Automatic 3d-parallelism distributed training

Dacheng Li
Carnegie Mellon University

Hao Zhang
UC Berkeley, Petuum, Inc.

Hongyi Wang
Carnegie Mellon University

Eric P. Xing
MBZUAI, Petuum, Inc., Carnegie Mellon University

Abstract

Nowadays, data, model, and pipeline parallelisms are combined to achieve scalable distributed training of massive scale models, which is also known as "*3D parallelism*". 3D parallelism has been widely deployed in existing distributed machine learning systems, *e.g.,* DeepSpeed and Megatron-LM and has attained promising training scalability. However, designing and deploying a good 3D parallelism strategy (*i.e.,* how to allocate degrees of each parallelism dimension and how to conduct device placement over a heterogeneous set of computing devices) leverages heavily on domain expertise of distributed training and high-performance computing. In this work, we take a step toward developing an automatic 3D parallelism strategy such that given an arbitrary set of computing devices (*e.g.,* homogeneous or heterogeneous). Our method leverages a carefully designed cost model that takes both properties of the machine learning task and device setup into consideration to conduct randomized searching of the desired solutions. Our results indicate that, on homogeneous distributed clusters, our method returns parallel strategy attains comparable throughput compared to the expert-tuned strategies adopted in Megatron-LM. On heterogeneous distributed clusters, our method leads to up to $1.5\times$ higher throughput compared to Megatron-LM.

1 Introduction

3D parallelism combines *data parallelism*, *model parallelism*, and *pipeline parallelism* to attain high scalability and throughput, especially for massive scale language model training [1, 2, 3]. The reason behind its promising scalability is that 3D parallelism takes advantage of each individual parallel strategy for better trade-offs among memory, communication, and computation. However, designing a good 3D parallelism strategy to achieve the optimal system throughput requires a lot of domain expertise. The default 3D parallelism strategy adopted by existing distributed training systems, *e.g.,* DeepSpeed and Megatron-LM make strong homogeneous assumptions on the clusters and models being used and often adopt simple strategies for load balancing [1, 2, 3]. However, we observed that those assumptions do not generally hold, especially when deploying over heterogeneous models on heterogeneous clusters (*e.g.,* mixture of different types of computing nodes, model architecture that is not a simple stack of a building block). In this paper, we try to answer the following question:*Is it possible to automatically find the best 3D training strategy given arbitrary cluster and model setup?*

We take a step toward tackling the aforementioned problem by designing a randomized search algorithm guided by a carefully designed cost model. Our cost model captures the properties of both the user-specified machine learning task and hyper-parameters (*e.g.,* model architecture, input dimension, batch size, and etc) as well as given device setups (*e.g.,* system bandwidths among workers). Our preliminary experimental results indicate that on heterogeneous cluster setup, our method finds strategies that attain comparable throughput compared to the expert-designed strategy adopted by Megatron-LM [3]. On heterogeneous clusters, our method attains up to $1.5\times$ higher

Pre-print, work in progress.

throughput compared to the best strategy found by Megatron-LM on GPT-2 pre-training task on the OpenWebText dataset [4].

2 Related work

In this section, we discuss other research efforts in scalable distributed training for large-scale machine learning tasks.

Data-parallel (DP) distributed model training. Data parallelism is the most widely adopted scheme for conducting distributed model training [5, 6]. *parameter server* (PS) [6, 7, 8, 9] and *all-reduce* [10, 11] are the most two common realization instances for data-parallel distributed training. Data-parallel distributed training attains promising scalability, however, each worker is required to hold the entire model replica. Therefore, for training models with a massive number of parameters, data parallelism has to be combined with model parallelism and pipeline parallelism [2, 1, 3].

Model-parallel (MP) distributed model training. DistBelief [5] proposed to shard the model parameters over various PSs, however, compute nodes are still required to compute the full stochastic gradient updates. Mesh-TensorFlow [12] proposed a programming language for easily specifying parallelism strategies that combine both data and model parallelism.

Pipeline-parallel (PP) distributed model training. Pipeline parallelism is another parallel model training scheme that attains several advantages over DP and MP, *e.g.,* the amount of communication required by PP is less compared to DP and MP. GPipe deploys the vanilla pipeline parallelism schedule via splitting the mini-batch of input samples into micro-batches for smaller pipeline bubble [13]. PipeDream further combines pipeline parallelism and data parallelism to reduce the communication cost among workers [14]. TeraPipe exposes fine-grained pipelining strategies across tokens in a single training sequence for auto-regressive models [15]. Pipeline parallelism also attains tolerance to weight/gradient staleness: PipeDream-2BW maintains two weight versions and guarantees 1-stale weight updates without expensive flushes [16] while PipeMare leverages asynchronous pipeline parallelism [17]. With introducing weight/gradient staleness, pipeline parallelism can attain higher throughput while potentially suffering from slower convergence (or even divergence). Another limitation of pipeline parallelism is that it can only scale to the number of workers, which is equal to the number of model layers. To scale to a larger cluster, PP has to be combined with DP and MP.

Automatic partition. Prior work, *e.g.,* FlexFlow, PipeDream, and DAPPLE explore automatic computation graph partition over a given devices map [18, 14, 19]. 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 attain good trade-offs among the scalability, memory footprint, and GPU utilization, DP, MP, and PP have to be used together (which is also known as *3D parallelism*). DeepSpeed leverages a carefully designed 3D parallelism strategy to train a massive scale language model with 17 billion parameters [2]. [3] leverages carefully explored heuristics to tune the degrees of DP, MP, PP such that the “tuned-by-hand" strategy scales up to 3072 GPUs. However, neither [2] nor [3] explore the possibility to find the optimal 3D parallel strategy automatically based on cost models. The work that bears the most similarity to our work is Piper [20], which leverages a simple cost model and a dynamic programming algorithm to search over the 3D parallel search space. In contrast, our approach leverages a more carefully designed, fine-grained cost model, which captures both properties of the given ML task (*e.g.,* GPT-2 training) and the deployed cluster (*e.g.,* DGX-2 boxes). Our cost model can capture a more representative search space while handling task and hardware heterogeneity.

3 Background: Distributed Model Training

As the scale of modern data and deep learning models grow massively, distributed training has became a popular and default paradigm for training large-scale models [5]. Three popular paradigm for parallel distributed machine learning are data parallelism, model parallelism, and pipeline parallelism [5, 1, 13, 14, 17, 15]. In this section, we walk through each parallelism strategy, discussing their pros and cons.

3.1 Data parallelism

Data parallelism partitions the input data batch evenly among workers, and each worker holds a entire model replica [6, 10]. For each iteration, workers computed gradients over the assigned sub-batch of data sample. The gradients are then synchronized among workers before entering the next iteration. Data parallelism splits the computation costs, but introduces the communication cost of one `all-reduce` over the gradients for each iteration. To handle large models, *e.g.,* GPT-3 [21] data parallelism is usually used in combination with model parallelism and pipeline parallelism where data parallelism can be used on smaller model shards.

3.2 Model parallelism

In model parallelism, individual layers of the model parameters are sharded over workers. One popular and specific approach to shard model layer parameters is proposed by NVIDIA Megatron-LM [1]. In this paper, we will follow the layer parameter sharding strategy of Megatron. Specifically, for a transformer encoder, it consists of a self-attention layer followed by a two-layer fully connected (FC) network [22].

The weights of a two-layer FC network can be represented by $\{W^{(1)}, W^{(2)}\}$, and the forward computation is:

$$Y = \text{GeLU}(XW^{(1)}); Z = \text{Dropout}(YW^{(2)}).$$

where Y, Z are the intemediate outputs of the first and second FC layers respectively. Megatron splits W_2 along its columns, *i.e.,* $W^{(1)} = [W_1^{(1)}, W_2^{(1)}]$ as it allows the GeLU activation function to be independently applied to the output of each partitioned intermediate output of the first FC layer, *i.e.,*

$$[Y_1, Y_2] = [\text{GeLU}(XW_1^{(1)}), \text{GeLU}(XW_2^{(1)})]$$

A noticeable advantage of this model sharding strategy is that it removes the need for synchronizing the intermediate output of the very first FC layer (for forward propagation of the second FC layer).

The rows of the second FC layer weight B can then be split, *i.e.,*

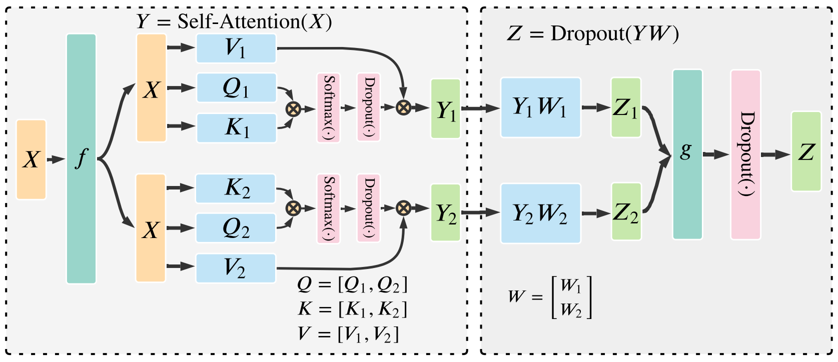
$$B = \begin{bmatrix} W_1^{(2)} \\ W_2^{(2)} \end{bmatrix}, Y = [Y_1, Y_2].$$

The output of the second FC layer is then reduced across the workers (GPUs) before the dropout operation.

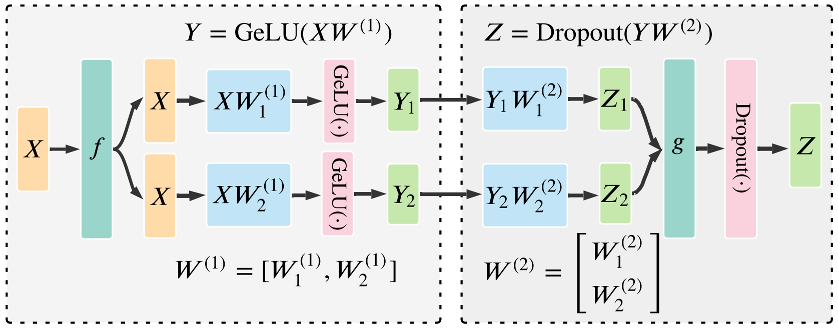
As in multi-head attention, computation among the attention heads are independent, the key (K), query (Q), and value V weights can be partitioned in a column-parallel fashion. And the output linear layer can then directly operate on the sharded output of the attention. It is noticed in Megatron that the aforementioned approach splits the matrix multiplication in a Transformer encoder block across multiple workers while requiring only two `all-reduce` operations in the forward propagation and two `all-reduce` operations in the backward propagation.

3.3 Pipeline parallelism

In pipeline parallelism, the stack of layers are sharded across workers. The mini-batch used in the first-order optimizers *e.g.,* SGD, Adam, and etc is further split into smaller micro-batches. The computation (*i.e.,* forward propagation and backward propagation) is then pipelined across micro-batches. Pipelining schemes have to schedule the order of computation and communication operations so that input samples can be fed with network layers with consistent weight versions. To make sure pipeline parallelism leads to the exact same computation to single machine training, periodic pipeline flushes are introduced so that optimizer steps are synchronized across workers. At the start and end of every batch, devices are idle. And the idle time is called the *pipeline bubble*. In general, a good design of pipeline parallelism should make the pipeline bubble as small as possible.



(a) Attention block sharding



$Y = \text{GeLU}(XW^{(1)})$
$Z = \text{Dropout}(YW^{(2)})$
X
f
$XW_1^{(1)}$
$XW_2^{(1)}$
GeLU(·)
Y_1
Y_2
$Y_1W_1^{(2)}$
$Y_2W_2^{(2)}$
Z_1
Z_2
g
Dropout(·)
Z
$W^{(1)} = [W_1^{(1)}, W_2^{(1)}]$
$W^{(2)} = \begin{bmatrix} W_1^{(2)} \\ W_2^{(2)} \end{bmatrix}$

(b) FC layers sharding

Figure 1: Model parameter sharding strategy introduced in Megatron-LM: (a) attention block; (b) FC layers.

There are several possible design of pipeline parallelism schemes. An vanilla pipelining scheme is proposed by GPipe [13]. It was proposed in GPipe that the forward passes for all micro-batches in a mini-batch are first executed, followed by backward passes for all micro-batches. One can quantify the size of pipeline bubble of GPipe (denoted as t_{pb} in this paper). Follow the notation of [3]. We denote the number of micro-batches in a batch as m, the number of pipeline stages (number of devices used for pipeline parallelism) as p, the ideal time per iteration as t_{id} (assuming perfect or ideal scaling), and the time of forward and backward pass to execute a single as t_f and t_b. In this schedule, the pipeline bubble consists of $p-1$ forward passes at the beginning of a batch, and $p-1$ backward passes at the end. The total amount of time spent in the pipeline bubble is then $t_{bp} = (p-1) \cdot (t_f + t_b)$. The ideal processing time for the batch is $t_{id} = m \cdot (t_f + t_b)$. Therefore, the fraction of ideal computation time spent in the pipeline bubble is:

$$\frac{t_{pb}}{t_{id}} = \frac{p-1}{m}.$$

Thus, we need $m \gg p$ to make sure the bubble time fraction to be small. However, for such large m, this approach has a high memory footprint as it requires stashed intermediate activation scores (or just input activation scores for each pipeline stage when using activation re-computation) to be kept in memory for all m micro-batches through the lifetime of a training iteration.

To reduce the size of the pipeline bubble, each worker can conduct forward and backward propagation over a subset of mode layers (*i.e.,* layer chunk). The aforementioned GPipe pielining schedule conducts "*all-forward, all-backward*" schedule. However, such pipelining strategy suffers from high memory footprint, which is propotional to the number of micro-batches m. Thus, we use the 1F1B schedule following the pipeline parallelism implementations in Megatron-LM and Deepspeed [1, 2].

4 Methods

4.1 Search Space - Candidate Representation

A training strategy can be represented by:

1. parallelism degrees: pp, dp, mp.
2. device placement.
3. micro-batch size in pipeline parallel strategy.
4. pipeline layer assignment: which set of layers is assigned to a pipeline stage.

We model a cluster as a $m \times n$ device mesh, whose underlying hardware topology may not be limited to such. A tuple (pp, dp, mp) is determined with $mp * dp * pp = m * n$, thus generating $m * n$ ranks that describe which GPUs are in a communication group. For example, if $mp = 2$, $(0, 0, 0)$ and $(0, 0, 1)$ forms a model parallelism group. Since the underlying GPUs have different connection speed from each other, which exact GPU gets assigned to which rank can affect the performance. For instance, if GPUs are all located in the same node, the communication speed for this group is fast. This is because intra-node bandwidth is generally higher than inter-node bandwidth, not to mention when NVLink is equipped by such node. As [3] suggests, different micro-batch sizes in pipeline parallelism lead to a noticeable difference in system throughput. While current systems often achieve load balance in a pipeline by using the same number of layers for each stage, we believe such strategies are sub-optimal when models become heterogeneous (*e.g.,* each layer has a different workload). In section 4.4, we present a **cost model-guided** randomized algorithm to optimize the first three dimensions. In section 4.5, we present a dynamic programming algorithm to optimize the last dimension. The overview of the system is shown in Figure 2

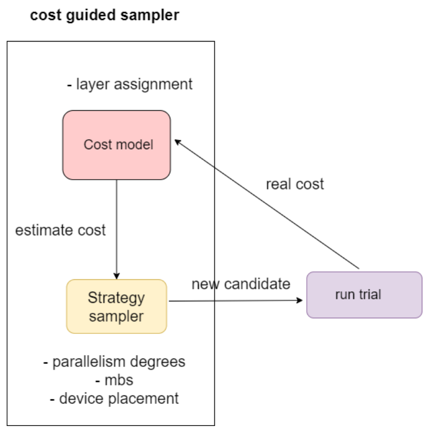


Figure 2: system overview

4.2 Communication Primitive

We consider all-reduce communication with ring topology in both data parallelism and mode parallelism [10]. Assume that the whole message size is M. In a ring all-reduce operation, all N workers form a circle and only neighbors pass information with split size $\frac{M}{N}$. Each split message would be passed through the ring twice for a reduce-scatter phase and an all-gather phase [9]. Denote the connectivity between workers with $B_1, ..., B_N$, the total time is thus

$$t_{comm} = \beta * \frac{2M}{N} \sum_{i=1}^{N} \frac{1}{B_i} \tag{1}$$

where β maps these values into wall clock time, and can be roughly estimated through simple profiling. It can also be improved by feedback from further real trials.

For point-to-point communication between pipeline stages, we assume it can always fully utilize the bandwidth [23], and model communication time as $\frac{A}{B}$ where A is the activation volume and B is the bandwidth connectivity between these two workers.

4.3 Cost Model

From a data parallelism perspective, The system runtime can be decomposed to two terms:

$$T_{all} = \max T_{pipeline} + \max T_{dpsync} \tag{2}$$

$T_{pipeline}$ is the time to finish executing a complete pipeline, where there are dp pipelines executed in parallel. T_{dpsync} is the time for all data parallelism workers to aggregate weight updates after execution, whose formula is given at equation 1 with message size equal to model parameter size and bandwidths looked up from actual GPUs. Since different pipelines and different data parallelism workers proceed with different speeds, we take a maximum over different pipelines, and a maximum over different data parallelism worker communication time.

A single pipeline consists of K stages. The communication between the i^{th} and $(i+1)^{th}$ is denoted as c_i. Execution time for the i^{th} stage t_i is the sum of each layer execution time in the stage.

$$t_i = \sum_{layers} \alpha * fp + T_{mpsync} \tag{3}$$

fp is the number of floating point operations for each model parallelism worker responsible for the layer, and T_{mpsync} is the allreduce time between these workers. α is another parameter that can be obtained by profiling or improved by further real trails.

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

$$T_{pipeline} = (M-1) * \max_{i \in \{1, \cdots, K\}} t_i + c_1 + \cdots + c_{K-1} + t_1 + \cdots + t_K \tag{4}$$

where M is the number of micro-batches in the pipeline. Intuitively, the first term models the straggler effect, which gets amplified by the number of time it appears. A dynamic programming algorithm can be adopted to directly solve for the layer assignments with objective 4.

4.4 Randomized optimization

Different than the objective in 4, the first three dimensions exhibit less structure that allows us to leverage a deterministic approach, while composing a large search space. For example, there are exponentially many possible device placement with less known orders [24]. Thus, we use a *simulated annealing* (SA) algorithm with the cost model in section 4.3 to iteratively find a good strategy. In addition, a locality-aware device placement strategy is in use: we assume that GPUs in the same pipeline shall be neighbors to each other in the device mesh. Specifically, we consider a problem setup similar to the Domino Tiling problem: given a $m \times n$ device mesh square, we first determine the size of the domino $mp \times dp$, and tile the device mesh using dominos either horizontally or vertically.

4.5 Pipeline layer assignment

5 Experiments

Megatron-LM presents a comprehensive analysis on 3D parallelism space, and proposes several heuristics [3]. We name their strategy **Megatron** and use as the baseline we are comparing to:

1. mp shall be used up to degree g where g is the number of GPUs in the smallest node.
2. micro-batch size shall be enumerated.
3. given micro-batch size, the product $mp \times pp$ shall be the smallest to hold the model.

To summarize, the Megatron strategy loops over all possible micro-batch size. For each micro-batch size, it selects the candidate with the smallest $mp \times pp$, whose mp is smaller than g.

The second optimization strategy we propose is Random Search (**RS**), which randomly choose candidates in our designed search space. The final optimization strategy is Simulated Annealing (**SA**) that incorporate our cost-guided optimizer to traverse the search space. In both experiments, we set the SA iteration to 500, and real trial budget to 10, which is similar for the number of trials Megatron need. We set the budget for random search to be 50 to understand the average strategies in our search space.

Algorithm 1: randomized optimization

```
Input: iteration, budget
/* Initialize a strategy */
t = 1.0 // Temperature for SA algorithm
s = initialize_strategy()
record = set()
/* estimate the runtime using our cost model */
cost = estimate(s)
for i in iteration do
    t = cool_down(t, i)
    new_s = s.copy()
    a = randn() // randomly select mp or dp degree to vary
    if a > 0.5 then
        new_s.mp = sample_mp(s.dp)
    else
        new_s.dp = sample_dp(s.mp)
    new_s.mbs = sample_mbs(new_s.dp)
    new_s.tile = sample_tile(new_s.dp, new_s.mp, new_s.mbs)
    /* Invoke algorithm 2 to determine pipeline layers */
    new_s.pipe = pipe_dp(s)
    new_cost = estimate(new_s)
    acc_prob = $e^{\frac{min(cost-new\_cost,0)}{t}}$
    if randn() < acc_prob then
        // accept the candidate strategy
        s = new_s
        record.add(s)
        cost = new_cost
/* find best strategies from top budget candidates using real trials. */
record = sorted(record)
best_s = s
best_cost = inf
for i in 1,...budget do
    real_cost = simulate(record[i])
    if real_cost < best_cost then
        best_s = record[i]
        best_cost = real_cost
return best_s
```

5.1 Homogeneous setup

Current systems [14] [2] assume a homogeneous cluster and models. Typically, they use several DGX2 nodes and Transformer models. We hypothesize that under such setups, Megatron strategy is (near) optimal. To examine this, We conduct experiments using GPT2 (L=24, H=1024) [4] on 4 AWS EC2 g4dn.16xlarge nodes. Each node has 4 K80 GPUs with 75 Gbps PCIe connection intra-node bandwidth, and 50 Gbps inter-node bandwidth.

Megatron finds better strategy compared to RS and SA. Our proposed search space would often propose strategies that include cross node model parallelism communication. While the designed cost model alleviate this issue (2.04 compard to 2.10), the Megatron baseline simply reject all such strategies.

conclusion: In a homogeneous cluster with homogeneous model, Megatron is (near) optimal.

Algorithm 2: pipe_dp

```
Input: L, K, A, B, M
Output: Optimal time and stage assignment
/* Initialize all possible stage time combination possible, and a
   dynamic programming table arr                                      */
possible = [0]
for i in 1.. L do
    ptr = 0
    while ptr+ i <= L do
        possible.append(sum(A[ptr:ptr+i]))
        ptr += 1
sorted(possible)
Initialize arr (size L*K*len(possible)) with ([], $\infty$).
for i in 0..L-1 do
    for j in 0..K-1 do
        for m in range(len(possible)) do
            if i + 1 <= j then
                /* Invalid, We need at least length $j+1$ for j cut   */
                Continue
            else
                if j == 0 then
                    // Base case, 0 cut
                    cur = sum(A[:i+1]) // Pythonic, not including i+1
                    arr[i][j][m] = ([i+1], (M-1) * max(0, cur - possible[m]))
                else
                    cost_best = $\infty$
                    S_best = []
                    for cut in j-1,...i-1 do
                        // Enumerate all possible last cut position
                        cur = sum(A[cut+1:i+1])
                        S, cost = arr[cut][j-1][possible.index(max(cur, possible[m]))]
                        /* Penalty and new communication cost         */
                        cost += (M-1) * max(0, cur - possible[m])
                        cost += B[cut][j-1]
                        if cost < cost_best then
                            cost_best = cost
                            S.append(i-cut)
                            S_best = S
                    arr[i][j][m] = (S_best, cost_best)
return arr[L-1][k-1][0]
```

5.2 Heterogeneous setup

We conduct experiments using GPT2 (L=24, H=1024) on 3 AWS EC2 P3.8xlarge nodes (each node has 4 A100 GPUs with NVLink intra-node connection and 10Gbps inter-node bandwidth) and 4 g4dn.8xlarge (each with 1 K80 GPU with 60 Gbps inter-node bandwidth).

In such setting, the effect of bad candidates in our search space get amplified (3.04 on average). However, the random search is able to find a better candidate than Megatron. Using a cost-guided optimizer in our search space, the algorithm can find a candidate with $1.5\times$ speedup compared to Megatron. Specifically, the third heuristics of Megatron is sub-optimal. When using micro-batch size 1, the best candidate shall be dp=4 and pp=4, which results in a 1.33 second/iteration performance. However, due to the third heuristic, Megatron selects dp=16, which results in 2.7 second/iteration performance. This candidate is much slower because it requires all GPUs to perform a data parallel

Table 1: Best and average candidates each strategy finds for homogeneous setup, scale: seconds per iteration.

-	Megatron	RS	SA
Min	**1.14**	1.27	1.39
Mean	$\mathbf{1.83 \pm 0.79}$	2.10 ± 0.83	2.04 ± 0.53

Table 2: Best and average candidates each strategy finds for hetergeneous setup, scale: seconds per iteration.

-	Megatron	RS	SA
Min	1.80	1.71	**1.31**
Mean	2.32 ± 0.37	3.04 ± 0.82	$\mathbf{1.68 \pm 0.33}$

synchronization with large message size (the whole model parameter), where some GPUs have low bandwidth (10 Gbps). On the contrary, our cost model considers difference in bandwidth, and punishes such candidates.

6 Conclusion

In a heterogeneous cluster with homogeneous model, Megatron is no longer optimal. Our algorithm can automatically find better strategies within similar budgets.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25:1223–1231, 2012.

[6] 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.

[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] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE transactions on Big Data*, 1(2):49–67, 2015.

[9] 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.

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

[11] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[12] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in Neural Information Processing Systems*, 31:10414–10423, 2018.

[13] 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.

[14] 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.

[15] 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.

[16] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[17] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems*, 3, 2021.

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

[19] 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.

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

[21] 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.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[23] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in accelerating convolutional neural networks. In *International Conference on Machine Learning*, pages 2274–2283. PMLR, 2018.

[24] 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.