Accelerating Large-Scale Distributed Neural Network Training with SPMD Parallelism

Shiwei Zhang
The University of Hong Kong
swzhang@cs.hku.hk

Lansong Diao
Alibaba Group
lansong.dls@alibaba-inc.com

Chuan Wu
The University of Hong Kong
cwu@cs.hku.hk

Siyu Wang
Alibaba Group
siyu.wsy@alibaba-inc.com

Wei Lin
Alibaba Group
weilin.lw@alibaba-inc.com

ABSTRACT

Deep neural networks (DNNs) with trillions of parameters have emerged, e.g., Mixture-of-Experts (MoE) models. Training models of this scale requires sophisticated parallelization strategies like the newly proposed SPMD parallelism, that shards each tensor along different dimensions. A common problem using SPMD is that computation stalls during communication due to data dependencies, resulting in low GPU utilization and long training time. We present a general technique to accelerate SPMD-based DNN training by maximizing computation-communication overlap and automatic SPMD strategy search. The key idea is to duplicate the DNN model into two copies that have no dependency, and interleave their execution such that computation of one copy overlaps with communication of the other. We propose a dynamic programming algorithm to automatically identify optimized sharding strategies that minimize model training time by maximally enabling computation-communication overlap. Experiments show that our designs achieve up to 61% training speed-up as compared to existing frameworks.

CCS CONCEPTS
• Computing methodologies → Distributed computing methodologies.

KEYWORDS
Distributed system, Neural networks, Pipeline parallelism

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 INTRODUCTION

Modern deep neural networks (DNNs) have been quickly expanding in size. M6 [19] for multimodal pretraining has 100 billion parameters and GPT-3 [4] for natural language processing includes over 175 billion parameters. The large models have exhibited unprecedented performance that reshaped DNN research, pushing the demand for further increasing the model capacity. Mixture-of-Experts (MoE) layers [33], exemplified by GShard [18] and Switch Transformer [9], have shown strong potential in building high capacity models with trillions or more parameters. Distributed training is necessary for learning these large models with accelerator devices such as GPUs.

A number of parallelisms have been exploited for distributed DNN training. With data parallelism (DP), each device has a full copy of the model, trains it with a distinct portion of the training data, and synchronizes model parameters at the end of each iteration. Model parallelism (MP) splits a DNN model into multiple parts and trains them on different devices with cross-device communication for aggregating shards of a tensor (e.g. a parameter that is partitioned on multiple devices). Pure DP and MP fall short when training very large models as those containing MoE layers [33]. MoE layers are sparse layers that consist of many conditionally activated experts. For each data sample, typically $k$ experts are activated regardless of the total number of experts. This allows the model developer to enlarge the model capacity without increasing computation. MoE models are often too large to fit into a single device as required by DP, and their large communication-to-computation ratios degrade the performance of MP training.
Popularized by GShard [18] and GSPMD [36], Single-Program-Multiple-Data (SPMD) parallelism has shown its success in training MoE models. SPMD can be seen as a generalization of data parallelism and intra-layer model parallelism. With SPMD, each tensor (e.g., parameters) can be split on any of its dimensions, and the compiled program to run on each device is the same [36]. This enables a constant compilation time regardless of the number of devices, important for scaling the training to thousands of devices.

State-of-the-art SPMD training faces two major challenges:

First, switching the sharding dimension (e.g., changing from model-parallel training of a layer to data parallelism of the next layer) in the forward computation pass requires communication. It has been shown that the All-to-All communication takes up to 11% of the per-iteration training time in GSPMD [36], which adopts TPUs with high-speed device-to-device links. In a GPU cluster with Ethernet interconnection, communication time can be substantially larger. How to mitigate the communication overhead is a key to expedite SPMD training.

Next, given a DNN model, how to identify a sharding strategy (i.e., sharding dimensions of each tensor) for SPMD training has not been carefully explored. Current SPMD frameworks [36, 38] require user annotations of the sharding strategy and heuristically infer the strategy for operators that are not explicitly annotated. With the fast emergence of new DNN models, manually designing SPMD strategies for each model is manpower intensive and time-consuming. Further, the optimal sharding strategy depends not only on the DNN structure but also on the configuration of the training cluster (number of devices, inter-device bandwidth, etc.). It is highly desirable to automatically find a good sharding strategy given a DNN model and the cluster specification.

Addressing these issues, we propose a novel design to mitigate communication overhead by overlapping communication with computation in SPMD training. We design a search algorithm to find near-optimal sharding strategies, taking computation-communication overlapping into account.

Our main contributions are summarized as follows:

- We propose a novel method to overlap computation and communication in SPMD training. We split the input data at each device into two microbatches with no dependency in between. Training of the two microbatches on each device can be carried out in parallel, which effectively overlaps the computation of one microbatch with the communication of the other microbatch.
- We build a training time cost model for SPMD sharding strategies and design a dynamic programming-based search algorithm to automatically identify an optimized sharding strategy for a given DNN model and training cluster. The time cost model captures the concurrency between GPU computation and network transfer. The algorithm incrementally finds good strategies for parts of the model and prunes those can be proven not to be a part of the optimal strategy using the time cost model. It produces an op-level sharding strategy that minimizes model training time without affecting model accuracy.
- We implement our system, HiDup, on a representative deep learning framework, PyTorch [24], and conduct experiments on a variety of workloads. Results show competitive performance in single-machine multi-GPU training, and up to 61% speed-up in a 64-GPU cluster, as compared with representative baselines.

2 BACKGROUND AND MOTIVATION

2.1 Neural Network Training

Tensors are the basic data elements in DNN models. A tensor is a multi-dimensional array. There are four main types of tensors in a DNN model: (1) **Activations** are intermediate data generated during forward computation of the model’s loss function. They usually have a “batch size” dimension which is the number of data samples used in a training iteration. (2) **Parameters** are updated during training of a DNN model, to minimize a pre-defined loss function. Parameters may have different dimensions, like “hidden size” for Dense layers and “expert” for MoE layers. (3) **Gradients** of the model parameters are tensors generated through backpropagation. They usually have the same dimensions as their corresponding parameters. (4) **Optimizer states** are used by different optimizers such as Adam [15] and AdaGrad [7], e.g., the momentum, which is of the same dimension as the corresponding parameter.

In each training iteration, a minibatch of data samples is used to train the DNN model, including a forward pass (when the loss of the model is calculated for the minibatch) and a backward pass (aka backpropagation, when gradients of the parameters are calculated in the reverse order of the forward pass). The parameters are synchronized across devices and updated in an update step at the end of each training iteration.

2.2 Mixture-of-Experts Models

Mixture-of-Experts (MoE) models [33] have shown strong potential in various tasks such as sentence completion [9], machine translation [33], multimodal pretraining [19], etc. MoE layers are sparse layers that contain a number of conditionally activated experts, where each expert processes only a selected portion of the data samples. In an MoE layer, a gating network is usually used to compute scores for each expert on each data sample, and each data sample is routed to k experts with the highest scores; then the results of the k experts are aggregated to produce the output, typically weighted by the gating scores.
An important property of MoE layers is that \( k \), the number of experts to activate for each data sample, is chosen independently from the total number of experts. This allows for enlarging the model capacity by adding more experts without increasing model computation time. The MoE layers in a DNN model usually contain the majority of parameters of the model [19]. A common practice is to scale the number of experts proportionally to the total number of GPUs [18]. As a result, MoE models are often too large to fit in a single GPU, and cannot be trained with pure DP. Most of the MoE models are trained using a simple hybrid sharding strategy: the MoE layers are partitioned using model parallelism, while other layers use data parallelism [18][36][19]. This strategy introduces large communication overhead that cannot overlap with computation with existing distributed training frameworks.

### 2.3 SPMD parallelism

To train large DNNs such as MoE models, thousands or more devices are often required [36][38]. SPMD parallelism is widely used in training MoE models [36][38][19], due to its excellent scalability onto a large number of devices.

With SPMD parallelism, tensors are partitioned along different dimensions. Each device only stores a slice of each tensor. Some operators can run on partitioned inputs and produce partitioned results. For example, most operators in a DNN can accept input tensors that are partitioned on the “batch size” dimension. Some operators with multiple inputs have more complex rules on acceptable input partitioning strategies, such as MatMul and Einsum. When the input tensors are partitioned in a manner that the operator does not accept, communication is required to aggregate the shards and recover the full tensor as input to the operator.

MPI-style collective communications [21] are used in SPMD parallelism for communication across layers using different sharding strategies. In collective communication, the devices execute the same communication operation with different ranks. Four types of collective communication are commonly used in SPMD parallelism. All-Gather concatenates the shards of a tensor along a partitioned dimension. All-Reduce aggregates the tensor shards with element-wise reduction (e.g. summation), resulting in identical copies of the aggregated tensor on all devices. Reduce-Scatter can be seen as All-Reduce followed by sharding the resulting tensor on a specified dimension. All-To-All switches the sharding dimension of a tensor, logically equivalent to first running All-Gather on a dimension that the tensor was sharded on and then partitioning the resulted tensors on another dimension.

A tensor may be partitioned on multiple dimensions at the same time and assigned to a mesh of devices [36][32][35]. This can better exploit the multi-dimensional mesh network on accelerators like TPU. As a GPU cluster is not typically organized in a mesh topology, we consider partitioning a tensor at one dimension at each time in this paper.

Example SPMD parallelism strategies on a MatMul layer are given in Fig. 1. Fig. 1a shows a single-card model that consists of a MatMul operator and an opaque loss layer. The input \( X \) can be output from previous layers. The outputs are \( \nabla W \) and \( \nabla X \), gradients of parameter \( W \) and input \( X \).

Fig. 1b gives an example of hybrid parallelism, where the previous layers (that produce \( X \)) are sharded using data parallelism and the MatMul layer is sharded using model parallelism. The half boxes denote sharded tensors and their relative position in the tensor of the equivalent single-card model. Boxes with hachures denote tensors that need to be summed to recover the single-card counterpart. Both \( X \) and \( W \) are sharded, revealing one of the major advantages of hybrid parallelism: it can enable large model training within the same memory consumption as compared with pure data parallelism and model parallelism, where only the activations or the parameters are distributed across devices. However, communication operators cannot run in parallel with any computation: for example, the Loss node can only be computed after the first Reduce-Scatter is done because of its dependency on \( Z \). Fig. 1c shows our proposed solution to this problem, to be detailed in Sec. 3.

### 2.4 Computation-Communication Overlapping

Computation-communication overlapping has been exploited in DNN training. For example of Horovod [30] (communication library used in DP training), gradient synchronization starts before the completion of the backward pass in each training iteration. Deep learning frameworks that advocate pipeline parallelism [11][22] support concurrent activation transfer and model computation. A number of studies have focused on further improving the overlapping ratio for data parallelism, e.g., PACE [3] and ByteScheduler [25].

Data dependency is the key obstacle to achieving a higher level of computation-communication overlapping in SPMD-parallel training. For example, in GShard [18], the expert layers require results of the All-To-All operations as inputs, and hence their computation cannot naturally overlap with the communication.

### 2.5 Opportunities and Challenges

Graph duplication to facilitate communication-computation overlapping. Since the major obstacle that prevents computation-communication overlapping in SPMD parallelism lies in data dependencies, if we can transform the DNN computation graph into a mathematically equivalent one with parallel
branches, we can schedule the operations such that computation in one branch runs in parallel with communication in another. Inspired by data parallelism and gradient accumulation, we propose one such transformation, referred to as Duplex, that is applicable to minibatch training of most DNNs, enabling communication-computation overlap in SPMD parallelism (Sec. 3).

**SPMD strategy needs to be overlapping-aware.** Overlapping computation and communication brings a new challenge: how do we identify SPMD strategies that make the best out of the overlapping? Existing SPMD systems [34, 35, 38] minimize communication costs when exploring sharding strategies, and the strategies are nonetheless inefficient when computation and communication can overlap. Further, algorithms used in these systems rely on the assumption that the total cost (per-iteration training time) is the sum of the costs of operators (computation or communication time). This assumption does not hold when computation time and communication time may overlap. We propose to explicitly formulate computation-communication overlapping in our cost model and use a Pareto optimization variant of dynamic programming to find the optimized SPMD strategy (Sec. 4).

Figure 1: Different SPMD parallelism strategies.
User-friendly implementation on PyTorch. PyTorch [24] is a popular deep learning framework featuring dynamic control flows and user-friendly APIs. Due to its dynamism and lack of static graph representation, graph transformation on PyTorch models is difficult. As a result, most distributed training systems that adopt automatic strategy search do not support PyTorch [2, 34, 35, 37, 40]. We implement HiDup for PyTorch based on the fx [28] module introduced in PyTorch 1.8, which allows us to trace and edit the forward method of a DNN model, such that we can implement op-level sharding on the whole computation graph without requiring the user to change the model code (Sec. 5).

3 DUPLEX

We propose a duplex design for distributed training of large DNN models (e.g., MoE models), allowing efficient overlap of communication and computation. Our idea is to duplicate the assigned computation graph at each device (according to the parallelism strategy adopted) into two copies, each trained using half of the input data. The gradients produced by the two copies are accumulated locally before parameter synchronization.

3.1 Design Principle

When training a DNN model using stochastic gradient descent (SGD), the set of parameters $\theta$ of the DNN model is updated as follows:

$$\theta(t) = \theta(t-1) - \alpha \nabla \ell(X(t), \theta(t-1))$$

where $X = (X_1, X_2, \ldots, X_B)^T$ is a minibatch of $B$ data samples, $\alpha$ is the learning rate and $\ell$ is the loss function. For most models (except those containing BatchNorm layers), the loss and gradients of a minibatch are the sums of those of each data sample:

$$\ell(X, \theta) = \sum_{i=1}^{B} \ell(X_i, \theta), \quad \nabla_{\theta} \ell(X, \theta) = \sum_{i=1}^{B} \nabla_{\theta} \ell(X_i, \theta)$$

This property serves as the basis of data parallelism and gradient accumulation [23]. Instead of calculating gradients of a minibatch all at once, we can split a minibatch into a number of microbatches and calculate gradients of each microbatch independently before updating the parameters:

$$\theta(t) = \theta(t-1) - \alpha \left[ \nabla_{\theta} \ell(X_a^{(t-1)}, \theta(t-1)) + \nabla_{\theta} \ell(X_b^{(t-1)}, \theta(t-1)) \right]$$

where $X_a$ and $X_b$ are the first half and latter half of the minibatch, respectively. Mathematically, (1) is equivalent to (3), while in this way, we can effectively separate the computation of gradients into two identical yet independent components that can be done in parallel.

3.2 Duplex Procedure

We make the training of the assigned computation graph duplex at each device as follows: (1) First, we make a copy of each operator and link its inputs to the copied input operators; (2) Next, we replace the inputs to the model with Chunk operators that split the input minibatch along the “batch size” dimension into two halves (aka two microbatches); (3) Then we add an element-wise summation operator for each pair of the gradients produced with the two microbatches to obtain the full gradient; (4) Finally, CUDA stream synchronization primitives are inserted before and after communication operators to enforce execution order of the two microbatches.

The execution order will be derived as part of our SPMD strategy in Sec. 4.

Fig. 2 illustrates the duplex procedure on a Transformer model consisting of a chain of interleaving attention layers and MoE layers [33]. The MoE layers are often sharded along their expert dimensions, and All-To-All communication operations are inserted before and after these layers to dispatch patch tensors across different workers to their corresponding experts and distribute them back after being processed by the experts [18]. For duplex, we first duplicate the graph into two, each taking half of the minibatch as input and computing the loss and gradients. Instead of sequentially training the two microbatches, we parallelize their training over the two copies of the graph, such that computation of one microbatch overlaps with communication of the other.

As a more concrete example, Fig. 1c shows the computation graph after applying duplex on Fig. 1b: the resulting graph on each worker has two chains that do not intersect until gradient aggregation; computation and communication can then be interleaved to reduce the execution time.

The computation-to-communication ratio is a key factor determining duplex’s performance. Ideally, if every computation block (a series of computation without communication) and communication operations take the same time, we can keep both GPUs and network links busy and save the overall
training time by 50%. On the other hand, duplex introduces additional overheads. CUDA synchronization is required to orchestrate the execution of the two microbatches. Smaller batch sizes may lead to GPU underutilization, as well as reduce the efficiency of collective communication. Nonetheless, they are negligible in most cases compared to the training time saving with duplex. The synchronization is local on each GPU and does not require cross-device communication. The element-wise addition operations used for gradient accumulation take less than 80μs.

The duplex’s design can be generalized to more than two microbatches, by which we have a larger scheduling space but the potential to achieve better overlapping. However, further splitting the tensors leads to more overheads due to even smaller tensor sizes and more CUDA stream synchronizations. We focus on duplex with two microbatches in our design, practically striking a good balance between training time saving and additional overhead.

We note that the memory usage with our duplex design is similar to one without duplex. The two microbatches share the same version of parameters and optimizer states. The activations produced by the two microbatches each are half of the size of activations in the original training graph, due to the reduced batch size per microbatch. Gradients are produced on two microbatches separately, but are accumulated as soon as produced by both microbatches. Therefore, the overall memory consumption is about the same as training the original graph without duplex.

Further, the duplex design can be applied to both computation graphs that contain only forward computation (e.g., PyTorch models [24]) and computation graphs that have both forward and backward passes (e.g., training graphs on TensorFlow [1]). In the former case, the forward pass and the backward pass use the same duplex execution: if a pair of computation and communication overlap in the forward pass, their corresponding backward operations also overlap in the backward pass.

4 DUPLEX-AWARE SPMD STRATEGY

We propose a dynamic programming-based searching method to decide the SPMD strategy for training a DNN model on a given GPU cluster. The strategy search explicitly considers computation-communication overlapping to minimize the per-iteration training time with our duplex design.

4.1 Problem Definition

Computation Graph. A DNN model is defined by a computation graph G, in which the nodes are operators and the edges represent tensors. There exists data dependency between operators that are connected by tensors: the operator that consumes a tensor can only start after its predecessors have been done. In the example graphs in Fig. 1, to illustrate the sharding strategies, we also plot the tensors as nodes.

Tensor Form. A tensor can be in different forms depending on the sharding strategy. When a tensor is not sharded, it is in the Full form, i.e., a complete copy of this tensor resides with each of some workers, identical to that in a single-card model. We define other forms of a sharded tensor according to the approach that can be used to transform the tensor into the Full form. If a tensor is sharded on its i-th dimension among a number of workers, performing All-Gather among the workers allows each worker to collect the missing parts from the other workers, which turns this tensor into the Full form; we specify this sharded form as Gather(i). The Reduce form indicates that the sharded tensor (usually the output of MatMul operators) can be transformed into the Full form with All-Reduce operations. The form of a tensor can be changed using collective communications. For example, a tensor can be transformed from the Reduce form into a Gather(i) form with Reduce-Scatter, or from the Gather(i) form to the Gather(j) form using All-To-All, where i and j indicate different dimensions on which the tensor is sharded.

Operator Signature. We use signatures of an operator to specify acceptable forms of input tensors and the corresponding forms of output tensors at the operator. For example, Gather(0), Full -> Gather(0) is a signature of the MatMul operator, indicating that MatMul can run on multiple devices with the first input tensor sharded on the first dimension (e.g., the “batch size”) and the second input tensor not sharded, and produces a tensor that is sharded on its first dimension. Similarly, Gather(1), Gather(0) -> Reduce is another signature of MatMul where both inputs are sharded and the output tensor can be aggregated using All-Reduce.

Stages. Training of a microbatch over the DNN computation graph on a device alternates between computation and communication. We divide the training process of a microbatch into stages; each stage consists of a communication step that contains a set of communication operators followed by a computation step that consists of multiple computation operators. Only the first stage does not include a communication step. In our duplex design, we always overlap the computation step of a stage in the first microbatch’s training with the communication step of the corresponding stage in the second microbatch’s training, and the computation step of the second microbatch with the communication step of the next stage of the first microbatch, to best exploit computation and communication resources. The division of stages, i.e., which communication operations and computation operations to put in a stage, is part of our strategy search space.

Strategy. A strategy describes how to shard tensors in a given DNN model and stage division of the computation
graph for parallel training with our duplex design. A strategy $Q = \{(O_1, S_1), (O_2, S_2), \ldots\}$ specifies a series of stages on the computation graph that starts from a stage including input nodes and leads to a certain sharding/stage division state of the DNN model, state($Q$). Here $O_i$ denotes the set of computation operators in the $i$-th stage and $S_i$ is the set of signatures of these operators. A stage $T = (C, F)$ is described by $C$, a cut in the computation graph that separates input nodes and the loss node (represented by the set of tensors on the cut), and $F$, the forms of the tensors in $C$. Executing the stages in $Q$ from the inputs leads to state($Q$). The final state $\tilde{T} = ((L), \{\text{Reduce}\})$, where $L$ denotes the loss tensor, corresponds to the state when all tensor sharding decisions and stage divisions are decided for the entire computation graph: the last computation in forward pass of SGD-based training (Eq. 2) is to sum the losses produced on the devices to obtain the loss of the whole minibatch and therefore we have the Reduce form of the loss tensor. We refer to a strategy that leads to the final state (i.e., state($Q$) = $\tilde{T}$) as a complete strategy.

The communication step in each stage is decided as follows. When a stage $(O_i, S_i)$ is added into a strategy $Q$, if the forms of the inputs to operators in $O_i$ required by $S_i$ mismatch those in state($Q$), communication operations are inserted at the beginning of the stage. For example, if a tensor in state($Q$) is in the Gather($\emptyset$) form while the same tensor is of the Gather(1) form in $S_i$, an AllToAll-$\text{ToAll}$ operator is included in the communication step of stage $(O_i, S_i)$.

Objective. The goal of our strategy search is to find the optimal complete strategy $Q^*$ that achieves the smallest training time of the DNN model.

4.2 Strategy Search Algorithm

To identify the optimal strategy, we carefully analyze costs associated with each strategy (that reflect the training time with our duplex training), and exploit them in a dynamic programming-based search algorithm.

With our duplex design, two microbatches are trained on two copies of the computation graph in parallel. The computation step in a stage of the second microbatch’s training may overlap with the communication in the next stage of the first microbatch’s training (Fig. 2). The end-to-end execution time of the computation graph under our duplex training is the time required to execute the complete computation graph for the two microbatches and produce the aggregated loss and gradients across all devices.

To capture the overlapping effect in end-to-end execution time, we define two costs associated with a strategy $Q$: (i) $\psi_Q$, the time to execute the computation and communication steps for the two microbatches from the input nodes till the cut $C$ in state($Q$) (referred to as the training time to reach state($Q$)), when the strategy $Q$ is used; (ii) $\phi_Q$, the additive inverse (i.e., negation) of one microbatch’s execution time of the computation step in the last stage in $Q$. We define $\phi_Q$ as a negative number to make our strategy searching a minimization problem. Intuitively, $\psi_Q$ is the training time required to reach state($Q$). $-\phi_Q$ quantifies how much the time in $\psi_Q$ can overlap with the execution of the next stage that can be appended to $Q$. For a complete strategy $Q$, $\psi_Q$ is the end-to-end execution time of the two microbatches on the complete computation graph. Tracking the two costs separately allows us to recursively calculate the costs of a strategy based on the costs of its sub-strategies, as well as facilitate more accurate estimation of the end-to-end execution time with our duplex training.

For a strategy $Q$ of $m$ stages, let $Q(i)$ denote the sub-strategy that contains the first $i$ stages in $Q$ $(i \leq m)$. The costs of $Q(i)$ can be calculated as follows, where $\text{comm}^{(i)}$ and $\text{comp}^{(i)}$ denote one microbatch’s communication time and computation time of the $i$-th stage, respectively:

$$
\phi_{Q(i)} = -\text{comp}^{(i)}
$$

$$
\psi_{Q(i)} = \psi_{Q(i-1)} + \phi_{Q(i-1)} + \max\{ -\phi_{Q(i-1)}, \text{comm}^{(i)} \}_{\text{microbatch 2}} - \text{microbatch 1}
$$

$$
+ \max\{ \text{comp}^{(i)}_{\text{microbatch 2}}, \text{comp}^{(i)}_{\text{microbatch 1}} \} - \phi_{Q(i)}
$$

In the RHS of (5), $\psi_{Q(i-1)} + \phi_{Q(i-1)}$ gives the training time to reach state($Q(i-1)$) (for both microbatches), without counting the second microbatch’s computation time in the $(i-1)$-th stage. The third term decides the time required to run the second microbatch’s computation in the $(i-1)$-th stage and the first microbatch’s communication step in the $i$-th stage, which can happen in parallel. The fourth term similarly describes the overlapped execution time of the first microbatch’s computation and the second microbatch’s communication in the $i$-th stage. The last term indicates the second microbatch’s computation time in the $i$-th stage, which is not overlapped with communication at this stage and will be considered again when we append the $(i+1)$-th stage into the strategy.

Fig. 3 illustrates cost calculation for the strategy in Fig. 1c. $Q^{(i-1)}$ reaches cut $\{Z\}$ (Fig. 1a) and stage $i$ is appended to $Q^{(i-1)}$ to obtain $Q^{(i)}$. The time segments in the bottom of the figure visualize Eq. (5). For example, the first segment indicates when the first microbatch finishes stage $i-1$ and the second microbatch finishes only AllToAll-$\text{ToAll}$ of stage $i - 1$. The second segment corresponds to the time of executing ReduceScatter and MatMul in parallel. The dotted lines separate stages.

Due to computation-communication overlapping across two microbatches’ training, we need to track both costs using...
the recursive computation in (4) and (5) in our dynamic programming algorithm, to identify the strategy minimizing the completion time of training both microbatches. In our strategy search, we need to preserve any strategy that is not dominated by other strategies [29], in terms of the two costs computed in (4) and (5) with the strategy. We define that strategy \( Q_1 \) dominates \( Q_2 \) if they reach the same state, while \( Q_1 \) is better in terms of both costs, i.e.:

\[
Q_1 \text{ dominates } Q_2 \iff \begin{aligned}
\text{state}(Q_1) &= \text{state}(Q_2) \\
\psi_{Q_1} &\leq \psi_{Q_2} \\
\phi_{Q_1} &\leq \phi_{Q_2} \\
\psi_{Q_1} &< \psi_{Q_2} \text{ or } \phi_{Q_1} < \phi_{Q_2}.
\end{aligned}
\]

The minimal time required to execute the remaining part of the graph from cut \( C \) to the loss node depends only on the forms of tensors in \( C \) (as we can treat the remaining part as a standalone model and tensors in \( C \) are its inputs). Therefore, if two strategies reach the same state \( T \), one can be regarded as strictly better than the other if it takes a shorter time to reach the state and brings more overlapping potential between the two microbatches’ training (indicated by longer computation time in the last stage of the strategy). We formalize the idea in Theorem 1. The proof is provided in a technical report.

**Theorem 1.** \( Q \not\leq Q' \) if \( \exists Q'' \) such that \( Q' \) dominates \( Q \).

Exploiting the result in Theorem 1, we propose an efficient dynamic programming algorithm to find the optimal strategy \( Q^* \) for a computation graph \( G \), as given in Fig. 4. We iterate through all possible cuts \( C_0, \ldots, C_n \) in the graph in an order that ensures that \( V_{e_1} \in C_i, V_{e_2} \in C_j, i < j, e_1 \neq e_2 \), there is no path from \( e_2 \) to \( e_1 \). The cuts can be enumerated with breadth-first search: starting with a set \( R \) that contains only the input nodes, we enumerate nodes whose input tensors are produced by nodes in \( R \) (as a set \( J \) and try to add a different node in \( J \) into \( R \) each time. The tensors produced by nodes in \( R \) and consumed by nodes not in \( R \) form a cut. The number of possible cuts is exponential to the maximum number of nodes in \( J \), which is 10 in our experiments. For each cut, we enumerate possible combinations of operators \( O \) and their signatures \( S \) to form stages \((O,S)\), that can reach a state including this cut and can be appended to the Pareto optimal strategies \( P \) (set of strategies that are not dominated by any other strategies). We only keep a strategy if it is not dominated by any other strategies (lines 8–10), and eliminate any strategies that are dominated by it (lines 11–15). Finally, we decide \( Q^* \) as the complete strategy achieving the smallest \( \psi_Q \), which is the end-to-end training time of the two microbatches (line 20).

### 5 IMPLEMENTATION

We implement HiDup as a graph transformation module on PyTorch [24], as shown in Fig. 5. HiDup takes as input a single-card PyTorch DNN model (as a PyTorch fx [28] graph) and the cluster specification (number of GPUs, interconnection bandwidth, etc.), and produces a model that can run on multiple GPUs. HiDup consists of three components.

**Annotator.** The annotator adds metadata to each node in the computation graph, including operator signatures, estimated computation time, and the output size. For each node, the output tensor size is inferred according to the sizes of input tensors. Possible signatures of an operator are derived according to the inputs and the operator type. The computation time of an operator is estimated using the number of floating-point operations required for the operator. We profile small models (e.g., a model with a reduced number of
layers) to obtain the device flops. The annotator supports 37 PyTorch operators that are used in our experiments, and can be easily extended to support more operators.

**Strategy Searcher.** The strategy searcher implements our SPMD strategy search algorithm to identify the best sharding strategy. The strategy searcher estimates the communication costs using the inferred size of the respective tensor and the inter-device bandwidth provided in the cluster specification.

**Compiler.** The compiler edits the single-card DNN computation graph according to the identified sharding strategy and adds synchronization operations to enforce stage execution order in our duplex training design. For each stage in the strategy, it generates communication operations required in the stage and copies the computation operators in the stage from the original single-card graph. It runs this procedure twice on two CUDA streams to generate operations for the two microbatches. Before and after each communication step, it inserts CUDA stream synchronization primitives to prevent consecutively running two stages on one microbatch without switching to the other one, which ensures alternating execution of the two microbatches.

### 6 EVALUATION

#### 6.1 Experimental Setup

**Testbed.** By default, we conduct experiments on 8 machines in a public cloud, each equipped with 8 NVIDIA V100 GPUs and NVLink. Inter-machine bandwidth is about 9.71Gbps, measured using `iperf3` [8]. Static resource allocation and exclusive access to the cluster are ensured during our experiments.

**Benchmarks.** We train four models that cover language modeling and image classification. We add MoE layers to the BERT [5] and ViT [6] models by replacing a feed-forward module every two layers, in a similar manner as in GShard [18]. We add two types of MoE layers to the models: BERT-SGMoE and ViT-SGMoE use Sparsely-Gated Mixture-of-Experts layers [33] with $k = 2$; BERT-Switch and ViT-Switch use Switch Transformer [9] layers. We follow the common practice of weak scaling in training these MoE models, i.e., set the global batch size and the total number of experts proportional to the number of GPUs. For language model pretraining, we train BERT-SGMoE and BERT-Switch on the WikiText-103 [20] dataset. For image classification, we train ViT-SGMoE and ViT-Switch on the Cifar-10 [16] dataset.

#### Baselines

Except for the MoE layers, we mostly use the same transformer configurations as in BERT-Base [5], and reduce the number of layers to 8 in order to allow for training the models with more experts in our testbed. We use 2 experts on each GPU for ViT-SGMoE and ViT-Switch, and 1 expert per GPU for BERT-SGMoE and BERT-Switch (because they are more memory-demanding). The models are trained with the Adam optimizer [15]. The numbers of operators and parameters in the models are summarized in Table 1. $n$ denotes the number of GPUs.

#### Table 1: Benchmark models

<table>
<thead>
<tr>
<th>Model</th>
<th>Operators</th>
<th>Parameters (Millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT-SGMoE</td>
<td>250</td>
<td>89 + 19n</td>
</tr>
<tr>
<td>BERT-Switch</td>
<td>250</td>
<td>89 + 19n</td>
</tr>
<tr>
<td>ViT-SGMoE</td>
<td>254</td>
<td>38 + 38n</td>
</tr>
<tr>
<td>ViT-Switch</td>
<td>254</td>
<td>38 + 38n</td>
</tr>
</tbody>
</table>

Directly profiling computation time in large MoE models is often costly and difficult without sharding the large models first.
through our experiments that setting it to the total number of GPUs always gives the best result.

Pure DP-based methods do not support MoE model training at large scale due to replicating every expert on every device and that the number of experts is proportional to the number of GPUs in the standard MoE settings. We hence only include PyTorch DDP and Horovod in single-machine experiments.

**Evaluation Method.** HiDup and the baselines do not change the training semantics (i.e., producing the same gradients as training over single-card models with only floating-point errors), such that the number of iterations required for model convergence to specified accuracies remains the same as single-card training. Therefore, our comparison of per-iteration training time reflects that of the end-to-end training time.

Under each configuration, we train the respective model for 100 iterations and show the average time of the last 50 iterations. The pre-training overheads of the evaluated systems, such as the fused operator compilation in DeepSpeed and strategy search in HiDup, are within tens of seconds and negligible compared to the DNN training time.

### 6.2 Scalability

We first evaluate the per-iteration training time of HiDup as compared to baselines on up to 64 GPUs across 8 machines. Fig. 6 shows that similar performance is achieved among HiDup, FastMoE, and DeepSpeed when training BERT-SGMoE and BERT-Switch with up to 16 GPUs, while HiDup achieves up to 18% speed-up as compared to the best baseline when more GPUs are in use. This is because when we add more devices, collective communication becomes slower due to bandwidth contention. Compared with FastMoE and DeepSpeed, our duplex design can mitigate the increased communication overhead by overlapping computation and communication, achieving better results.

Benefited from its optimized implementation, FastMoE achieves the best performance in 8-GPU training, but is bottlenecked by communication with more GPUs. It also experiences OOM in 64-GPU training, due to the lack of per-expert processing capacity limit. DeepSpeed’s MoE module shows a similar scaling trend as HiDup, but is consistently slower than HiDup as it does not overlap computation and communication.

When training ViT-SGMoE and ViT-Switch, HiDup achieves significantly better performance (up to 61% faster) than the baselines. Computation time and communication time in these models are closer to each other and HiDup can achieve higher overlapping ratios.

### 6.3 Single-Machine Performance

We also evaluate HiDup in a single machine of 8 GPUs, which represents a high-bandwidth inter-connect scenario
as the devices are connected by NVLinks of 200 Gbps uni-
directional pairwise bandwidth. As Fig. 7 shows, pure DP-
based methods (Horovod and PyTorch DDP) generally achieve
good performance in this single-machine scenario, because
communication can fully overlap with backward pass compu-
tation under the very high bandwidth. HiDup automatically
identifies similar strategies and achieves comparable perfor-
ance as Horovod and PyTorch DDP, despite the additional
overhead introduced by our duplex design.

6.4 Performance under Different
Bandwidth Levels
This set of experiments were conducted on an on-premise
cluster of 2 machines connected to a Dell Z9100-ON switch,
each equipped with 4 NVIDIA V100 GPUs and NVLink. We
evaluate HiDup under different inter-machine bandwidth
levels, by limiting the bandwidth using tc tool in iproute2
package [17]. As Fig. 8 shows, HiDup outperforms the base-
ilines by up to 14% as it automatically optimizes the SPMD
strategies for different bandwidth levels and hides communi-
tication time within computation time with our duplex design
when the bandwidth is smaller.

6.5 Performance with Different Batch Sizes
We train BERT-SGMoE using different per-card batch sizes
on two machines that are connected by a 100Gbps RDMA
network. Each machine is equipped with 4 NVIDIA V100
GPUs and NVLink. As Fig. 9a shows, HiDup outperforms
the two baselines when batch size is larger than 16. When
the batch size is smaller, HiDup’s performance is similar to
the baselines, because its duplex training further splits the
minibatch and results in lower GPU utilization.

To further measure GPU underutilization caused by de-
creased batch sizes, we run the Einsum operation used in
BERT-SGMoE with different batch sizes and calculate the
throughput as batch size divided by computation time. As
shown in Fig. 9b, the computation throughput is similar when
the batch size is larger than 64.

6.6 Interference between Computation and
Communication
When computation and communication overlap, they may
compete for GPU resources including processors, memory
bandwidth, caches, etc. This interference may reduce the
benefits brought by duplex training. We measure the inter-
ference between computation and communication by pro-
fileing the performance of MatMul operations on one CUDA
stream while running AllReduce on another CUDA stream
and compare it with the performance of MatMul without
communication. This shows the worst case of interference
under 100% overlapping. We conduct this experiment on

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Tensor Size</th>
<th>Additional time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Machine</td>
<td>256MB</td>
<td>14.0%</td>
</tr>
<tr>
<td>Single Machine</td>
<td>256KB</td>
<td>3.5%</td>
</tr>
<tr>
<td>Two Machines (100Gbps)</td>
<td>256MB</td>
<td>1.9%</td>
</tr>
<tr>
<td>Two Machines (100Gbps)</td>
<td>256KB</td>
<td>2.0%</td>
</tr>
</tbody>
</table>

Table 3: Peak GPU memory usage (GB).

<table>
<thead>
<tr>
<th></th>
<th>B-SGMoE</th>
<th>B-Switch</th>
<th>V-SGMoE</th>
<th>V-Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>HiDup</td>
<td>6.77</td>
<td>6.34</td>
<td>2.79</td>
<td>2.60</td>
</tr>
<tr>
<td>DDP</td>
<td>9.87</td>
<td>9.44</td>
<td>4.92</td>
<td>4.68</td>
</tr>
<tr>
<td>Horovod</td>
<td>8.98</td>
<td>8.55</td>
<td>4.21</td>
<td>3.97</td>
</tr>
<tr>
<td>FastMoE</td>
<td>7.63</td>
<td>7.30</td>
<td>2.78</td>
<td>2.63</td>
</tr>
<tr>
<td>DeepSpeed</td>
<td>8.09</td>
<td>7.80</td>
<td>2.53</td>
<td>2.52</td>
</tr>
</tbody>
</table>

the same cluster as in Sec. 6.5 (two machines with 100Gbps
inter-connection). As Table 2 shows, the interference is only
significant when the tensor size is relatively large and the
communication is within a machine. Since HiDup targets
distributed DNN training, the impact of interference is small.

6.7 GPU Memory Usage
We show the peak GPU memory usage when training the
benchmark models with 4 GPUs on one machine. The usage
is recorded using torch.cuda.max_memory_allocated. As
Table 3 shows, HiDup uses the least memory for BERT mod-
els and similar memory levels as the best baseline for ViT
models. Duplex training slightly increases the memory con-
sumption due to producing each gradient twice, but HiDup’s
automatic strategy search can shard more tensors than the
baselines and reduce the per-card memory usage.

6.8 Training Time Breakdown
To gain further insights on HiDup’s performance, we ana-
lyze computation time and communication time separately
from the execution trace of BERT-SGMoE. In Table 4, the
wall time is the real-world time spanning an iteration. Total
time is the sum of computation time and communication
time without considering their overlapping. The overlapping
ratio is calculated by dividing the duration when compu-
tation and communication are overlapped by the smaller
one between computation time and communication time.
We see that HiDup can hide more than 91% and 41% of the
communication in computation in the single and two ma-
chines (100Gbps) cases, respectively. It does not completely
hide all communication because of uneven communication
times and computation times in the pipeline. As illustrated
in Fig. 3, a communication step needs to be shorter than the
computation step of the same stage and that of the previous
stage to be fully overlapped. We see that HiDup’s total time
We analyze the strategies found by HiDup under different bandwidth levels. We first show the placement strategies of parameters. With SPMD parallelism, there are two major placement strategies for a parameter: replication and sharding. When using replication, each device holds a full copy of the parameter, and All-Reduce is needed to aggregate the gradients in the backward pass. When using sharding, the parameter is split along a dimension and each device stores a slice. All-Gather is used to recover the full parameter in the forward pass and Reduce-Scatter is used to aggregate the gradients in the backward pass. Conceptually, an All-Reduce operation is equivalent to Reduce-Scatter followed by All-Gather. However, due to some low-level optimizations [12], All-Reduce in NCCL can be faster than running Reduce-Scatter and All-Gather separately.

Intuitively, the placement strategies for the parameters can be chosen according to the following principles: (1) When the bandwidth is low, replication is preferred, because the communication time dominates the training time which can hardly be hidden within computation time, and the communication time of All-Reduce is lower which leads to a shorter training time than sharding. (2) When the bandwidth is high, the communication may be hidden within the computation time and the overlapping ratio plays a more important role in the overall training time. In this case, sharding could be preferred. Even though it leads to longer communication time, the communication lies in both the forward pass and the backward pass, while the All-Reduce operation in the replication case can only overlap with the backward computation. Therefore, sharding can lead to a higher overlapping ratio and may lead to faster training in high bandwidth cases.

Fig. 11 shows the number of parameter tensors that use the two placement strategies for BERT-SGMoE under different bandwidth levels, respectively. We find that HiDup’s decisions follow our analysis above and it can automatically identify the optimal placement strategy for each parameter.

Next, we show the partition strategy found by HiDup for MoE layers in Fig. 12. We use the same notation as in GShard [18], where lower case letters denote sharded dimensions. $B$ is the batch size, $S$ is the sequence length, $M$ is the embedding size, $E$ is the number of experts, $C$ is the per-expert capacity and $H$ is the hidden size. The dashed boxes show stages. The left side of Fig. 12 shows the strategy on two machines connected by a 100Gbps link. It is the same as the expert-designed strategy in GShard [18]. However, when
Table 4: Per-iteration time breakdown.

<table>
<thead>
<tr>
<th>System</th>
<th>Single Machine</th>
<th>Two Machines (100Gbps)</th>
<th>Two Machines (30Gbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HiDup</td>
<td>DeepSpeed</td>
<td>FastMoE</td>
</tr>
<tr>
<td>Wall time (s)</td>
<td>0.2972</td>
<td>0.3988</td>
<td>0.3985</td>
</tr>
<tr>
<td></td>
<td>0.3245</td>
<td>0.3813</td>
<td>0.4611</td>
</tr>
<tr>
<td>Total time (s)</td>
<td>0.3245</td>
<td>0.3813</td>
<td>0.6857</td>
</tr>
<tr>
<td>Computation (s)</td>
<td>0.2948</td>
<td>0.3531</td>
<td>0.3088</td>
</tr>
<tr>
<td>Communication (s)</td>
<td>0.0297</td>
<td>0.0282</td>
<td>0.1523</td>
</tr>
<tr>
<td>Overlapping ratio</td>
<td>91.92%</td>
<td>0</td>
<td>41.10%</td>
</tr>
</tbody>
</table>

Table 5: Impact of inaccurate computation and communication time estimation.

<table>
<thead>
<tr>
<th>Noise</th>
<th>Relative time</th>
</tr>
</thead>
<tbody>
<tr>
<td>20% random noise</td>
<td>100.2%</td>
</tr>
<tr>
<td>50% random noise</td>
<td>112.9%</td>
</tr>
<tr>
<td>+20% communication</td>
<td>100.0%</td>
</tr>
<tr>
<td>+50% communication</td>
<td>100.0%</td>
</tr>
<tr>
<td>−20% communication</td>
<td>100.0%</td>
</tr>
<tr>
<td>−50% communication</td>
<td>117.2%</td>
</tr>
</tbody>
</table>

Figure 12: Partition strategies for MoE layers.

The bandwidth is reduced to 10Gbps, HiDup switches to the strategy shown on the right side. This strategy duplicates the calculation of the gating layer on every card, reducing communication at the cost of more computation. The input to the MoE layer is not partitioned, which is highly coupled with the strategy used by the previous layer, indicating that HiDup can holistically consider the partition strategy across layers.

6.11 Computation/Communication Time Estimation

We evaluate the impact of our computation and communication time estimation (using device flops and bandwidth, as discussed in Sec. 5) on the strategy found by HiDup for BERT-SGMoE on the same cluster as in Sec. 6.5. We add two types of noises to authentic profiling results, generate the best strategy based on these noisy estimations, and derive the average ratio of per-iteration training time achieved with the generated strategy over that of the optimal strategy generated using the unmodified estimations (the 'Relative Time' in Table 5). The first noise type is "x% random noise", with which we randomly change the estimated time of each operator by up to x%. We conduct the experiments 10 times for each noise level. The second type is "±y% communication", which means we increase/decrease the estimated time of all communication operators by y%. We observe from Table 5 that HiDup can find near-optimal strategies with noise up to 20%, suggesting that it is resilient to estimation errors.

We show the estimated computation time and profiling results for operators used in BERT-SGMoE in Fig. 13. The flops-based estimation tends to under-estimate the computation time for small operators like element-wise addition, because these operators may be memory-bound and flops do not reflect the memory accessing time. For larger operators like MatMul, the maximum estimation error is 17%.

6.12 Strategy Searching Time

Fig. 14 shows HiDup’s strategy search time for BERT-SGMoE. In Fig. 14a, we keep the number of GPUs as 4 and alter the number of layers in the model. The overall search time increases linearly with the number of layers. In Fig. 14b, we fix the number of layers at 6 and change the number of GPUs.

Figure 13: Flops-based computation time estimation.
As HiDup uses SPMD parallelism where all devices use the same computation graph, its search time is independent of the number of GPUs.

7 RELATED WORK

7.1 MoE training systems

GShard [18] and GSPMD [36] parallelize computation in large model training based on user-annotated SPMD strategies and heuristic parallelism strategies for nodes not annotated. For MoE models, their heuristic strategies are to alternate between MP for MoE layers and DP for other layers. GSPMD supports a special case of pipeline training for models consisting of identical consecutive layers, by treating the layers as an additional dimension. In contrast, HiDup automatically finds sharding strategies. We do not experimentally compare with GShard and GSPMD as they are mainly designed for TPU clusters and implemented on Mesh Tensorflow [32], while we focus on GPU clusters and implement HiDup for PyTorch.

DeepSpeed [14, 27] uses 3D parallelism including DP, MP, and pipeline parallelism to support MoE model training. Fast-MoE [10] implements MoE layers for PyTorch with low-level optimizations such as customized CUDA kernels. Both of them provide special MoE layer implementation with built-in parallelisms. HiDup uses graph transformation to support MoE model training so that it can holistically consider the sharding strategy for both MoE layers and other layers.

7.2 Automatic parallelism strategy search

Tofu [35] and HyPar [34] use dynamic programming to find a DNN partition strategy that minimizes the total communication cost. Alpa [39] solves an integer linear program for intra-op partition strategies and applies dynamic programming to identify inter-op partition strategies. Flexflow [13] uses a Markov Chain Monte Carlo (MCMC) algorithm for strategy search on the SOAP space, including sharding strategies and placements. HeteroG [37] utilizes a graph neural network to generate distributed training strategies for heterogeneous clusters. HiDup is different from these studies in that we consider computation-communication overlap in strategy search, introduced by our duplex design, while no overlapping is assumed in these systems. Such overlapping complicates the dynamic programming approach design.

8 DISCUSSIONS

8.1 Heterogeneous Clusters

Existing SPMD frameworks are mostly used on homogeneous clusters. However, GPUs allocated in public clouds are often scattered in different machines or racks, resulting in heterogeneous inter-connections. As a possible future direction, HiDup can be extended to support this kind of heterogeneity by using topology-aware communications (e.g., Hoplite [41], TACCL [31]) that exploit faster links and avoid slow links.

8.2 Pipeline Parallelism

GPipe [11] and PipeDream [22] propose the pipeline parallelism for DNN training, where the model is divided into stages (each consisting of multiple computation operators - note that it is different from our stage definition in Sec. 4) and multiple microbatches are trained at different stages at the same time. In HiDup, our duplex design effectively forms a pipeline between the computation devices (GPUs) and communication devices (network links); however, this pipeline is quite different from the pipeline parallelism above, as we focus on SPMD parallelism and do not put different layers on different devices. HiDup can be used in conjunction with the pipeline parallelism: many pipeline training systems embed data parallelism [27, 39] as well, by putting a single stage on multiple devices for DP training; HiDup can be used to replace the DP strategy with more sophisticated SPMD parallelism, to further accelerate pipeline training.

9 CONCLUSION

We present HiDup, an automated module to accelerate SPMD training with duplex and automatic SPMD strategy searching. Our duplex design introduces computation-communication overlapping to the SPMD parallelism, and our dynamic programming-based strategy searching algorithm automatically identifies sharding strategies that exploit the overlapping opportunities provided by duplex. We implement HiDup for PyTorch and show that it achieves up to 61% faster training of MoE models as compared with representative frameworks.

ACKNOWLEDGMENTS

This work was supported in part by Alibaba Group through Alibaba Innovative Research (AIR) Program, and Hong Kong RGC under the contracts HKU 17204619 and 17208920.
REFERENCES


URL: https://github.com/shemminger/iproute2


