# Fast Performance Prediction for Efficient Distributed DNN Training

Yugyoung Yun Eunhyeok Park
Graduate School of Artificial Intelligence, POSTECH, Korea

***Abstract*—Training large-scale DNN models requires parallel distributed training using hyper-scale systems. To make the best use of the numerous accelerators, it is essential to intelligently combine different parallelization schemes. However, as the size of DNN models increases, the possible combinations of schemes become enormous, and consequently, finding the optimal parallel plan becomes exceedingly expensive and practically unfeasible. In this paper, we introduce a novel cost model, the Markovian Performance Estimator (MPE). This model provides affordable estimates of the throughput of various parallel plans, promoting efficient and fast searches for the ideal parallel plan, even when resources are limited. Significantly, this work is pioneering in explaining the expensive nature of searching for an optimal plan and addressing it using intuitive performance estimations based on real device evaluations. Our experiments demonstrate the effectiveness of the MPE, revealing that it accelerates the optimization process up to 126x faster (36.4 on average) than the existing state-of-the-art baseline, Alpa.**

***Index Terms*—Distributed training, performance modeling, large language model, 3d parallelism**

## I. Introduction

LARGE language models (LLMs) demonstrate remarkable generative capabilities. However, these models, due to their enormous parameter count and high computational demands, necessitate a hyper-scale system for training, such as thousands of A100 GPUs for GPT-3 training [1]. The challenge lies in utilizing these large systems efficiently. Improving the throughput of distributed training could be achieved by finding the ideal combination of 3D parallelism [1], [2], i.e., pipeline parallel (PP) [3], [4], data parallel (DP) [5], [6], and operator parallel (OP) [7]–[9], specifically designed for the system and networks. However, searching for the optimal plan is an NP-hard problem, and due to the vast number of possible combinations, making a greedy search practically unfeasible [10], [11]. At present, optimization relies on human expertise, resulting in suboptimal configurations [1], [2].

Recent studies [10], [11] have indicated the potential for machine learning-based auto-optimization to provide more efficient solutions, offering higher throughput than human experts. For instance, Alpa [10], a state-of-the-art optimization study, proposes a hierarchical search algorithm that identifies intra-op and inter-op parallelism independently. This approach involves finding the best candidate for intra-op parallelism by partitioning the tensor operators along internal axes based on DP or OP and estimating the computational and communication overhead based on compilation metrics. On the other hand, the maximum throughput of inter-op parallelism, which divides the model into separate stages and executes them sequentially on different devices based on PP, is assessed through real device profiling. The combination of performance estimation for fine-gained optimization and precise profiling for coarse-grained optimization helps identify optimal candidates.

However, these studies are barely used in practice due to their high search cost. Despite Alpa's use of numerical estimation for intra-op tuning, optimization requires hundreds of compilations, resulting in notable overhead. The profiling of several stages also requires running a subset of models numerously, and the frequent communication due to the intra-op parallelism slows down the profiling even more. According to our observation, the optimization can take several weeks as the size of the LLM increases, which hinders the widespread adoption of automated optimization.

In this work, we propose a novel cost model called the Markovian Performance Estimator (MPE) that reduces the optimization overhead substantially by utilizing the recurring pattern of operators in LLMs and the Markovian property of sequential tensor execution. This straightforward but insightful idea considerably decreases both compilation and profiling overhead, allowing the search for the optimal parallelization policy to be completed within an hour, even for the currently largest LLM. To the best of our knowledge, we tackle the expense of cost modeling for the first time for LLM training.

## II. Background and Related Works

In this section, we aim to provide a comprehensive overview of essential background knowledge and significant research advancements that form the basis to understand our idea.

### A. 3D Parallelism

LLMs necessitate distributed systems for training, owing to their extensive memory needs exceeding the capacities of typical server-grade accelerators and substantial computational efforts to train the datasets having trillions of tokens. Recently, several mapping techniques (e.g., PP, DP, and OP) have been introduced for distributed training.

PP divides tensor operations into stages, processed sequentially across separate computational resources, leading to increased memory footprint but potentially reduced throughput due to inter-device communication. Conversely, DP segments

This work was supported by IITP grant (MSIT, No.2021-0-00310) and NRF grant (MSIT, RS-2023-00213611) funded by the Korea government and SAIT.
Manuscript received June 19, 2023; revised August 11, 2023.

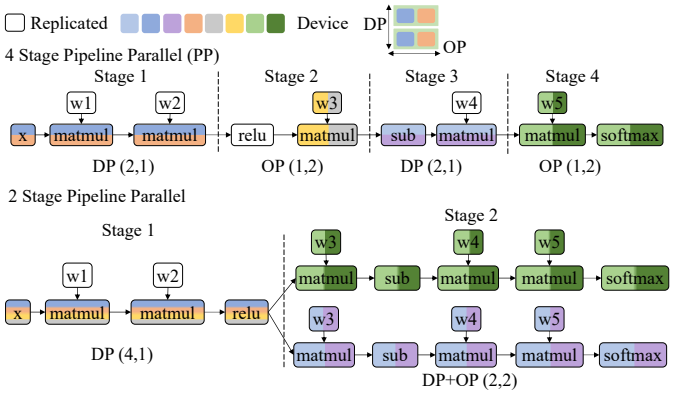


Fig. 1. 3D Parallelism Example. (X, Y) represents the device mesh, the degree of data parallel and operator parallel, respectively. Note that X·Y devices are used per stage. The horizontal split of the operator signifies batch parallelism for DP, whereas the vertical split denotes operator parallelism

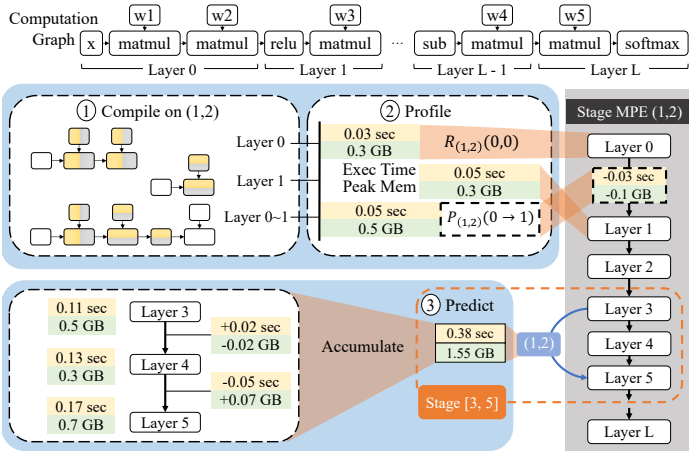


Fig. 2. Overview of Stage MPE-based Optimization.

input data for parallel computation on multiple devices, thus enhancing throughput proportional to the device count with minor synchronization overhead, although storage benefits are curtailed due to parameter duplication. As tensor dimensions expand, OP has been introduced, employing multiple devices to handle a single operation by slicing the tensor along input or output axes. While OP boosts both throughput and memory footprint, it necessitates frequent synchronization due to input/output separation and merging.

Given their distinct characteristics, a combination of these techniques can significantly improve throughput. However, the complexity of finding the optimal combination of PP, DP, and OP presents a significant challenge due to the increasing number of potential combinations with the growth of operators.

### B. Alpa

Alpa is an auto-tuning framework that optimizes the combination of PP, DP, and OP, as shown in Fig 1, through a hierarchical optimization process based on the granularity of layer, stage, and pipeline. Initially, tensor operators in a network are grouped into $L$ divisions, each having similar FLOPs. We refer to these groups as layers. The subset of layers is then grouped into stages, generating $L \times (L-1)$ candidates. For each stage, Alpa allocates a certain number of devices and determines the optimal DP and OP schedule, called a device mesh (denoted as (X, Y), where X and Y stand for the degree of parallelism of DP and OP, respectively). When the device mesh is given, the corresponding optimal slicing dimension are determined for each operator within a stage, aiming to minimize the collective communication cost. All potential slicing dimensions for each operator are evaluated, and the corresponding communication costs are evaluated based on the cost model in XLA HLO [12]. The optimal plan is identified using integer linear programming (ILP) and then compiled into executable code with XLA HLO optimizations like operation fusion and memory planning. After performing the intra-op optimization, Alpa profiles the memory footprint and performance of the compiled code for each device mesh per stage. Then it performs an inter-op optimization that minimizes PP latency within available resources through dynamic programming.

However, the hierarchical search in Alpa has significant overhead from repeated computations across all stage and device mesh combinations, necessitating $O(L^2 \times log(D))$ compilations and profiles when $D$-devices are available. Alpa attempts to address this cost through a parallel evaluation environment using distributed software, Ray [13], but this concurrent evaluations increase the error of profiling due to contention among the distributed workers and inaccuracies in the cost model. Our novel approach minimizes evaluations to $O(L \times log(D))$, offering fast, accurate measurements with an error comparable to existing Alpa's methods.

## III. Markovian Performance Estimator (MPE)

In this paper, we introduce a new concept called the Markovian Performance Estimator (MPE), which accelerates optimization by up to two orders of magnitude.

### A. Stage MPE

Stage MPE is a performance estimator for a given stage and device mesh. It replaces the intra-op optimization process of Alpa, which is designed based on two intuitive concepts: 1. The performance of a particular layer relies solely on the preceding layer. 2. Since LLM has a repetitive structure, corresponding layers, which are the sequences (groups) of operators in Alpa, are also repeated. The first concept allows us to focus the compilation and profiling efforts on adjacent layers, while the second facilitates the reutilization of profiling results for recurring patterns. With the assumption of the Markovian property, we construct a Markov chain (or directed graph) for each device mesh to predict performance.

In the graph, we identify unique layers. Using string matching to $L$ layers, we quickly discern layers with identical operators, often in under a second. Layers with the same functionalities are profiled once and then mapped to a node. To access a layer's performance, we reference the mapped node's value and edge. The value of each node, denoted as $P(a)$, is assigned as follows:

$$P_{(X,Y)}(i) = R_{(X,Y)}(i,i) \quad (1)$$

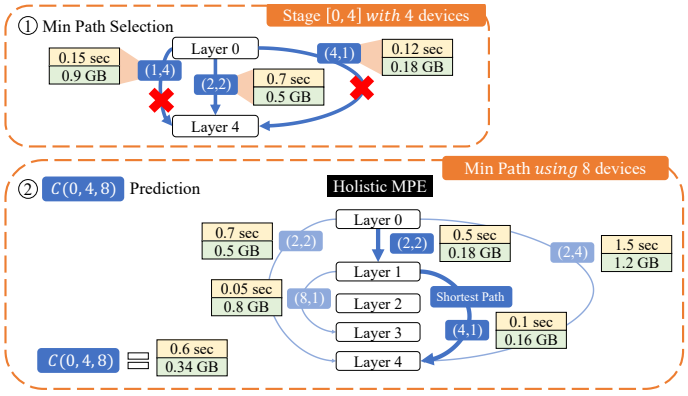


Fig. 3. Overview of Holistic MPE-based Optimization. After constructing the Holistic MPE by selecting the minimum path for each node from stage MPEs, we find the overall shortest path.

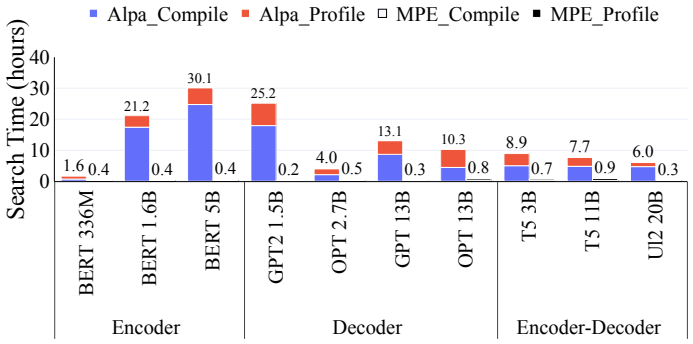


Fig. 4. Search time of Alpa with or without MPE for various LLMs.

where $R(x, y)$ is the measured performance on real devices with device mesh (X, Y), when executing layers from $x$- to $y$-th (Fig. 2, (2)). Additionally, an edge $i \rightarrow i+1$ is added between two adjacent nodes $i$- and $i+1$-th, whose value is calculated by subtracting each node's execution time from the time taken when the two nodes are run sequentially as follows:

$$P_{(X,Y)}(a \rightarrow b) = R_{(X,Y)}(a, b) - R_{(X,Y)}(a, a) - R_{(X,Y)}(b, b). \quad (2)$$

The edge represents the communication overhead or performance enhancement from additional optimizations, such as operator fusion, when layers are executed sequentially. For instance, the edge value reflects the frequent occurrence of changes in the optimal slicing dimension at the boundary between the two layers (Fig. 2, (1)).

We apply the same approach to estimate the memory footprint with an individual graph. The overall performance or memory usage of running a single stage (executing the layers from $a$ to $b$ in Fig. 2, (3)) given (X, Y) can be estimated by aggregating all values of visited nodes and edges.

In Alpa, a network with $L$ layers requires $O(L^2)$ evaluations per device mesh. However, the proposed method reduces this cost to $O(L)$ and further reduces it when there are repetitive patterns. Moreover, the introduction of an edge to consider optimization between adjacent layers greatly enhances the precision of the proposed method. For instance, GPT2 1.5B on a single node (8 GPUs) demands 17850 compilation and 35700 profiling with Alpa, whereas our MPE constructs a comprehensive performance model for the entire network using only 1386 compilation and 2772 profiling. This gap is larger as the model gets larger, so the benefits of MPE are more pronounced in larger LLMs.

### B. Holistic MPE

After constructing the stage MPE, we build a high-level graph $G_H$, referred to as the *Holistic MPE*, to replace the inter-op optimization in Alpa. In the Holistic MPE, each layer is mapped to a corresponding node. However, unlike the edges in the stage MPE that indicate performance differences between adjacent nodes, we add paths (using different notation intentionally) between nodes in the Holistic MPE. These paths offer the fastest execution within the given number of devices and fit within the available memory (Fig. 3, (1)). As a result, there are multiple paths for layers 0 to 4, but only one path is shown in the Holistic MPE in Fig. 3, (2). Through the Holistic MPE, we can estimate and find the shortest (minimum) execution latency $C(i, j, d)$ incurred when executing from the i-th to the j-th layers using d devices, as follows:

$$C(i, j, d) = min \begin{cases} G_H(i, j, d), \\ C(i, a, d') + C(a, j, d - d'), \end{cases} \quad (3)$$

where $0 < d' \leq d$ and $i \leq a \leq j$. Unlike Alpa's exhaustive evaluation of all conceivable combinations, MPE simplifies cost estimation by aggregating per-layer cost based on the stage MPE. This allows for a bottom-up cost estimation approach, accumulating per-layer costs into stages to construct a holistic MPE.

In the Holistic MPE, each path requires a different number of GPUs. When executing layers sequentially by traversing multiple paths, the cumulative GPU count is necessary. Thus, for optimal execution from layer $i$ to $j$, we must search for a trajectory in the Holistic MPE, starting from node $i$ and ending at node $j$, that minimizes the aggregated latency while ensuring the cumulative GPU count remains below the available GPU resources. As an example, in Fig. 3, (2), the shortest path from layer 0 to 4, utilizing 8 devices, occurs when executing 2 pipeline stages. The first stage (layer 0 to 1) employs a (2,2) device mesh, utilizing 4 devices, while the second stage (layer 1 to 4) utilizes a (4,1) device mesh, also utilizing 4 devices. The expected execution latency for this path is 0.6 sec, with a memory requirement of 0.34 GB. In this study, we employ dynamic programming to determine the optimal traversal plan $C(0, L, D)$ for covering all layers from 0 to $L$ using a total of $D$ available devices. Running the identified shortest path accelerates LLM training significantly.

## IV. Evaluation

We validate the performance of the proposed scheme (MPE) compared to 1. the baseline, Alpa, and 2. the naive estimator, which disregards the edge between nodes and estimates performance solely based on layer-wise compiling/profiling. All experiments are conducted on real devices equipped with 8 RTX 3090 GPUs. For a fair comparison, we use the on-device profiling setting of Alpa.

### A. Compiling and Profiling time

Fig. 4 illustrates that Alpa's compile/profiling time can exceed a day, even for smaller models like BERT 5B, due to the exploration overhead involved in considering all stage and device mesh combinations. In contrast, our MPE method significantly reduces this time by conducting layer-level compilation within a specific device mesh, leading to a substantial reduction in compile/profiling time. The optimization is accelerated up to 126x faster (36.4x on average), and MPE typically optimizes any model in under an hour, demonstrating its remarkable effectiveness.

### B. Prediction Quality

Fig. 5 compares the predictions achieved by stage MPE and the naive method in terms of real runtime and peak memory of Alpa for each stages with consists of arbitrary layer (e.g., *[3,7]*) on particular device mesh (e.g., *(1,4)*). The results show that MPE delivers reliable predictions on Alpa, whereas the naive method estimated higher memory usage and slower performance in certain cases, while MPE stably predicts the performance value in the range of sample variance. MPE considers the communication overhead and optimization between consecutive layers based on the edge, leading to reliable output. When evaluating the mean-average percentage error (MAPE) as shown in Fig. 6, MPE significantly outperforms the naive method, even with a lower error margin compared to the profiling error of Alpa. These findings indicate that, despite incorporating several approximations, MPE provides reliable predicted results to the baseline, Alpa.

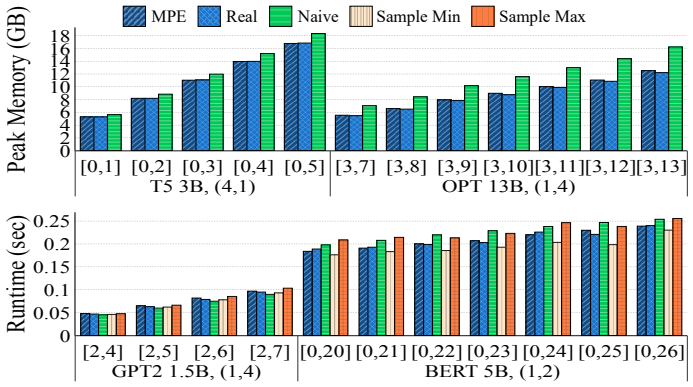


Fig. 5. Performance Prediction Quality between our *MPE* and *Naive*. *Real* means performance value on real execution in Alpa. *Sample min,max* indicates the minimum/maximum values during profiling.

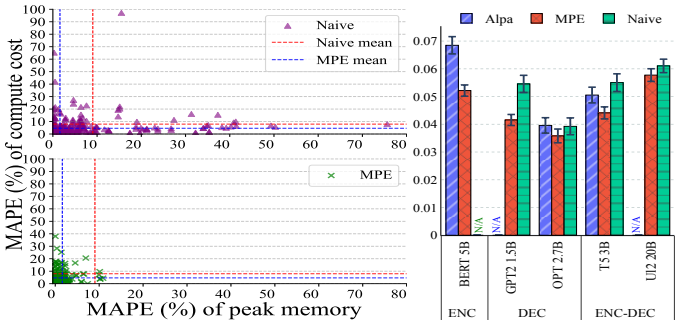


Fig. 6. Distribution of MAPE of compute cost[ms] and peak memory[GB] among various LLMs and device mesh choices.

Fig. 7. Comparison of the searched result, the runtime[s] of training per iteration.

### C. Training Performance Comparison

We evaluated the performance of various training plans, as illustrated in Fig. 7. For MPE and Naive, we utilize our shortest path algorithm. Some networks in Alpa, such as GPT2 1.5B, couldn't find an optimal solution due to the contention among numerous profiling workers, and on models like UI2 20B, Alpa failed to establish a solution while there is a viable solution within the memory limitations. Conversely, our MPE method excelled at identifying optimal solutions even where Alpa and Naive stumbled, owing to its smooth representative capability and reliable profiling using adequate warmup/repeat (number of on-devie profiling samples) iterations per profiling of layer. These results showcase the robustness and reliability of MPE in finding the optimal training schedule.

## V. Conclusion and Future Work

In this paper, we introduce a novel concept, the Markovian Performance Estimator (MPE), designed for rapid and accurate estimation of performance and memory footprint. MPE takes into account the repetitive operators in LLMs and the overhead or optimizations between adjacent operators, thereby enabling precise modeling with a minimal need for compilations and profiling. Our extensive experiments validate that incorporating MPE accelerates the optimization process up to 126 times compared to Alpa, providing a practical solution for efficient LLM training.

## References

[1] D. Narayanan *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.
[2] M. Shoeybi *et al.*, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv:1909.08053*, 2019.
[3] Y. Huang *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Neural Information Processing Systems (NeurIPS)*, 2019.
[4] A. Harlap *et al.*, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv:1806.03377*, 2018.
[5] S. Fan *et al.*, "Dapple: A pipelined data parallel approach for training large models," in *Principles and Practice of Parallel Programming (PPOPP)*, 2021.
[6] S. Rajbhandari *et al.*, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.
[7] D. Lepikhin *et al.*, "Gshard: Scaling giant models with conditional computation and automatic sharding," *arXiv:2006.16668*, 2020.
[8] Y. Xu *et al.*, "Gspmd: general and scalable parallelization for ml computation graphs," *arXiv:2105.04663*, 2021.
[9] N. Shazeer *et al.*, "Mesh-tensorflow: Deep learning for supercomputers," *Neural Information Processing Systems (NeurIPS)*, 2018.
[10] L. Zheng *et al.*, "Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning," in *Operating Systems Design and Implementation (OSDI)*, 2022.
[11] Z. Shi *et al.*, "Tap: Accelerating large-scale dnn training through tensor automatic parallelisation," *arXiv:2302.00247*, 2023.
[12] Tensorflow. (2023) Xla: Optimizing compiler for machine learning. [Online]. Available: https://www.tensorflow.org/xla
[13] P. Moritz *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *Operating Systems Design and Implementation (OSDI)*, 2018.