OSDP: Optimal Sharded Data Parallel for Distributed Deep Learning

Youhe Jiang[1] , **Fangcheng Fu**[1] , **Xupeng Miao**[2] , **Xiaonan Nie**[1] , **Bin Cui**[1,3]
[1]School of CS & Key Lab of High Confidence Software Technologies (MOE), Peking University
[2]Computer Science Department, Carnegie Mellon University
[3]Institute of Computational Social Science, Peking University (Qingdao)
youhejiang@gmail.com, {ccchengff, xupeng.miao, xiaonan.nie, bin.cui}@pku.edu.cn

Abstract

Large-scale deep learning models contribute to significant performance improvements on varieties of downstream tasks. Current data and model parallelism approaches utilize model replication and partition techniques to support the distributed training of ultra-large models. However, directly deploying these systems often leads to sub-optimal training efficiency due to the complex model architectures and the strict device memory constraints. In this paper, we propose Optimal Sharded Data Parallel (OSDP), an automated parallel training system that combines the advantages from both data and model parallelism. Given the model description and the device information, OSDP makes trade-offs between the memory consumption and the hardware utilization, thus automatically generates the distributed computation graph and maximizes the overall system throughput. In addition, OSDP introduces operator splitting to further alleviate peak memory footprints during training with negligible overheads, which enables the trainability of larger models as well as the higher throughput. Extensive experimental results of OSDP on multiple different kinds of large-scale models demonstrate that the proposed strategy outperforms the state-of-the-art in multiple regards.

1 Introduction

Large-scale deep learning (DL) models have achieved great success in the last few years. For example, the pre-trained models, such as ELMo, GPT-3, and LLaMA [Devlin *et al.*, 2018; Raffel *et al.*, 2019; Kaplan *et al.*, 2020; Liu *et al.*, 2021; Touvron *et al.*, 2023], achieve significant accuracy gains with the explosion of the number of model parameters [Shoeybi *et al.*, 2020; ChatGPT authors, 2022; Radford *et al.*, 2019; Brown *et al.*, 2020; Peters *et al.*, 2018; Lin *et al.*, 2021]. However, how to load large models into the limited device (e.g., GPU) memory and perform efficient training remains a huge challenge. For instance, none of existing single GPU devices could accommodate a Transformer-based GPT-3 model with 175 billion parameters without involving model distillation or compression techniques. Therefore, building an efficient distributed training system is becoming increasingly important and indispensable for the advanced exploration of deep learning approaches.

There are a variety of literature on distributed training methodologies such as data parallel [Dean *et al.*, 2012; Shallue *et al.*, 2018; Zinkevich *et al.*, 2010], model parallel, pipeline parallel [Harlap *et al.*, 2018; Huang *et al.*, 2019; Narayanan *et al.*, 2021; Miao *et al.*, 2023; Yang *et al.*, 2021; Nie *et al.*, 2023b] and so on. Data parallelism accelerates the model training by making each device to be responsible for only a fraction of the input data. However, it requires each device to hold a whole model replica during the training process and collaborate with each other through model synchronizations. Apparently, such a redundant model storage does not resolve the memory bottleneck per device. Model parallelism and pipeline parallelism are promising research directions. For example, Megatron-LM [Shoeybi *et al.*, 2020] partitions the model parameters and computation in each layer to multiple devices. But they also bring significant inter-device communications on the intermediate results and lead to unacceptable training efficiency. Recently, Zero Redundancy Optimizer (ZeRO) has been proposed to eliminates the memory redundancies while retaining low communication overheads. It only partitions the model parameters to reduce the memory usage but remains the data parallel computation through sharding and gathering model states across the devices. There are several popular implementations, such as DeepSpeed [Rajbhandari *et al.*, 2020], AngelPTM [Nie *et al.*, 2023a] and Fully Sharded Data Parallel (FSDP) in FairScale [FairScale authors, 2021], and the latter has been integrated into PyTorch [Paszke *et al.*, 2017]. These systems have been successfully used in producing large pretrained models in real industrial scenarios like Microsoft and Meta.

However, these ZeRO-based systems have two major defects: (1) The zero memory redundancy (i.e., all model parameters are sharded) target is overambitious and brings an additional 50% communication overhead compared to vanilla data parallel, resulting in a great hardware efficiency reduction. (2) The gigantic tensors involved during the training process may cause peak memory usage beyond the device's memory capacity. To address the above problems, in this approach, we propose an novel distributed training system Optimal Sharded Data Parallel (OSDP) to achieve a better trade-off between the memory consumption reduction and the training efficiency

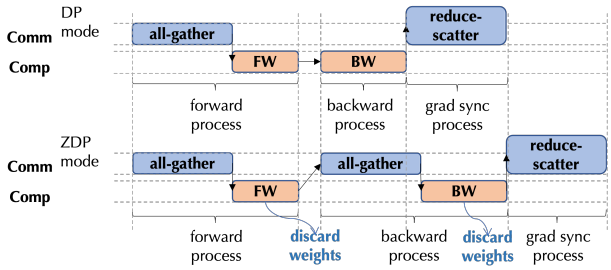


Figure 1: The gantt chart of processing one operator in the Data Parallel (DP) mode and the ZeRO Data Parallel (ZDP) mode, respectively.

improvement. Specifically, OSDP breaks the memory redundancy minimization limitation in previous ZeRO-based systems and enables to determine whether to perform parameter sharding for each operator individually. Moreover, to avoid the gigantic tensors (e.g., `MatMul` outputs), OSDP supports operator splitting and fine-grained memory management, enlarging the entire decision space. Given the specific model description and device information, OSDP provides an efficient search engine to automatically find the optimal parallel strategies for each operator in the computation graph and eventually generates the execution plans. In general, OSDP provides both flexibility and universal applicability to large model distributed training as well as maximizes training efficiency.

We demonstrate the flexibility and efficiency of OSDP on GPT-like Transformers with varying scales and architectures under device memory constraints of 8G and 16G, respectively. Experimental results show that OSDP improves the overall training throughput by up to $2.84\times$ compared with the state-of-the-art parallel training systems.

2 Background and Related Works

2.1 Data Parallel

With the ever-increasing data volume in modern deep learning tasks, data parallelism has become one of the most popular distributed training schemes. When training in data parallel, each process maintains a full copy of the model parameters. In each iteration, each device reads in a mini-batch from different data shards to perform the forward and backward propagation accordingly. Then, model gradients are synchronized among the data parallel processes to update the model parameters. In the era of deep learning, the most commonly used technique for gradient synchronization is the all-reduce collective communication operation. In this work, to ease the analysis of communication cost, we follow the previous studies [Rajbhandari *et al.*, 2020; FairScale authors, 2021] to dissect an all-reduce operation into a reduce-scatter operation and an all-gather operation, as shown in Figure 1.

2.2 Zero Redundancy Data Parallel

As the model becomes larger, data parallelism can hardly afford the memory required to maintain the full copies of model parameters, gradients, and optimizer states on each device. Thus, Zero Redundancy Optimizer (ZeRO) is incorporated into data parallelism, well-known implementations include DeepSpeed and Fully Sharded Data Parallel (FSDP) proposed by FairScale and the latter has been integrated into PyTorch [Rajbhandari *et al.*, 2020; FairScale authors, 2021].

For ZeRO-based systems such as FSDP, the model parameters, gradients, and optimizer states are partitioned according to the data parallel processes. Each process only stores and updates part of the models. The workflow of one iteration is demonstrated in Figure 1. During the training stage, all-gather operations are performed among all processes to get the fully updated parameters, reduced gradients and parameters outside its partition to confirm the complete forward and backward propagation of the model, which greatly reduces the memory consumption of model training while incurs $1.5\times$ communication overhead. During gradient synchronization stage, reduce-scatter operations are performed to synchronize gradients on each device. ZeRO-based systems significantly eliminate model state redundancies during data parallel training, the memory consumption of model states is reduced to $\frac{1}{N}$ compared with before, which ensure continuously efficient training of large-scale models under limited GPU resources.

2.3 Checkpointing

Checkpointing [Chen *et al.*, 2016; Jain *et al.*, 2020; Nie *et al.*, 2022] is a widely-used technique to reduce the memory footprint of intermediate activations during training with additional recomputation, which trades roughly 30% additional computation cost with linear memory allocation. Many recent efforts have been made to incorporate checkpointing with ZeRO data parallelism for better memory management [Rajbhandari *et al.*, 2020; Ren *et al.*, 2021; Rajbhandari *et al.*, 2021]. However, when checkpointing is used in ZeRO-based systems (such as DeepSpeed, FairScale, and PyTorch), an additional round of communication is required for the recomputation phase since model parameters are sharded across data parallel processes.

2.4 Other Parallel Training Strategies

In addition to data parallelism, model parallelism also plays an important role in large-scale model training. There are numerous model parallelism approaches targeting at reducing the memory cost while sustaining training efficiency. For example, Tensor Parallelism (TP) [Shoeybi *et al.*, 2020] divides the model tensors into multiple parts and train them separately on different devices, which greatly reduces the memory cost while incurs frequent communication during training. Pipeline Parallelism (PP) [Huang *et al.*, 2019; Harlap *et al.*, 2018], different from TP, treats the model as a sequence of layers and partitions them into multiple stages across devices to minimize the memory cost, while consistent communication of the intermediate results is necessary to complete the network propagation. Moreover, recent approaches demonstrate that better performance could be achieved in distributed training with the combination of different parallel strategies. For example, PipeDream [Harlap *et al.*, 2018] adopts data parallelism to duplicate the pipeline stages and maximize the system throughput during pipeline parallel training. And a lot of frameworks such as DeepSpeed [Rasley *et al.*, 2020] and Hetu [Miao *et al.*, 2022a; Miao *et al.*, 2022b;

Miao *et al.*, 2022c] provide efficient realization of 3D parallelism (a combination of data, tensor and pipeline parallelism) for large model training.

3 OSDP: Optimal Sharded Data Parallel

In this section, we first formulate the searching problem of execution plans, and then introduce the proposed Optimal Sharded Data Parallel (OSDP) framework. We first present the frequently used notations:

- N: the parallelism degree;
- n: the number of operators in the DL model;
- b: the training batch size;
- p_i: the parallel mode of the i-th operator;

3.1 Motivation and Problem Formulation

Motivation. As introduced in Section 2, Data Parallel (DP) mode and Zero redundancy Data Parallel (ZDP) mode have distinct memory consumption and communication load — as shown in Figure 1, ZDP consumes fewer memory by discarding model states at the cost of extra communication for re-gathering them. Motivated as such, we wish to utilize the trade-offs between the memory consumption reduction and the training efficiency improvement, and eventually maximize the overall system throughput.

To achieve this goal, we develop Optimal Sharded Data Parallel (OSDP), a parallel training framework that guarantees training efficiency as well as breaks the memory bottlenecks of existing data parallel implementations. Given a DL model, OSDP automatically searches for the optimal execution plan that maximizes training throughput while satisfying the device memory limits.

Problem Formulation. To be formal, we formulate the optimal execution plan searching problem as follows. Given a DL model with n operators, where each operator is processed in either DP or ZDP mode, assuming the available device memory is denoted as M_limit, OSDP searches for the optimal parallel modes for all operators $\boldsymbol{p} = \{p_i\}_{i=1}^n$ and the corresponding training batch size b to minimize the averaged training time (i.e., maximize the overall training throughput):

$$\begin{aligned} \boldsymbol{p}^*, b^* = \arg\min_{\boldsymbol{p},b} T(\boldsymbol{p}, b) &:= \frac{1}{b}\sum_{i=1}^{n} T_i(p_i, b) \\ \text{s.t. } M(\boldsymbol{p}, b) &:= \sum_{i=1}^{n} M_i(p_i, b) \le M\_limit, \\ p_i &\in \{DP, ZDP\} \text{ for } i \in \{1, 2, ..., n\}, \\ b &\in Z^+, \end{aligned} \tag{1}$$

where $M_i(p_i, b), T_i(p_i, b)$ denote the memory and time cost of the i-th operator when training in the parallel mode of p_i and with a batch size of b. Then, we introduce how OSDP estimates the memory and time cost, respectively.

To estimate the *memory cost*, we take three types of data into account: model states (including model parameters and optimizer states), intermediate activations, and the extra overhead (such as the temporary workspaces required by the operator). For simplicity, for the i-th operator, the memory consumed by these three types of data are denoted as three factors $M_i^{(model)}, M_i^{(act)}, M_i^{(extra)}$, respectively. Since ZDP shards the model states across the parallel processes, the memory consumption of model states could be amortized to $1/N$, where N is the number of parallel processes. Therefore, the memory cost can be expressed as

$$M_i(p_i, b) = \begin{cases} M_i^{(model)} + bM_i^{(act)} + M_i^{(extra)}, & \text{if } p_i \text{ is } DP \\ \frac{M_i^{(model)}}{N} + bM_i^{(act)} + M_i^{(extra)}, & \text{if } p_i \text{ is } ZDP \end{cases}.$$

It is worthy to note that although the memory factors (i.e., $M_i^{(model)}, M_i^{(act)}, M_i^{(extra)}$) vary for different operators, they can be calculated according to the definition of operators (e.g., types and shapes). Consequently, after the *model description* is provided, OSDP immediately computes the memory factors for the searching of execution plans.

The *time cost* consists of communication and computation cost, where the communication cost is related to the amount of model parameters, while the computation cost is related to the training batch size. Thus, we model the communication and computation cost through the (α, β, γ)-model [Hockney, 1994; Thakur *et al.*, 2005; Cai *et al.*, 2021], where α, β, γ represent the network latency, transfer time per byte, and computation coefficient, respectively.

To model the communication cost, we follow the ring-based all-gather and reduce-scatter operations as supported by NVIDIA Collective Communication Library (NCCL) [Chan *et al.*, 2007]. To accomplish one all-gather or reduce-scatter operation, $N-1$ communication steps are required and the amount of communication in each step is S_i/N, where N is the number of parallel processes and S_i is the size of model parameters for the i-th operator. As depicted in Figure 1, if an operator is processed in ZDP mode, three collective operations (two all-gather operations and one reduce-scatter operation) are needed, resulting in $3(N-1)$ communication steps, while $2(N-1)$ communication steps are expected for DP mode. As for the computation cost, it is proportional to the training batch size and the computation coefficient. Putting them together, the time cost can be modeled as

$$T(i; p_i, b) = \begin{cases} 2(N-1)(\alpha + \frac{S_i}{N}\beta) + b\gamma_i, & \text{if } p_i \text{ is } DP \\ 3(N-1)(\alpha + \frac{S_i}{N}\beta) + b\gamma_i, & \text{if } p_i \text{ is } ZDP \end{cases}.$$

Similar to the memory factors, the size of model parameters (i.e., S_i) can be calculated via the model description. However, the values of α, β, γ_i vary according to hardware ability, experimental environments, and operator types. In practice, we require that such *device information* has been profiled in advance and is provided for the optimal plan searching in OSDP. Our problem formulation does not consider the overlapping between communication and computation, as the communication cost usually dominates in large model training.

3.2 System Overview

Figure 2 demonstrates the workflow of OSDP, which consists of three major modules: the *Profiler*, the *Search Engine*, and the *Scheduler*. As shown in Algorithm 1, these modules work together to maximize the overall system throughput adaptively and automatically. Below we introduce the workflow in depth.

Algorithm 1 Routines of OSDP.

Input: Model Description MD, Device Information DI.
Output: The optimal execution plan $\boldsymbol{p}^*$ and the corresponding batch size b.

Initialize candidate plans $\mathcal{P} \leftarrow \{\}$
// Iteratively increase the training batch size.
for training batch size $b \in \{1, 2, 3, ...\}$ **do**
  $T^*(b) \leftarrow$ INF, $\boldsymbol{p}^*(b) \leftarrow$ None
  // Traverse execution plans via Depth First Search.
  for execution plan $\boldsymbol{p} \in \{DP, ZDP\}^n$ **do**
    Estimate memory and time cost $M(\boldsymbol{p}, b), T(\boldsymbol{p}, b)$
    if $M(\boldsymbol{p}, b) \leq M\_limit$ and $T(\boldsymbol{p}, b) < T^*(b)$ **then**
      $T^*(b) \leftarrow T(\boldsymbol{p}, b), \boldsymbol{p}^*(b) \leftarrow \boldsymbol{p}$
    end if
  end for
  if $\boldsymbol{p}^*(b)$ is None **then**
    // Stop searching since all plans exceed memory limit
    // under the current batch size.
    break
  else
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{(T^*(b), \boldsymbol{p}^*(b), b)\}$
  end if
end for
// Return the optimal execution plan $\boldsymbol{p}^*$ and the
// corresponding batch size b^*.
$\boldsymbol{p}^*, b^* \leftarrow \arg\min_{\boldsymbol{p}, b} \{T(b) | (T(b), \boldsymbol{p}, b) \in \mathcal{P}\}$
return $\boldsymbol{p}^*, b^*$

Profiler. The *Profiler* is responsible for estimating the memory and time cost. In each time of profiling, the *Search Engine* suggests an execution plan $\boldsymbol{p}$ and a batch size b (along with the model description and device information provided by users). The *Profiler* follows the cost model as discussed in Section 3.1 and outputs the estimated memory and time cost.

Search Engine. The *Search Engine* takes as input the memory cost and time cost estimated by the *Profiler*, and adopts Depth First Search (DFS) as the search method. DFS traverses and makes decisions on each operator in the model based on their time and memory cost, ensures the generated plan minimizes the overall time cost while the memory cost is under device memory limit, and eventually, outputs the optimal execution plan and its corresponding estimated system throughput. Additionally, two intuitive pruning schemes are introduced — if the current memory usage exceeds memory limit or the current time cost exceeds the best plan so far, we will prune the searching immediately. Eventually, it takes merely 9-307 seconds in our experiments to complete the search process, which is worthy as the improvement in efficiency can save hours or even days in large model training.

Scheduler. The *Scheduler* iteratively collects the output plan and throughput from *Search Engine* as candidates, and increases the training batch size output to *Profiler* until the minimum possible overall memory cost exceeds device memory limit. Normally when memory is sufficient, a larger training batch size demonstrates a higher system throughput, we only need to output the last candidate of *Scheduler* as the overall optimal execution plan. However, OSDP makes full use of the device memory in every batch size training, which makes it possible to train with a smaller batch size to get a higher system throughput. In this case, *Scheduler* chooses the plan with the highest estimated system throughput among all the candidates and outputs it as the final execution plan.

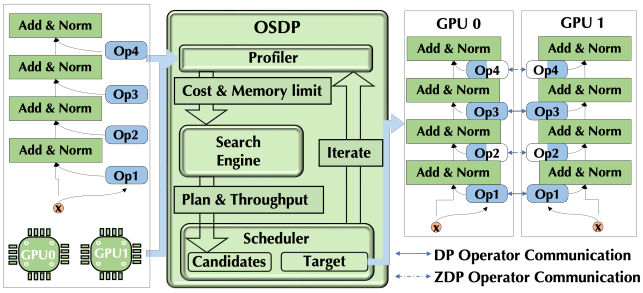


Figure 2: Workflow of OSDP.

3.3 Implementation and Optimization

Implementation. We implement OSDP on top of PyTorch and FairScale. To be specific, OSDP is designed as an efficient distributed training framework that adaptively and automatically determines the data parallel mechanisms. Furthermore, to be user-friendly, we provide a simple but convenient API interface following the FSDP module in FairScale. As shown in Figure 3, by only modifying a few lines of codes, users can easily switch from FSDP to OSDP for better efficiency and scalability. Our code is available[1].

Operator Splitting. In order to alleviate peak memory footprints brought by the huge, bottleneck operators, we further introduce the operator splitting scheme to cooperate with OSDP. Specifically, existing ZeRO-based systems (such as DeepSpeed and FairScale) shard model states across workers during model initialization stage and re-gather them during training stage. Obviously, there is a memory surge during the gathering process — for the huge operators (e.g., `MatMul` operators with large hidden sizes), gathering the corresponding gigantic tensors could turn into peak memory usage and might exceed the device memory. For instance, one of the `MatMul` operators in the GPT-3 model contains 0.6 billion parameters, which ends up to consume 2.24 GB of memory.

The operator splitting aims at reducing the impact of such gigantic tensors in OSDP. Intuitively, given a huge `MatMul` operator, the essential idea is to split the model into several slices and sequentially process them. By doing so, the memory consumed by different slices can be released serially so that the peak memory can be reduced greatly. Figure 4 illustrates the workflow, including three steps. First, both the last dimension of the input data and the first dimension of the operator are partitioned into multiple slices according to an artificially determined slice granularity. Then, the computation of each slice is executed sequentially. Finally, all computation results are summed as the final output.

Combined with ZDP, operator splitting amortizes the memory from $size(\texttt{MatMul})$ to $\frac{size(\texttt{MatMul})}{slice\_granularity}$, which is ex-

[1]https://github.com/Youhe-Jiang/IJCAI2023-OptimalShardedDataParallel

```
  # construct FSDP model
- sharded_module = FSDP(my_module)
  # construct OSDP model
+ sharded_module = OSDP(my_module,
+                       model_description,
+                       device_information)
  # define optimizer
  optim = torch.optim.Optimizer(sharded_module.params(),
                                lr=lr)
  # model training
  for sample, label in dataload.next_batch:
      out = sharded_module(sample)
      loss = criterion(out, label)
      loss.backward()
      optim.step()
```

Figure 3: Comparison between FSDP and OSDP.

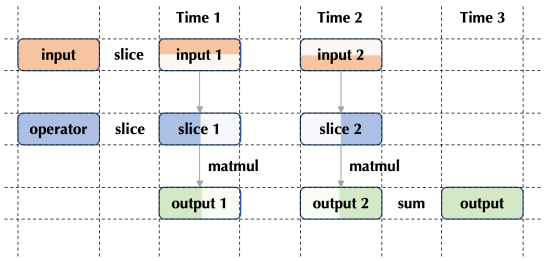


Figure 4: Workflow of operator splitting.

tremely beneficial for large-scale models. In addition, since our distributed training deployment supports the overlapping between computation and communication [Li *et al.*, 2020], as long as the communication cost remains a system bottleneck, the computational overhead caused by slicing and summation can be completely hidden. Thus, the extra overhead of operator splitting on the overall training time is almost negligible.

Putting them together, the performance of OSDP can be improved with operator splitting. As operator splitting partitions one operator into multiple slices, OSDP can treat each slice individually. For instance, instead of assigning the DP or ZDP mode to an entire operator, OSDP can first partition the operator into 4 slices and process 1 of them in the ZDP mode and 3 of them in the DP mode. By such means, OSDP can provide a variety of choices for a single operator and search for a more fine-grained execution plan for the model.

4 Experiments

4.1 Experimental Setup

We first introduce the experimental setup in this work.

Environments. We conduct experiments on two types of hardware environments. Most of our experiments are performed on a laboratorial server equipped with 8 NVIDIA RTX TITAN 24 GB GPUs using PCIe 3.0. We mainly use this hardware environment to assess the effectiveness and efficiency of OSDP. For the multi-server experiments, two cloud servers equipped with NVIDIA A100 GPUs are utilized. The network bandwidth between the two servers is 100 Gb.

Baselines. We compare OSDP with both pure and hybrid parallel strategies. For pure parallel strategies, we choose PyTorch DDP, GPipe, and Megatron-LM [Li *et al.*, 2020; Huang *et al.*, 2019; Shoeybi *et al.*, 2020] as the representatives of DP, PP, and TP, respectively. FairScale [FairScale authors, 2021] is chosen as the representative of FSDP/ZeRO. OSDP-base represents OSDP without operator splitting. We also conduct experiments on hybrid parallelism. To be specific, we compare with DeepSpeed 3D parallelism [Rasley *et al.*, 2020], which integrates DP, PP, and TP together. In addition, since OSDP can be regarded as a substitute of DP, we further replace the DP dimension in 3D parallelism to form a new hybrid parallel strategy called 3D+OSDP, which demonstrates the compatibility of OSDP with existing hybrid strategies. To achieve a fair comparison, we tune the combinations of parallel strategies for hybrid parallelism and report the one with the best performance. By default, we set the slice granularity of our operator splitting technique as 4, and we will conduct more experiments with varying granularities in Section 4.3.

Model	Layer Num	Operator Num	Hidden Size	Param. Num
N&D	48-96	98-194	1024-1536	1.3-2.9B
W&S	2-4	6-10	6144-12288	1.7-4B
I&C	24-96	50-194	1024-4096	0.9-2.3B

Table 1: Statistics of Models

Models. We choose minGPT[2] as our experimental model base, which is a well-known PyTorch re-implementation of GPT training. As shown in Table 1, We propose three different types of models: narrow & deep (**N&D**) models, wide & shallow (**W&S**) models, and inconsistent & consecutive models (**I&C**) models. N&D models have numerous layers with small hidden sizes, which represent models such as GPT-2, Bert, and T5 [Radford *et al.*, 2019; Devlin *et al.*, 2018; Raffel *et al.*, 2019]. W&S models have few layers with large hidden sizes, which represent models such as GPT-3 [Brown *et al.*, 2020] that can only place part of its layer on one device. I&C models have layers with different hidden sizes, which represent models such as Swin transformer [Liu *et al.*, 2021]. For each type of models, we conduct experiments with several configurations to evaluate the universal applicability of our work. All experiments are executed for 100 iterations and the averaged statistics are reported.

4.2 End-to-end Comparison

We first compare the overall training throughput of all counterparts by conducting experiments on N&D, W&S and I&C under the GPU memory limit of 8G and 16G, respectively. The results are provided in Figure 5 and Figure 6.

Comparison with Pure Parallelism. We first discuss the empirical results of pure parallel strategies (i.e., DP, PP, TP, FSDP, and OSDP).

As shown in Figure 5, on the N&D tasks, OSDP achieves a maximum of 174% acceleration compared with the other pure parallel strategies. In particular, OSDP outperforms FSDP with a maximum and an average of 23% and 22% speedup, respectively. It verifies that fine-grained memory management method of OSDP is able to make a balance between the memory consumption and training efficiency, and therefore, provide the system with a higher end-to-end training throughput. On the W&S tasks, due to the huge size of operators, ZeRO optimizer is unsuitable for such a type of models, which leads

[2]https://github.com/karpathy/minGPT

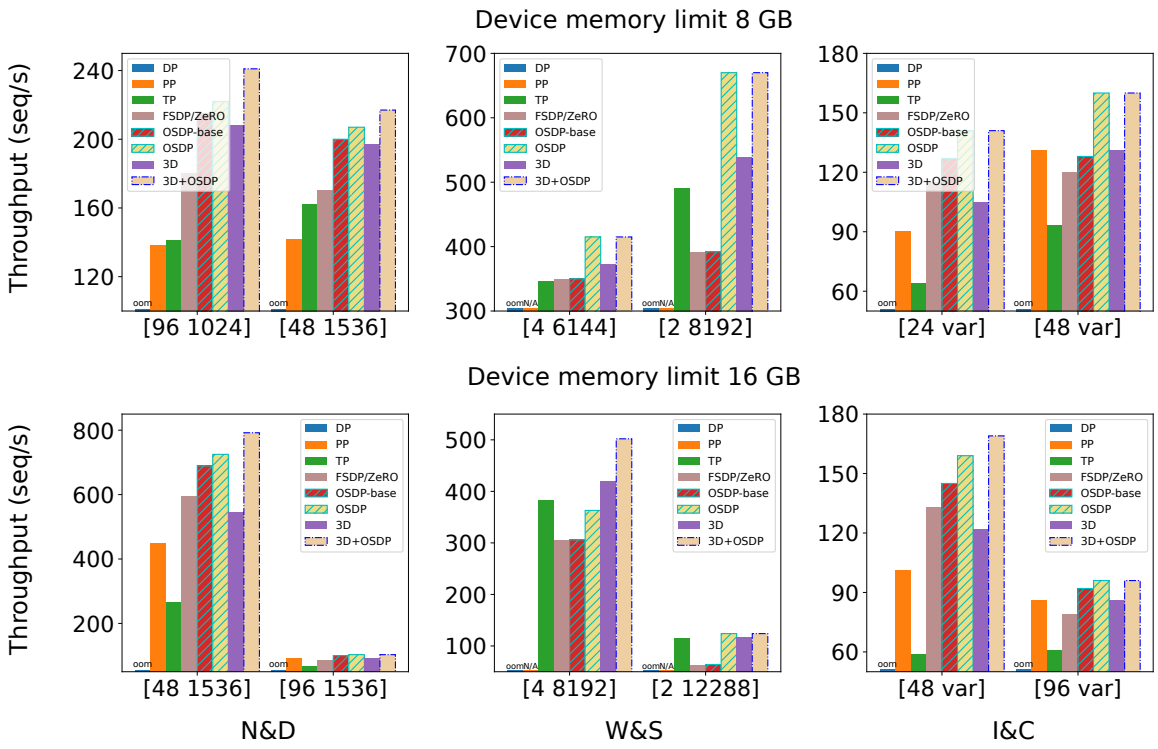


Figure 5: End-to-end comparison of different parallel strategies with 8 GPUs. The x-axis represents different settings for the number of model layers and hidden sizes, the x label represents the model type, and the y-axis represents the overall training throughput. "OOM" indicates out of memory and "N/A" indicates not applicable (PP requires at least 8 layers, so it is not applicable on W&S models).

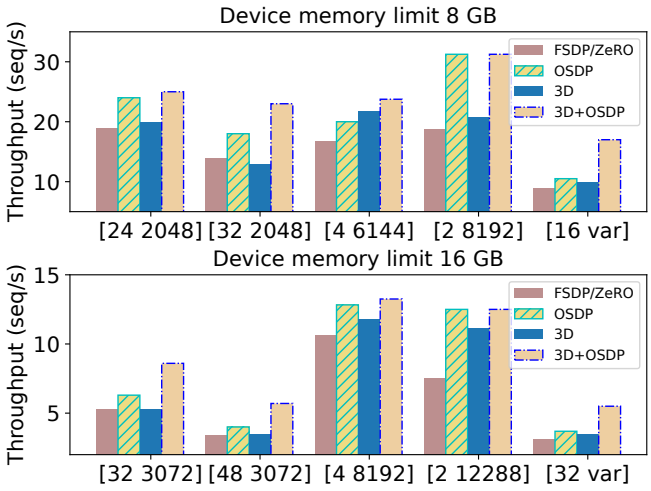


Figure 6: End-to-end comparison of different parallel strategies with 16 GPUs.

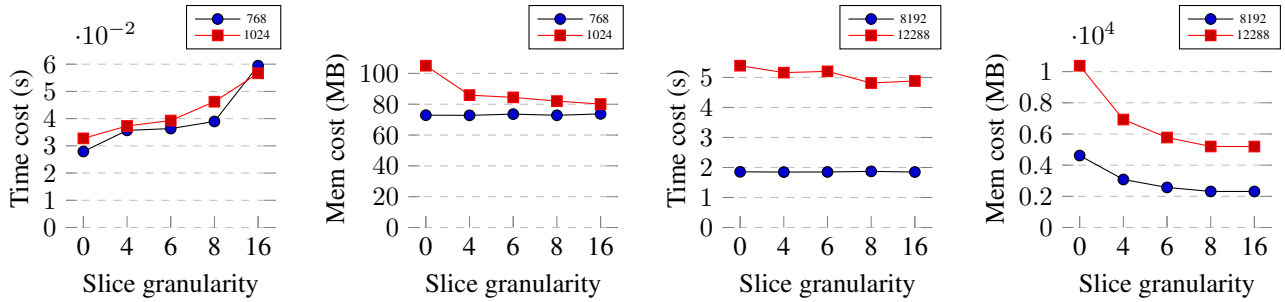
to the unsatisfactory performance of FSDP. However, since OSDP alleviates the peak memory footprint by splitting the huge operators and making fine-grained execution plans, it exibits a much better performance than FSDP — compared with the pure parallel counterparts, OSDP achieves a maximum of 92% and an average of 32% speedup, respectively. Finally, on the I&C tasks, OSDP achievies a maximum of 168% and an average of 33% speedup, demonstrating its flexibility.

The results of two-server experiments also verify the ability of OSDP. As shown in Figure 6, OSDP outperforms FSDP by a maximum of 67% and an average of 29%.

Comparison with Hybrid Parallelism. In addition to pure parallelism, recent studies have proved that using combinatorial parallel strategies could bring further improvement to large-scale model training. In order to assess the performance of OSDP interacting with other parallelism, we further incorporate OSDP with TP and PP to obtain a stronger hybrid parallel strategy, i.e., 3D+OSDP. As shown in Figure 5 and Figure 6, 3D+OSDP consistently achieves the highest training throughput in all experiments. In short, 3D+OSDP outperforms DeepSpeed 3D parallelism by a maximum of 73% and an average of 31%, and achieves a maximum of 184% and an average of 38% acceleration compared with the other baselines. These empirical results prove that OSDP can well fit with other parallel strategies, and therefore, make the training system more flexible and universally applicable.

4.3 More Experiments

Effectiveness of Operator Splitting. To evaluate the effectiveness of operator splitting, we conduct experiments to investigate its impact on memory and time cost, and assess the



(a) Operators with hidden size 768 and 1024. (b) Operators with hidden size 768 and 1024. (c) Operators with hidden size 8192 and 12288. (d) Operators with hidden size 8192 and 12288.

Figure 7: Figure(a)-(b) demonstrate the impact of operator splitting on operators with small hidden sizes, and (c)-(d) present the impact of operator splitting on operators with large hidden sizes (8 GPUs).

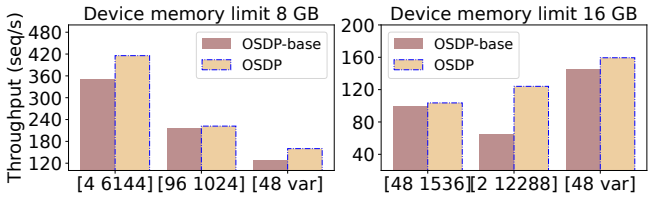


Figure 8: Throughput comparison of OSDP with and without the operator splitting technique (8 GPUs).

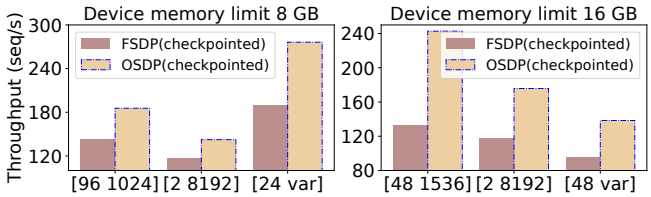


Figure 9: Throughput comparison of OSDP and FSDP with the checkpointing technique (8 GPUs).

improvement it contributes to the overall training efficiency.

We first evaluate the impact of operator splitting on ZDP training. We conduct experiments on operators with hidden sizes from 1024 to 12288 and vary the slice granularity from 0 to 16 (a slice granularity of 0 indicates no operator splitting is performed). As shown in Figure 7, operator splitting alleviates peak memory footprints during training in all experiments, and a maximum of 50% reduction in memory cost is observed. In addition, for operators with small hidden sizes (i.e., 768 and 1024), larger slice granularity leads to an increase in the time cost, while for operators with large hidden sizes (i.e., 8196 and 12288), smaller slice granularity cannot fully reduce the memory cost. These empirical results demonstrate that by tuning the slice granularity for different models and even different operators, OSDP is able to achieve better training performance. And it is an interesting topic to explore how to automatically suggest a desirable slice granularity to each operator according to the model description and device information.

Next, we evaluate the overall training efficiency of OSDP with and without the operator splitting technique under the GPU memory limit of 8G and 16G, respectively. The results are shown in Figure 8. In N&D, approximately 25% of operators are partitioned using splitting for finer mode selection and higher throughput. In W&S, all operators are partitioned, reducing peak memory and enabling larger batches, enhancing throughput. In I&C, around 50% of operators are partitioned, prioritizing larger ones and selectively partitioning smaller ones based on demand, maximizing throughput. In short, the operator splitting technique consistently improves the training throughput by 3%-92%.

Integrating with Checkpointing. As introduced in Section 2, checkpointing is a widely used technique to eliminate the impact on memory brought by activations. Furthermore, checkpointing is usually integrated with the ZDP mode in many real-world applications [Rajbhandari *et al.*, 2020; Ren *et al.*, 2021; Rajbhandari *et al.*, 2021]. To evaluate the impact of checkpointing, we compare OSDP and FSDP with checkpointing enabled. As shown in Figure 9, checkpointing increases the training throughput of OSDP and FSDP (compared with the results in Figure 5). However, the improvement on OSDP is larger — when intergrating with checkpointing, OSDP achieves up to 108.3% and an average of 52.9% speedup compared with FSDP. In fact, when checkpointing is enabled with the ZeRO optimizer, the recomputation process before backward propagation requires an additional round of gathering since each GPU does not maintain a full copy of model states, which has a big impact on the overall training efficiency. By making fine-grained decisions, OSDP can leave operators with smaller memory overhead in DP mode, and therefore, mitigate the side-effect of checkpointing.

5 Conclusion

In this work, we proposed a novel automatic parallel system OSDP, which optimizes data parallel training by making fine-grained trade-offs between memory consumption reduction and the training efficiency improvement. In addition, OSDP supports operator splitting for more fine-grained execution plan decisions and memory optimization. Empirical results demonstrate that OSDP outperforms the state-of-the-art parallel training systems in multiple regards, and achieves up to $2.84\times$ of speedup in terms of the overall system throughput.

Acknowledgments

This work is supported by National Key R&D Program of China (2022ZD0116315), National Natural Science Foundation of China (61832001,U22B2037), and PKU-Tencent joint research Lab. Fangcheng Fu and Bin Cui are the corresponding authors.

References

[Brown *et al.*, 2020] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[Cai *et al.*, 2021] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 62–75, 2021.

[Chan *et al.*, 2007] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[ChatGPT authors, 2022] ChatGPT authors. Chatgpt: Optimizing language models for dialogue. https://openai.com/blog/chatgpt, 2022. Accessed: 2022-11-01.

[Chen *et al.*, 2016] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[Dean *et al.*, 2012] 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. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, volume 25, 2012.

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

[FairScale authors, 2021] FairScale authors. Fairscale: A general purpose modular pytorch library for high performance and large scale training. https://github.com/facebookresearch/fairscale, 2021. Accessed: 2022-11-01.

[Fu *et al.*, 2020] Fangcheng Fu, Yuzheng Hu, Yihan He, Jiawei Jiang, Yingxia Shao, Ce Zhang, and Bin Cui. Don't waste your bits! squeeze activations and gradients for deep neural networks via tinyscript. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020*, volume 119, pages 3304–3314. PMLR, 2020.

[Fu *et al.*, 2022] Fangcheng Fu, Xupeng Miao, Jiawei Jiang, Huanran Xue, and Bin Cui. Towards communication-efficient vertical federated learning training via cache-enabled local update. *Proceedings of the VLDB Endowment*, 15(10):2111–2120, 2022.

[Harlap *et al.*, 2018] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[Hockney, 1994] Roger W Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel computing*, 20(3):389–398, 1994.

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

[Jain *et al.*, 2020] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.

[Kaplan *et al.*, 2020] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

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

[Lin *et al.*, 2021] Junyang Lin, Rui Men, An Yang, Chang Zhou, Ming Ding, Yichang Zhang, Peng Wang, Ang Wang, Le Jiang, Xianyan Jia, Jie Zhang, Jianwei Zhang, Xu Zou, Zhikang Li, Xiaodong Deng, Jie Liu, Jinbao Xue, Huiling Zhou, Jianxin Ma, Jin Yu, Yong Li, Wei Lin, Jingren Zhou, Jie Tang, and Hongxia Yang. M6: A chinese multimodal pretrainer. *arXiv preprint arXiv:2103.00823*, 2021.

[Liu *et al.*, 2021] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. *arXiv preprint arXiv:2103.14030*, 2021.

[Miao *et al.*, 2022a] Xupeng Miao, Xiaonan Nie, Hailin Zhang, Tong Zhao, and Bin Cui. Hetu: A highly efficient automatic parallel distributed deep learning system. *Sci. China Inf. Sci.*, 2022.

[Miao *et al.*, 2022b] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proceedings of the VLDB Endowment*, 16:470–479, 2022.

[Miao *et al.*, 2022c] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. HET: scaling out huge embedding model training via cache-enabled distributed framework. *Proc. VLDB Endow.*, 15(2):312–320, 2022.

[Miao *et al.*, 2023] Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, and Zhihao Jia. Sdpipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training. *Proc. VLDB Endow.*, 16, 2023.

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

[Nie *et al.*, 2022] Xiaonan Nie, Xupeng Miao, Zhi Yang, and Bin Cui. Tsplit: Fine-grained gpu memory management for efficient dnn training via tensor splitting. In *International Conference on Data Engineering*, pages 2615–2628. IEEE, 2022.

[Nie *et al.*, 2023a] Xiaonan Nie, Yi Liu, Fangcheng Fu, Jinbao Xue, Dian Jiao, Xupeng Miao, Yangyu Tao, and Bin Cui. Angel-ptm: A scalable and economical large-scale pre-training system in tencent. *Proceedings of the VLDB Endowment*, 2023.

[Nie *et al.*, 2023b] Xiaonan Nie, Xupeng Miao, Zilong Wang, Zichao Yang, Jilong Xue, Lingxiao Ma, Gang Cao, and Bin Cui. Flexmoe: Scaling large-scale sparse pre-trained model training via dynamic device placement. In *SIGMOD*. ACM, 2023.

[Paszke *et al.*, 2017] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS 2017 Workshop on Autodiff*, 2017.

[Peters *et al.*, 2018] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

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

[Raffel *et al.*, 2019] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[Rajbhandari *et al.*, 2020] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2020.

[Rajbhandari *et al.*, 2021] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. *arXiv preprint arXiv:2104.07857*, 2021.

[Rasley *et al.*, 2020] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[Ren *et al.*, 2021] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. *arXiv preprint arXiv:2101.06840*, 2021.

[Shallue *et al.*, 2018] Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.

[Shoeybi *et al.*, 2020] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2020.

[Thakur *et al.*, 2005] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[Touvron *et al.*, 2023] 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. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[Yang *et al.*, 2021] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Re, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. In *Proceedings of Machine Learning and Systems*, volume 3, pages 269–296, 2021.

[Zinkevich *et al.*, 2010] Martin Zinkevich, Markus Weimer, Alexander J Smola, and Lihong Li. Parallelized stochastic gradient descent. In *NIPS*, volume 4, page 4. Citeseer, 2010.