are presented in 3. Since we only use transformer layers with two different hidden size, a simple definition of heterogeneity is:

$$heterogeneity = \frac{\text{h=16 layers time}}{\text{h=1024 layers time}} \tag{9}$$

Optimization cost. The dynamic programming algorithm takes the major time in our searching algorithm. In experiments with heterogeneous models with 24 transformer layers, our method takes 140.7 seconds for profiling and 235.6 seconds for 65 iterations of the optimization procedure. In particular, 230.8 seconds are spent in the dynamic programming algorithm. The additional cost (around 5 minutes in total) introduced by our method is negligible compared to usually days to weeks of training time in real-world DNNs applications.

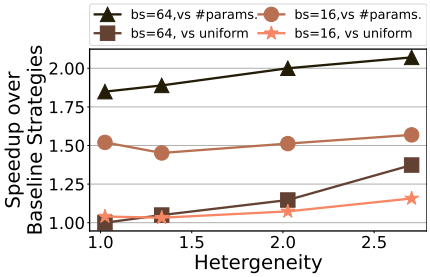

Figure 3: Speedup against heterogeneity versus two Megatron layer assignment methods. Numbers obtained when using the best parallelism strategy: mp=1, pp=4, dp=4.

Comparison with Piper. Piper planner [22] also explores the 3D parallelism space but with a different problem setup. They assume that the cluster has a uniform bandwidth. Our method attains its most speedup when the cluster has non-uniform bandwidth, which is out of the domain of Piper input. Additionally, Piper reports results on simulation, while we report numbers on real trials based on DeepSpeed, which can not support most of Piper's solutions.

Limitations. The main limitation of our method is that we do not model memory footprint. During the experiments, we discover that a small number of top strategies are out of memory (i.e. 7th and 9th of top 10 strategies in the TransGAN experiment). We plan to incorporate a memory model in our future work.

5 Conclusion

We present AMP, an automatic algorithm that can find model parallel training strategies with high system throughput. AMP is equipped with an expert-design cost model that considers cluster and model configurations. We show that when heterogeneity exists in cluster or model setup, current heuristics are sub-optimal. Our automatic algorithm can find strategies with $1.37\times$ and $1.76\times$ speedup within similar budgets.

References

[1] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millican, Malcolm Reynolds, et al. Flamingo: a visual language model for few-shot learning. *arXiv preprint arXiv:2204.14198*, 2022.

[2] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[3] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the eleventh european conference on computer systems*, pages 1–16, 2016.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[5] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 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*, pages 431–445, 2021.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[7] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[8] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32:103–112, 2019.

[9] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *SysML 2019*, 2019.

[10] Yifan Jiang, Shiyu Chang, and Zhangyang Wang. Transgan: Two pure transformers can make one strong gan, and that can scale up. *Advances in Neural Information Processing Systems*, 34, 2021.

[11] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong 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)*, pages 463–479, 2020.

[12] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[13] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[14] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. *arXiv preprint arXiv:2102.07988*, 2021.

[15] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*, pages 2430–2439. PMLR, 2017.

[16] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[17] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[18] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[19] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong 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*, pages 3505–3506, 2020.

[20] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[21] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[22] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. *Advances in Neural Information Processing Systems*, 34, 2021.

[23] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, 2017.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [No]
   (c) Did you discuss any potential negative societal impacts of your work? [No]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Codes and how to reproduce results are included in the supplementary materials.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] We do not have randomness in our main experiments.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] We include cluster setup in § 4.3, and total time used in § 4.5
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We cite the underlying system Deepspeed in § 4.4
   (b) Did you mention the license of the assets? [No]
   (c) Did you include any new assets either in the supplemental material or as a URL? [No] No new assets are introduced.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [No] We focus on system performance under different setups, which does not need to use actual data.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [No] Since we focus on the system performance, we do not release model weights.

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]