AMSP: Super-Scaling LLM Training via Advanced Model States Partitioning

Qiaoling Chen
S-Lab,
Nanyang Technological University
& Shanghai AI Laboratory

Qinghao Hu
S-Lab,
Nanyang Technological University
& Shanghai AI Laboratory

Zhisheng Ye
Peking University
& Shanghai AI Laboratory
China

Guoteng Wang
Shanghai AI Laboratory
China

Peng Sun
Shanghai AI Laboratory
China

Yonggang Wen
Nanyang Technological University
Singapore

Tianwei Zhang
Nanyang Technological University
Singapore

Abstract

Large Language Models (LLMs) have demonstrated impressive performance across various downstream tasks. When training these models, there is a growing inclination to process more tokens on larger training scales but with relatively smaller model sizes. Zero Redundancy Optimizer (ZeRO), although effective in conventional training environments, grapples with scaling challenges when confronted with this emerging paradigm. To this end, we propose a novel LLM training framework AMSP, which undertakes a granular partitioning of model states, encompassing parameters (P), gradient (G), and optimizer states (OS). Specifically, AMSP (1) builds a unified partitioning space, enabling independent partitioning strategies for P, G, and OS; (2) incorporates a scale-aware partitioner to autonomously search for optimal partitioning strategies: (3) designs a dedicated communication optimizer to ensure proficient management of data placement discrepancies arising from diverse partitioning strategies. Our evaluations show that AMSP achieves up to 90.3% scaling efficiency across 1024 GPUs.

1 Introduction

LLMs, when trained on extensive text corpora, have demonstrated their prowess in undertaking new tasks [9, 11, 43] either through textual directives or a handful of examples [34]. These few-shot properties first appeared when scaling models to a sufficient size [21], followed by a line of works that focus on further scaling these models [4, 34, 35, 53, 55]. These efforts are grounded in the scaling law posted by OpenAI [21, 22]. However, DeepMind [15] presented a contrarian perspective that smaller models supplemented with vast datasets can potentially yield enhanced results. Meta, while considering the inference budgets, trained the LLaMA model [45], which has a tenth of the model size of DeepMind's model but uses 1 TB tokens. The performance brought by this methodology outperformed those from larger models. Since

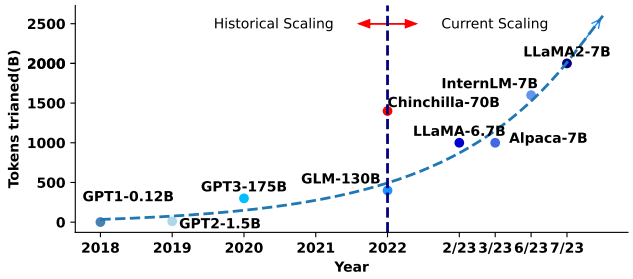

Figure 1. The token size for cutting-edge language models has been growing at an exponential rate over time, while the model size is reducing. The legend is represented in the format *ModelName* – *ModelSize*. For example, LLaMA-6.7B means the LLaMA model with 6.7 billion parameters.

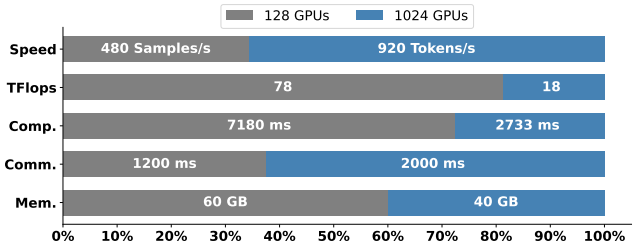

Figure 2. Comparison of training a LLaMA-6.7B model with ZeRO-1 using 2T data and micro-batch-size of 4 on 128 GPUs and 1024 GPUs. When training on 1024 GPUs instead of 512 GPUs, the overall throughput is doubled, but per GPU efficiency drops by 77% due to the increased ratio of communication and computation and suboptimal memory usage.

the introduction of Chinchilla [15] by DeepMind, the token sizes for cutting-edge language models have been growing at an exponential rate. However the parameter sizes of the model have remained around 7B, as depicted by the likes of LLaMA2 [46], Alpaca [42], and InternLM [6] in Figure 1.

Consequently, there is a preference to process this exponentially growing number of tokens using smaller models on larger training scales. For instance, as illustrated in

Figure 2, by training LLaMA-6.7B using micro-batch-size of 4 on a scale of 1024 GPUs [45, 46] expanded from 128 GPUs [30, 37, 57], we can halve the time [15, 45] required to process data. However, the decreasing computational efficiency and rising communication latency in Figure 2 are attributed to the following challenges: **(1) Per-GPU batch size is limited.** Due to the requirements of convergence and reproducibility, the global batch size remains constant. This limitation compromises the computational efficiency achieved on GPUs, especially at large scales. **(2) The latency of communication grows super-linearly** as the number of GPUs increases.

There are two popular solutions to alleviate these challenges, namely 3D parallelism [30, 40, 58] and ZeRO [37, 38, 49]. However, they have limited scalability and cannot handle such super-scale scenarios. Therefore, based on **(1)** and **(2)** we can trade GPU memory for communication. More specifically, instead of spreading model states across all the GPUs, we maintain copies of them. Besides, within the model state, there are three distinct components: parameters (P), gradient (G), and optimizer states (OS). Each of them exhibits varying degrees of communication overhead and memory cost. This heterogeneity offers the potential to flexibly select the redundancy level for each component individually.

Incorporating the above observations, we design AMSP to promote a more granular and adaptive strategy, allowing for component-specific optimization based on their individual communication and memory constraints. AMSP consists of three key modules, as shown in Figure 6. Specifically, (1) it constructs a unified partition space for P, G, and OS (grey block in Fig 3) based on the number of computing resources used by the user and the provided model size. This allows for finer-grained partitioning of P, G, and OS. (2) It introduces a *Scale-Aware partitioner*, which consists of two key components: data dependency rules to avoid high data transfer costs, and a cost model focusing on communication and GPU memory for combination strategies. An established Integer Linear Programming (ILP) algorithm is then used to determine the best partitioning approach. (3) Communication Optimizer constructs a tree-based stratified communication strategy tailored for cross-node exchanges within model states. Furthermore, for specialized scenarios involving data exchanges between P and OS, we incorporate a prefetch algorithm to ensure correct data sequencing.

Extensive evaluations show a significant system throughput and scaling efficiency improvement of AMSP on training LLaMA-based models. On A100 (80GB) GPU clusters with 800Gbps network, the throughput of AMSP is 4 × larger than that of DeepSpeed, which is the state-of-the-art DP framework for large model training. Compared to Megatron-LM-3D, a state-of-the-art system specialized for training Transformer models, AMSP achieves up to 36.9% larger throughput. AMSP gets near-linear (90.3%) strong scaling efficiency in 1024 GPU training, which is up to 55% better than DeepSpeed.

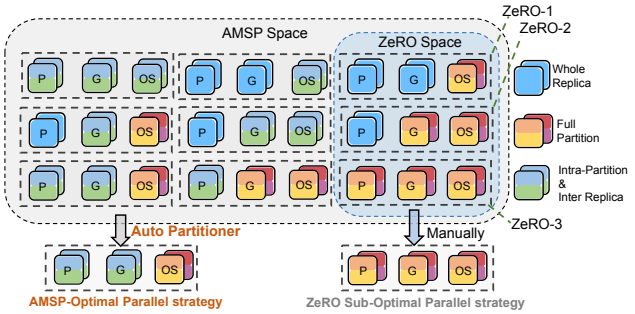

Figure 3. Comparing the search spaces of AMSP and prior work. By decomposing the three components of the model state and independently selecting their partitioning strategies and introducing a different level of redundancy strategy named *IntraPartition&InterReplica*, we can tap into a more expansive search space than prior work (ZeRO).

In summary, we make the following contributions:

- We build a unified partitioning space for model states, enabling independent and fine-grained partitioning strategies for P,G, and OS of the model states.
- We design a scale-aware partitioner in AMSP to automatically derive the optimal partition strategy.
- We evaluate AMSP on training large models with 1024 GPUs and get near-linear (90.3%) strong scaling efficiency.

Based on our current understanding and research, AMSP aims to address some of the challenges observed in distributed training frameworks given the prevalent training trends. We've explored the possibility of fine-grained partitioning of model states' members as a potential solution.

2 Background

2.1 LLM Training

There is a power law relationship between the parameter number of an autoregressive language model and its performance [21, 22]. This results in the trend of training larger and larger models [4, 41, 44, 53, 55], for higher performance. However, [15, 45] demonstrated that the best performance is not always achieved with the largest models, but can be with smaller models trained on more data. For instance, Chinchilla [15], while operating within the same computational budget as Gopher (280B) [36], employs only 70B parameters but leverages 4 × more data, surpassing Gopher's performance. Furthermore, LLaMA-6.7B [45], maintaining the same computational budget as Chinchilla, trains a 6.7B model on 1T tokens, diverges from the suggested 200B tokens for a 10B model, and achieves a performance improvement over Chinchilla. This shift underscores an emerging trend where researchers are increasingly emphasizing extensive data training on smaller models, with similar GPU resource demands.

2.2 3D parallelism

Training large models across multi-GPU clusters necessitates the use of advanced parallelism techniques, and Data Parallelism (DP) [25, 59], Pipeline Parallelism (PP) [13, 16, 29], and Tensor Parallelism (TP) [30, 40]are the three predominant strategies. Data Parallelism, or DP, is best suited for scenarios where the model's size can comfortably fit within a single GPU's memory. In this approach, each GPU retains a complete set of the model weights and processes distinct batches of input data concurrently, essentially duplicating the model across GPUs while splitting the data [7, 12, 27, 59]. On the other hand, when a model's size exceeds an individual GPU's memory capacity, Model Parallelism (MP) comes into play. Rather than dividing the data as in DP, MP divides the model itself, assigning segments of the model to different GPUs. Within MP, there are two primary techniques. The first, Pipeline Parallelism, divides the model into sequential stages. Each stage comprises a continuous sequence of layers, and though each stage relies sequentially on the previous one for a specific micro-batch, multiple micro-batches can be processed simultaneously at different stages, but it introduces empty bubbles that cause inefficiencies. The second technique, Tensor Parallelism, distributes individual layers of the model across several GPUs. For a single input, every GPU manages a unique section of the layer, allowing layer computations to happen concurrently, but this creates additional communication overhead that cannot be overlapped.

2.3 ZeRO

ZeRO is a memory optimization technique specifically tailored for data parallel training. It functions by partitioning and distributing model states, including P, G, and OS, across the GPUs being utilized. Model states are aggregated only when required for computation at a specific layer. The optimization provided by ZeRO is categorized into three stages as shown in Figure 3). In ZeRO-1, only the OS is partitioned and distributed across the GPUs. In ZeRO-2, both the OS and G undergo partitioning and distribution. ZeRO-3 advances this by partitioning all components of model states. ZeRO-3 offers the highest memory efficiency for large-scale model training, but this comes with the requirement of increased collective communications overhead.

Besides, when running on thousands of GPUs, the batch size per GPU is limited by the maximum global batch size that can be used during the training without sacrificing convergence efficiency, which will lead to lower computation efficiency. Therefore, ZeRO has a high ratio of communication and computation when training on thousands of GPUs.

Aiming to curtail the costly inter-node communication inherent in collective exchanges, recent refinements to ZeRO-3, exemplified by methods like MiCS [56], leverage on-device memory to facilitate more efficient communication. Under MiCS, the GPU cluster is segmented into distinct sub-groups.

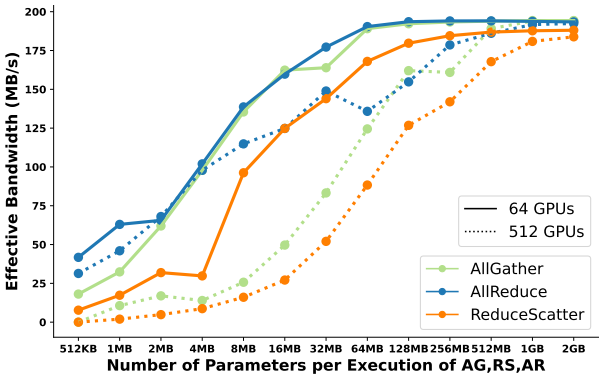

Figure 4. Effective bandwidths of three common collective communication algorithms at varying communication volumes, evaluated both at 64 and 512 A100 GPUs with 200GB/s NvLink and 800Gb/s InfiniBand network.

Algorithm 1 ZeRO-3 algorithm

Input: model,world size
Output: model

```
while model not converged do
    AllGather(P,world size);
    model.forward();
    partition(P,world size);
    AllGather(P,world size);
    model.backward();
    partition(P,world size);
    ReduceScatter(G,world size);
    optimizer.step();
```

Within these sub-groups, model states are divided, yet consistently duplicated among the different sub-groups.

2.4 Mix-Precision Training

Mixed precision is a renowned Large Model Training (LLM) technique that can reduce memory consumption and enhance training efficiency. Within this method, the forward and backward passes for P are conducted in the FP16 format, resulting in FP16 G. However, the OS and master weights are preserved in the FP32 format. Given this arrangement, there is a noticeable disparity in the memory footprint of P, G, and OS within the model state. When leveraging optimizers of the Adam [24], they maintain a copy of the master weights as well as corresponding momentum and bias. This translates to a storage requirement of three times the FP32 data in the form of OS. As a result, the proportional memory footprint of P, G, and OS stands at a ratio of 2:2:12, respectively [28].

3 Challenges and Motivation

3.1 Challenges for Linear-Scaling Training

Limited scalability of communication operator. Allreduce operations used to sum gradients over multiple GPUs have usually been implemented using rings [39] to achieve full bandwidth. The downside of rings is that latency scales linearly with the number of GPUs, preventing scaling above

hundreds of GPUs. NCCL 2.4 adds double binary trees [23, 26], which offer full bandwidth and a logarithmic latency even lower than 2D ring [14, 48] latency though not perfect, the performance shown in Figure 4. However, ZeRO-3 uses extra other two primitive Allgather [2] and ReduceScatter [10] to aggregate partitioned P and G shown in Algorithm 1 line 2,5,8, which shows a very limited scalability in Figure 4 when scale LLM training from 64 to 512 GPUs.

Rigid model-states partitioning. ZeRO-1 achieves a four-fold reduction in memory compared to DP but also introduces additional overhead in the form of ReduceScatter and AllGather communications [37]. ZeRO-2 goes further by achieving an eightfold memory reduction; however, it also introduces more gradient synchronization overhead compared to ZeRO-1, especially when utilizing gradient accumulation techniques [49]. ZeRO-3's memory reduction scales linearly with the DP degree, but it also adds approximately 50% more communication overhead [38, 49]. Notably, when there's a requirement for a ninefold memory reduction during training, ZeRO-2 becomes infeasible due to OOM. Using ZeRO-3 in such cases would involve excessive costs associated with aggregating parameters on a larger scale. These constraints underscore ZeRO's inherent rigidity in model state partitioning. As a result, user-chosen strategies are frequently sub-optimal due to the limited choices among ZeRO-1, ZeRO-2, and ZeRO-3. Moreover, MiCS [56] is a strategy rooted in ZeRO3, segmenting the model state by forming subgroups. It introduces an alternative partitioning strategy beyond ZeRO's complete partition, giving users an added choice. However, at its core, it remains a ZeRO approach and still faces the rigidity issues associated with the model state.

3.2 Opportunities for Linear-scale Training

Unified partition space for model-states. To address the rigidity issue of model state partitioning, we propose constructing a unified partitioning space. This space goes beyond the conventional strategies of Data Distributed Parallelism (DDP), where each GPU redundantly holds a complete set of model states, and the ZeRO approach that assigns each GPU an independent $1/N$ portion of the model states, ensuring the entire training cluster retains just one set of model states. We introduce a partitioning strategy with partial redundancy, inspired by the concept of MiCS. This partitions within n nodes and introduces redundancy across n/N nodes, which we term *intra-partition&inter-replica* redundancy strategy.

As illustrated in Figure 3, the partitioning space of ZeRO can be represented by the blue block. However, by decomposing the three components of the model state and independently selecting their partitioning strategies, we can tap into a more expansive gray search space. Notably, ZeRO stages 1, 2, and 3 are encompassed within this unified space.

Scale-aware Partitioner. To automatically search for an optimal combination for the user at the partition space, we can construct a generic memory and communication cost model

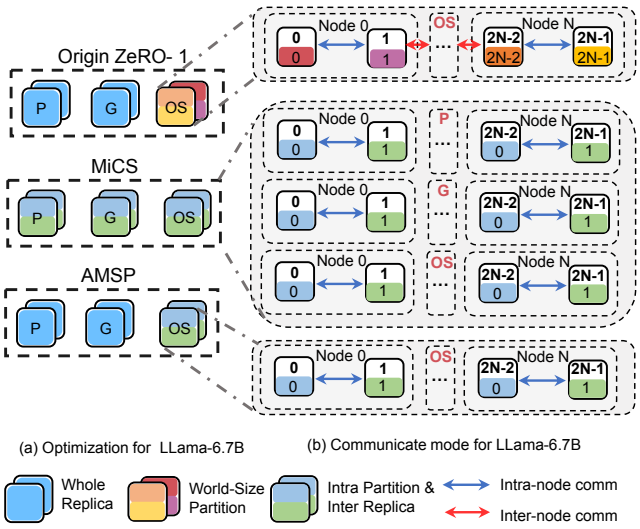

Figure 5. An optimization example of AMSP. Different model state partition strategies result in varied memory usage and affect which components participate in communication. AMSP minimizes both the number of participants and the range of communication without stressing the memory.

within this larger partitioning space. Considering the data dependencies among P, G, and OS, and based on the principles of minimal memory consumption and communication, we can search for the most cost-efficient partitioning strategy combination for a specific scale of compute resource.

Extra communication optimization. When the three components of the model state are divided in a different way than the DP approach, the data synchronization method differs from the traditional approach. This provides additional opportunities for communication optimization. For example, we can design hierarchical collection and reduction methods for cross-machine partitioning of P and G to circumvent the overhead associated with naive global synchronization

3.3 Motivating Example

Figure 5 showcases an optimization example identified by AMSP when training a 6.7B LLaMA model on 1024 A100(80GB) GPUs, with each node comprising 8 GPUs. We apply ZeRO1 as a comparison, which only partitions model-states OS into 1024 GPUs. Besides, following the insight from MiCS [56], we partition the model states into one node, and there are 128 replicas of model states in the training cluster. However, AMSP selects a strategy that remains a full replica of P and G like ZeRO, while sharding OS to one node like MiCS.

Memory usage analysis. When utilizing mixed precision in the training of the 6.7B LLaMA model, the memory proportions for P, G, and OS are 2, 2, and 12 respectively, as detailed in Section §2.4. In the case of ZeRO-1, which partitions only OS across 1024 GPUs, the memory footprint becomes $2+2+12/1024$. On the other hand, MiCS, which distributes P, G, and OS over 8 GPUs, has a memory footprint of

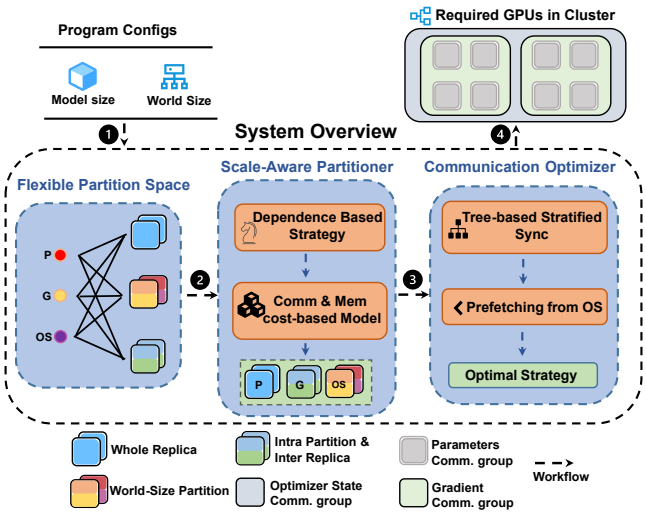

Figure 6. Overview of the AMSP System.

$(12+2+2)/8$. Meanwhile, AMSP, which allocates only *OS* to 8 GPUs, has a memory footprint of $2+2+12/8$. Consequently, the memory consumption for ZeRO-1, MiCS, and AMSP are approximately $4.01GB$, $2GB$, and $14.5GB$ respectively. Although AMSP exhibits a higher memory consumption in this context, it remains sufficient for the maximum micro-batch-size required for convergent training.

Communication overhead analysis. Figure 5 (b) illustrates the communication of the three approaches. Both ZeRO-1 and AMSP, having partitioned only *OS*, necessitate the communication solely for *OS*. Specifically, Zero-1's *OS* communication spans the entire world size, encompassing cross-node communication for each instance within the world size. In contrast, AMSP restricts its communication to intra-node, thus eliminating any cross-node communication. However, since MiCS segments *P*, *G*, and *OS*, all three components require communication. Yet, all these communications are confined within the node, resulting in zero cross-node exchanges. Consequently, when solely considering cross-node communications, the overhead for ZeRO-1 is up to the number of world size, while MiCS and AMSP are 0. However, noting that MiCS incurs additional intra-node communication for *P* and *G* compared to AMSP.

As a result, AMSP reduces the number of cross-node and intra-node communications to 0 and 8 without increasing the pressure of GPU memory.

4 AMSP Design

To deliver a linear-scaling LLM training framework, we introduce AMSP. It leverages an expanded model state partitioning space and possesses the capability to pinpoint the most communication-efficient partition combinations. In the following sections, we will outline its architectural design and delve into the specifics of each module.

Notation	Meaning
M	Model size of the given model.
N	World size of compute nodes.
R	Number of GPUs on each compute node.
G_n	Number of gradient accumulation steps.
Shard_P	Number of GPUs that a *parameter* is partitioned.
Shard_G	Number of GPUs that a *gradient* is partitioned.
Shard_OS	Number of GPUs that *optimizer states* are partitioned.

Table 1. Notations used in AMSP

4.1 Overview

Architecture & Workflow. Figure 6 illustrates the architecture overview and workflow of the AMSP system. Initially, AMSP built a unified partition space based on the program configuration inputs (model size and world size). Subsequently, AMSP leverages its scale-aware partitioner to produce an optimal combination of model state partitions, minimizing the communication overheads subject to the GPU memory constraints. This partitioner consists of two vital elements: the data dependency rules and a cost model. The rules can preemptively filter out strategies that could potentially lead to significant data transfer costs. The cost model is designed for communication and GPU memory related to combination strategies. Once this model is established, an off-the-shelf ILP algorithm is employed to pinpoint the optimal partitioning strategy. Considering the data movement resulting from various data placements post-slicing, a unified communication protocol is incorporated to further minimize inter-node communication.

Table 1 defines the notations used in AMSP.

4.2 Flexible Partitioning Space

ZeRO partitioning all model states across all devices can result in substantial communication overhead, especially at large scales. MiCS reduces this overhead by redundantly storing all model parameters within smaller sub-groups. It is worth noting that both ZeRO and MiCS treat the three components within the model state as an entire entity. This perspective restricts the possible strategies users could apply.

In our approach, we consider decoupling the model state into its individual constituents. This allows users to specify unique partitioning strategies for each component. Given the different communication patterns and memory footprints of these three model state components, our method facilitates more fine-grained control over communication and memory overheads. Next, we present the enhanced and more flexible partition space within the AMSP.

Partitioning stage. In AMSP, there are two primary components in the partition space: the model state and partition strategy. The partitioning strategy can be categorized into three main types, ordered by their memory reduction capabilities: *world-size partition*, *intra-partition & inter-replica*, and *whole replica*. AMSP allows each of the three members within

Spec	Device0	Device1	Device2	Device3
$P^0G^0OS^0$	[P,G,OS]	[P,G,OS]	[P,G,OS]	[P,G,OS]
$P^0G^0OS^1$	[P,G,0:$\frac{OS}{2}$]	[P,G,$\frac{OS}{2}$:OS]	[P,G,0:$\frac{OS}{2}$]	[P,G,$\frac{OS}{2}$:OS]
$P^0G^0OS^2$	[P,G,0:$\frac{OS}{4}$]	[P,G,$\frac{OS}{4}$:$\frac{OS}{2}$]	[P,G,$\frac{OS}{2}$:3$\frac{OS}{4}$]	[P,G,3$\frac{OS}{4}$:OS]
$P^0G^1OS^1$	[P,0:$\frac{G}{2}$,0:$\frac{OS}{2}$]	[P,$\frac{G}{2}$:G,$\frac{OS}{2}$:OS]	[P,0:$\frac{G}{2}$,0:$\frac{OS}{2}$]	[P,$\frac{G}{2}$:G,$\frac{OS}{2}$:OS]
$P^0G^1OS^2$	[P,0:$\frac{G}{2}$,0:$\frac{OS}{4}$]	[P,$\frac{G}{2}$:G,$\frac{OS}{4}$:$\frac{OS}{2}$]	[P,0:$\frac{G}{2}$,$\frac{OS}{2}$:3$\frac{OS}{4}$]	[P,$\frac{G}{2}$:G,3$\frac{OS}{4}$:OS]
$P^0G^2OS^2$	[P,0:$\frac{G}{4}$,0:$\frac{OS}{4}$]	[P,$\frac{G}{4}$:$\frac{G}{2}$,$\frac{OS}{4}$:$\frac{OS}{2}$]	[P,$\frac{G}{2}$:$\frac{3G}{4}$,$\frac{OS}{2}$:3$\frac{OS}{4}$]	[P,$\frac{3G}{4}$:G,3$\frac{OS}{4}$:OS]
$P^1G^1OS^1$	[0:$\frac{P}{2}$,0:$\frac{G}{2}$,0:$\frac{OS}{2}$]	[$\frac{P}{2}$:P,$\frac{G}{2}$:G,$\frac{OS}{2}$:OS]	[0:$\frac{P}{2}$,0:$\frac{G}{2}$,0:$\frac{OS}{2}$]	[$\frac{P}{2}$:P,$\frac{G}{2}$:G,$\frac{OS}{2}$:OS]
$P^1G^1OS^2$	[0:$\frac{P}{2}$,0:$\frac{G}{2}$,0:$\frac{OS}{4}$]	[$\frac{P}{2}$:P,$\frac{G}{2}$:G,$\frac{OS}{4}$:$\frac{OS}{2}$]	[0:$\frac{P}{2}$,0:$\frac{G}{2}$,$\frac{OS}{2}$:3$\frac{OS}{4}$]	[$\frac{P}{2}$:P,$\frac{G}{2}$:G,3$\frac{OS}{4}$:OS]
$P^1G^2OS^2$	[0:$\frac{P}{2}$,0:$\frac{G}{4}$,0:$\frac{OS}{4}$]	[$\frac{P}{2}$:P,$\frac{G}{4}$:$\frac{G}{2}$,$\frac{OS}{4}$:$\frac{OS}{2}$]	[0:$\frac{P}{2}$,$\frac{G}{2}$:$\frac{3G}{4}$,$\frac{OS}{2}$:3$\frac{OS}{4}$]	[$\frac{P}{2}$:P,$\frac{3G}{4}$:G,3$\frac{OS}{4}$:OS]
$P^2G^2OS^2$	[0:$\frac{G}{4}$,0:$\frac{G}{4}$,0:$\frac{OS}{4}$]	[$\frac{P}{4}$:$\frac{P}{2}$,$\frac{G}{4}$:$\frac{G}{2}$,$\frac{OS}{4}$:$\frac{OS}{2}$]	[$\frac{P}{2}$:3$\frac{P}{4}$,$\frac{G}{2}$:$\frac{3G}{4}$,$\frac{OS}{2}$:3$\frac{OS}{4}$]	[3$\frac{P}{4}$:P,$\frac{3G}{4}$:G,3$\frac{OS}{4}$:OS]

Table 2. Partition specs of a 2-dimensional tensor on a 2 × 2 device mesh. [P, G, OS] shows a complete model state. The device mesh is [[Device 0, Device 1], [Device 2, Device 3]]. Each device stores a partition of [P, G, OS]. The first column is the conbination of the Partition spec. The latter columns use Numpy syntax to describe the partitions stored on each device.

the model state to select a partition strategy independently. Here are the details of the three partition stages:

World-size Partition. This strategy revolves around the concept of partitioning data across all available GPUs named *world-size*. Therefore, each GPU retains 1/N of data and the synchronization should span across the entire world size.

Whole Replica. This strategy maintains a full replica of a component's data within each GPU, thus resulting in maximum memory usage and significant optimization of communication synchronization.

Intra-Partition & Inter-Replica. This strategy involves data segmentation within n nodes, with the same data redundantly stored across N/n nodes. This method leverages the imbalanced intra- and inter-node bandwidths. Our main goal is to maximize the usage of the rapid intra-node bandwidth while minimizing high-overhead inter-node communication. To achieve this, the framework partitions model states inside nodes and maintains redundant data replicas across multiple nodes. With this strategy, there are $log2\frac{N}{R}$ available options, each denoting a distinct degree of redundancy.

Concurrently, we represent the chosen partitioning strategies for these three components as *Shard_P*, *Shard_G*, and *Shard_OS*, respectively. When the value is equal to 1 and N, means using *World-size partition* and Whole Replica partition, while the value from 2 to $N-1$ means one of the subsets of Intra-Partition & Inter-Replica partition strategy.

Device Mesh. To better express the *intra−partition&inter−replica* policy and the communication cost associated with it, we express a set of computational resources with bandwidth imbalance using a device mesh. The *Intra-Partition* refers to the first dimension within the mesh, while *Inter-Replica* pertains to the second dimension of the device mesh. The bandwidths of the first and second dimensions differ.

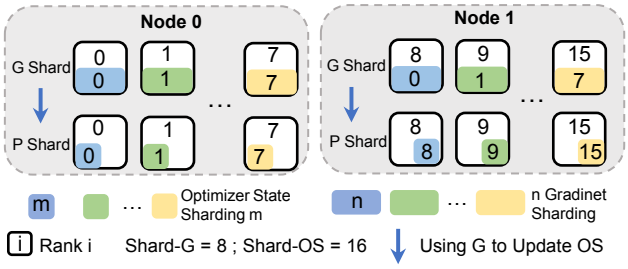

Figure 7. Sharding on 16 GPUs utilizing the Dependence-based rule. A larger (1/8) data chunk of gradient ensures enough gradient data to update the optimizer (1/16) without extra aggregating data from other GPUs.

4.3 Scale-aware Partitioner

Based on the partition space built by 4.2, we leverage Partitioner to search for a communication optimal combination of strategies from *P*, *G*, and *OS*. In this section, we detail the two components of a partitioner, including a pre-filter based on data dependence, and solving an ILP formulation built by the communication and memory cost model.

Dependence-based rules. The model training process includes a data dependency flow of all three model states, consisting of several steps: utilizing *parameters* to compute *gradients*, updating *optimizer states* with *gradients*, and updating *parameters* with *optimizer states*.

Extra Data Transfer. Utilizing varying partitioning strategies leads to complexities and potential inconsistencies. To maintain result accuracy and integrity, additional data movements are required, introducing communication overhead.

Dependence Rules. To avoid these extra data transmissions, we institute a set of combination rules that are firmly grounded in data dependencies. Before combining the strategies for the members, we filter out and eliminate strategies that are predisposed to incurring substantial data transfer

costs. Mathematically, this relationship is represented as:

$$Shard\_OS = R \times 2^i \times Shard\_G = R \times 2^j \times Shard\_P$$

Here, i and j are exponents such that $i \leq j$. Noting that the value 2^i is always less than the world size. At its essence, this rule signifies a hierarchical relationship: the partition number for the *OS* is a multiple of the number used for both *G* and *P*, and *G*'s partition number is a multiple of *P*'s nu. Adhering to this pattern ensures that any data dependency in the training flow from the upstream can fully access its proportionally allocated downstream data, facilitating the computation of *G* and updates of *OS*. Take the optimizer update as an example shown in Figure 7. A larger (1/8) data chunk of gradient ensures enough gradient data to update the optimizer, which has a smaller data chunk (1/16). The process of gradient generation follows the same pattern. However, the *optimizer step* is an example in the training flow where the most downstream *OS* accesses the most upstream parameters. As a result, we design a prefetching mechanism (§4.4) to minimize the data movement brought by this partitioning.

Partition Specs. We utilize the "partition spec" to represent the partitioning choices when *P*, *G*, and *OS* are combined. The partitioning strategies for *P*, *G*, *OS* can be articulated as P^a, G^b, and OS^c, where a, b, and c can be formulated as:

$$a, b, c = \frac{\text{shard\_P}}{R}, \frac{\text{shard\_G}}{R}, \frac{\text{shard\_OS}}{R}$$

Table 2 displays all potential partition spec combinations for a 2-dimensional tensor operating on a 2 × 2 mesh with 4 devices.

Memory Cost. Due to the inherent memory constraints, not all of the combined strategies that filter by rules §4.3 can fit within the available memory. Moreover, certain partitioning strategies for individual members might already breach these constraints. For example, in a 50B LLaMA model, if Shard_OS equals 1 or 8, the memory consumption for the optimizer state alone on a single GPU would skyrocket to 600GB and 75GB, respectively, exceeding the hardware capacities. Given these constraints, we must prune some partitioning strategies based on the memory bounds before consolidating the partition stages for the three members.

In mixed-precision training, the memory consumption ratio of *P*, *G*, and *OS*, generally stands at 2 : 2 : 12 when using the Adam [24] optimizer. The memory cost of the model state and activation memory cost can be expressed as:

$$memory\_cost = \frac{2M}{shard\_P} + \frac{2M}{shard\_G} + \frac{12M}{shard\_OS}$$

Depending on the size of the input model and the GPU resources required for the task, we only retain those partition combinations with memory consumption costs below a specified memory threshold. For simplicity, we model the activation [37] as $(34bsh+5bs^2a)\times l$ and add it to $memory\_cost$,

Spec	Comm Cost
$P^0G^0OS^2$	[0, 0, (AG,4)]
$P^0G^1OS^1$	[0, [$G_n \times M$(RS,2), (AR,2)], M(AG,2)]
$P^1G^1OS^1$	[2 × M(AG,2), [G_n×(RS,2),(AR,2)], M(AG,2)]

Table 3. Several cases of the communication cost. Only cross-node communication is included. AG denotes All-Gather, RS denotes Reduce-Scatter, and AR denotes All-Reduce. [0,0(AG, i)] denotes allgather in i ranks for complete OS communication. [a,b,c] represent a complete *P*, *G*, and *OS*, respectively.

where b, s, h, a, l represent batch-size, sequence-length,hidden-size,number-of-attention-head and the number of layers.

Communication Cost. When deploying thousands of GPUs, cross-node communication often becomes a pronounced limitation for collective communication. Aware of this obstacle, our refined model to calculate communication costs narrows its attention solely to cross-node exchanges, sidelining other possible communication routes. In the sections that follow, we elucidate the process of estimating the communication expenses for each component of the model state:

Parameters: Each parameter must aggregate shards from other partitions during both the forward and backward passes. Thus the communication cost for P, is expressed as:

$$Comm\_P = 2 \times M \times \frac{Shard\_P}{R}$$

Gradient: In scenarios where gradient accumulation techniques are employed, every micro-step of gradient accumulation necessitates aggregation with shards from other partitions. Additionally, at the boundary of gradient accumulation, a world-size 'All-Reduce' operation (a feature specific to DDP, and is not considered part of the gradient's communication cost) is executed. Therefore, the communication cost for the gradient can be described as:

$$Comm\_G = G_n \times M \times \frac{shard\_P}{R}$$

Optimizer state: The communication requirements for this are quite direct. Once updated, it merely sends its shard to the relevant partition housing its associated parameters.

$$Comm\_OS = M \times \frac{shard\_OS}{R}$$

Table 3 lists some examples of the communication costs for different partition specs.

ILP Formulation. We frame the minimization of communication costs as an Integer Linear Programming problem and utilize an off-the-shelf solver to find its optimal solution. For three model state members, each has $log2^{\frac{N}{R}} + 2$ strategies. We denote the communication cost and memory cost for the i^{th} member using the j^{th} strategy as C_{ij}, and A_{ij} respectively. Then we define our decision variable as X_{ij} to indicate whether the i^{th} member uses the j^{th} strategy. The specified $memory\_threshold$ is usually set to the

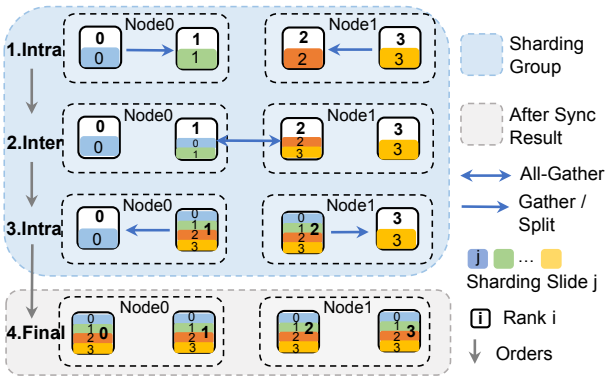

Figure 8. An example of all-gather parameters within one partition when employing a Tree-based stratified sync strategy for parameters. The data slice of P is first aggregated locally within nodes to specific ranks, which then combine their data inter-node, before distributing the complete data slice to other ranks in their respective nodes.

Algorithm 2 AMSP algorithm

Input: **model,world size**
Output: **model**

while model not converged **do**
  *- AllGather(**P**,world size);*
  *+ Stratified_AllGather(**P**, Shard_P);*
  model.forward();
  *- partition(**P**,world size);*
  *+ partition(**P**,Shard_P);*
  *- AllGather(**P**,world size);*
  *+ Stratified_AllGather(**P**, Shard_P);*
  model.backward();
  *- partition(**P**,world size);*
  *+ partition(**P**,Shard_P);*
  *- ReduceScatter(**G**,world size);*
  while gradient not reach boundary **do**
    *+ Stratified_ReduceScatter(**G**,Shard_G);*
  *+ Stratified_ReduceScatter(**G**,world size);*
  if *Shard_OS < Shard_P* **then**
    + Prefetching_AllGather(OS,Shard_OS);
  optimizer.step();

GPU memory threshold. The objective is to minimize the following

$$\min\left(\sum_{i=1}^{3}\sum_{j=1}^{log2\frac{N}{R}+2} C_{ij}\times X_{ij}\right)$$

Where the decision variable X_{ij} must satisfy the dependencies mentioned above and:

$$\sum_{i=1}^{3}\sum_{j=1}^{log2\frac{N}{R}+2} A_{ij} < memory\_threshold.$$

4.4 Communication Optimization

The unique model states sharding lowers the communication costs compared to existing works. Besides, we also make communication optimizations to further reduce the overhead, including a tree-based stratified communication optimization and parameter update-based prefetching strategy. Algorithm 2 exhibits a different communication optimization strategy for AMSP compared to ZeRO-3.

Tree-based Stratified Synchronization. When the splitting strategy for some members of the model state is set to shard-X > R, it will introduce cross-node communication. However, considering the disparity between intra-node and inter-node bandwidths, it is desirable to reduce the frequency of inter-node communications. Furthermore, when shard-X > 1 and the gradient-accumulation technique [33] is applied, a global reduction in each micro-step will introduce multiple times of cross-node communication. The key solution is to reduce the inter-node communication volume as much as possible. Therefore, we introduce a three-tiered conditional step-wise algorithm to diminish communication instances and ensure the sequential integrity of data. It is a tree-based hierarchical approach designed to supersede the traditional singular global synchronization methods.

Our optimization consists of different communication patterns based on the networking topology.

Compared to the global sync. approach with latency $(x-1)\times M/x$, our method streamlines the cross-node communication. In our design, such communication is chiefly confined to the second stage and occurs with a frequency of x/R, leading to a reduced latency of:

$$\left(\frac{x}{\mathrm{R}}-1\right)\times\left(\frac{\mathrm{R}}{x}\right)\times M$$

The latency ratio between AMSP and the global one is:

$$\frac{x-\mathrm{R}}{x-1}$$

The benefit of AMSP decreases as the value of x increases.

Prefetching from OS. When updating parameters based on the rules defined in §4.3, *OS* only possesses a portion of P, preventing local exchanges. As shown in Figure 9(a), a naive approach involves a global scatter reduction, potentially causing data sequence misalignment and necessitating additional data copy and replacement. To address this issue, we devise an efficient prefetching communication strategy. It pre-transmits the required data chunk for each portion of P that needs an update prior to executing the parameter update via a separate channel, as shown in Figure 9(b).

5 Evaluation

In this section, we evaluate the following three aspects:

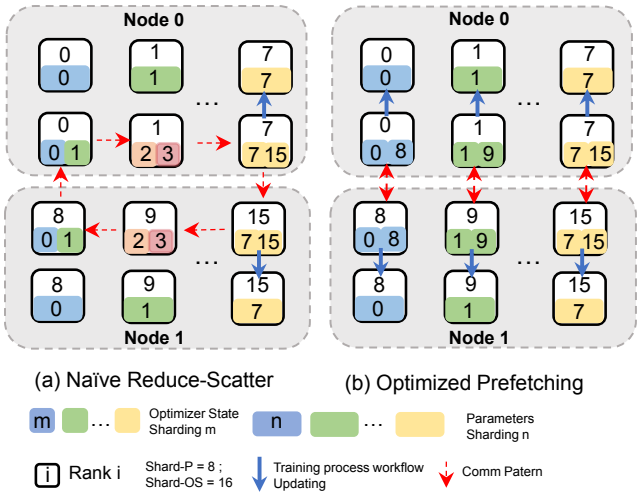

Figure 9. Prefetching mechanism. A naive approach for the optimizer step leads to the consequence of misalignment in the data sequence and requires extra data copy and replacement. AMSP prefetch the data chunk of *OS* before the optimizer step begins.

- **End-to-end experiments with up to 1024 GPUs** (§5.2): Does AMSP demonstrate near-linear scalability? Why do we claim that AMSP unifies the ZeRO family?
- **Throughput breakdown and analysis** (§5.3): Which partitioning approaches have a profound influence on performance outcomes? And do the refinements we've made on *P*, *G*, and *OS* lead to enhanced performance?
- **Fidelity** (§5.4): Does AMSP inadvertently compromise crucial conditions for successful training convergence? Can it align precision with existing training methods?
- **Case study** (§5.5): We selected a representative model training scenario, analyzed the memory and communication overhead under various sharding strategies, and derived insightful conclusions.

5.1 Experimental Setup

Implementation. We implement AMSP on top of DeepSpeed 0.9.0 and Pytorch 2.0. We also present a user-friendly interface that necessitates only minimal modifications to the existing DeepSpeed code. Upon initializing AMSP, we employ an ILP solver to automatically optimize the communication based on the parsed configuration. The resultant fine-grained communication patterns are then realized using PyTorch's NCCL functions to establish the corresponding replication and partitioning groups.

Hardware and Software. Our testbed comprises 128 nodes, collectively harnessing the power of 1024 GPUs. Each node is equipped with 8 NVIDIA A100 80GB GPUs, 128 CPUs, and 2 TB memory. The GPUs within a node are interconnected via NVLink, while there is a 4*200Gb Infiniband for inter-node communication. For the LLM training configuration, we use the O2 level mixed-precision technique [8] and gradient accumulation is also enabled.

Table 4. Language model structure for LLaMA variants. We use sequence lengths of 512 and 1024 for all models.

model	hidden size	layer	attention heads
LLaMA 6.7B	4096	32	32
LLaMA 13B	5120	40	40
LLaMA 30B	6144	60	48

Table 5. The configs of Megatron-LM used in experiments.

Configs	TP Size	PP Size
Megatron-LM (1)	4	1
Megatron-LM (2)	8	1
Megatron-LM (3)	8	2

Table 6. In the experiments, we explored various partitioning configurations for AMSP. The numbers presented here correspond to the value of Shard-X.

States	AMSP (1)	AMSP (2)	AMSP (3)	AMSP (4)	AMSP (5)
P	1	1	8	8	1
G	8	1	8	8	1
OS	DP	DP	DP	8	8

Metrics. Our primary assessment metric is throughput - samples/s. The experiments adopt a sequence length of 512. The global batch size is set to 4096 unless specifically mentioned and is kept constant among different experiments. For optimal system performance, our foremost strategy is to expand the micro-batch size to its highest permissible limit—ensuring it doesn't lead to CUDA OOM—while maintaining a consistent global batch size. When the micro-batch size hits this ceiling, there's an escalation in the number of gradient accumulations per step. To determine the per-GPU and overall system throughput, we assess the training step's time and quantify the tokens processed within that duration. This is done by multiplying the samples, batch size, sequence length, and the number of gradient accumulation steps.

Model Configurations. We opt for cutting-edge and widely recognized model architectures, with all our models being derivatives from LLaMA [45]. For a range of model configurations, we vary the count of transformer layers and their dimensions. The details can be found in Table 4.

Baselines. We select Megatron-LM v2 [30], DeepSpeed-ZeRO[37], and DeepSpeed-MiCS [56] as the baselines for LLaMA models. We use different stages for DeepSpeed-ZeRO. We grid-search the optimal parallel strategy of these baseline systems. We implement AMSP on top of DeepSpeed-ZeRO. The parallel configurations of Megatron-LM-3D used in the experiments are provided in Table 5.

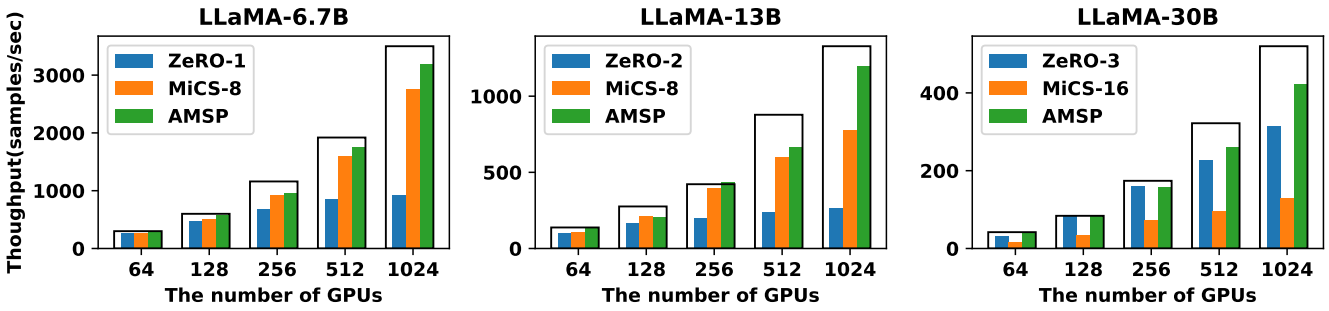

Figure 10. Scalability on up to 1024 GPUs of LLaMA model size range from 6.7B ~ 30B. For the LLaMA-6.7B and 13B models, we use ZeRO-1 and ZeRO-2 as the respective baselines. For the LLaMA-30B model, ZeRO-3 serves as the baseline. Additionally, MiCS Shard8 acts as a benchmark for both LLaMA-6.7B and 13B, while MiCS Shard16 is designated as the baseline for the 30B model. AMSP, equipped with its refined model-state partitioning approach driven by its integral partitioner, consistently outperforms the competitors. AMSP gets near-linear (90.3%) strong scaling efficiency in LLaMA-6.7B training using 1024 GPU.

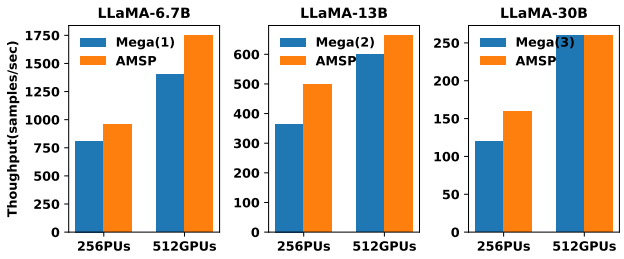

Figure 11. Performance Comparison to Megatron-LM. For model sizes 6.7B, 13B, and 30B, we use Megatron-LM(1), Megatron-LM(2), and Megatron-LM(3) as baselines, respectively. The throughput of AMSP outperforms Megatron-LM up to 37% at LLaMA-13B training on 256 GPUs.

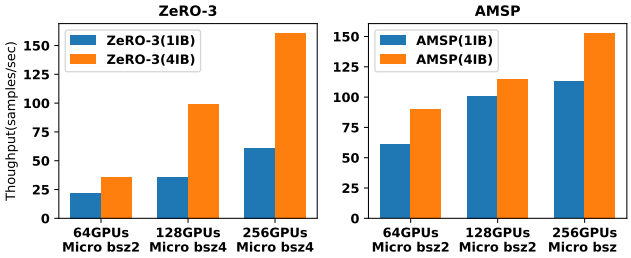

Figure 12. Throughput of training LLaMA-30B with different numbers of InfiniBand connection. The performance of ZeRO-3 is more sensitive to network alterations, while AMSP is more stable and resilient amidst the network change

.

5.2 End-to-end System Evaluation

Performance Scalability. In Figure 10, we present a systematic analysis of the scalability of AMSP from 128 GPUs up to 1024 GPUs, using models LLaMA-6.7B, LLaMA-13B, and LLaMA-30B. From the 10, it is evident that ZeRO experiences scalability problems, especially when scaled to 512 GPUs. These limitations are directly attributed to the constraints of communication operator scalability, as referenced in 4.

MiCS, aiming to address these limitations, reduces the communication scale at a single node for the 6.7B and 13B models, resulting in near-linear scaling during training. However, for the 30B training, the memory requirements necessitate cross-node sharding. In this scenario, MiCS experiences increased inter-node communication and its performance falls behind that of ZeRO.

Contrastingly, AMSP, with its optimized model-state partition strategy determined by its integral partitioner, consistently exhibits superior performance. Specifically, during the 30B model training, AMSP achieves a throughput almost four times higher than the baseline.

In summation, AMSP demonstrates robustness, delivering consistent speed enhancements across varying model complexities and GPU counts. This reinforces its effectiveness and adaptability in diverse training scenarios.

Comparing to Megatron-LM-3D. For 6.7B,13B, and 30B Megatron follow the configurations in Table 5 (1), (2), and (3) as baseline respectively. As illustrated in Figure 11, AMSP consistently outperforms Megatron in speed and efficiency. For the 6.7B, 13B, and 30B models, Megatron uses configurations from tables (1), (2), and (3) as baselines. As illustrated in Figure 11, AMSP consistently surpasses Megatron in both speed and efficiency. Specifically, when training the 6.7B, 13B, and 30B models on 256 GPUs, AMSP outperforms Megatron by 18%, 37%, and 32% respectively. On a 512 GPU setup, these performance gains are 25%, 10%, and 0.9%. It can be seen that for this scenario where small models scale to large-scale training, Megatron achieves the same performance as AMSP at the configuration of Megatron-LM(3) that $PPsize = 2, TPsize = 8$.

Stability in different InfiniBand network. In this section, we delve into how AMSP fares over distinct inter-node InfiniBand network configurations, specifically 1*HDR (200 Gbps) and 4*HDR (800 Gbps). As demonstrated in Figure 12, the performance of ZeRO-3 is considerably contingent upon the underlying network. When scaling the training of the

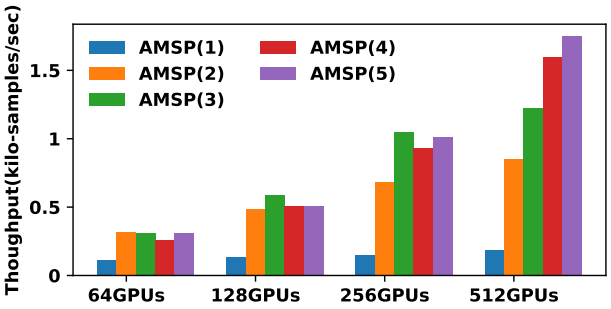

Figure 13. Throughput change with different AMSP strategy when training a LLaMA-6.7B with 800Gbps network. AMSP (2) emerges as the most efficient configuration at 64 GPUs, AMSP (3) is the most efficient strategy training on 128 or 256 GPUs, AMSP (5) becomes the best on a 512GPUs scale.

LLaMA-30B model from 64 GPUs to 128 GPUs and further to 512 GPUs, under 1*HDR, ZeRO3's throughput decreased by 1.63, 1.75, and 0.63 times respectively compared to 4*HDR. In contrast, AMSP experienced an average decline of only 0.32 times, which is relatively marginal. The cornerstone of this stability can be attributed to AMSP's design philosophy: to minimize the amount of cross-node communication, thus making it less susceptible to network fluctuations.

5.3 Design Analysis

Analysis of partition strategy from AMSP. In our research, we investigate the efficacy of different partitioning strategies of AMSP in response to varying scales. The AMSP configurations with different partitioning strategies are presented in Table 6. We base our experimentation on the LLaMA-6.7B model, keeping the micro-batch size constant at 4 and deploying an 800Gbps inter-node network. Figure 13 elucidates the comparative performance of these strategies. Notably, AMSP (2) emerges as the most efficient configuration at 64 GPUs, registering a performance that is 6 % superior to AMSP (3). However, when extended to 128 or 256 GPUs, AMSP (3) takes the lead in throughput. Interestingly, the dynamics shift again at 512 GPUs: AMSP (5) becomes the frontrunner, closely trailed by AMSP (4). This analysis underscores the significance of choosing an optimal partitioning strategy in AMSP, contingent on the specific scale and architecture of the training environment.

Throughput breakdown and analysis of Communication Optimizer. In Figure 14, we show the individual impact of communication optimizer for P, G, and OS on the throughput of the LLaMA-30B model on 256 and 512 GPUs. For this high bandwidth cluster, the individual speedup range between 1.13-1.16 ×, for a combined speedup of up to 1.3 ×. P, G as members of the model state with high communication costs, as shown in §4.3, the performance of AMSP will be greatly degraded when they are not optimized. Meanwhile, OS, as the least expensive member, can improve the throughput by about 8% when we use prefetching.

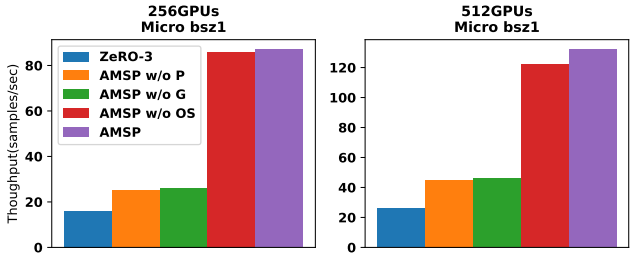

Figure 14. Throughput of LLaMA-30B model on 256 and 512 GPUs with AMSP, without optimization of P, G, OS and baseline ZeRO-3, and keep micro batch size as 1.

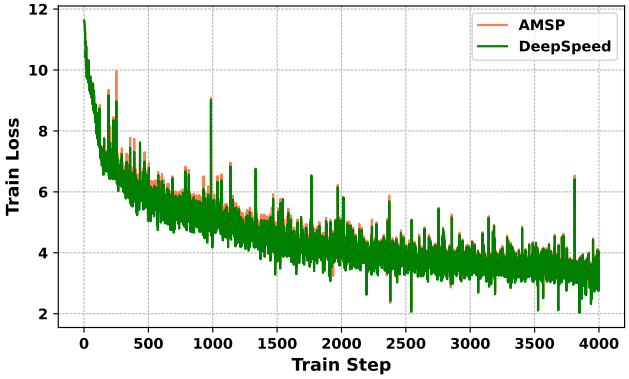

Figure 15. The training loss curve of 4000 steps under the same model configuration and random seeds.

5.4 Fidelity

In this section, we show that AMSP achieves consistent convergence as DeepSpeed, which validates the correctness of our system. We provide the training loss curves Figure 15 for training a LLaMA-6.7B model on the Wikipedia-en dataset. The global batch size is 4096. And the micro-batch size is 4 (the number of gradient accumulation steps is 4). The loss validation process does not aim to produce exactly the same loss as DeepSpeed but to ensure the convergence behaviors are the same. We report the training losses on 1 million sequences. As shown in Figure, AMSP provides the same convergence as DeepSpeed.

5.5 Case Study: LLaMA-30B

To demonstrate how different state partitioning strategies affect memory size, communication efficiency, and, ultimately, throughput, we selected the LLaMA-30B model for training at a scale of 512 GPUs as a typical case. This allows us to showcase our findings and the insights we gained from this scenario. Figure 17 provides information about the memory size occupied by the training state after partitioning for four representative optimizer partitioning strategies, as well as the activation occupied memory and throughput.

MiCS(Shared-16) and AMSP (3) exhibit the same runtime memory peak; however, the former achieves nearly half the throughput of the latter. From Figure 17, it is evident that

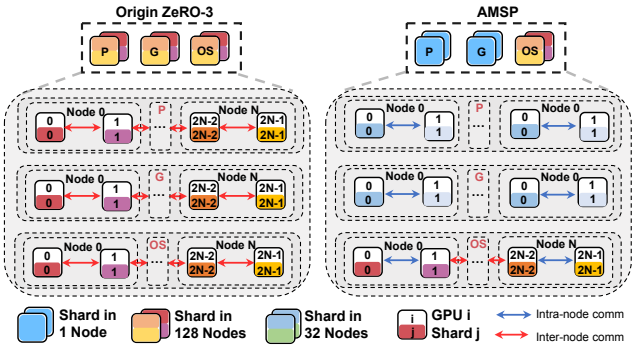

Figure 16. Visualization of the LLaMA-30B training partitioning strategy for AMSP searching in 512 GPUs.

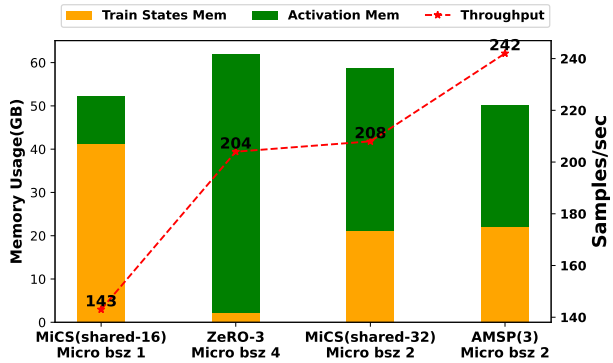

Figure 17. LLaMA-30B training with 512 GPUs throughput and memory usage under different train states sharding strategies.

MiCS(Shared-16), after partitioning, retains too much redundant training state to further increase the micro-batch size. As a result, it needs to perform gradient accumulation to satisfy the global batch size constraint. Nevertheless, the inter-node gradient accumulation in MiCS(Shared-16) is more costly compared to the intra-node gradient accumulation in AMSP (3). The results of this discussion apply similarly to the comparison between MiCS(Shared-32) and AMSP (3).

Conversely, ZeRO-3, which entirely eliminates the redundant storage of the training state compared to MiCS(Shared-16), allows for a larger micro-batch size of 4. A larger micro-batch size increases the proportion of computation to communication time and reduces the number of communications. Finally, when we compare AMSP (3) and ZeRO-3, we observe that AMSP (3) achieves higher throughput with a smaller micro-batch size. This is due to AMSP (3) substantially reducing the communication scope compared to ZeRO-3, avoiding the inefficient DP scope collective communications. In Figure 16, AMSP's optimal partition strategy for LLaMA-30B on 1024 GPUs is displayed. Through the case study of LLaMA-30B, we observed several intriguing phenomena:

- **The cost of redundancy cannot be ignored**, different sharding strategies exhibit significant differences in training efficiency at similar memory footprints. Too much redundancy will lose the opportunity to increase micro-batch-size. A special case arises when limiting redundancy within one node. In this scenario, communication overhead is very low, and additional gradient accumulations are no longer an issue.
- **Memory sharding has marginal effect**, especially when the GPU count exceeds 256. For a ZeRO-3 based 30B model, the training states are already divided into relatively small portions(<5GB), and at this point, the vast majority of the memory is occupied by activations. This suggests that, when scaling the training, it's essential not to expand the partitioning scope of training states without bounds.

6 Related Work

Model state sharding techniques. We summarize several techniques that save GPU memory and support large-scale model training by partitioning the model states. DP [3] serves as the fundamental approach to model training parallelism and has been extensively adopted across various frameworks, such as PyTorch-DDP [25], Horovod [39], and Tensorflow-DDP [1]. In DP, the complete training state is replicated among each rank with distinct input data. The communication overhead arises from the synchronization of gradients among DP ranks. ZeRO [37] and Fully Sharded Data Parallel(FSDP [51]), split parameters, gradients, and optimizer state in the scope of DP. ZeRO family serves as a crucial memory optimization technique in LLM training. ZeRO demonstrates limited horizontal scalability as the scale of the DP scope increases, primarily due to frequent and inefficient inter-node communication. Inspired by these sharding techniques, recent work dynamically selects the model state to partition, balancing memory consumption with communication overhead, thus achieving scalability efficiency. For instance, MiCS[56] chooses to partition the training state within subgroups, trading partial redundancy in the training state for improved communication efficiency. ZeRO++[49] takes a different approach by redundantly storing an additional set of secondary parameters on each node, in exchange for enhanced communication efficiency through parameter pre-fetching. PyTorch FSDP[57] controls the split range of the training state by setting a sharding factor. Setting the sharding factor to 1 results in a fully redundant setup, while aligning it with the DP world size achieves zero redundancy. The aforementioned efforts break the constraint that the split of the training state must be on a global scale, while not attempting to decompose the data dependency of the training state and adopt different partitioning strategies for different training states.

Model parallelism and 3D parallelism. Model parallelism is represented by two approaches: tensor parallelism and pipeline parallelism. Tensor parallelism [30] involves partitioning specific layer weights and introducing additional AllReduce communication. Pipeline parallelism[13, 16, 29,

52] divides the layers of the model horizontally among each rank. Recent innovations have proposed methods that autonomously discern parallelism approaches by intricately melding both data and model parallelism for distinct operators within the model. To illustrate, solutions like Alpa [58], OptCNN [19], FlexFlow [20, 47], and TensorOpt [5] incorporate both data and tensor parallelism. These leverage a variety of search algorithms to refine and enhance the execution of blueprints. However, while these automated parallelism solutions focus on optimizing the partitioning and placement strategies for the optimal operators within the computational graph, they overlook strategies related to the orthogonal placement of the model states.

Large-scale communication optimization. Some works[17, 32, 57] try to overlap communication with computation to mitigate communication costs. ZeRO++ and Espresso[50] utilize quantization and compression techniques to reduce communication volume, albeit at the expense of precision. DEAR[54] aggregates multiple small communications using fixed-size buffers to reduce communication overheads. Hetu[31] leverages hierarchical all-to-all to minimize inter-node communication volume under poor inter-node communication. Similarly, Hybrid AllReduce[18] attempts to decompose a single collective communication primitive into a combination of multiple subgroup communications, while targeting on large scales.

7 Conclusion

Large Language Models (LLMs) are increasingly being trained with more tokens but smaller model sizes. The traditional Zero Redundancy Optimizer (ZeRO) struggles to adapt to this new trend. To address this, we introduced AMSP, a novel training framework. This framework efficiently partitions model states and optimally manages data placement, achieving a 90% scaling efficiency on 1024 GPUs.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. 2016. TensorFlow: A system for large-scale machine learning. *CoRR* abs/1605.08695 (2016).

[2] M. Barnett, L. Shuler, R. van de Geijn, S. Gupta, D.G. Payne, and J. Watts. 1994. Interprocessor collective communication library (InterCom). In *Proceedings of IEEE Scalable High Performance Computing Conference*. 357–364. https://doi.org/10.1109/SHPCC.1994.296665

[3] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Comput. Surv.* 52 (2019), 1–43.

[4] 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 *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20)*. Curran Associates Inc.

[5] Zhenkun Cai, Kaihao Ma, Xiao Yan, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. 2020. TensorOpt: Exploring the Tradeoffs in Distributed DNN Training with Auto-Parallelism. *CoRR* abs/2004.10856 (2020).

[6] InternLM Contributors. 2023. InternLM. https://github.com/InternLM/InternLM.

[7] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems (NeurIPS '12)*.

[8] Apex Developers. 2023. Apex Mixed Precision. https://nvidia.github.io/apex/amp.html#o2-almost-fp16-mixed-precision.

[9] GitHub Developers. 2023. GitHub Copilot. https://copilot.github.com/.

[10] NVIDIA Developers. 2023. NCCL User Guide: ReduceScatter. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/operations.html#reducescatter.

[11] OpenAI Developers. 2023. GPT-3 Apps. https://openai.com/blog/gpt-3-apps/.

[12] Nikoli Dryden, Sam Ade Jacobs, Tim Moon, and Brian Van Essen. 2016. Communication quantization for data-parallel training of deep neural networks. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments (MLHPC '16)*. IEEE Press.

[13] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: a pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. Association for Computing Machinery.

[14] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2018. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR* abs/1706.02677 (2018).

[15] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training Compute-Optimal Large Language Models. *CoRR* abs/2203.15556 (2022).

[16] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao 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 *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NIPS'19)*. Curran Associates Inc.

[17] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based Parameter Propagation for Distributed DNN Training. *CoRR* abs/1905.03960 (2019).

[18] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. 2018. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. (2018).

[19] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. *CoRR* abs/1802.04924 (2018).

[20] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond Data and Model Parallelism for Deep Neural Networks. *CoRR* abs/1807.05358 (2018).

[21] 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).

[22] 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).

[23] John Kim, William J. Dally, and Dennis Abts. 2007. Flattened butterfly: a cost-efficient topology for high-radix networks. *SIGARCH Comput. Archit. News* 35 (2007), 126–137.

[24] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2017).

[25] 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. *CoRR* abs/2006.15704 (2020).

[26] Luo Mai, Chuntao Hong, and Paolo Costa. 2015. Optimizing Network Performance in Distributed Machine Learning. In *7th USENIX Workshop on Hot Topics in Cloud Computing (undefined)*. USENIX Association, undefined.

[27] 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 International Conference on Management of Data (SIGMOD '21)*. Association for Computing Machinery.

[28] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. *CoRR* abs/1710.03740 (2018).

[29] 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 '19)*. Association for Computing Machinery.

[30] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand 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. *CoRR* abs/2104.04473 (2021).

[31] Xiaonan Nie, Pinxue Zhao, Xupeng Miao, Tong Zhao, and Bin Cui. 2022. HetuMoE: An Efficient Trillion-scale Mixture-of-Expert Distributed Training System. *CoRR* abs/2203.14685 (2022).

[32] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 16–29. https://doi.org/10.1145/3341301.3359642

[33] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. 2020. Training Large Neural Networks with Constant Memory using a New Execution Algorithm. *CoRR* abs/2002.05645 (2020).

[34] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[35] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[36] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Ed Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. 2022. Scaling Language Models: Methods, Analysis & Insights from Training Gopher. *CoRR* abs/2112.11446 (2022).

[37] 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 '20)*. IEEE Press.

[38] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. *CoRR* abs/2104.07857 (2021).

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

[40] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2020).

[41] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, 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).

[42] Stanford CRFM Team. 2023. Alpaca: A Strong, Replicable Instruction-Following Model. https://crfm.stanford.edu/2023/03/13/alpaca.html.

[43] Techcrunch. 2023. Microsoft uses GPT-3 to let you code in natural language. https://techcrunch.com/2021/05/25/microsoft-uses-gpt-3-to-let-you-code-in-natural-language/.

[44] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. 2022. LaMDA: Language Models for Dialog Applications. *CoRR* abs/2201.08239 (2022).

[45] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric

Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023).

[46] 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, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023).

[47] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. 2022. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. USENIX Association, 267–284.

[48] Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. 2020. RAT - Resilient Allreduce Tree for Distributed Machine Learning. In *Proceedings of the 4th Asia-Pacific Workshop on Networking (APNet '20)*. Association for Computing Machinery.

[49] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. 2023. ZeRO++: Extremely Efficient Collective Communication for Giant Model Training. *CoRR* abs/2306.10209 (2023).

[50] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. 2023. Hi-Speed DNN Training with Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery.

[51] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. 2020. Automatic Cross-Replica Sharding of Weight Update in Data-Parallel Training. *CoRR* abs/2004.13336 (2020).

[52] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R. Aberger, and Christopher De Sa. 2020. PipeMare: Asynchronous Pipeline Parallel DNN Training. *CoRR* abs/1910.05124 (2020).

[53] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, Peng Zhang, Yuxiao Dong, and Jie Tang. 2022. GLM-130B: An Open Bilingual Pre-trained Model. *CoRR* abs/2210.02414 (2022).

[54] Lin Zhang, Shaohuai Shi, Xiaowen Chu, Wei Wang, Bo Li, and Chengjian Liu. 2023. DeAR: Accelerating Distributed Deep Learning with Fine-Grained All-Reduce Pipelining. *CoRR* abs/2302.12445 (2023).

[55] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. *CoRR* abs/2205.01068 (2022).

[56] 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. *Proceedings of the VLDB Endowment* 16 (2022), 37–50.

[57] 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, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *CoRR* abs/2304.11277 (2023).

[58] 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. *CoRR* abs/2201.12023 (2022).

[59] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. 2010. Parallelized Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems (NeurIPS '10)*.