MALLEUS: Straggler-Resilient Hybrid Parallel Training of Large-scale Models via Malleable Data and Model Parallelization

Haoyang Li*
lihaoyang@stu.pku.edu.cn
Peking University

Fangcheng Fu*
ccchengff@pku.edu.cn
Peking University

Hao Ge
gehao@stu.pku.edu.cn
Peking University

Sheng Lin
linsh@stu.pku.edu.cn
Peking University

Xuanyu Wang
wxyz0001@pku.edu.cn
Peking University

Jiawen Niu
niujiawen705@stu.pku.edu.cn
Peking University

Yujie Wang
alfredwang@pku.edu.cn
Peking University

Xiaonan Nie
xiaonan.nie@pku.edu.cn
Peking University

Hailin Zhang
z.hl@pku.edu.cn
Peking University

Bin Cui
bin.cui@pku.edu.cn
Peking University

Abstract

As the scale of models and training data continues to grow, there is an expanding reliance on more GPUs to train large-scale models, which inevitably increases the likelihood of encountering dynamic stragglers that some devices lag behind in performance occasionally. However, hybrid parallel training, one of the de facto paradigms to train large models, is typically sensitive to the stragglers.

This paper presents MALLEUS, a straggler-resilient hybrid parallel training framework for large-scale models. MALLEUS captures the dynamic straggler issues at the nuanced, per-GPU granularity during training. Once a shift in the GPU ability is detected, MALLEUS adaptively adjusts the parallelization of GPU devices, pipeline stages, model layers, and training data through a novel planning algorithm, accommodating the dynamic stragglers in real time. In addition, MALLEUS seamlessly and efficiently migrates the model states to fulfill the adjusted parallelization plan on the fly, without sacrificing the stability of the training tasks. Empirical results on large language models with up to 110B parameters show that MALLEUS consistently outperforms existing parallel training frameworks under various straggler situations, delivering on average 2.63-5.28× of efficiency improvement.

1 Introduction

Recent years have witnessed the unprecedented success of large-scale deep learning models. Notably, large-scale Transformer models, particularly the large language models (LLMs), have achieved remarkable advancements in various applications [5, 30, 54]. Their exceptional performance can be largely attributed to the expanding scale of both the models themselves and the volumes of training data [23]. Meanwhile, this trend also makes the training of such large models more resource-intensive. For instance, Microsoft takes 4480 A100 GPUs for training MT-NLG [47], whilst Meta has announced that two 24000-GPU clusters are used to train Llama 3 [31].

Hybrid parallel [37] is known as the combination of data parallel and model parallel, and has become a pivotal foundation for training large-scale models over multiple GPU devices. When training with hybrid parallel, synchronization is expected among these GPUs. For instance, model parallel techniques like tensor parallel and pipeline parallel necessitate exchanging the intermediate results [15, 36, 46], whilst data parallel techniques require synchronizing the model replicas [27, 28, 45]. Undoubtedly, it prefers a homogeneous environment to avoid idle waiting. Nevertheless, the hardware environment would be far from ideally homogeneous in the training of large-scale models — involving more GPUs in the training inevitably raises the probability of encountering *dynamic stragglers*. Several studies have observed that a few GPUs occasionally become slower than expected [22, 51], which can be introduced by various unforeseeable issues like GPU auto-throttling, network link jitter, overheating, unstable voltage, and unknown sharing. When stragglers exist, all the other GPUs suffer from long periods of idle time waiting for the stragglers, causing severe efficiency degradation.

To mitigate the dynamic straggler problem, there have been many efforts and they can be categorized into two lines. The first line of efforts [13, 14, 20, 32, 61] relaxes the

*Equal contribution.

synchronization protocol of data parallel, allowing the non-straggling devices to update models earlier than the straggling ones. The second line [2, 3, 29, 42] elastically changes the number of devices involved in training to remove the stragglers, and dynamically adjusts the global batch size (i.e., the number of training data per step). However, these approaches are developed for data parallel, and address the dynamic straggler problem at the granularity of model replicas. Whilst in hybrid parallel, each model replica is served by multiple devices, making existing works ineffective (detailed in §2.3). Worse still, impacts on the model convergence are inevitable with these approaches. Since large-scale models require an extremely long time to train, it is impractical to run numerous trials to tune hyper-parameters for these lossy approaches.

Beyond data parallel, straggler-resilient hybrid parallel for large models is under-explored yet. As a result, it demands engineers to laboriously discover the stragglers during training and manually replace them with backup devices [22]. However, let alone the substantial time cost in problem shooting and restarting, this works only when there are spare nodes in the cluster, which is unrealistic in general due to the GPU shortage problem [49, 58]. Besides, it necessitates removing the entire node (machine). However, existing studies have found that the straggler problem usually appears at the GPU granularity rather than the node granularity [22, 51]. Thus, simply removing the entire node would lead to a waste of computing resources when there are non-straggling GPUs on it, calling for a more nuanced solution.

To fill this gap, this work introduces Malleus[1], a straggler-resilient hybrid parallel training framework for large-scale models. Malleus pinpoints the dynamic stragglers at the nuanced, per-GPU granularity, and promptly adjusts the parallelization to maintain high performance. The adjustment takes the straggler situation into account and strikes a good balance among the training workloads on different GPUs through a series of non-uniform partitioning w.r.t. GPU devices, pipeline stages, model layers, and training data.

Given the GPUs of diverse straggling behaviors and the training task, we formulate a bi-level optimization problem for the deduction of a parallelization plan that maximizes the training efficiency. In the upper-level problem, the objective is to discover how to partition the GPUs into tensor parallel groups and how to orchestrate multiple training pipelines with these groups, with each group serving as one stage of a training pipeline. Given any possible solution to the upper-level problem, the lower-level problem aims to determine the best assignment of model layers within each pipeline as well as the best assignment of training data across the pipelines. Based on this bi-level problem, we develop a novel planning algorithm to deduce the optimal parallelization.

For the upper-level problem, we tackle it with two major efforts. First, we establish theorems to analyze how to partition the available GPUs into groups with performance guarantees, based on which a partition-then-split approach is devised. Second, given these groups, we formulate a mixed-integer non-linear programming (MINLP) problem in response to the varying efficiencies among groups. By solving the problem, we achieve the optimal orchestration of training pipelines.

For the lower-level problem, we reformulate it as a joint-optimization problem of layer and data assignments, which aims to minimize the training time by balancing the workloads across the stages within each pipeline and across the pipelines simultaneously. Then, we decouple the joint problem into multiple sub-problems in the form of integer linear programming (ILP), and solve them to obtain the best plan.

To facilitate the adaptation to the dynamic stragglers, we introduce an efficient re-planning process that adjusts the training task to accommodate the straggler situations in real time. For one thing, we design an asynchronous re-planning mechanism, which derives the optimal parallelization based on the immediate straggler situation without halting the training task. For another, according to the new plan, Malleus automatically migrates the model partitioning and device mapping on the fly, enhancing the stability and straggler-resilience in the training of large-scale models.

The contributions of this work are summarized as follows:

- We develop Malleus, the first straggler-resilient hybrid parallel training framework for large-scale models.
- Malleus quantifies the straggler problems into straggling rates at the nuanced, per-GPU granularity, and introduces a novel parallelization planning algorithm to automatically deduce the optimal model and data parallelization that maximizes the training efficiency given the straggling rates.
- In response to the dynamic changes in straggling situations, Malleus supports adjusting the parallelization plan on the fly, eliminating the need of re-starting the training task.
- We conduct experiments using LLMs with up to 110B parameters. The results in various straggler situations show that Malleus outperforms existing parallel training frameworks in terms of training speed by up to 6.73× and 2.63-5.28× on average. Furthermore, Malleus accommodates dynamic changes in straggler situations and consistently achieves at least 90% of the theoretic optimal performance.

2 Preliminaries

2.1 Parallelization in Model Training

With the explosive growth in model sizes and training data, parallelization of data and models has become an essential bedrock to train large-scale models.

[1] In our work, the parallelization is designed to be malleable in order to accommodate the dynamic straggler problem. Thus, we name our framework "Malleus", which is the Latin etymological origin of the word "Malleable".

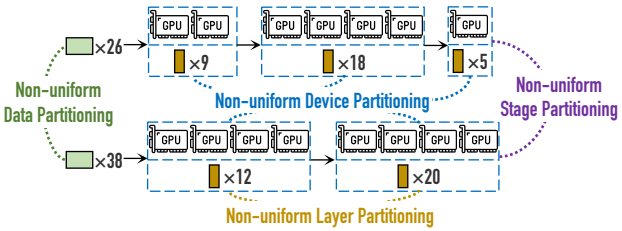

Figure 1. Key characteristics of the parallelization plans in our work, illustrated with the example workload in Figure 2.

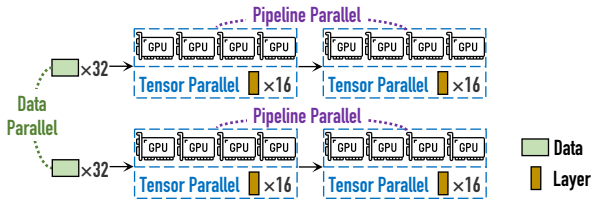

Figure 2. An example of 3D parallel. The model consists of 32 layers and the global batch size is 64.

Data parallel. Data parallel (DP) [11, 21, 27, 28, 38, 45] scatters the training data to multiple devices, yet requires each device to hold one model replica. In each training step, the devices execute forward and backward propagation individually to compute the model gradients, which are then synchronized through communication (e.g., `all-reduce`) to ensure the model replicas on all devices are updated in the same way.

Model parallel. Model parallel splits the model states across multiple devices to support large-scale models. Notably, tensor parallel (TP) [46] and pipeline parallel (PP) [15, 35, 36] are two well-known categories of model parallel.

TP splits the model parameters of computationally intensive operations (e.g., matrix multiplication) across devices. However, TP necessitates network communications to exchange the intermediate results in both forward and backward propagation, implying a need for high communication bandwidth among the devices. Thus, TP is typically applied on GPUs within the same node (machine), because intra-node connections have higher communication bandwidth than inter-node connections in most cases.

PP treats a model as a sequence of layers, and partitions the layers into multiple stages. These stages are distributed across devices to form a pipeline, and peer-to-peer communication is leveraged to transmit the intermediate results between consecutive stages. Since only the activations across stages need to be transmitted, PP usually entails a much lower communication volume than TP, and is able to accommodate both intra- and inter-node network connections.

Hybrid parallel. Hybrid parallel [19, 33, 37, 50, 60, 63] incorporates the strengths and weaknesses of different parallel techniques. Particularly, one of the most widely used hybrid parallel approach is the 3D parallel approaches in Megatron-LM [24, 37], which is illustrated in Figure 2. Such a 3D parallel approach encompasses DP, TP, and PP, and has become fundamental for training large-scale models.

2.2 Straggler-resilient Parallel Training

Since parallel training needs to synchronize model gradients and/or intermediate results to ensure correct convergence, when stragglers exist, most GPUs must sit idle until the slowest stragglers have caught up. Obviously, it results in a huge waste of resources and poor efficiency. There have been numerous approaches to mitigate the straggler problem. We classify them into two lines.

The first line of approaches [13, 14, 20, 32, 61] is developed based on the idea of stale synchronous parallel (SSP), which relaxes the synchronization protocol and allows updating different model replicas asynchronously. Specifically, fast DP groups can synchronize model gradients and update models without waiting for the slow DP groups. There is usually a threshold that constrains the differences in terms of the training steps between the fastest and slowest groups. By doing so, the SSP-based approaches reduce the idle periods of the non-straggling devices and thereby improve efficiency.

The second line [2, 3, 29, 42] addresses the straggler problem by dynamically adjusting the number of devices during training. Particularly, once any stragglers are detected, the system removes the related devices from the training task, and (optionally) tries to request new devices from the cluster. Consequently, all devices involved in the training task will not be slowed down by stragglers. In essence, these approaches manage the devices at the granularity of model replicas, which is beneficial to ensure aligned performance among DP groups. When the number of DP groups changes, the global batch size is also adjusted proportionally.

2.3 Limitation Analysis

Despite the numerous efforts of straggler mitigation, we find that they are developed for DP, but fall short in the scenario of hybrid parallel training of large-scale models.

First, existing efforts solely aim at DP, whilst model parallel, which is indispensable due to the tremendous scale of models, is not their primary focus. In hybrid parallel, each model replica is served by a training pipeline consisting of multiple GPUs, making existing efforts ineffective. For one thing, SSP-based approaches cannot prevent the non-straggling GPUs from being affected by the stragglers within the same pipeline. For another, for approaches based on straggler removal, removing one straggler would make an entire pipeline inexecutable, so they must keep the other GPUs in the pipeline idle until a new device is ready to join.

Second, existing efforts inevitably impact the model convergence. For instance, SSP-based approaches are proven to slow down the convergence rate and may even lead to

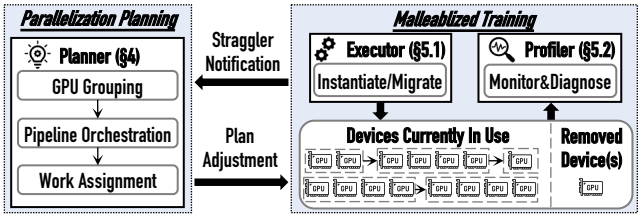

Figure 3. Architecture overview of Malleus.

divergence due to the staleness in model gradients [14]. For approaches that based on elastic scaling, they also need to adapt the batch size to the DP degree, which influences the model convergence. Since the training of large-scale models is time-consuming (e.g., it takes weeks or even months to train an LLM), it is impractical to run numerous trials to tune the hyper-parameters.

To tackle these limitations, this work proposes to solve the straggler problem with a nuanced, per-GPU granularity, rather than the coarse, per-replica or per-node granularity. Besides, regardless of the straggler situations, our work does not adjust the global batch size and the synchronization protocol across pipelines to be lossless.

3 System Overview

In this section, we present an overview of our straggler-resilient hybrid parallel training framework, namely Malleus.

3.1 Design of Parallelization Plans

Figure 1 illustrates the key characteristics of the hybrid parallel training in Malleus. To harness the stragglers, Malleus integrates non-uniform partitioning of GPU devices, pipeline stages, model layers, and training data, as discussed below.

① Since GPUs in a TP group should synchronize frequently, in response to the efficiency variation among GPUs, Malleus supports *non-uniform device partitioning* by strategically allowing the numbers of GPUs in TP groups to be different.

② In hybrid parallel, each TP group serves as one unit that executes a pipeline stage. To accommodate the efficiency variation among TP groups, Malleus enables *non-uniform stage partitioning*, which allows the pipelines to have varying numbers of stages (i.e., TP groups).

③ Within each pipeline, the TP groups would differ in efficiency. To address this issue, Malleus uses *non-uniform layer partitioning*, which supports appointing different numbers of model layers to the stages within each pipeline.

④ Since the efficiencies of different pipelines would also be unaligned, Malleus facilitates *non-uniform data partitioning* that allocates different volumes of training data to the pipelines to balance their training time.

In short, a solution of such non-uniform partitioning is determined to accommodate the stragglers. We call the solution a ***parallelization plan***. And the goal of Malleus is to adaptively adjust the parallelization plan against the dynamic straggler situations to maximize the training efficiency.

Table 1. Notations used in this work.

$\overline{DP}$	DP degree (the number of pipelines)
$\overline{PP}_i$	PP degree (the number of stages) of the i-th pipeline
x	The GPU straggling rate of a specific GPU, which denotes its slowdown rate compared to a normal GPU (higher indicates slower and $x = 1$ indicates the GPU is not a straggler)
y	The group straggling rate of a specific TP group
L	The number of layers in the model
$l_{i,j}$	The number of model layers assigned to the j-th stage in the i-th pipeline
B	The global batch size (number of training data per step)
b	The micro-batch size
m_i	The number of micro-batches assigned to the i-th pipeline
$\mathbb{N}_0$	The set of non-negative integers, i.e., $\{0, 1, 2, \cdots\}$

3.2 System Components

The architecture overview of Malleus is shown in Figure 3. There are three major components, as introduced below.

Planner. The planner is responsible for analyzing and deducing the most suitable parallelization plans. In particular, it takes as input the task description (e.g., model architecture, mini-batch size, etc.) provided by the user and the profiled information collected by the profiler. Then, it deduces the parallelization plan aiming to adapt the training plan to the stragglers in real time. The deduction is done based on a brand new planning algorithm, which will be detailed in §4.

Executor. The executor is in charge of the training based on the parallelization plan. Specifically, whenever the planner makes a new decision, the executor triggers a migration process that adjusts the model partitioning and device mapping immediately. Besides, to realize efficient training of our parallelization plans, the executor manages the model sharding and gradient synchronization across the pipelines in a non-uniform manner. More details will be elaborated in §5.1.

Profiler. The profiler monitors the real-time hardware efficiency of each device and provides the GPU straggling rates on the fly. To be specific, it measures the running time of each GPU, identifies the stragglers, and calculates their straggling rates by comparing to the non-stragglers. In addition to recording the running time on devices that are currently in use, it also examines devices that were previously removed due to high straggling rates since these devices may be useful later (detailed in §5.2). Once an obvious shift in the GPU straggling rates is detected, the profiler will notify the planner immediately.

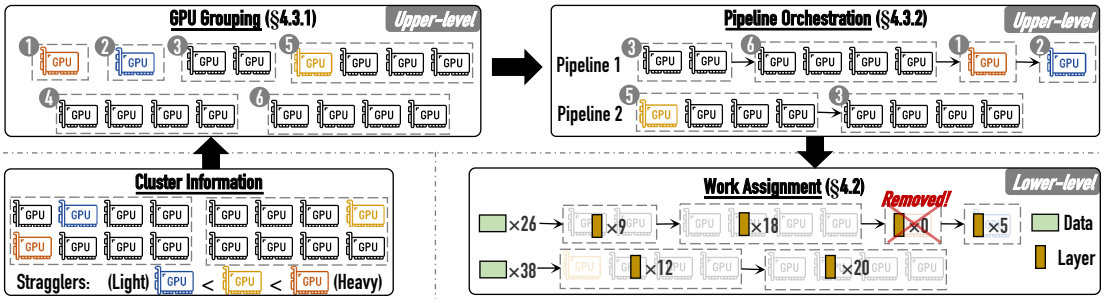

Figure 4. Overview of the planning routine.

Overall Routine. The overall routine of Malleus is as follows. ① The training process begins with an initial parallelization plan, which can be provided by the user or suggested by the planner. ② The executor instantiates the model states over the GPUs and carries out the training. ③ Meanwhile, the profiler consistently tracks the performance and examines the GPU straggling rates. ④ Once an obvious shift in the straggling rates is detected, the planner is informed and then triggers a re-planning process to adjust the parallelization plan to adapt to the dynamic stragglers.

4 Parallelization Planning

In this section, we introduce our planning algorithm, which aims to optimize the overall training efficiency.

4.1 Problem Formulation

In a nutshell, a parallelization plan consists of four components. ① The *GPU grouping* that indicates how the available GPUs are partitioned into different TP groups. ② The *pipeline orchestration* that describes how to organize multiple training pipelines with the TP groups, where each group serves as one stage of one pipeline. ③ The *layer assignment* of each pipeline, which represents how the model layers are assigned to the corresponding groups. ④ The *training data assignment* of each pipeline, which corresponds to how a global batch of training data is scattered among the pipelines.

To ease the planning, we formulate the deduction of parallelization plan as a bi-level optimization problem as follows.

- **Upper-level problem:** Suppose there are N GPUs available, and denote the straggling rate of the i-th GPU as x_i. The objective of the upper-level problem is to find out the best combination of GPU grouping and pipeline orchestration that minimizes the training time.
- **Lower-level problem:** Suppose there is an arbitrary combination of GPU grouping and pipeline orchestration, and denote the straggling rate of the j-th stage in the i-th pipeline as $y_{i,j}$. The objective of the lower-level problem is to determine the best combination of layer assignment and training data assignment minimizing the training time.

Undoubtedly, the Cartesian product of the solution regions of the two problems covers all feasible parallelization plans, so we only need to solve the bi-level problem.

Figure 4 presents an overview of our planning routine. There are three processes. Given the profiled information, Malleus initiates the GPU grouping process (§4.3.1), which aims to construct TP groups given the available GPUs. Then, the pipeline orchestration process (§4.3.2) determines how to organize multiple training pipelines with these groups. Both the GPU grouping and pipeline orchestration processes are for the upper-level problem. Finally, the work assignment process (§4.2) solves the lower-level problem by a joint optimization of layer and training data assignments.

4.2 Solving the Lower-level Problem

Suppose we have orchestrated $\overline{DP}$ pipelines, and the i-th pipeline consists of $\overline{PP}_i$ stages. As introduced in §3.1, the pipeline stages may vary in the number of GPUs and performance due to the stragglers. The lower-level problem aims to jointly deduce the optimal assignments of model layers and training data, in order to minimize the training time.

Definition of Group Straggling Rates. We first define the straggling rates of TP groups, which are utilized to estimate the efficiency of a group in our work. Suppose there is a TP group containing n GPUs with straggling rates of $\{x_{k_1}, \cdots, x_{k_n}\}$. Such GPU straggling rates are given by the profiler, as introduced in §3.2. Then, the group straggling rate is calculated based on two considerations:

- The slowest straggler dominates the running time of the whole group due to the synchronous nature of TP. Hence, the group straggling rate is dependent on $\max\{x_{k_1}, \cdots, x_{k_n}\}$.
- The workload in a TP group is evenly distributed, so the groups' numbers of GPUs matter. Denote ζ_n as the time cost of a unit workload (e.g., one Transformer layer with a batch size of 1) with n non-straggling GPUs. Then, we compute $\rho_n = \zeta_n / \max_{n'}\{\zeta_{n'}\}$ as the coefficient of efficiency degradation when the group consists of n GPUs. Such coefficients can be profiled and computed beforehand.

Putting them together, the group straggling rate is calculated as $y = \rho_n \times \max\{x_{k_1}, \cdots, x_{k_n}\}$.

Cost Model. For all $i \in [1, \overline{DP}]$, $j \in [1, \overline{PP}_i]$, denote m_i as the number of micro-batches assigned to the i-th pipeline, and $l_{i,j}$ as the number of layers assigned to the j-th stage in the i-th pipeline. We introduce the cost modelling for training time and memory consumption, respectively.

To begin with, we focus on the training time. For a micro-batch size of b, we denote $\tau(b)$ as the running time (including forward and backward) of one layer when the group straggling rate is 1, which can be profiled in advance. Since modern large models (e.g., LLMs) typically consist of identical layers, the running time of the j-th stage in the i-th pipeline for one micro-batch can be modelled as $t_{i,j} = y_{i,j} \times l_{i,j} \times \tau(b)$.

Following prior works [17, 55, 63], we model the running time of the i-th pipeline as $T_i = (m_i - 1) \times \max_j\{t_{i,j}\} + \sum_j t_{i,j}$, where the first term represents the running time of the 1F1B phase and the second term is for the warm-up and cool-down phases. Since large models are usually trained with a huge batch size (e.g., the global batch size in LLM training is usually a few millions of tokens) whilst the maximum accommodable micro-batch size is relatively small due to the limited GPU memory, the number of micro-batches (m_i) is usually much larger than that of pipeline stages ($\overline{PP}_i$). Therefore, we simplify the modelling for training time as $T_i \approx m_i \times \max_j\{t_{i,j}\} = m_i \times \max_j\{y_{i,j} \times l_{i,j}\} \times \tau(b)$. Empirical results in §7 show that the estimated running time given by our cost model is extremely close to the actual running time.

Next, for memory modelling, similar to prior works [55, 63], we focus on the memory of model states and forward activations. Both are proportional to the number of layers, whilst the memory of activations is also related to the stage index j in 1F1B pipeline execution. For simplicity, we denote $l_{i,j} \times \mu_{i,j}(b) + \nu_{i,j}(b)$ as the memory usage for the j-th stage in the i-th pipeline, where $\mu_{i,j}(b), \nu_{i,j}(b)$ are stage-specific coefficients calculated by profiling. Besides, as groups can vary in the number of GPUs, we denote $C_{i,j}$ as the memory capacity. Due to space constraints, we leave the details of how to calculate $\mu_{i,j}(b), \nu_{i,j}(b), C_{i,j}$ to Appendix B.4.

Deriving the Assignments. Based on the cost model, we wish to deduce the optimal values for $l_{\cdot,\cdot}, m_\cdot, b$. Recalling that the maximum accommodable micro-batch size is usually small, so we decide to enumerate the value for $b \in \{1, 2, \cdots\}$ until all assignments exceed the memory constraint. For a given candidate b, the lower-level problem can be written as

$$\begin{aligned}
&\arg\min_{l_{i,j}, m_i} \max_{i \in \left[1, \overline{DP}\right]} \left\{ \max_{j \in \left[1, \overline{PP}_i\right]} \left\{ y_{i,j} \times l_{i,j} \right\} \times m_i \times \tau(b) \right\} \\
&\text{s.t.} \sum_{i=1}^{\overline{DP}} m_i \times b = B, \ \sum_{j=1}^{\overline{PP}_i} l_{i,j} = L, \forall i \in \left[1, \overline{DP}\right] \\
&\quad l_{i,j} \times \mu_{i,j}(b) + \nu_{i,j}(b) \le C_{i,j}, \forall i \in \left[1, \overline{DP}\right], \forall j \in \left[1, \overline{PP}_i\right] \\
&\quad l_{i,j}, m_i \in \mathbb{N}_0, \forall i \in \left[1, \overline{DP}\right], \forall j \in \left[1, \overline{PP}_i\right]
\end{aligned} \tag{1}$$

Solving Eq. (1) is equivalent to solving two orthogonal types of sub-problems for $l_{i,j}$ and m_i, respectively (detailed deduction provided in Appendix B.5). The first type consists of $\overline{DP}$ sub-problems, with the i-th sub-problem written as

$$\begin{aligned}
&\arg\min_{l_{i,j}} \max_{j \in \left[1, \overline{PP}_i\right]} \left\{ y_{i,j} \times l_{i,j} \right\} \\
&\text{s.t.} \sum_{j=1}^{\overline{PP}_i} l_{i,j} = L, \forall i \in \left[1, \overline{DP}\right] \\
&\quad l_{i,j} \times \mu_{i,j}(b) + \nu_{i,j}(b) \le C_{i,j}, \ \ l_{i,j} \in \mathbb{N}_0, \forall j \in \left[1, \overline{PP}_i\right]
\end{aligned} \tag{2}$$

Eq. (2) is a well-formulated integer linear programming (ILP) problem, which can be solved efficiently. Denote o_i as the optimal value for the i-th sub-problem of the first type, then the second type of sub-problem is as follows.

$$\begin{aligned}
&\arg\min_{m_i} \max_{i \in \left[1, \overline{DP}\right]} \{ o_i \times m_i \} \times \tau(b) \\
&\text{s.t.} \sum_{i=1}^{\overline{DP}} m_i \times b = B, \ \ m_i \in \mathbb{N}_0, \forall i \in \left[1, \overline{DP}\right]
\end{aligned} \tag{3}$$

Again, Eq. (3) is also a well-formulated ILP problem. By solving these $\overline{DP} + 1$ sub-problems, we obtain the optimal assignments of layers and data.

It is worth noting that solving these ILP problems can automatically assign zero layers to groups with high straggling rates. The GPUs in these groups are removed from the training for better efficiency, as exemplified in Figure 4.

4.3 Solving the Upper-level Problem

Given N available GPUs, there are tremendous ways to organize the pipelines, and finding the best one is far from trivial. For one thing, as introduced in §3.1, we allow the number of stages to be varied among pipelines and the number of GPUs to be varied among the stages, so how to partition the GPUs into different groups is challenging. For another, to construct the pipelines given a feasible grouping result, we are required to determine which pipeline stage should each group serve as, and it is impossible to enumerate all candidate constructions. In this section, we focus on the two processes of GPU grouping and pipeline orchestration.

4.3.1 GPU Grouping.

We first describe the GPU grouping process in our work.

Even Partitioning. Suppose all GPU straggling rates are close, then the GPUs are expected to be evenly partitioned into TP groups to balance their performance. As mentioned in §2.1, it is a common practice to enforce TP within the same node due to its high communication cost. Thus, we can break down the GPU grouping into different nodes individually. In particular, we have the following Theorem.

Theorem 1. *Suppose there are n GPUs in a node with straggling rates $\{x_1, \cdots, x_n\}$, and we need to partition them into*

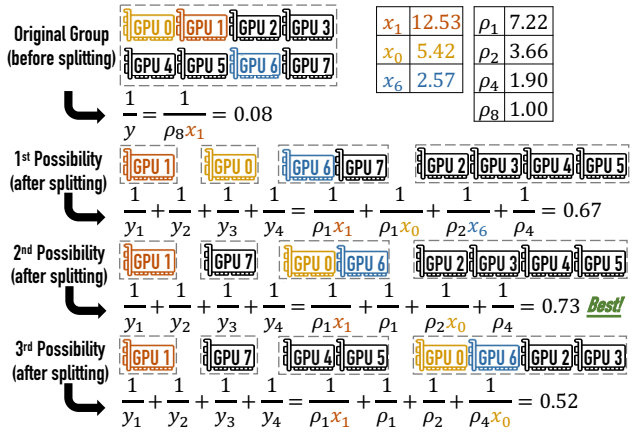

Figure 5. An example of GPU grouping with splitting.

n/k groups (each with k GPUs). Denote $\{i_1, \cdots, i_n\}$ as the ordering satisfying $x_{i_1} \geq \cdots \geq x_{i_n}$. Then, the best grouping result is $\{\{x_{i_1}, \cdots, x_{i_k}\}, \{x_{i_{k+1}}, \cdots, x_{i_{2k}}\}, \cdots, \{x_{i_{n-k+1}}, \cdots, x_{i_n}\}\}$.

Theorem 1 suggests that putting GPUs with similar performance into the same group is preferable. Besides, Theorem 1 does not make any assumptions on how the pipelines are constructed based on these groups nor how the layers and training data are assigned. Recalling that we do not consider cross-node grouping, we can perform the partitioning within each node and summarize the groups on all nodes.

Group Splitting. In practice, there may exist heavy stragglers that slow down the corresponding groups severely. In such cases, solving the lower-level problem (Eq. (1) in §4.2) would give the result that very few or even none of the layers should be assigned to the corresponding groups, leading to the waste of resources since the other GPUs in these groups become extremely under-utilized. Undoubtedly, it would be preferable if we split the groups to isolate the heavy stragglers (i.e., letting them form individual groups with a TP degree of 1).

Suppose there are 8 GPUs in the original group, after a heavy straggler has been isolated, then we need to re-group the rest 7 GPUs. However, Theorem 1 becomes inapplicable as 7 GPUs cannot be partitioned into groups with equal size. Besides, to obtain three groups with 1, 2, and 4 GPUs, respectively, there exist up to 6 possible grouping results (see details in Appendix B.7). To address this problem, we devise a mechanism to estimate the theoretical efficiency of an arbitrary grouping result, motivated by Theorem 2.

Theorem 2. *Suppose there are two different grouping results that consist of M' and M'' groups with straggling rates of $\{y'_1, \cdots, y'_{M'}\}$ and $\{y''_1, \cdots, y''_{M''}\}$, respectively. If we ignore the memory constraints in Eq. (1), and further assume the layer and training data assignments are not restricted to integers (i.e., $l_{i,j}, m_i$ in Eq. (1) can be any positive real numbers), then the minimum training time of the two grouping results satisfy $T'/T'' = (\sum_{i=1}^{M''} 1/y''_i)/(\sum_{i=1}^{M'} 1/y'_i)$.*

Following Theorem 2, it only takes constant time to examine how each possible grouping result after splitting performs compared with that before splitting. If none of the possible grouping results improve the estimated efficiency, then we will keep the straggling GPU. Otherwise, the best one after splitting will be chosen, as depicted in Figure 5.

To sum up, all GPUs are first evenly partitioned into groups by Theorem 1. Then, we iterate the straggling GPUs in descending order w.r.t. their straggling rates. For each straggling GPU, we examine whether we should isolate it and update the grouping result by Theorem 2. Such a routine will be executed for each candidate TP degree in $\{1, 2, 4, 8\}$, producing 4 grouping results in total, which are then forwarded to the pipeline orchestration process.

4.3.2 Pipeline Orchestration.

For each grouping result, to orchestrate multiple pipelines, there are two decisions to make: (i) how to divide the groups into multiple pipelines, and (ii) how to order the groups within each pipeline. Malleus makes the decisions through two steps, namely pipeline division and group ordering, respectively.

Pipeline division. We first focus on the first step. A naïve approach is to enumerate all possible division results and solve the lower-level problem to achieve the best one. However, the number of possible division results (a.k.a. the Bell number) grows exponentially w.r.t. the number of groups, making the naïve approach infeasible.

To cope with this issue, we simplify the problem from two perspectives. First, we leverage the fact that most GPUs are not stragglers, which further indicates most groups are associated with the same value of y. Thus, we treat these groups as identical to reduce the possible division results. Second, we loosen the constraints in Eq. (1) in order to accelerate the evaluation of each division result.

To elaborate, suppose there are M groups in total, where the majority of them share the same straggling rate of $\hat{y}$ (fast groups), and meanwhile M_s of them are different from the majority with straggling rates $\{y_1, \cdots, y_{M_s}\}$ (slow groups). If we ignore the memory constraints in Eq. (1), and further assume the layer assignments are not restricted to integers (i.e., $l_{i,j}$ in Eq. (1) can be any positive real numbers), then the problem of finding the best pipeline division can be written as (detailed deduction provided in Appendix B.6)

$$\begin{aligned} &\arg\min_{m_i, h_i, q_{i,k}} \max_{i \in \left[1, \overline{DP}\right]} \left\{ \frac{m_i \times \tau(b)}{h_i \times \hat{y} + \sum_{k=1}^{M_s} q_{i,k}/y_k} \right\} \\ &\text{s.t.} \sum_{i=1}^{\overline{DP}} m_i = \frac{B}{b}, \sum_{i=1}^{\overline{DP}} h_i = M - M_s, \sum_{i=1}^{\overline{DP}} q_{i,k} = 1, \forall k \in [1, M_s] \\ &\quad q_{i,k} \in \{0, 1\}, m_i, h_i \in \mathbb{N}_0, \forall i \in \left[1, \overline{DP}\right], \forall k \in [1, M_s] \end{aligned} \tag{4}$$

where h_i denotes the number of fast groups in the i-th pipeline, $q_{i,k} = 1$ indicates the k-th slow group is partitioned to the i-th pipeline, and $q_{i,k} = 0$ vice versa. The problem in Eq. (4) is a mixed-integer non-linear programming (MINLP) problem, a kind of complex combinatorial optimization problems that is usually more time-consuming to solve compared with ILP problems. Nevertheless, thanks to the reduction in possible division results and the loosened constraints, there is only a handful of decision variables in Eq. (4). As a result, we can solve the problem within a manageable time.

Group ordering. It is noteworthy that Eq. (4) is irrelevant to the ordering of groups within each pipeline. Thus, before feeding the pipelines to the lower-level problem, we need to determine the group ordering.

When there are no heavy stragglers to isolate, our GPU grouping will produce groups with the same number of GPUs. In such cases, the group ordering can be determined straightforwardly. To be formal, we have the following theorem.

Theorem 3. *Suppose the groups assigned to the same pipeline have the same number of GPUs, then the best ordering of pipeline stages satisfies that the groups are in descending order w.r.t. the group straggling rates.*

It suggests that faster groups should serve as the ending stages in a pipeline. The rationale is that the beginning stages need to reserve more memory for the forward activations, so putting faster groups to the ending stages allows them to process more layers for better efficiency. Therefore, we can determine the ordering easily but effectively.

When some heavy stragglers are isolated, our GPU grouping may produce groups with unequal numbers of GPUs, so we cannot apply Theorem 3 directly. Fortunately, the number of GPUs in a group is exactly the TP degree of the corresponding pipeline stage, which is typically restricted in $\{1, 2, 4, 8\}$. Consequently, we can enumerate the ordering of TP degrees. To be specific, for each pipeline, groups with the same number of GPUs are first bundled together, and groups in the same bundle are sorted via Theorem 3. Then, we enumerate the ordering of bundles and solve the lower-level problem to evaluate the efficiency of each enumeration. Eventually, the best enumeration will be selected. Considering that there are 24 ordering of bundles at most, and solving the ILP problems in §4.2 is extremely fast, such an enumeration-based approach works well.

4.3.3 Putting Them Together.

We finish this section by summarizing the overall routine. In the GPU grouping process, we enumerate the maximum TP degrees in $\{1, 2, 4, 8\}$ to obtain 4 grouping results. Then, the pipeline orchestration process takes each grouping result as input, and forms $\overline{DP}$ pipelines[2]. By doing so, there are 4 candidate solutions to the upper-level problem, which will be fed into the lower-level problem to determine the best one.

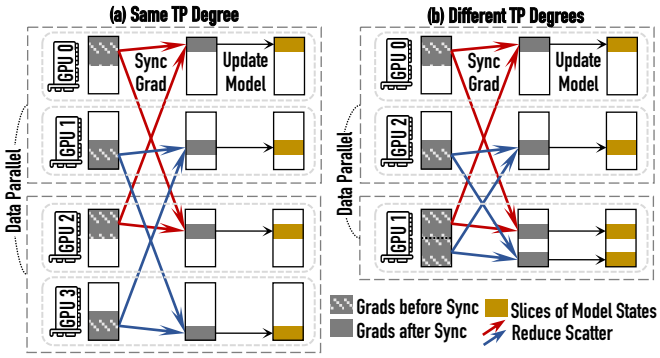

Figure 6. Illustration of model sharding. The red and blue arrows indicate different `reduce-scatter` communications for gradient synchronization. The `all-gather` communications after model update work inversely and are omitted.

5 Malleableized Training

This section describes how Malleus manages the model states and hardware devices to achieve malleableized training, enabling the immediate adjustment of parallelization plans to handle dynamic changes in straggler situations.

5.1 Model Management

Model Sharding. It is a popular choice to partition the model states via the ZeRO-1 optimizer [44] in hybrid parallel. Suppose $\overline{TP}$ is the TP degree of an arbitrary model layer, then the associated model states are sharded into $\overline{DP} \times \overline{TP}$ slices, and these slices are scattered to unique GPUs, as shown in Figure 6(a). Whilst in Malleus, we adjust the model sharding to accommodate varying TP degrees, as shown in Figure 6(b). To be precise, for an arbitrary model layer, suppose its TP degree in the i-th pipeline is $\overline{TP}_i$ and let $\overline{TP}_{max} = \max_i\{\overline{TP}_i\}$, then the corresponding model states are sharded into $\overline{DP} \times \overline{TP}_{max}$ slices, where each GPU in the i-th pipeline is responsible for $\overline{TP}_{max}/\overline{TP}_i$ slices.

After the backward propagation, each GPU holding two or more slices should invoke multiple `reduce-scatter` communications to synchronize the gradients, along with multiple `all-gather` communications to retrieve the updated models. Malleus automatically identifies these GPUs and handles the ordering of communication calls to avoid deadlocks.

Model Migration. Once the parallelization plan is adjusted, we must migrate the model slices to meet the new plan. For each layer, we locate the source and destination of each model slice, summarizing the many-to-many communication. Then, we fuse the migration of different slices with the

[2]Since the memory consumption of model parameters increases w.r.t. the DP degree in hybrid parallel, we maintain the DP degree before and after the parallelization plan adjustment. It is also feasible to consider different DP degrees, e.g., by simply enumerating DP degrees within a small range.

batched-send-recv primitive for better efficiency. In addition, we pack the migration of multiple layers (4 layers by default) together to make full use of the network bandwidth.

However, if a failure occurs, some GPUs may not be responding and the model states owned by them become unavailable. Although it is possible to incorporate the idea of storing redundant model states [52], it increases memory consumption and leads to performance degradation. Thus, in such cases, we recover the training task by loading the latest model checkpoint onto the remaining GPUs and setting the straggling rates of unresponsive GPUs as infinite.

5.2 Device Management

Straggler Detection. During the training, some devices may occasionally lag behind and become stragglers. To detect such dynamic stragglers, the profiler in Malleus records the hardware efficiency based on CUDA events. In particular, we assess the computation and communication time cost of each GPU to distinguish the slower ones, and compute their GPU straggling rates by comparing them with the normal ones. Besides, we add a threshold for communication calls during training in order to detect failures.

Elastic Scaling. As introduced in §4.2, Malleus strategically removes heavy stragglers by assigning zero layers. However, these GPUs could be back to normal or become light stragglers later. Thus, instead of removing these GPUs permanently, we maintain them as standby devices, and periodically conduct micro-benchmarks to assess their GPU straggling rates. During each time of re-planning, our planning algorithm is able to adaptively determine whether there are removed GPUs to be involved as well as whether there are new heavy stragglers to be removed. By doing so, Malleus supports elastic scaling of involved GPUs during the training.

5.3 Re-planning with Overlapping

When any of the GPU straggling rates have changed considerably, Malleus triggers a re-planning process to accommodate the dynamicity, involving the execution of the planning algorithm to derive a new parallelization plan and the migration of model states. However, although the time cost of our planning algorithm is not substantial (around 10-30 seconds in our experiments), it still leads to non-negligible idle periods if we halt the training task during the planning. To cope with this problem, we devise an asynchronous re-planning mechanism — instead of leaving the GPUs idle, we continue training with the current parallelization plan, and execute the planning algorithm concurrently. In practice, we find that the planning finishes within one training step, achieving satisfactory overlapping. Although the model migration cannot be overlapped, it only takes a short (around 1-5 seconds in our experiments), which is acceptable.

6 Implementation

Malleus is designed to adapt to the dynamic stragglers, featuring a series of non-uniform partitioning and real-time adjustments in the parallelization plans. To fulfill this aim, we develop a prototype hybrid parallel training system for Malleus, consisting of 76K LoC in C++/CUDA and 6.4K LoC in Python. To implement the planning algorithm, we use the PuLP [1] and Pyomo [6] libraries to solve the ILP and MINLP problems, respectively. Our prototype system is particularly optimized for LLM training, with communication primitives implemented with NCCL [41] and computation kernels accelerated via libraries such as FlashAttention [8, 9], cuBLAS [39], and cutlass [40], matching the performance of Megatron-LM when there are no stragglers (evaluated in §7.2). Note that our implementation and evaluation focus on LLMs due to their huge model sizes and the demand of training with massive GPUs, whilst the proposed designs for parallelization planning and malleableized training are applicable to more forms of deep learning models. We leave them as potential future extensions.

7 Experimental Evaluation

7.1 Experimental Setup

Hardware Environments. We conduct all experiments on 8 GPU servers equipped with 8×A800 (80G) GPUs, totaling 64 GPUs. The GPUs within the same server are connected via NVLink with a bandwidth of 400GB/s, and the servers are connected via InifiBand with a bandwidth of 200GB/s.

Workloads. We consider three LLMs in the LLaMA architecture [53] with 32B, 70B, and 110B parameters, respectively. We train the 32B model over 32 GPUs and the other two models over 64 GPUs. The context length is set as 4K following most open-sourced LLMs. The global batch size (i.e., B) is set as 64 by default.

Baselines. To the best of our knowledge, none of the existing hybrid parallel training frameworks address the dynamic straggler problem. Thus, we compare Malleus with two prestigious LLM training frameworks: (1) Megatron-LM, a powerful hybrid parallel training framework that integrates DP, TP (empowered by sequence parallel [24]), and PP; (2) DeepSpeed, which utilizes the ZeRO-3 optimizer [44] to scatter model states across the devices (a.k.a. Fully Sharded Data Parallel [62]) and requires gathering model parameters for each layer in both forward and backward propagation.

Straggler Simulation. As analyzed in recent studies [22, 51], there are various kinds of root causes that lead to stragglers. It is difficult to develop a benchmark for reproducing these root causes since they are hard to control. Thus, we control the dynamic straggler patterns by simulation for all competitors to achieve fair comparison. To be specific, we launch extra computing processes on some GPUs to make them straggling, and we consider three levels of stragglers

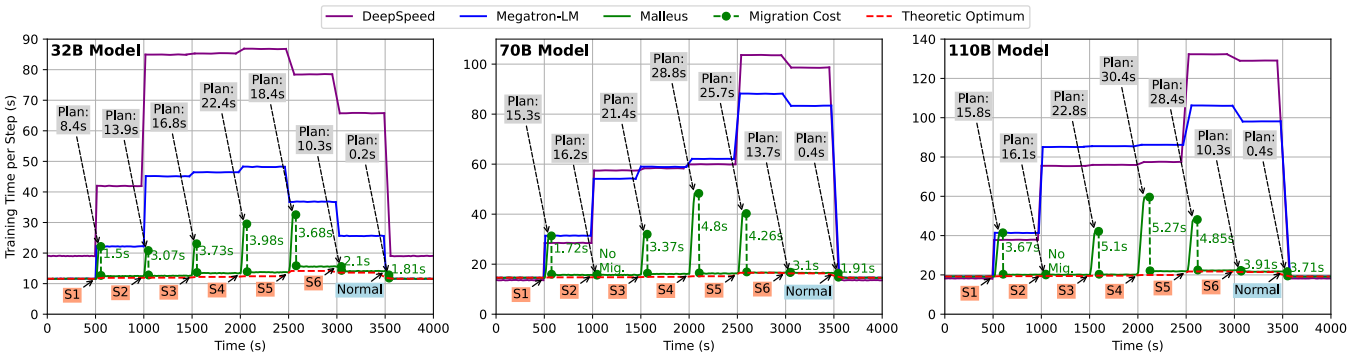

Figure 7. End-to-end evaluation on a trace consisting of six straggler situations ("Normal" indicates there are no stragglers). The x-axis represents the trace and the y-axis represents the running time per step (in seconds). During each time of re-planning of Malleus, the time cost of planning (highlighted in background gray color) is overlapped by the training, and the time cost of migration is provided (in green).

Table 2. Averaged running time per step (in seconds) under the straggler situations in Figure 7. The values in parentheses represent the improvement achieved by Malleus compared to the baselines. "Theoretic Opt." indicates the theoretic optimum, which is provided as reference. "Avg. Improv." indicates the average improvement measured in terms of geometric mean.

		Normal	S1	S2	S3	S4	S5	S6	Avg. Improv.
32B	DeepSpeed	19.0	42.0 (3.36×)	84.9 (6.73×)	85.3 (6.36×)	86.8 (6.38×)	78.5 (5.03×)	65.8 (4.67×)	5.28×
	Megatron-LM	11.6	22.2 (1.77×)	45.1 (3.58×)	46.4 (3.46×)	48.2 (3.54×)	36.8 (2.35×)	25.6 (1.81×)	2.63×
	Malleus	11.6	**12.5**	**12.6**	**13.4**	**13.6**	**15.6**	**14.1**	-
	Theoretic Opt.	-	11.9	11.9	12.2	12.4	14.2	13.6	-
70B	DeepSpeed	13.6	28.5 (1.81×)	57.4 (3.65×)	58.3 (3.62×)	59.9 (3.67×)	103.6 (6.24×)	98.6 (5.97×)	3.85×
	Megatron-LM	14.6	31.3 (1.99×)	54.2 (3.45×)	58.9 (3.66×)	62.1 (3.81×)	88.1 (5.30×)	83.3 (5.05×)	3.70×
	Malleus	14.6	**15.7**	**15.7**	**16.1**	**16.3**	**16.6**	**16.5**	-
	Theoretic Opt.	-	14.7	14.7	14.9	15.2	16.5	16.3	-
110B	DeepSpeed	18.2	37.9 (1.81×)	75.4 (3.75×)	76.0 (3.76×)	77.4 (3.55×)	132.3 (5.93×)	129.0 (5.94×)	3.84×
	Megatron-LM	19.2	41.4 (2.06×)	85.1 (4.23×)	85.4 (4.22×)	86.1 (3.95×)	106.1 (4.75×)	98.1 (4.52×)	3.82×
	Malleus	19.2	**20.1**	**20.1**	**20.2**	**21.8**	**22.3**	**21.7**	-
	Theoretic Opt.	-	19.4	19.4	19.6	20.0	21.7	21.5	-

by launching 1-3 processes, indicated as level-1, -2, and -3 stragglers, respectively. We generate a trace containing six straggler situations to simulate diverse scenarios: (S1) one level-1 straggler; (S2) one level-3 straggler; (S3) one level-1 straggler and one level-3 straggler, residing in different nodes; (S4) one level-1 straggler, one level-2 straggler, and one level-3 straggler, residing in different nodes; (S5) eight level-1 stragglers on the same node and one level-2 straggler on another node; (S6) eight level-1 stragglers on the same node. The generated trace consists of both GPU- and node-granular straggling situations, and contains the transitions where straggler appears or disappears. Therefore, it simulates the dynamic straggler problems in real-world scenarios well.

Protocols. We focus on the training time of each competitor under various straggler situations. For the baselines, we tune their configuration for each training task to achieve the best performance. Malleus adopts the same 3D parallel configuration as Megatron-LM when there are no stragglers, and switches to the parallelization plans generated by our planning algorithm when stragglers exist.

7.2 End-to-end Evaluation

We first conduct experiments under the six straggler situations. Figure 7 presents how the efficiency of each competitor changes when there is a shift in the stragglers, and Table 2 lists the running time in detail. These empirical results demonstrate that Malleus consistently achieves the best performance under all straggler situations for all models.

Individual Evaluation. To begin with, we focus on the results in Table 2. Both Megatron-LM and DeepSpeed suffer from the existence of stragglers, leading to significant performance degradation. For instance, when training the 110B model under the most severe straggler situations (S5), their running time bumps by 5.52× (from 19.2 to 106.1 seconds per step) and 7.27× (from 18.2 to 132.3 seconds per step), respectively. Even under the mildest straggler situations (S1), the performance reduction is still around 2×, which is quite

Table 3. The ratios of time cost of training with stragglers to that of training without stragglers, where R_{actual} is the actual ratio computed by the values in Table 2, R_{opt} is the theoretically optimal ratio, and R_{est} is the ratio estimated by our planning algorithm.

		R_{actual}	R_{opt}	$1-\frac{R_{\text{opt}}}{R_{\text{actual}}}$	R_{est}	$1-\frac{R_{\text{est}}}{R_{\text{actual}}}$
32B	S1	1.08	1.03	4.63%	1.06	1.85%
	S2	1.08	1.03	4.63%	1.06	1.85%
	S3	1.16	1.05	9.48%	1.13	2.58%
	S4	1.17	1.07	9.32%	1.18	0.00%
	S5	1.34	1.22	8.95%	1.37	-2.24%
	S6	1.22	1.17	4.10%	1.20	1.64%
70B	S1	1.08	1.01	6.48%	1.03	4.63%
	S2	1.08	1.01	6.48%	1.03	4.63%
	S3	1.10	1.02	7.27%	1.04	5.45%
	S4	1.11	1.04	6.30%	1.04	6.30%
	S5	1.14	1.13	0.88%	1.15	-0.88%
	S6	1.13	1.12	0.88%	1.13	0.00%
110B	S1	1.05	1.01	3.81%	1.03	1.90%
	S2	1.05	1.01	3.81%	1.03	1.90%
	S3	1.05	1.02	2.85%	1.05	0.00%
	S4	1.14	1.04	8.77%	1.08	3.70%
	S5	1.16	1.13	2.59%	1.17	-0.01%
	S6	1.13	1.12	0.88%	1.13	0.00%

unsatisfactory since the other 63 GPUs are not stragglers. In contrast, the performance reduction of Malleus is merely 1.05-1.16× (from 19.2 to 20.1-22.3 seconds per step). It validates that Malleus is resilient to various straggler situations — by adjusting the parallelization plan adaptively, it is capable of harnessing the stragglers and thereby maintaining a high performance. Eventually, Malleus outperforms Megatron-LM and DeepSpeed by up to 5.30× and 6.73×, respectively. And the strength of Malleus is consistent over the three models, provisioning 2.63-5.28× of speed up on average.

Besides, we find that DeepSpeed is more sensitive to stragglers — when there are no stragglers, it runs a bit faster than Malleus and Megatron-LM on the 70B and 110B model, whilst gradually surpassed when the straggler situation becomes more severe. This is not surprising since the ZeRO-3 optimizer needs to gather model parameters for each layer, which is globally synchronous by nature. Instead, for hybrid parallel approaches, only GPUs within the same TP group need to synchronize per layer, so the idle periods are shorter compared with DeepSpeed. As a result, hybrid parallel is a better fit for straggler-resilient training of large-scale models.

Adaption to Dynamic Stragglers. Next, we focus on Figure 7, which examines the ability of adaption to dynamic stragglers. Megatron-LM and DeepSpeed have poor ability to handle dynamic stragglers, leading to varying performance when facing diverse straggler situations. On the contrary, Malleus can automatically determine the stragglers, deduce a new parallelization plan for better efficiency, and migrate the model states in real time. More importantly, the asynchronous re-planning mechanism perfectly hides the planning time by overlapping, so the only overhead is brought by model migration, which is negligible (around 5 seconds or shorter). Consequently, Malleus is superior in handling dynamic stragglers through malleableizing the parallelization plans on the fly, improving the stability and straggler-resilience in the training of large-scale models.

Table 4. Case studies of parallelization plans. Straggling GPUs are highlighted in red. Groups after splitting or containing stragglers are highlighted in blue background color.

110B under S4 ($x_0 = 5.42$, $x_8 = 3.75$, $x_{16} = 2.57$)				
$m_1 = 33$ (8 stages)	x_7 $l_{1,1} = 2$	x_{15} $l_{1,2} = 2$	x_{23} $l_{1,3} = 2$	$x_1 \sim x_4$ $l_{1,4} = 10$
	$x_9 \sim x_{12}$ $l_{1,5} = 11$	$x_{17} \sim x_{20}$ $l_{1,6} = 11$	$x_{40} \sim x_{47}$ $l_{1,7} = 21$	$x_{56} \sim x_{63}$ $l_{1,8} = 21$
$m_2 = 31$ (6 stages)	x_5, x_6 $l_{2,1} = 4$	x_{13}, x_{14} $l_{2,2} = 5$	x_{21}, x_{22} $l_{2,3} = 5$	
	$x_{32} \sim x_{39}$ $l_{2,4} = 22$	$x_{48} \sim x_{55}$ $l_{2,5} = 22$	$x_{24} \sim x_{31}$ $l_{2,6} = 22$	

32B under S5 ($x_0 \sim x_7 = 2.62$, $x_8 = 3.8$)				
$m_1 = 7$ (4 stages)	x_2, x_1 $l_{1,1} = 15$	x_4, x_3 $l_{1,2} = 17$	x_0, x_5 $l_{1,3} = 17$	x_{15} $l_{1,4} = 11$
$m_2 = 17$ (4 stages)	x_6, x_7 $l_{2,1} = 7$	$x_{20} \sim x_{21}$ $l_{2,2} = 17$	$x_{26} \sim x_{27}$ $l_{2,3} = 18$	$x_{10} \sim x_9$ $l_{2,4} = 18$
$m_3 = 20$ (4 stages)	$x_{16} \sim x_{17}$ $l_{3,1} = 15$	$x_{22} \sim x_{23}$ $l_{3,2} = 15$	$x_{28} \sim x_{29}$ $l_{3,3} = 15$	$x_{12} \sim x_{11}$ $l_{3,4} = 15$
$m_4 = 20$ (4 stages)	$x_{18} \sim x_{19}$ $l_{4,1} = 15$	$x_{24} \sim x_{25}$ $l_{4,2} = 15$	$x_{30} \sim x_{31}$ $l_{4,3} = 15$	$x_{14} \sim x_{13}$ $l_{4,4} = 15$

Comparison with Theoretic Optimum. Suppose there are N GPUs in total and n of them are stragglers with rates $\{x_1, \cdots, x_n\}$. Theoretically speaking, if the hardware ability (e.g., TFLOPs) is inversely proportional to the straggling rates, then the optimal ratio of the time cost of running with stragglers to that of running without stragglers should be $N/((N-n)+\sum_{i=1}^{n} 1/x_i)$. As shown in Table 2, the performance loss of Malleus compared with the theoretic optimum is within 10% under all situations and even within 5% in more than half of the cases, verifying that the performance of Malleus is very close to the theoretic optimum. In addition, we also present the estimated performance obtained by our planning algorithm (i.e., via the solution to Eq. (1)), which shows that our cost model is accurate — the estimated errors are not higher than 6.3% in all experiments. Undoubtedly, this is vital to the deduction of parallelization plans.

7.3 Case Studies and Ablation Studies

Case Studies. Table 4 presents two parallelization plans discovered by Malleus. When training the 110B model under the S4 situation, Malleus eliminates stragglers on all three nodes and forms new GPU groups with 1, 2, and 4 GPUs, respectively, summing up to 9 groups in total. Malleus distributes these 9 groups (in different sizes) to two pipelines to

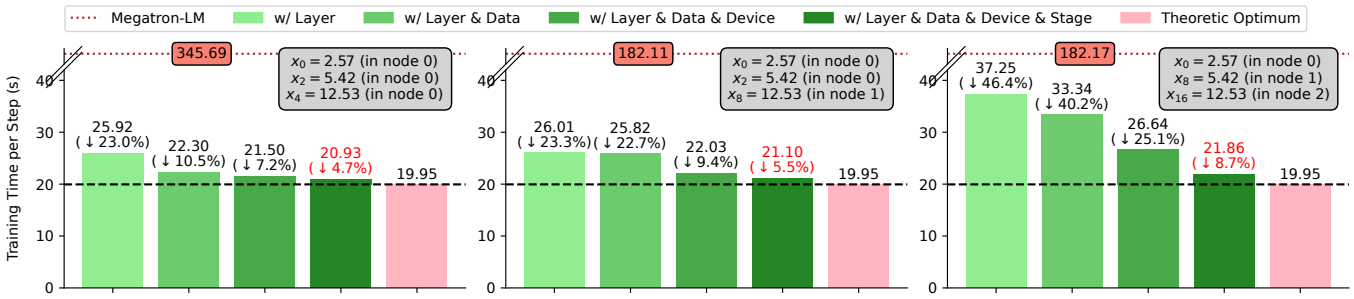

Figure 8. Effectiveness of the non-uniform partitioning on each dimension, evaluated on the 110B model. The ratios represents the gap from the theoretic optimum, computed by $1 - T_{\text{opt}}/T_{\text{actual}}$, where T_{opt} and T_{actual} denote the theoretic optimal time and actual running time.

achieve load balancing. When training the 32B model under the S5 situation, Malleus eliminates the level-2 straggler on the second node and retains all level-1 stragglers on the first node. By assigning these stragglers fewer layers (x_6, x_7) and less data ($x_0 \sim x_5$), the overall training time is minimized.

Ablation Studies. As introduced in §3.1, our parallelization features four types of non-uniform partitioning. We assess the effectiveness of them using the 110B model. To better demonstrate the agility of Malleus in dealing with complex straggler situations, we further introduce a severe straggler in level-8 (i.e., running 8 extra processes). Depicted in Figure 8, we experiment with three stragglers in level-1, level-3, and level-8, appearing on one node, two nodes, and three nodes, respectively.

When the stragglers are on only one node, we find that the non-uniform partitioning of layers and data (solved by the lower-level problem) can greatly alleviate the impact of stragglers, with a gap of only about 10% from the theoretic optimum. For instance, our planning algorithm strikes a good balance by strategically assigning only 2 layers to the slowest group (containing all three stragglers) whilst evenly distributing the remaining 78 layers to the other three non-straggling groups in the same pipeline.

However, when the stragglers appear on multiple nodes, adjusting the partitioning of layers and data alone is no longer sufficient to produce satisfactory outcomes, with a significant gap of 20-40% from the theoretic optimum. At this time, the introduction of non-uniform partitioning of devices and stages (solved by the upper-level problem) becomes particularly important. By isolating the stragglers and orchestrating pipelines in diverse forms, we can make better use of the non-straggling GPUs, reducing the gap from the theoretic optimum to at most 8.7%.

8 Related Works

Heterogeneous Training. Due to the rising concerns of the GPU shortage problem [49, 58], several approaches have been developed for distributed training over heterogeneous types of GPUs [18, 25, 48, 57, 59]. Our work differs from them for two reasons. Firstly, these works focus on static heterogeneity — the efficiency differences among the GPUs do not change during training. In contrast, our work considers a more complex scenario of dynamic stragglers, where the efficiency variation is dynamic and unforeseeable. Secondly, these works assume the GPUs within the same node are of the same type and thereby provision identical hardware efficiency, whilst our work addresses the straggler problem at the nuanced, per-GPU granularity.

Elastic Training. Elastic training is an essential technique to handle failures or device defectiveness. Checkpointing is the most common choice for elastic training. For instance, TorchElastic [3] and HorovodElastic [2] support restarting and loading the model checkpoints at failure. Gemini [56] studies how to accelerate failure recovery. There are also approaches that focus on resilience when the failures are informed or detected. For instance, Varuna [4] and Bamboo [52] consider training over a kind of preemptible cloud instances called spot instances, whilst Oobleck [17] and SlipStream [12] detect the failures in dedicated clusters. These approaches typically re-configure the training task before or upon failures. However, they are orthogonal to our work since straggler mitigation is not their focus. Elasticity is also important in job scheduling to improve cluster usages [16, 26, 34, 43], yet these works mainly focus on data parallel and do not consider the straggler-resilience of a single job.

Besides, these approaches primarily tackle node-level failures and recoveries, rather than considering the fine-grained removal or addition of individual GPUs. In contrast, Malleus supports elasticity at the GPU granularity. It is noteworthy that node failure problem is essentially a subset of GPU failure problem (i.e., the simultaneous failure of 8 GPUs means a node failure), and the GPU failure problem could be regarded as a special case of straggler problem as well (i.e., resulting in a straggling rate of infinity). The parallelization planning process used in Malleus can be effectively applied to these scenarios by simply setting the straggling rate of completely failed GPU(s) to infinity. And thanks to the GPU-granular elasticity offered by Malleus, we can handle the failure of individual GPUs, whilst existing approaches need

to remove/replace the entire node. Hence, we believe that Malleus addresses a broader set of problems compared to previous elastic training approaches.

Relaxed Synchronization Protocols. Relaxing the synchronization protocol in data parallel for straggler mitigation has been explored for long. This line of research breaks the barrier of gradient/model synchronization to reduce idle periods of non-straggling devices. Notable efforts include asynchronous parallel [7, 10] and stale synchronous parallel [14, 20, 32, 61]. However, these works are developed for data parallel, and inevitably impact the model convergence. On the contrary, our work focuses on straggler-resilient hybrid parallel training of large-scale models without affecting the model convergence.

9 Conclusion

This work focuses on the straggler-resilience of hybrid parallel training for large-scale models. Specifically, we introduced Malleus, a hybrid parallel training framework that captures the dynamic stragglers at the nuanced, per-GPU granularity. We developed a parallelization planning algorithm that co-optimizes the non-uniform partitioning of GPU devices, pipeline stages, model layers, and training data, speeding up the training under various straggler situations. In response to the dynamicity in stragglers, we proposed a re-planning process that adjusts the parallelization plan and migrates model states on the fly. Empirical results show that Malleus can be on average 2.63-5.28× faster than existing training frameworks and handle the straggler dynamicity efficiently.

References

[1] 2009. Optimization with PuLP. https://coin-or.github.io/pulp/.

[2] 2019. Elastic Horovod. https://horovod.readthedocs.io/en/latest/elastic_include.html.

[3] 2023. Torch Distributed Elastic. https://pytorch.org/docs/stable/distributed.elastic.html.

[4] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys 2022)*. 472–487.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Annual Conference on Neural Information Processing Systems 2020 (NeurIPS 2020)*.

[6] Michael L Bynum, Gabriel A Hackebeil, William E Hart, Carl D Laird, Bethany L Nicholson, John D Siirola, Jean-Paul Watson, David L Woodruff, et al. 2021. *Pyomo-optimization modeling in python*. Vol. 67. Springer.

[7] Trishul M. Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*. USENIX Association, 571–582.

[8] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *CoRR* abs/2307.08691 (2023).

[9] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Annual Conference on Neural Information Processing Systems 2022 (NeurIPS 2022)*.

[10] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *26th Annual Conference on Neural Information Processing Systems 2012 (NeurIPS 2022)*. 1232–1240.

[11] Shaoduo Gan, Xiangru Lian, Rui Wang, Jianbin Chang, Chengjun Liu, Hongmei Shi, Shengzhuo Zhang, Xianghong Li, Tengxu Sun, Jiawei Jiang, Binhang Yuan, Sen Yang, Ji Liu, and Ce Zhang. 2021. BAGUA: Scaling up Distributed Learning with System Relaxations. *Proc. VLDB Endow.* 15, 4 (2021), 804–813.

[12] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. 2024. SlipStream: Adapting Pipelines for Distributed Training of Large DNNs Amid Failures. *CoRR* abs/2405.14009 (2024).

[13] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2016. Addressing the straggler problem for iterative convergent parallel ML. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SOCC 2016)*. 98–111.

[14] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Annual Conference on Neural Informatio Processing Systems 2013 (NeurIPS 2013)*. 1223–1231.

[15] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Annual Conference on Neural Information Processing Systems 2019 (NeurIPS 2019)*. 103–112.

[16] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2021)*. 721–739.

[17] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. 2023. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP 2023)*. 382–395.

[18] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. 2022. Whale: Efficient Giant Model Training over Heterogeneous GPUs. In *2022 USENIX Annual Technical Conference (ATC 2022)*. 673–688.

[19] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems 2019 (MLSys 2019)*.

[20] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware Distributed Parameter Servers. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 2017)*. 463–478.

[21] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. 2018. SketchML: Accelerating Distributed Machine Learning with Data Sketches. In *Proceedings of the 2018 ACM International Conference on Management of Data (SIGMOD 2018)*. ACM, 1269–1284.

[22] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou,

Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10, 000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 2024)*. 745–760.

[23] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *CoRR* abs/2001.08361 (2020).

[24] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing Activation Recomputation in Large Transformer Models. *CoRR* abs/2205.05198 (2022).

[25] Dacheng Li, Hongyi Wang, Eric P. Xing, and Hao Zhang. 2022. AMP: Automatically Finding Model Parallel Strategies with Heterogeneity Awareness. In *Annual Conference on Neural Information Processing Systems 2022 (NeurIPS 2022)*.

[26] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic Scheduling for Deep Learning Clusters. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys 2023)*. 835–850.

[27] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*. 583–598.

[28] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020), 3005–3018.

[29] Haibin Lin, Hang Zhang, Yifei Ma, Tong He, Zhi Zhang, Sheng Zha, and Mu Li. 2019. Dynamic Mini-batch SGD for Elastic Distributed Training: Learning in the Limbo of Resources. *CoRR* abs/1904.12043 (2019).

[30] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual Instruction Tuning. In *Annual Conference on Neural Information Processing Systems 2023 (NeurIPS 2023)*.

[31] Meta. 2024. Introducing Meta Llama 3: The most capable openly available LLM to date. https://ai.meta.com/blog/meta-llama-3/.

[32] Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, and Bin Cui. 2021. Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce. In *Proceedings of the 2021 ACM International Conference on Management of Data (SIGMOD 2021)*. 2262–2270.

[33] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2022. Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism. *Proc. VLDB Endow.* 16, 3 (2022), 470–479.

[34] Zizhao Mo, Huanle Xu, and Chengzhong Xu. 2024. Heet: Accelerating Elastic Training in Heterogeneous Deep Learning Clusters. In *Architectural Support for Programming Languages and Operating Systems 2020 (ASPLOS 2020)*. 499–513.

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

[36] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-Efficient Pipeline-Parallel DNN Training. In *International Conference on Machine Learning 2021 (ICML 2021)*, Vol. 139. 7937–7947.

[37] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *International Conference for High Performance Computing, Networking 2021 (SC 2021)*. 58.

[38] Xiaonan Nie, Yi Liu, Fangcheng Fu, Jinbao Xue, Dian Jiao, Xupeng Miao, Yangyu Tao, and Bin Cui. 2023. Angel-PTM: A Scalable and Economical Large-scale Pre-training System in Tencent. *Proc. VLDB Endow.* 16, 12 (2023), 3781–3794.

[39] NVIDIA. 2024. cuBLAS. https://docs.nvidia.com/cuda/cublas/.

[40] NVIDIA. 2024. cutlass. https://github.com/NVIDIA/cutlass/.

[41] NVIDIA. 2024. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[42] Andrew Or, Haoyu Zhang, and Michael J. Freedman. 2020. Resource Elasticity in Distributed Deep Learning. In *Proceedings of Machine Learning and Systems 2020 (MLSys 2020)*.

[43] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2021)*.

[44] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2020)*. 20.

[45] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018).

[46] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019).

[47] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zheng, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *CoRR* abs/2201.11990 (2022).

[48] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2020. AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators. In *IEEE International Symposium on High Performance Computer Architecture, 2020 (HPCA 2020)*. 342–355.

[49] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. 2024. ML Training with Cloud GPU Shortages: Is Cross-Region the Answer?. In *Proceedings of the 4th Workshop on Machine Learning and Systems, EuroMLSys 2024*. 107–116.

[50] Jakub Tarnawski, Deepak Narayanan, and Amar Phanishayee. 2021. Piper: Multidimensional Planner for DNN Parallelization. In *Annual Conference on Neural Information Processing Systems 2021 (NeurIPS 2021)*. 24829–24840.

[51] The Imbue Team. 2024. From bare metal to a 70B model: infrastructure set-up and scripts. https://imbue.com/research/70b-infrastructure/.

[52] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2023)*. 497–513.

[53] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes,

Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023).

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Annual Conference on Neural Information Processing Systems 2017 (NeurIPS 2017)*. 5998–6008.

[55] Yujie Wang, Youhe Jiang, Xupeng Miao, Fangcheng Fu, Xiaonan Nie, and Bin Cui. 2023. Improving Automatic Parallel Training via Balanced Memory Workload Optimization. *CoRR* abs/2307.02031 (2023).

[56] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP 2023)*. 364–381.

[57] Si Xu, Zixiao Huang, Yan Zeng, Shengen Yan, Xuefei Ning, Haolin Ye, Sipei Gu, Chunsheng Shui, Zhezheng Lin, Hao Zhang, Sheng Wang, Guohao Dai, and Yu Wang. 2024. HetHub: A Heterogeneous distributed hybrid training system for large-scale models. *CoRR* abs/2405.16256 (2024).

[58] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. 2023. SkyPilot: An Intercloud Broker for Sky Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2023)*. 437–455.

[59] Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. 2024. HAP: SPMD DNN Training on Heterogeneous GPU Clusters with Automated Program Synthesis. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys 2024)*. 524–541.

[60] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. 2022. MiCS: Near-linear Scaling for Training Gigantic Model on Public Cloud. *Proc. VLDB Endow.* 16, 1 (2022), 37–50.

[61] Xing Zhao, Aijun An, Junfeng Liu, and Bao Xin Chen. 2019. Dynamic Stale Synchronous Parallel Distributed Training for Deep Learning. In *IEEE International Conference on Distributed Computing Systems (ICDCS 2019)*. 1507–1517.

[62] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (2023), 3848–3860.

[63] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*. 559–578.

A More Experiment Results

A.1 Effectiveness of the Cost Model

As discussed in §7.2, the results in Table 4 show that our cost model is accurate. To further examine the effectiveness of our cost model, i.e., whether it is useful in helping us deduce the optimal plan, we conduct a representative testbed using the 32B model. In particular, we employ a fixed hybrid parallel strategy with DP, PP, and TP degrees of 4, 2, and 2, respectively, and we further decrease the sequence length from 4K, as used in previous experiments, to 1K to bypass all memory constraints. Additionally, we increase the global batch size from 64 to 512 and keep the micro-batch size as 1, allowing for a more refined granularity of data assignment and, consequently, a more rigorous validation for the precision of our cost model. In the experiment, we introduce a level-1 straggler, without the need of isolating heavy stragglers or non-uniform stages across the pipelines. Due to symmetry, we can actually enumerate all possibilities by traversing the layers and data allocated to the straggler GPU and measure the end-to-end performance for each partitioning.

For layer partitioning, as each pipeline only involves two stages, after enumerating the number of layers l allocated to the stage containing the straggler, the remaining stage in the pipeline will be allocated $60 - l$ layers for sure, while for the other pipelines composed of non-straggling GPUs, the optimal layer partitioning remains evenly assigning 30 layers to each stage. As For data partitioning, since DP is 4, we only need to enumerate the number of micro-batches m allocated to the pipeline containing the straggler, and the remaining three pipelines, being completely isomorphic, will evenly distribute the remaining $512 - m$ micro-batches (ideally, without considering the integer constraint of micro-batches, each normal pipeline should be assigned $(512 - m)/3$ micro-batches).

Following the aforementioned approach, we first enumerate all layer partitioning possibilities and select the optimal from them. Based on this, we again enumerate all data partitioning possibilities. At each step of enumeration, we test the actual profiling time on norm (i.e., non-straggling) GPUs, profiling time on the straggling GPU, and the overall end-to-end time, and compare them with the estimated time given by our cost model. As shown in Figure 9, it can be observed that our cost model well approximates the actual running time, and the final layer and data partitioning solutions also coincide with the optimal solution found through actual end-to-end enumeration. This demonstrates that our cost model can effectively identify the optimal load balancing point, achieving the optimal solution in practical.

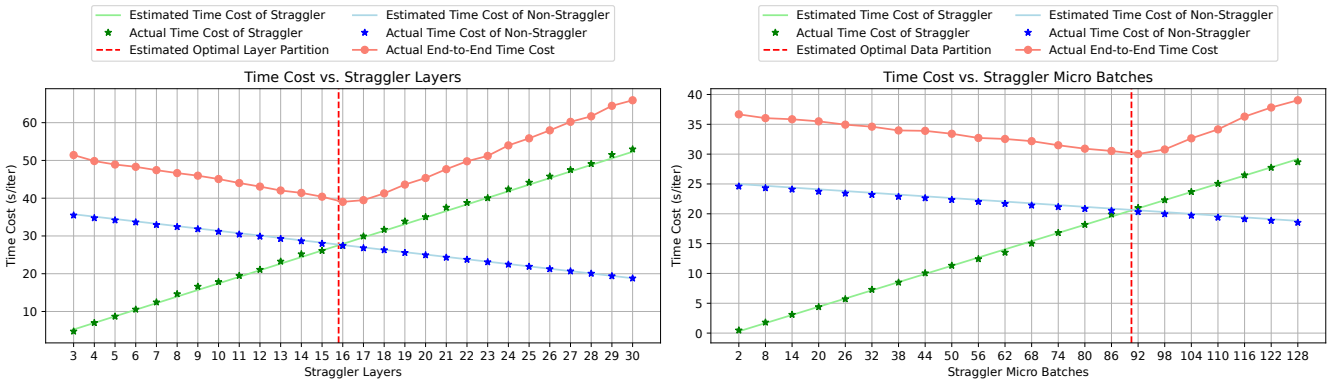

Figure 9. Enumeration on layer and data partitioning. The results given by our cost model coincide precisely with the optimal solution towards the final load balancing.

A.2 Scalability of Parallelization Planning

According to the experiment results in §7.2, the parallelization planning process can be fully overlapped by the training of 1 iteration. In this experiment, we wish to examine the scalability of our parallelization planning algorithm to more GPUs. Unfortunately, due to the high expense of GPUs, we cannot evaluate the training efficiency of Malleus over more GPUs. Thus, we only focus on the time cost of planning here. Specifically, we assume a total of 1024 GPUs (128 8-GPU nodes) are used to train the 110B model. Meanwhile, we assume the global batch size is linearly scaled to 1024 (originally 64 in §7), constituting each batch with 4 million tokens, which is a reasonable configuration in LLM training. We further assume there are 32 stragglers (approximately 3% of the cluster). Below, we discuss the breakdown of the time taken for each step (shown in Table 5) of our parallelization planning algorithm in this scenario:

- **Enumeration overhead of GPU grouping (§4.3.1)**: The overhead of this part is negligible (0.01 second). Although this part involves the enumeration of group splitting, each enumeration only requires using Theorem 2 to calculate a ratio, and

the number of enumerations is not large. Specifically, we began with running four processes to solve the optimal strategy for TP limits of 1, 2, 4, and 8, respectively. With the overall TP degree fixed, we then considered how to split the TP groups for the stragglers. For example, when TP degree is 1, there is no group splitting, and when TP degree is 8, a node with straggler(s) needs to enumerate up to 6 times of group splitting (as discussed in §4.3.1 and detailed in Appendix B.7). With 32 straggler GPUs across a maximum of 32 nodes, we only need to enumerate 32 times. Putting them together, there are only a few hundreds of enumerations.

- **Pipeline division overhead (first half of §4.3.2)**: This part has the largest overhead (51.23 seconds) due to the complexity of the MINLP problem and its correlation with the DP degree as well as the number of straggling TP groups. However, it can still finish within reasonable time even for more than one thousand GPUs.
- **Optimal group ordering overhead (second half of §4.3.2)**: After solving the MINLP problem, there are 16 straggling groups appeared in 5 pipelines (the actual result solved by MINLP was $2 + 3 + 3 + 4 + 4 = 16$). Then, our parallelization planning algorithm needs to enumerate the permutations within these 5 pipelines (up to 24 enumerations per pipeline as mentioned in §4.3.2 of our manuscript) and solve Eq. 2 for each permutation in order to achieve the best one. Since these 5 pipeline permutations are orthogonal and do not affect each other, we computed up to $24 \times 5 = 120$ ILPs in total, with multi-threading optimization, resulting in an overhead of 0.59s (a single ILP took about 0.40s).
- **Work (layer and data) assignment overhead (§4.2)**: At this point, we only need to solve a few ILP problems in Eq. 2 and Eq. 3, with the ones in Eq. 2 being orthogonal and solved using multi-threading optimization. This part took 0.75s.

Table 5. Time taken for each part of our algorithm in the 64-GPU S3 scenario and the simulated 1024-GPU scenario.

	GPU Grouping	Pipeline Division	Group Ordering	Work Assignment
1024 GPUs	0.01s	51.23s	0.59s	0.75s
64 GPUs	0.01s	22.61s	0.07s	0.11s

Table 5 shows that although the overhead increases, Malleus can still complete planning within a minute. Theoretically speaking, when training with a global batch size of 1024 on 1024 GPUs, each iteration's time is similar to that of training with a global batch size of 64 on 64 GPUs. Therefore, we can complete planning within 1-2 iterations, meaning that we can obtain a new optimal parallelization plan and perform migration within a maximum of 2 iterations. Consequently, our parallelizatio planning algorithm has a sound scalability.

B Proofs

In this section, we provide proofs for the theorems and some omitted deduction in our paper.

B.1 Proof for Theorem 1

In Section 4.3.1, we propose to partition GPUs with similar performance to the same group according to Theorem 1. Below we provide the proof for it.

Theorem 1. *Suppose there are n GPUs in a node with straggling rates $\{x_1, \cdots, x_n\}$, and we need to partition them into n/k groups (each with k GPUs). Denote $\{i_1, \cdots, i_n\}$ as the ordering satisfying $x_{i_1} \geq \cdots \geq x_{i_n}$. Then, the best grouping result is $\{\{x_{i_1}, \cdots, x_{i_k}\}, \{x_{i_{k+1}}, \cdots, x_{i_{2k}}\}, \cdots, \{x_{i_{n-k+1}}, \cdots, x_{i_n}\}\}$.*

Proof. The proof of this theorem only leverages the following two simple statements.

Statement 1. ***The slowest straggler in the tensor parallel group dominates the time of the whole group.*** *Assume each GPU group consists of k GPUs, define $T(y_1, \ldots, y_{\frac{n}{k}})$ as the minimum cost when the straggling rate of group i is y_i. Denote X_i as the i-th GPU with straggling rate x_i and Y_i as the i-th GPU group. Then we have $y_i = \max_{X \in Y_i} \{x\}$. Note here we drop the ρ discussed in Section 4.2 because all the groups have the same amount of GPUs (equals to k).*

Statement 2. ***If a single straggler worsens, it is impossible to find a lower minimum cost.*** *For any given i, if $y_i \geq y_i'$, we have $T(y_1, \ldots, y_{\frac{n}{k}}) \geq T(y_1, \ldots, y_{i-1}, y_i', y_{i+1}, \ldots, y_{\frac{n}{k}})$. Note that no other assumptions are made about the properties of the function T.*

For any GPU groups combination $(Y_1, \ldots, Y_{\frac{n}{k}})$, where $Y_i = (X_{i,1}, \ldots, X_{i,k})$, we guarantee that $\max_{X \in Y_1} x \geq \ldots \geq \max_{X \in Y_{\frac{n}{k}}} x$ and $x_{i,1} \geq \ldots \geq x_{i,k}$ to ensure the representation is free of rotation. Then, we could denote the grouping of GPUs as a single array

$$G = \begin{cases} Y_1 = \left(X_{1,1}, \ldots, X_{1,k}\right) \\ \cdots \\ Y_{\frac{n}{k}} = \left(X_{\frac{n}{k},1}, \ldots, X_{\frac{n}{k},k}\right) \end{cases} \implies \left(x_{1,1}, \ldots, x_{1,k}, x_{2,1}, \ldots, x_{\frac{n}{k},k}\right)$$

This sequence has a finite number of inverse pairs. We sequentially exchange these inverse pairs until the sequence is in descending order. Next, we will prove that each exchange of the inverse pair does not increase the eventual cost of T.

Assume that we are exchanging $X_{i,p}$ with $X_{j,q}$, where $x_{i,p} < x_{j,q}$. Let Y_i and Y_j be the groups before exchanging, Y_i' and Y_j' be the new groups after exchanging, we have

$$\begin{gathered} x_{i,p} < x_{j,q} \\ \implies i < j \text{ and } p \neq 1, \text{meaning } X_{i,p} \in Y_i' \\ \implies y_i = \max_{X \in Y_i} \{x\} = x_{i,1} \geq y_j = \max_{X \in Y_j} \{x\} = x_{j,1} \end{gathered}$$

Two situations may occur during the exchange process. If $q = 1$, we have $x_{i,p} < x_{j,1}$. According to Statement 2, the exchange will benefit T because

$$\begin{gathered} y_i' = \max_{X \in Y_i'} \{x\} = \max\left\{x_{i,1}, x_{j,1}\right\} = x_{i,1} = y_i \\ y_j' = \max_{X \in Y_j'} \{x\} = \max\left\{x_{j,2}, \ldots, x_{j,k}, x_{i,p}\right\} \leq \max\left\{x_{j,2}, \ldots, x_{j,k}, x_{j,1}\right\} = y_j \\ \implies T(y_1, \ldots, y_{\frac{n}{k}}) \geq T(y_1, \ldots, y_{j-1}, y_j', y_{j+1}, \ldots, y_{\frac{n}{k}}) \end{gathered}$$

Otherwise, if $q \neq 1$, meaning that $X_{j,q} \in Y_j'$, the exchange will have no effect

$$\begin{gathered} y_i' = \max_{X \in Y_i'} \{x\} = \max\left\{x_{i,1}, x_{j,q}\right\} = x_{i,1} = y_i \\ x_{i,p} < x_{j,q} \leq x_{j,1} \implies y_j' = \max_{X \in Y_j'} \{x\} = \max\left\{x_{j,1}, x_{i,p}\right\} = x_{j,1} = y_j \\ \implies T(y_1, \ldots, y_{\frac{n}{k}}) \text{ remains the same} \end{gathered}$$

Therefore, for any GPU groups combination $(Y_1, \ldots, Y_{\frac{n}{k}})$, after a finite number of inverse pair exchanges, it can be transformed into the grouping result mentioned in Theorem 1, which has no inverse pairs and therefore is the best grouping result among all. □

B.2 Proof for Theorem 2

In Section 4.3.1, we compare two possible grouping results according to Theorem 2. Below we provide the proof for it.

Theorem 2. *Suppose there are two different grouping results that consist of M' and M'' groups with straggling rates of $\{y'_1, \cdots, y'_{M'}\}$ and $\{y''_1, \cdots, y''_{M''}\}$, respectively. If we ignore the memory constraints in Eq. (1), and further assume the layer and training data assignments are not restricted to integers (i.e., $l_{i,j}, m_i$ in Eq. (1) can be any positive real numbers), then the minimum training time of the two grouping results satisfy $T'/T'' = (\sum_{i=1}^{M''} 1/y''_i)/(\sum_{i=1}^{M'} 1/y'_i)$.*

Proof. Denote M as the number of total groups, $(y_1, \ldots, y_M)$ as the straggling rate of all groups, L as the number of total layers and B as the number of training samples in one step. And let the running time of the j-th stage in the i-th pipeline for one micro-batch be $t_{i,j} = y_{i,j} \times l_{i,j} \times \tau(b)$, where b is the size of micro-batch. Then, for 1F1B pipeline, considering the heterogeneous layers assignment, the minimum cost of a training iteration for the i-th pipeline could be formulated as

$$
\begin{aligned}
T_i &= \min_{l_{i,1},\ldots,l_{i,\overline{PP}_i}} \left\{ (m_i - 1) \times \max_{1 \le j \le \overline{PP}_i} \left\{ t_{i,j} \right\} + \sum_{1 \le j \le \overline{PP}_i} t_{i,j} \right\} \\
&= \min_{l_{i,1},\ldots,l_{i,\overline{PP}_i}} \left\{ (m_i - 1) \times \max_{1 \le j \le \overline{PP}_i} \left\{ y_{i,j} \times l_{i,j} \right\} + \sum_{1 \le j \le \overline{PP}_i} y_{i,j} \times l_{i,j} \right\} \times \tau(b) \\
&\Longrightarrow T_i \approx \min_{l_{i,1},\ldots,l_{i,\overline{PP}_i}} \max_{1 \le j \le \overline{PP}_i} \left\{ y_{i,j} \times l_{i,j} \right\} \times m_i \times \tau(b), \text{ when } m_i >> \overline{PP}_i
\end{aligned}
$$

Since we have ignored the memory constraints in Eq. (1) and further assumed the layers assignment is not restricted to integers, given $l_{i,1} + \ldots + l_{i,\overline{PP}_i} = L$, the T_i could be reached when

$$
\begin{aligned}
\left(l_{i,1}, \ldots, l_{i,\overline{PP}_i} \right) &= \left(\frac{\frac{L}{y_{i,1}}}{\sum_{1 \le j \le \overline{PP}_i} \frac{1}{y_{i,j}}}, \ldots, \frac{\frac{L}{y_{i,\overline{PP}_i}}}{\sum_{1 \le j \le \overline{PP}_i} \frac{1}{y_{i,j}}} \right) \\
\Longrightarrow T_i &= \frac{1}{\sum_{1 \le j \le \overline{PP}_i} \frac{1}{y_{i,j}}} \times L \times m_i \times \tau(b)
\end{aligned}
$$

Then, the minimum cost of the whole system given current pipelines training cost $(T_1, \ldots, T_{\overline{DP}})$ should be

$$
T = \min_{m_1 \ldots, m_{\overline{DP}}} \max_{1 \le i \le \overline{DP}} \{T_i\} = \min_{m_1, \ldots, m_{\overline{DP}}} \max_{1 \le i \le \overline{DP}} \left\{ \frac{m_i}{\sum_{1 \le j \le \overline{PP}_i} \frac{1}{y_{i,j}}} \right\} \times L \times \tau(b)
$$

Still, under the assumption that the micro-batches assignment is not restricted to integers and $(m_1 + \ldots + m_{\overline{DP}}) \times b = B$, we can directly solve the problem

$$
\begin{aligned}
\left(m_1, \ldots, m_{\overline{DP}} \right) &= \left(\frac{\sum_{1 \le j \le \overline{PP}_i} \frac{\frac{B}{b}}{y_{1,j}}}{\sum_{1 \le 1 \le \overline{DP}} \sum_{1 \le j \le \overline{PP}_i} \frac{1}{y_{i,j}}}, \ldots, \frac{\sum_{1 \le j \le \overline{PP}_i} \frac{\frac{B}{b}}{y_{\overline{DP},j}}}{\sum_{1 \le 1 \le \overline{DP}} \sum_{1 \le j \le \overline{PP}_i} \frac{1}{y_{i,j}}} \right) \\
\Longrightarrow T &= \frac{\frac{B}{b} \times L \times \tau(b)}{\sum_{1 \le 1 \le \overline{DP}} \sum_{1 \le j \le \overline{PP}_i} \frac{1}{y_{i,j}}} = \frac{\frac{B}{b} \times L \times \tau(b)}{\sum_{1 \le i \le M} \frac{1}{y_i}}
\end{aligned}
$$

We could find out that T is independent of the layers and data assignment. And the ratio of the optimal training cost T', T'' between two grouping results will always be

$$
\frac{T'}{T''} = \frac{\sum_{1 \le i \le M'} \frac{1}{y'_i}}{\sum_{1 \le i \le M''} \frac{1}{y''_i}}
$$

□

B.3 Proof for Theorem 3

In Section 4.3.1, we propose to sort GPU groups within the same pipeline according to their group straggling rates when the groups have the same number of GPUs, which is based on Theorem 3. Below we provide the proof for it.

Theorem 3. *Suppose the groups assigned to the same pipeline have the same number of GPUs, then the best ordering of pipeline stages satisfies that the groups are in descending order w.r.t. the group straggling rates.*

Proof. For a pipeline with w stages, where each stage consists of an equal number of GPUs, let the straggling rate for the groups from the first to the last stage be denoted as $(y_1, \ldots, y_w)$. Assume the optimal layer assignment in this scenario is represented by $(l_1, \ldots, l_w)$. For a single GPU, let a_f represent the peak memory consumption of activations during forward propagation for one layer when the micro-batch size is 1. Similarly, let a_{f+b} denote the peak memory consumption of activations during both forward and backward propagation for one layer when the micro-batch size is 1, and let s be the memory consumption of the parameters, gradients, and optimization states for one layer. Given that the memory constraint is uniform, denoted as C, and assuming the memory consumption of non-uniform layers in the first and last stage (such as the embedding table) is negligible compared with a bunch of uniform layers, the constraints in Eq.(1) can be theoretically expressed as (please refer to Proposition B.4 for more details)

$$\forall j \in [1, w], l_j \times \left\{ b \times \left[a_f \times (w - j) + a_{f+b} \right] + s \right\} \leq C$$

From this expression, it is evident that the maximum number of layers is constrained by the stage number j. Therefore, let $(max\_l_1, \ldots, max\_l_w)$ represent the maximum number of layers for each stage. We have two following properties:

1. $\forall j \in [1, w], l_j \leq max\_l_j$.
2. $max\_l_1 \leq \ldots \leq max\_l_w$.

Assume there exists a group pair satisfying $y_p < y_q$, where $p < q$. If $l_p \leq l_q$, then by swapping the groups while maintaining the layers, the new solution becomes $(l_1, \ldots, l_p, \ldots, l_q, \ldots, l_w)$, with groups arrangement $(y_1, \ldots, y_q, \ldots, y_p, \ldots, y_w)$. We can prove that this will result in a better (or at least equal) training cost due to the objective function in Eq. (1) when other variables remain constant

$$\begin{gathered} y_p \times l_q < y_q \times l_q \\ y_q \times l_p \leq y_q \times l_q \\ \Longrightarrow \ max\left\{y_p \times l_q, y_q \times l_p\right\} \leq max\left\{y_p \times l_q, y_q \times l_p\right\} \end{gathered}$$

Otherwise, if $l_p > l_q$, we have

$$\begin{gathered} max\_l_p \geq l_p > l_q \\ max\_l_q \geq max\_l_p \geq l_p \end{gathered}$$

In this case, by swapping both groups and layers, we obtain another valid solution that still satisfies the memory constraints: $(l_1, \ldots, l_q, \ldots, l_p, \ldots, l_w)$ under the groups arrangement $(y_1, \ldots, y_q, \ldots, y_p, \ldots, y_w)$. Similarly, according to the objective function in Eq. (1), the new solution is as good as the previous one because $max\left\{y_p \times l_p, y_q \times l_q\right\} = max\left\{y_q \times l_q, y_p \times l_p\right\}$.

Thus, swapping any pair that satisfies $y_p < y_q$ and $p < q$ will enhance the overall performance. Consequently, the strategy that arranges the groups in total descending order of the straggling rate will be the most optimal among all configurations. □

B.4 Memory Cost Model (Deduction of $\mu_{i,j}(b), \nu_{i,j}(b), C_{i,j}$)

In this section, we provide more details about how to calculate the coefficients of the memory cost model in Section 4.2.

Proposition 1. *In the hybrid parallel training scenario of large-scale models, when the size of the micro-batch b is given, the memory constraint condition on a single GPU is a linear function that depends only on the number of layer assignments $l_{i,j}$. And the number of GPUs that share the layers in the same group determines the upper limit of the memory constraint condition.*

Proof. A large-scale model can usually be divided into many uniform layers (e.g., Transformer blocks) and a handful of non-uniform layers (e.g., the embedding table and LM head). And in the context of a pipeline parallelism, these non-uniform layers will only appear in the first and last stages of the pipeline, while the remaining uniform layers are partitioned into all stages.

For any GPU group Y with k GPUs, on each GPU, let a_{f_k} represent the peak memory consumption of activations during forward propagation for one layer when the micro-batch size is 1. Similarly, let a_{f+b_k} denote the peak memory consumption of activations during both forward and backward propagation for one layer when the micro-batch size is 1, and let s_k be the memory consumption of the parameters, gradients, and optimization states for one layer. Theoretically, the memory

consumption is proportional to the number of GPUs in a group, denoted as k. This is because the sizes of activations, parameters, gradients and optimization states are all directly related to the hidden states dimension, which is evenly distributed across the GPUs within a group. Consequently, for two GPU groups with k' and k'' GPUs, rates k'/k'' would hold for $a_{f_{k'}}/a_{f_{k''}}$, $a_{f+b_{k'}}/a_{f+b_{k''}}$ and $s_{k'}/s_{k''}$.

As for the non-uniformed layers, let $\dot{a}_{f_k}$ and $\ddot{a}_{f_k}$ represent the peak memory consumption of activations during forward propagation for the first and last several non-uniform layers of the model, respectively, when the micro-batch size is 1. And let $\dot{a}_{f+b_k}$ and $\ddot{a}_{f+b_k}$ denote the peak memory consumption of activations during both forward and backward propagation for the first and last several non-uniform layers of the model, respectively, when the micro-batch size is 1. Finally, let $\dot{s}_k$ and $\ddot{s}_k$ be the memory consumption of the parameters, gradients, and optimization states for the first and last several non-uniform layers of the model, respectively, when the micro-batch size is 1.

Considering the i-th 1F1B pipeline with $\overline{PP}_i$ stages in total, for a GPU within stage j ($1 \le j \le \overline{PP}_i$), it will first accumulate $\overline{PP}_i - j$ rounds forward activations with a micro-batch size of b. Then, for the remaining micro-batches, it will execute a forward propagation followed by a backward propagation, until all the micro-batches are processed. Therefore, the peak memory consumption for a GPU within a group consisting of $k_{i,j}$ GPUs in stage j would be

$$\begin{cases} l_{i,1} \times \left\{ b \times \left[a_{f_{k_{i,j}}} \times \left(\overline{PP}_i - 1 \right) + a_{f+b_{k_{i,j}}} \right] + s_{k_{i,j}} \right\} + b \times \left[\dot{a}_{f_{k_{i,j}}} \times \left(\overline{PP}_i - 1 \right) + \dot{a}_{f+b_{k_{i,j}}} \right] + \dot{s}_{k_{i,j}}, & \text{for } j = 1 \\ l_{i,\overline{PP}_i} \times \left(b \times a_{f+b_{k_{i,j}}} + s_{k_{i,j}} \right) + b \times \ddot{a}_{f+b_{k_{i,j}}} + \ddot{s}_{k_{i,j}}, & \text{for } j = \overline{PP}_i \\ l_{i,j} \times \left\{ b \times \left[a_{f_{k_{i,j}}} \times \left(\overline{PP}_i - j \right) + a_{f+b_{k_{i,j}}} \right] + s_{k_{i,j}} \right\}, & \text{for } 2 \le j \le \overline{PP}_i - 1 \end{cases}$$

Assume that the memory limit for GPU X is denoted by C_X. In most cases, $C_X = C$, where C represents the GPU memory bound. However, if a GPU straggler experiences memory pressure, the GPU memory usage within the group should be limited by the minimum C_X. To accommodate practical scenarios and prevent out-of-memory (OOM) errors, we also introduce a reserved memory gap G (4096MiB in our experimental setup) to allocate memory for essential operations such as NCCL and CUDA contexts. Thus, From the perspective of $k = 1$, the memory constraint can be formulated as

$$\begin{cases} l_{i,1} \times \left\{ b \times \left[\frac{a_{f_1}}{k_{i,j}} \times \left(\overline{PP}_i - 1 \right) + \frac{a_{f+b_1}}{k_{i,j}} \right] + \frac{s_1}{k_{i,j}} \right\} + b \times \left[\frac{\dot{a}_{f_1}}{k_{i,j}} \times \left(\overline{PP}_i - 1 \right) + \frac{\dot{a}_{f+b_1}}{k_{i,j}} \right] + \frac{\dot{s}_1}{k_{i,j}} \le \min\limits_{X \in Y_{i,j}} \{C_X\} - G, & \text{for } j = 1 \\ l_{i,\overline{PP}_i} \times \left(b \times \frac{a_{f+b_1}}{k_{i,j}} + \frac{s_1}{k_{i,j}} \right) + b \times \frac{\ddot{a}_{f+b_1}}{k_{i,j}} + \frac{\ddot{s}_1}{k_{i,j}} \le \min\limits_{X \in Y_{i,j}} \{C_X\} - G, & \text{for } j = \overline{PP}_i \\ l_{i,j} \times \left\{ b \times \left[\frac{a_{f_1}}{k_{i,j}} \times \left(\overline{PP}_i - j \right) + \frac{a_{f+b_1}}{k_{i,j}} \right] + \frac{s_1}{k_{i,j}} \right\} \le \min\limits_{X \in Y_{i,j}} \{C_X\} - G, & \text{for } 2 \le j \le \overline{PP}_i - 1 \end{cases}$$

We can directly derive $\mu_{i,j}(b)$, $v_{i,j}(b)$ and $C_{i,j}$ from the above equation as

$$\begin{cases} \mu_{i,1}(b) = b \times \left[a_{f_1} \times \left(\overline{PP}_i - 1 \right) + a_{f+b_1} \right] + s_1, & \text{for } j = 1 \\ \mu_{i,\overline{PP}_i}(b) = b \times a_{f+b_1} + s_1, & \text{for } j = \overline{PP}_i \\ \mu_{i,j}(b) = b \times \left[a_{f_1} \times \left(\overline{PP}_i - j \right) + a_{f+b_1} \right] + s_1, & \text{for } 2 \le j \le \overline{PP}_i - 1 \end{cases}$$

$$\begin{cases} v_{i,1}(b) = b \times \left[\dot{a}_{f_1} \times \left(\overline{PP}_i - 1 \right) + \dot{a}_{f+b_1} \right] + \dot{s}_1, & \text{for } j = 1 \\ v_{i,\overline{PP}_i}(b) = b \times \ddot{a}_{f+b_1} + \ddot{s}_1, & \text{for } j = \overline{PP}_i \\ v_{i,j}(b) = 0, & \text{for } 2 \le j \le \overline{PP}_i - 1 \end{cases}$$

and

$$C_{i,j} = k_{i,j} \times \left(\min_{X \in Y_{i,j}} \{C_X\} - G \right), \text{where } k_{i,j} = \left| \left\{ X | X \in Y_{i,j} \right\} \right|$$

□

B.5 Deduction of Equivalence Between Eq. (1) and Eq. (2), (3)

In Section 4.2, we decouple the problem in Eq. (1) into the sub-problems in Eq. (2) and Eq. (3). Below we provide the detailed deduction.

Proposition 2. *Any optimal solution of Eq. (2) combined with Eq. (3) will also be one of the optimal solutions of Eq. (1)*

.

Proof. Let $(\bar{l}_{1,1},\ldots,\bar{l}_{\overline{DP},\overline{PP}_{\overline{DP}}})$ and $(\overline{m}_1,\ldots,\overline{m}_{\overline{DP}})$ be one of the optimal solutions of Eq. (1). And let $\overline{o}_i := \max_{1\le j\le\overline{PP}_i}\{y_{i,j}\times\bar{l}_{i,j}\times\tau(b)\times\overline{m}_i\}$. Let $(\hat{l}_{i,1},\ldots,\hat{l}_{i,\overline{PP}_i})$ be one of the optimal solutions of Eq. (2), the i-th sub-problem, where i is fixed here. Assume $(\hat{l}_{i,1},\ldots,\hat{l}_{i,\overline{PP}_i})\neq(\bar{l}_{i,1},\ldots,\bar{l}_{i,\overline{PP}_i})$, then we consider $\hat{o}_i := \max_{1\le j\le\overline{PP}_i}\{y_{i,j}\times\hat{l}_{i,j}\times\tau(b)\times\overline{m}_i\}$. Since $(\hat{l}_{i,1},\ldots,\hat{l}_{i,\overline{PP}_i})$ is one of the optimal solutions of Eq.(2), and $(\bar{l}_{i,1},\ldots,\bar{l}_{i,\overline{PP}_i})$ is also a feasible solution for it, we have $\max_{1\le j\le\overline{PP}_i}\{y_{i,j}\times\hat{l}_{i,j}\}\le\max_{1\le j\le\overline{PP}_i}\{y_{i,j}\times\bar{l}_{i,j}\}$. Therefore, it must be the case that

$$\hat{o}_i=\max_{1\le j\le\overline{PP}_i}\left\{y_{i,j}\times\hat{l}_{i,j}\times\tau(b)\times\overline{m}_i\right\}\le\overline{o}_i=\max_{1\le j\le\overline{PP}_i}\left\{y_{i,j}\times\bar{l}_{i,j}\times\tau(b)\times\overline{m}_i\right\}$$

This implies that the solution $(\bar{l}_{i,1},\ldots,\bar{l}_{i,\overline{PP}_1},\ldots,\hat{l}_{i,1},\ldots,\hat{l}_{i,\overline{PP}_i},\ldots,\bar{l}_{\overline{DP},1},\ldots,\bar{l}_{\overline{DP},\overline{PP}_{\overline{DP}}})$ is a better (or at least equal) solution for Eq.(1) compared to $(\bar{l}_{i,1},\ldots,\bar{l}_{i,\overline{PP}_1},\ldots,\bar{l}_{i,1},\ldots,\bar{l}_{i,\overline{PP}_i},\ldots,\bar{l}_{\overline{DP},1},\ldots,\bar{l}_{\overline{DP},\overline{PP}_{\overline{DP}}})$. Since we have already assumed that one of the optimal solutions of Eq.(1) is $(\bar{l}_{i,1},\ldots,\bar{l}_{i,\overline{PP}_1},\ldots,\bar{l}_{i,1},\ldots,\bar{l}_{i,\overline{PP}_i},\ldots,\bar{l}_{\overline{DP},1},\ldots,\bar{l}_{\overline{DP},\overline{PP}_{\overline{DP}}})$, it follows that $(\bar{l}_{i,1},\ldots,\bar{l}_{i,\overline{PP}_1},\ldots,\hat{l}_{i,1},\ldots,\hat{l}_{i,\overline{PP}_i},\ldots,\bar{l}_{\overline{DP},1},\ldots,\bar{l}_{\overline{DP},\overline{PP}_{\overline{DP}}})$ is equally optimal. And the combined solution $(\hat{l}_{i,1},\ldots,\hat{l}_{i,\overline{PP}_1},\ldots,\hat{l}_{i,1},\ldots,\hat{l}_{i,\overline{PP}_i},\ldots,\hat{l}_{\overline{DP},1},\ldots,\hat{l}_{\overline{DP},\overline{PP}_{\overline{DP}}})$ after solving Eq.(2) for each given i, should also be one of the optimal solutions of Eq.(1). Therefore, we can solve Eq.(1) by solving all Eq.(2) (totally $\overline{DP}$ sub-problems) and obtain the optimal solution $(\hat{l}_{i,1},\ldots,\hat{l}_{i,\overline{PP}_1},\ldots,\hat{l}_{i,1},\ldots,\hat{l}_{i,\overline{PP}_i},\ldots,\hat{l}_{\overline{DP},1},\ldots,\hat{l}_{\overline{DP},\overline{PP}_{\overline{DP}}})$. Subsequently, with $l_{i,j}$ fixed, the only variables are m_i. The problem then reduces to a single ILP problem (Eq.(3)) represented as

$$\begin{aligned}\arg\min_{m_i}\ \max_{i\in\left[1,\overline{DP}\right]}&\left\{\max_{1\le j\le\overline{PP}_i}\left\{y_{i,j}\times\hat{l}_{i,j}\right\}\times m_i\right\}\times\tau(b)\\ \text{s.t.}\ &\sum_{i\in\left[1,\overline{DP}\right]}m_i\times b=B,\\ &m_i\in\mathbb{N}_0\ \text{for}\ \forall i\in\left[1,\overline{DP}\right]\end{aligned}$$

□

B.6 Deduction of Eq. (4)

In Section 4.3.2, we formulate the problem of finding the best pipeline division into Eq. (4) by relaxing a few constraints in Eq. (1). Below we provide the detailed deduction.

Proposition 3. *If we ignore the memory constraints in Eq.* (1)*, and further assume the layer assignments are not restricted to integers (i.e., $l_{i,j}$ in Eq.* (1) *can be any positive real numbers), then the problem of finding the best pipeline division can be written as Eq.* (4)*.*

Proof. We have already proved the equivalence between the ultimate problem, Eq. (1) and two sub-problems, Eq. (2) and Eq. (3). Thus, when first solving Eq. (2), if we neglect the memory constraints and further assume the layer assignments are not restricted to integers, the optimal solution could be directly obtained as follows

$$\left(l_{i,1},\ldots,l_{i,\overline{PP}_i}\right)=\left(\frac{\frac{L}{y_{i,1}}}{\sum_{1\le j\le\overline{PP}_i}\frac{1}{y_{i,j}}},\ldots,\frac{\frac{L}{y_{i,\overline{PP}_i}}}{\sum_{1\le j\le\overline{PP}_i}\frac{1}{y_{i,j}}}\right)\Longrightarrow\ o_i=\frac{1}{\sum_{1\le j\le\overline{PP}_i}\frac{1}{y_{i,j}}}\times L$$

Substituting these results into Eq. (3), the problem becomes

$$\begin{aligned}\arg\min_{m_i}\ \max_{i\in\left[1,\overline{DP}\right]}&\left\{\frac{m_i\times\tau(b)}{\sum_{1\le j\le\overline{PP}_i}\frac{1}{y_{i,j}}}\right\}\\ \text{s.t.}\ &\sum_{i\in\left[1,\overline{DP}\right]}m_i\times b=B\\ &m_i\in\mathbb{N}_0\ \text{for}\ \forall i\in\left[1,\overline{DP}\right]\end{aligned}$$

Assuming there are M GPU groups in total, with M_s groups having stragglers. For the i-th pipeline, let h_i be the number of normal groups (without stragglers). Considering the placement of each straggler group, we can form a $\{q_{i,k}\}_{\overline{DP},Ms}$ matrix.

In this matrix, each column j contains exactly one 1, representing which pipeline the j-th straggler group is assigned. Each row consists of several 1s, indicating how many straggler groups the pipeline contains, summing up to h_i. Thus, for the i-th pipeline, its term in the objective function can be transformed to

$$\frac{m_i \times \tau(b)}{\sum_{1 \le j \le \overline{PP}_i} \frac{1}{y_{i,j}}} = \frac{m_i \times \tau(b)}{h_i \times \hat{y} + \sum_{k=1}^{M_s} q_{i,k}/y_k}$$

We also have additional constraints for h_i and $\{q_{i,k}\}_{\overline{DP},M_s}$

$$\sum_{i=1}^{\overline{DP}} h_i = M - M_s$$

$$q_{i,k} \in \{0,1\} \text{ for } \forall i \in [1, \overline{DP}], \forall k \in [1, M_s]$$

$$\sum_{i=1}^{\overline{DP}} q_{i,k} = 1 \text{ for } \forall k \in [1, M_s]$$

$$h_i \in \mathbb{N}_0 \text{ for } \forall i \in [1, \overline{DP}]$$

□

B.7 Deduction of Possible Grouping Results after Splitting

In Section 4.3.1, we make the statement that there exist up to 6 possible grouping results to classify 7 GPUs into three groups with 1, 2, and 4 GPUs, respectively. Below we first prove a related proposition, and then elaborate the 6 possible grouping results.

Proposition 4. *For an optimal grouping solution, regardless of the number of GPUs each group possesses (non-uniform device partitioning may happens), the GPUs within each group must be arranged in descending order of their straggling rates and form a consecutive sequence.*

Proof. The underlying principle is that we can demonstrate the existence of an optimal solution by means similar to the proof in Theorem 1, that is, through the exchange of GPU pairs. Suppose there exists a non-consecutive group $Y = (X_{i_1}, \ldots, X_{i_k})$, and assuming the corresponding straggling rates satisfy $x_{i_1} \ge \ldots \ge x_{i_k}$. Then we consider the GPU with the highest straggling rate in the group, X_{i_1}. Assuming it is in the p-th position in the GPU sequence arranged in descending order of the straggling rate. We then consider $(x_p, x_{p-1}, \ldots, x_{p-k+1})$. Because the non-consecutive group already has $k-1$ GPUs with a smaller straggling rate than X_p, it is certain that such a consecutive sequence can be extracted without the issue of insufficient GPU amount. And for the new continuous GPU group with descending order $Y' = (X_p, X_{p-1}, \ldots, X_{p-k+1})$, it necessarily possesses the property that

$$x_p = x_{i_1}$$
$$x_{p-1} \ge x_{i_2}$$
$$x_{p-2} \ge x_{i_3}$$
$$\ldots$$
$$x_{p-k+1} \ge x_{i_k}$$
$$x_p \ge x_{p-1} \ge \ldots \ge x_{p-k+1}$$

Then, we swap each X_{i_j} with $X_{p-(j-1)}$. For the group who owns $X_{p-(j-1)}$ before, it gets a better GPU with smaller straggling rate, and therefore would have a less (or at least equal) group straggling rate in total. And for the newly formed group Y', since the X_p is unchanged, it shares the same group straggling rate with Y before. Thus, the total training cost will only get better after swapping to turn Y into Y'. And a grouping result that only consists of consecutive GPUs in each group would be better compared to all. □

Then, we consider the circumstance that an original group consisting of 8 GPUs is split due to a heavy straggler in it, and we need to classify the 7 remaining GPUs into three groups with 1, 2, and 4 GPUs, respectively. Note that the following enumeration of grouping is generalized and applicable to various partitioning scenarios and straggler situations.

Without loss of generality, assuming that the remaining GPUs are sorted in descending order by straggling rate $x_1 \ge \ldots \ge x_7$, corresponding sequentially to GPU $X_1, \ldots, X_7$. Let Y_1, Y_2 and Y_3 represent the groups with 1, 2 and 4 GPUs, respectively.

According to Proposition 4, if X_1 is partitioned to Y_3, then we must partition X_2, X_3, X_4 to Y_3 to achieve better performance. Similarly, if X_1 is partitioned to Y_2, then we must partition X_2 to Y_2. Consequently, we only need to consider the following 6 grouping results:

$$\begin{cases} Y_1 = (X_1) \\ Y_2 = (X_2, X_3) \\ Y_3 = (X_4, X_5, X_6, X_7) \end{cases} \quad \begin{cases} Y_1 = (X_1) \\ Y_2 = (X_6, X_7) \\ Y_3 = (X_2, X_3, X_4, X_5) \end{cases}$$

$$\begin{cases} Y_1 = (X_3) \\ Y_2 = (X_1, X_2) \\ Y_3 = (X_4, X_5, X_6, X_7) \end{cases} \quad \begin{cases} Y_1 = (X_5) \\ Y_2 = (X_6, X_7) \\ Y_3 = (X_1, X_2, X_3, X_4) \end{cases}$$

$$\begin{cases} Y_1 = (X_7) \\ Y_2 = (X_1, X_2) \\ Y_3 = (X_3, X_4, X_5, X_6) \end{cases} \quad \begin{cases} Y_1 = (X_7) \\ Y_2 = (X_5, X_6) \\ Y_3 = (X_1, X_2, X_3, X_4) \end{cases}$$

We will subsequently apply Theorem 2 to identify the most favorable among these candidates.

To further substantiate the efficacy of Theorem 2 in selecting the optimal one of all possibilities, we evaluate the end-to-end performance and the estimated performance provided by Theorem 2 on the 110B model. We introduce three stragglers with straggling rates of 2.57, 5.42 and 12.53 within a single node. Then we assess the performance of Malleus by crafting parallelization plans with the three different grouping results illustrated in Figure 5. From Figure 10, we can find that the correlation between the training time estimated using Theorem 2 and the actual training time of Malleus is coherent, indicating that the lower the estimated time, the lower the actual training time of Malleus. Consequently, we can deduce that Theorem 2 efficiently guides us in identifying the optimal grouping result without the necessity to evaluate all potentialities by solving the pipeline division problem as well as the lower-level model. Instead, the overhead incurred by employing Theorem 2 to filter the best grouping result is negligible.

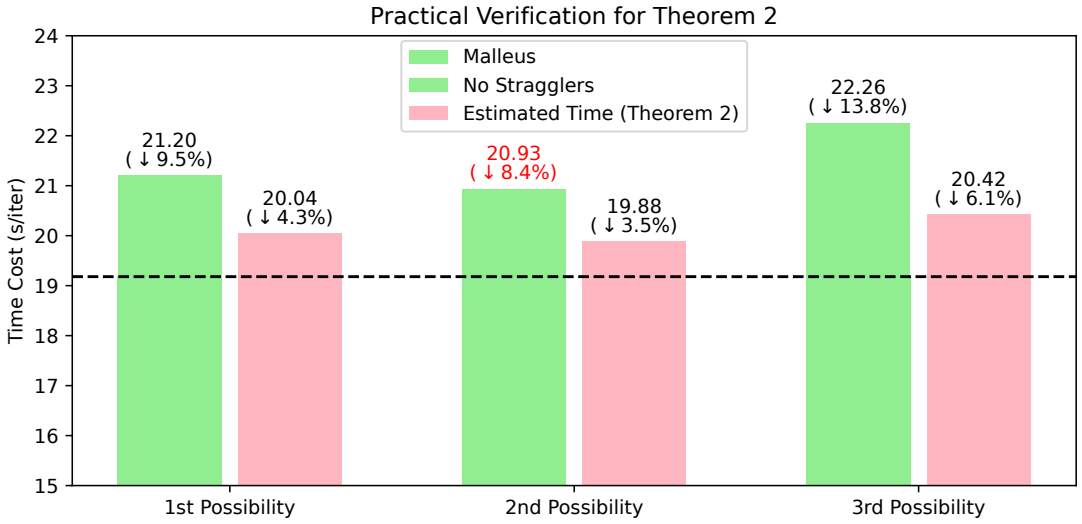

Figure 10. Effectiveness of Theorem 2, evaluated on the 110B model. Three scenarios correspond to three different grouping possibilities after splitting depicted in Figure 5.