AutoDist: A Composable and Automated Synchronization System for Distributed Deep Learning

Hao Zhang [1] **Peng Wu** [2] **Zhijie Deng** [3] **Christy Li** [4] **Qirong Ho** [2] **Aurick Qiao** [2] **Zeya Wang** [2] **Eric P. Xing** [5]

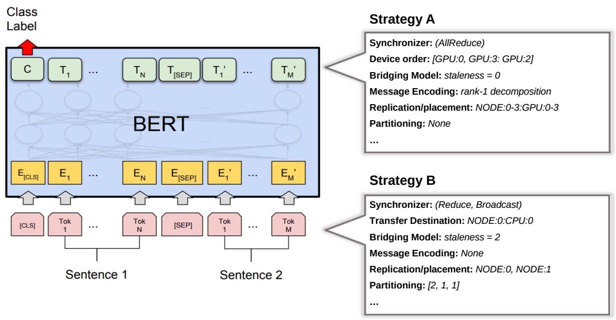
Abstract

Efficient data-parallel distributed training has been a key driver behind recent innovations in deep learning (DL). However, achieving satisfactory distributed performance involves making difficult system-level decisions related to diverse synchronization aspects. We present *AutoDist*, which automatically composes parallel synchronization strategies for DL models by rewriting their original dataflow graphs into parallel versions. Unlike existing training systems with fixed strategies, AutoDist adaptively composes strategies by jointly optimizing multiple aspects, each applied to different parts of the DL model. Compared to other graph rewriting systems, AutoDist deliberately breaks seemingly distinct synchronization optimizations into atomic graph rewriting kernels, and allows mechanically assembling them to express new strategies that extrapolate to new models and clusters. We show that AutoDist can find high-performance strategies quickly, and enables model training 1.2x to 1.6x faster than hand-optimized baselines. Critically, AutoDist does not require manual tuning when faced with new DL models or cluster configurations.

1 Introduction

Data-parallel distributed training using CPU or GPU clusters has become a key requirement for recent deep learning (DL) models, and is offered by every modern DL framework (Paszke et al., 2019; Abadi et al., 2016). Core to the performance of data-parallel training is the strategy by which model parameters are synchronized. Such a synchronization strategy can be composed of diverse aspects, spanning communication architectures (Li et al., 2014; Sergeev & Del Balso, 2018), placement and partitioning (Mirhoseini et al., 2017; Jia et al., 2018b), message computation and augmentation (Lin et al., 2017; Xie et al., 2018). Existing systems which implement a single strategy can improve the training speed and scalability for *specifically-chosen* DL model architectures, but may have trouble extending their improvements to other models which may have preferences for different synchronization strategies. As a result, performance-conscious users face the challenge of selecting the right systems, which implement the right synchronization strategies, that achieve high performance for their specific training tasks. In particular, designing the right synchronization strategy or system requires careful consideration of the following properties of DL training.

Heterogeneity within the model structure. Contemporary DL models exhibit a rich variety of sub-structures. For example, BERT (Devlin et al., 2018) (Figure 1) is composed of several components: word and positional embedding, transformer encoders, and multiple loss-related layers. These components may exhibit diverse computational characteristics and connectivity with each other, resulting in different preferences for parameter synchronization across different parts of the same model. Rather than applying the same mechanism uniformly across all components (e.g., AllReduce in Horovod (Sergeev & Del Balso, 2018)), the framework should allow flexible composition of synchronization strategies that better match the heterogeneous structure and connectivity within DL models, which we refer to as *strategy composition* (Figure 1 right).



Strategy A

Synchronizer: *(AllReduce)*
Device order: *[GPU:0, GPU:3: GPU:2]*
Bridging Model: *staleness = 0*
Message Encoding: *rank-1 decomposition*
Replication/placement: *NODE:0-3:GPU:0-3*
Partitioning: *None*
...

Strategy B

Synchronizer: *(Reduce, Broadcast)*
Transfer Destination: *NODE:0:CPU:0*
Bridging Model: *staleness = 2*
Message Encoding: *None*
Replication/placement: *NODE:0, NODE:1*
Partitioning: *[2, 1, 1]*
...

Figure 1. Example: strategy composition for BERT.

Dependence on cluster environment. The highest performing synchronization strategy may differ between cluster environments with different performance characteristics. For example (§3.2), when a high-bandwidth connection

[1]UC Berkeley [2]Petuum, Inc. [3]Tsinghua University [4]Duke University [5]Carnegie Mellon University. Correspondence to: Hao Zhang <hao@cs.berkeley.edu>.

Prepint, 2020.

such as GPUDirect is available, using AllReduce to collect gradients outperforms using a parameter server, whereas the opposite may be true when using different hardware. Thus, the framework should adapt to different environments by using the best synchronization strategy for the given cluster.

Interplay between multiple synchronization aspects. Decisions made for different aspects of synchronization may exhibit strong inter-dependence on each other. For example, the ideal number of parameter servers to spawn may depend on hardware performance and model structure. For each part of the model, the ideal parameter partitioning/placement scheme and whether/how to use collective or parameter server in turn depends on the number of parameter servers available (§3.2). The framework should be able to navigate the space of synchronization strategies when faced with complex inter-dependencies.

Whereas state-of-the-art training systems (Zhang et al., 2017; Kim et al., 2018; Peng et al., 2019) employ rule-based heuristics to optimize a small number of synchronization aspects, such as differentiating between sparse and dense gradients, these rules can hardly scale with and may not hold across diverse models or cluster setups. A desired framework should enable composability between diverse synchronization aspects to match heterogeneity in model structures, and adapt to diverse cluster environments. Due to the complex interplay between multiple synchronization aspects, designing the right synchronization strategy requires ML and system savvy. Thus, for practical applicability of such a framework, it is also essential to provide *automatic strategy composition* to eliminate mandatory manual tuning.

1.1 Our Approach

We present AutoDist, a composable and automated synchronization system for data-parallel DL training. AutoDist supports *strategy composition* by automatically generating a synchronization strategy, tailored for a given model and resource specification, by selecting from a pool of synchronization aspects that can come from prior systems literature, and be extended as new research emerges. We address the aforementioned challenges with the following contributions.

Synchronization strategy representation. We design a *unified representation for synchronization strategies* that jointly encodes multiple synchronization aspects, including graph replication, variable partitioning and placement, synchronization architecture and aggragation structures, message encoding, and bridging models (§2.2). With the representation, AutoDist generates a strategy composition suited with the specific model and cluster setup. The strategy is explicitly represented, exercise-able by low-level systems, and transferable across applications (Figure 1).

Composable execution system. We implement a system architecture for strategy composition and execution. AutoDist maps each synchronization aspect to a modular and composable *dataflow graph rewriting kernel* in the execution system (§2.3). A complete synchronization strategy is materialized by connecting and exercising multiple primitive kernels based on a given strategy representation. Our approach separates strategy specification from the distributed execution system, which allows composition and efficient execution of complex synchronization strategies. Furthermore, our design ensures that future advances in synchronization may be easily incorporated into AutoDist.

Automatic strategy optimization. Oceanus performs end-to-end strategy auto-optimization adaptively for different model structures and hardware resources (§2.4). This not only improves parallel performance, but also eliminates the need for manual effort by an expert user. Our optimizer combines mathematical performance models and ML-based simulator, and can improve with experience.

We evaluate AutoDist over multiple ML models and cluster setups, and present the following findings: (1) AutoDist demonstrates matched or better performance than specialized systems when composing their proposed strategies using AutoDist composable kernels. (2) we reinspect several synchronization aspects, and study how their effectiveness varies with different model architectures and resource configurations, which deviates from existing rules or heuristics and necessitates a composable and automated synchronization system design like AutoDist. (3) When auto-optimizing strategies for novel architectures and cluster setups, AutoDist, without modifying low-level system runtime or tuning, finds strategies 1.2x-1.6x faster than hand-optimized baselines, with an acceptable auto-optimization budget.

2 AutoDist Design and Architecture

Systems for implementing DL models, such as TensorFlow (Abadi et al., 2016), commonly use *dataflow graphs* (Murray et al., 2013) to represent models. Throughout this paper, we refer to the dataflow graph of DL models and its computational function using $\mathcal{G}=\{(\mathcal{O},\mathcal{V}_\Theta),\mathcal{E}\}$, where the node of $\mathcal{G}$ is either a computational operation $o\in\mathcal{O}$ or a stateful variable $v\in\mathcal{V}_\Theta$, and edges ($\mathcal{E}$) are tensors. Parallelizing $\mathcal{G}$ over multiple devices using *data-parallelism* follows

$$\mathcal{V}_\Theta^{(t+1)} \leftarrow \mathcal{V}_\Theta^{(t)} + \epsilon \sum_{p=1}^{P} \nabla_{\mathcal{G}_p}(\mathcal{V}_\Theta^{(t)}, \mathcal{X}_p), \tag{1}$$

where the original single-device graph $\mathcal{G}$ is replicated into P replicas $\{\mathcal{G}_p\}_{p=1}^P$, and each replica $\mathcal{G}_p$ performs computation on the pth device over its independent split of training data $\mathcal{X}_p$, but shares a consistent set of trainable parameters $\mathcal{V}_\Theta$, synchronized per iteration. We use $\mathcal{R}$ to describe the specification of the target cluster containing these devices. $\mathcal{R}$ includes information such as the number of nodes, their addresses, number of CPUs and GPUs per node, and their

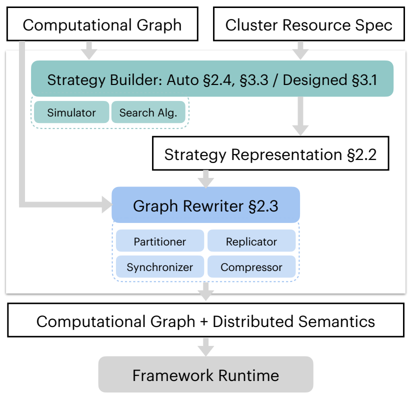


Figure 2. An overview of AutoDist workflow and key components.

interconnection information (e.g., network bandwidth).

2.1 Overview

Figure 2 illustrates the high-level workflow of Oceanus and its key components. AutoDist takes the dataflow graph $\mathcal{G}$ and the cluster specification $\mathcal{R}$ as inputs. It makes an appropriate choice for each aspect in its synchronization representation space, taking into consideration the computational properties present in $\mathcal{G}$ and the resource condition in $\mathcal{R}$. It then composes them into an expression $\mathcal{S}$, which precisely instructs how each part of $\mathcal{G}$ should be synchronized when distributed on the cluster $\mathcal{R}$. We call $\mathcal{S}$ a *strategy*. It spans multiple atomic synchronization aspects developed in §2.2, and is model-dependent and resource-dependent. A toy example $\mathcal{S}$ is shown on Figure 1 right.

AutoDist decouples the strategy generation from specific system implementation (unlike existing systems). The generation of $\mathcal{S}$, depending on user choices, is done either via *hand-designed strategy builders*, or an *AutoStrategy-Builder* that derives the optimal $\mathcal{S}$ via automatic strategy optimization (§2.4). For instance, we can have a predefined *HorovodBuilder* that takes $\mathcal{G}$ and $\mathcal{R}$ and applies collective ring allreduce as the communication architecture, which precisely recovers Horovod (Sergeev & Del Balso, 2018).

The expression is then broadcasted across all related nodes of $\mathcal{R}$ mentioned in $\mathcal{S}$. An AutoDist process on each node is invoked locally to exercise $\mathcal{S}$, by transforming $\mathcal{G}$ based on it into a distributed graph. AutoDist backend implements a library of primitive and generic graph rewriting functions, which we will call *graph transformation kernels*. By mapping each aspect in $\mathcal{S}$ with a graph transformation kernel, the rewriting happens in a *composable* way, by applying a series of kernels instructed in $\mathcal{S}$ (§2.3) stage by stage. Upon the completion of rewriting, AutoDist executes the obtained distributed graph to start distributed training.

We next elaborate the three key components: the strategy representation (§2.2), the composable system backend (§2.3), and the auto-optimization pipeline (§2.4). We defer descriptions of implementations to Appendix B.

2.2 Strategy Representation

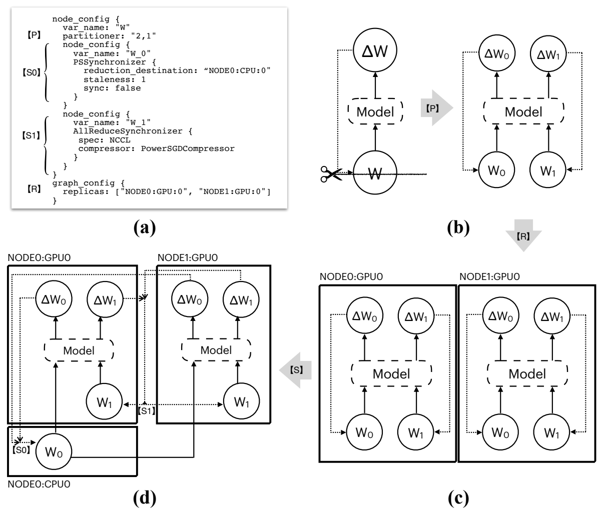
We want the strategy representation to be sufficiently expressive to capture many synchronization-affecting factors, but not to specialize to certain optimizations that otherwise sacrifices its simplicity, and hinders the composability of our backend (§2.3) when exercising the strategy. To formalize the representation, we first break existing instance-based synchronization strategies, and re-identify their common semantics, cataloged as the following orthogonal aspects:

- *Replication* of the model graph $\mathcal{G}$: how and which devices to replicate the model graph on for data parallelism.
- *Partitioning and placement* of variables $\mathcal{V}_\Theta$: partitioning, sharing and placement mechanisms of trainable variables.
- *Synchronization architecture and aggregation structures*: How to set up a network topology for message synchronization (e.g., bipartite PS, P2P broadcast, or fully-connected AllReduce). How and where to aggregate updates $\nabla_{\mathcal{G}_p}$, e.g., full summation at a central device, partial summation across a tree, etc.
- *Message encoding*: How to encode and decode messages $\nabla_{\mathcal{G}_p}$ before communication, such as various forms of gradient updates, compression, sufficient factors, etc.
- *Bridging models*: How to bridge computation of $\sum_{p=1}^{P} \nabla_{\mathcal{G}_p}$ and communication – various synchronous, asynchronous, or bounded-asynchronous methods.

To cover these aspects, we define the representation of $\mathcal{S}$ with two-level semantics: *graph level* that expresses how $\mathcal{G}$ will be transformed globally to fit a parallel environment, and *node level* that specifies per-variable synchronization configurations – thus supporting strategy composition. Figure 3(a) illustrates a real strategy snippet. At the graph level, we introduce the *graph config* to include a single semantic, *replication*, to capture the aspect of how $\mathcal{G}$ should be replicated across multiple devices in data-parallel environment. We use a tuple $(\mathcal{G}, \{d_i\}_{i=1}^m)$ to notate that $\mathcal{G}$ will be replicated on a set of destination devices $\{d_i\}_{i=1}^m \subset \mathcal{R}$.

At the node level, for each trainable variable $v_i \in \mathcal{V}_\Theta$, we introduce a *node config* to express its variable-specific synchronization setup, specifying the following aspects.

Variable partitioning. Operation partitioning in dataflow graphs is a widely explored topic (Wang et al., 2019; Jia et al., 2018b; Shazeer et al., 2018). Complement to existing works that partition computational operations, AutoDist only concerns the partitioning of nodes that participate in data-parallel synchronization: variable nodes $v_i \in \mathcal{V}_\Theta$. Unlike operation partitioning, a variable could be partitioned along any axis of its tensor buffer, we thus represent the partitioning of v_i simply as a vector $\boldsymbol{p}_i = [p_i^j]_{j=1}^{k_i}$, with k_i



```
        node_config {
          var_name: "W"
【P】      partitioner: "2,1"
          node_config {
            var_name: "W_0"
            PSSynchronizer {
【S0】          reduction_destination: “NODE0:CPU:0"
              staleness: 1
              sync: false
            }
          }
          node_config {
            var_name: "W_1"
【S1】        AllReduceSynchronizer {
             spec: NCCL
             compressor: PowerSGDCompressor
            }
          }
        }
        graph_config {
【R】      replicas: ["NODE0:GPU:0", "NODE1:GPU:0"]
        }
```

Figure 3. A strategy expression (§2.2) and its mapped graph-rewriting operations (§2.3). A solid arrow is from a tensor producer to a stateless consumer; dotted refers to a stateful variable update. (b) It partitions variable W into W_0, W_1 along axis 0. (c) It replicates $\mathcal{G}$ onto two devices. (d) W_0 is place one one device, shared, and synchronized with (`Reduce`, `Broadcast`), W_1 is placed on both devices, synchronized via `AllReduce`.

equals the number of dimensions of v_i, and p_i^j is an integer representing how many partitions are generated along the jth axis of v_i – we obtain $\prod_{j=1}^{k^i} p_i^j$ *partitioned variables* of v_i after the partitioning. The representation is variable-specific, and allows to express a rich set of existing synchronization optimizations, such as various sharding strategies for load balancing in PS (Peng et al., 2019; Li et al., 2014).

Variable sharing and placement. A variable, partitioned or unpartitioned, can be either shared or replicated across multiple devices for synchronization. We bring in *placement* that decides where the variable locates. When it is instantiated as a set of devices $\{d_i\}_{i=1}^m \subset \mathcal{R}$, the variable is replicated across all $\{d_i\}_{i=1}^m$. A single device value $v_i : d_i$ then indicates v_i is placed on d_i; computation or synchronization of different devices have shared access to v_i on d_i. Compared to general device placement (Mirhoseini et al., 2017), AutoDist only models the placements of variables, as the rest of nodes in $\mathcal{G}$ are replicated in data-parallel training, captured by the *replication* semantic defined earlier.

Synchronization architecture. Data-parallel synchronization requires collecting and applying updates $\nabla\mathcal{G}_p$ from distributed replicas, and synchronizing all replicas with new states of v_i, which distinguishes synchronization from normal communication between devices addressed in model-parallel systems (Jia et al., 2018b; Wang et al., 2019). Hence, for each v_i, we introduce a *synchronizer config* to express synchronization-related semantics. It has its leading dimension as *synchronizer type*, indicating the communication primitives used for synchronization, notated as combinations of communication primitives (MPI). For example, v_i with synchronizer type as (`Reduce`, `Broadcast`) and placement as d_i follows a bipartite PS architecture to *reduce* and apply the updates $\{\nabla\mathcal{G}_p\}_{p=1}^P$ onto the destination d_i, and *broadcast* the updated states back to all participating replica. Similarly, type (`AllReduce`) straightforwardly indicates using AllReduce for joint aggregation and synchronization. This forms a unified representation of diverse synchronization architectures.

Based on the synchronizer type, we bring in more critical type-specific semantics such as: (1) *transfer destination*: $d_t (d_t \in \mathcal{S})$, to express a two-level hierarchy to reduce updates or broadcast parameters (of v_i) first to the transfer station d_t, then to the final destination d_i. (2) *reduction sequence*: list of devices $(d_1,...,d_n)$ indicating the order of devices when performing collective communication. (3) *group*: integer to indicate which group the communication primitives shall be merged, together into one single operation to reduce constant min-cost (see §3.2 for a thorough study). The offers flexibility to specify, for each v_i, various forms of communication topologies, aggregation structure, grouping mechanisms, and other optimizations.

Message encoding and decoding. Many synchronization systems exploit certain structures (e.g., low-rank) present in the original message $\nabla_{\mathcal{G}_p}$, and encode them into alternative forms for faster network transfer. We introduce *encoder* and *decoder* for each v_i to model this aspect. Optimizations like sufficient factor broadcasting (Xie et al., 2018), or compression (Lin et al., 2017) hence can be expressed by specifying each variable with a corresponding encoder/decoder.

Bridging models. This aspect specifies how far the parallel replicas can allow computations and messages to occur out-of-order. We use an integer *staleness* to express the degree of consistency for synchronizing the variable v_i – when staleness is set to K, the fastest worker replica cannot be more than K steps ahead of the slowest worker.

We discuss the expressiveness of this representation in Appendix A. This representation serves as an instruction for low-level composable dataflow graph rewriting (§2.3), also forms a space for automatic strategy optimization (§2.4).

2.3 Composable System via Graph Rewriting

Given $\mathcal{G}$ and $\mathcal{S}$, AutoDist incorporates distribution semantics by rewriting $\mathcal{G}$: adding, deleting or modifying nodes and edges and their attributes in $\mathcal{G}$, while keeping the computational results. It is realized by applying a series of atomic graph rewriting kernels configured by $\mathcal{S}$, composably.

Graph rewriting kernel library. AutoDist backend offers a library of primitive graph rewriting kernels. Each sub-expression in $\mathcal{S}$ is mapped to a kernel type with specific configurations. Each type of kernel defines how to rewrite the graph from its current state to the next, which manifests the corresponding semantics in the sub-expression. By designing each kernel at the right level of abstraction, they can be flexibly composed to alter the graph based on $\mathcal{S}$. We describe a few representative kernels illustrated in Figure 3.

Partitioner (Figure 3(b)) reads node-level partitioning subexpression from $\mathcal{S}$ for each $v_i \in \mathcal{V}_\Theta$. It splits the variable across given axes into multiple smaller variables, as well as its gradients and subgraphs corresponded to its state-updating operations. However, it does not split the operations that consume the original variable – which will instead consume the value re-stitched from all partitions. Each partitioned variable, with its own node config (generated at strategy generation time), will be added into $\mathcal{V}_\Theta$ as an independent variable identity. *Replicator* (Figure 3(c)) reads graph-level configuration from $\mathcal{S}$. It replicates the original graph onto target devices. Unless overridden by other kernels, the replicated operations or variables have their placements as the replication destination. *Synchronizer* (Figure 3(d)) reads the synchronizer config for each $v_i \in \mathcal{V}_\Theta$. Depending on the synchronizer type, it rewrites the graph: either to share a variable on a destination device across replicas ((`Reduce`, `Broadcast`) synchronizer) with specified staleness in $\mathcal{S}$, or to synchronize states of replicated variables via collective communication (`AllReduce` synchronizer) following specified device structures in $\mathcal{S}$. Moreover, the kernel implementations are dispatched to handle either dense or sparse cases. A *compressor* kernel appends compression operations afterwards to transform the message based on encoding and decoding semantics in $\mathcal{S}$.

Two-stage graph rewriting. By applying a series of above graph-rewriting kernels configured in $\mathcal{S}$, a single-device graph $\mathcal{G}$ is transformed into a distributed graph $\mathcal{G}'$ following a compilation process. Generating a single complete $\mathcal{G}'$ possessing all distributed semantics makes the size of $\mathcal{G}'$ growing proportionally with both the size of $\mathcal{G}$ and the number of devices in $\mathcal{R}$, which quickly becomes unmanageable if any of $\mathcal{G}$ or $\mathcal{R}$ is large. To scale the graph rewriting with large $\mathcal{G}$ and many devices, AutoDist proposes two-stage, local and deferred graph rewriting. Instead of writing all distributed semantics in one graph, AutoDist first broadcasts the strategy expression (whose size only corresponds to original single-device $\mathcal{G}$) across $\mathcal{R}$, and invokes graph rewriting on each node afterward. Each node starts graph rewriting in parallel, but toward a local snapshot of $\mathcal{G}'$ that corresponds to graph execution on itself. They maintain a consistent picture of $\mathcal{G}'$ by checking against $\mathcal{R}$ and $\mathcal{S}$, and uses absolute device notations when generating shared or distributed nodes. This allows to scale to large graphs or clusters. After transformation, the executions of the distributed graph is delegated to specific framework runtime on each node.

2.4 Automatic Strategy Optimization

With an explicit representation of $\mathcal{S}$ and a ready-to-evaluate system in place, we seek to auto-optimize $\mathcal{S}$ conditioned on $\mathcal{G}$ and $\mathcal{R}$, formulated as

$$\mathcal{S}^* = \underset{\mathcal{S}}{\operatorname{argmax}} \quad u(\mathcal{S}, f(\mathcal{G},\mathcal{R},\mathcal{S}),\mathcal{R}), \tag{2}$$

where $f(*)$ denotes the rewriting by AutoDist, and u characterizes the system throughput that we want to maximize. While it is possible to optimize other unities, e.g., convergence, in this paper, we focus on *system throughput* as the utility since it is easier to characterize. Hence, in auto-optimization, we only consider synchronous training, and exclude aspects that would impact the convergence, e.g., lossy compression or staleness. However, there is a large space of $\mathcal{S}$ for considerations. The solver needs to efficiently locate performant strategies, so the overhead due to auto-optimization does not cancel out its efficiency gains, even if the strategy is used to train the model only for once. To this end, we develop a new auto-optimization Algorithm 1, which performs budgeted search for $\mathcal{S}^*$ as follows: it first samples a set of strategy candidates using a guided *strategy sampler*; the candidates are evaluated by a performance *simulator* to generate scores, and a set of elite candidates are selected based on the simulator *fidelity* and diversity for trial execution. The real runtime data is feedback to refine the simulator. The process repeats until the search budget is exhausted, and the best $\mathcal{S}$ found is returned. We describe each component below.

Algorithm 1: Automatic strategy optimization

```
   Input: model $\mathcal{G}$, resource $\mathcal{R}$, budget $B$
   Output: near-optimal strategies $\mathcal{S}^*$
1  while $b < B$ do
2      Identify the fidelity level $l$ of the simulator $\hat{u}$
3      Sample $\{\mathcal{S}_i\}_{i=1}^N$ using the guided strategy sampler
4      Filter $\{\mathcal{S}_i\}_{i=1}^N$ by rejecting $\mathcal{S}_i$ with $\hat{u}(\mathcal{S}_i) < l$
5      Select trial candidates $\{\mathcal{S}_k\}_{k=1}^K \subset \{\mathcal{S}_i\}_{i=1}^N$ based on
         both scores and their diversity
6      Launch trials and get $\{u_k\}_{k=1}^K, u_k = u(\mathcal{S}_k)$
7      update $S^*$ with the best-performed $\mathcal{S}$
8      Improve $\hat{u}$ using new data $\{u_k\}_{k=1}^K$
9      $b = b + K$
10 return $\mathcal{S}^*$
```

Strategy sampler. The space spanned by certain semantics of $\mathcal{S}$ (e.g., partitioning config, transfer destination, group) is discrete and has a combinatorial number of choices growing with $\mathcal{G}$ and $\mathcal{R}$. Randomly sampling their values is prohibitive. We regulate their choices using a few constraints, including: (1) avoid partitioning variables of small size but always partitioning variables of size over certain threshold; (2) given a set of variables $\mathcal{V}_\Theta^{ps}$ using PS-based synchronizer, post-placing $\mathcal{V}_\Theta^{ps}$ on $\mathcal{R}$ so each node in $\mathcal{R}$ has approximately balanced loads, (3) randomly grouping collective operations so every collective group has similar loads. The sampler still generates randomized strategies, but are directed by these constraints from regions where $\mathcal{S}^*$ most likely locate.

Performance simulator. We build a simulator, denoted as $\hat{u}$, to estimate u given $\mathcal{S}$,$\mathcal{G}$,$\mathcal{R}$ without launching real execution. The tight budget prevents building a fully data-driven simulator (Chen et al., 2018b) as it requires massive groundtruth data to train. Instead, we develop a hybrid simulator: the simulator starts as a model- and resource-agnostic predefined performance model based on communicating modeling. Once trial data on ($\mathcal{G}$,$\mathcal{R}$) are acquired, it morphs to an ML model and generates more accurate predictions for the specific ($\mathcal{G}$,$\mathcal{R}$).

The predefined model takes ($\mathcal{G}$,$\mathcal{R}$,$\mathcal{S}$) as input and estimates per-iteration runtime T, decomposed as parameter synchronization time T_{sync} across devices, and computation time T_{comp} happening in parallel on each device. Since the former often dominates the latter in distributed ML, we estimate $T = \max(T_{\text{comp}}, T_{\text{sync}}) = T_{\text{sync}}$. To get T_{sync}, we compute T_{sync} of each variable v, precisely following the synchronization expression of v, using a predefined network communication model from existing literature, and then accumulate the results. The communication model calculates 6 related components (see Appendix C), denoted as $\mathbf{z}_{\text{predefined}}$, and T_{sync} is estimated as a linear combination of them, as $\hat{u} = \mathbf{z}_{\text{predefined}} * \theta$, where θ are fixed constants associated with the communication model. This model roughly tells how each strategy performs by directly estimating communication without any runtime data.

If there is budget for trial run, we can improve the prediction by adapting the predefined model to an ML model trained using collected runtime data on a specific ($\mathcal{G}$,$\mathcal{R}$). To train the ML model, we treat $\mathbf{z}_{\text{predefined}}$ as a hand-extracted feature, and additionally extract raw features from $\mathcal{G}$ and $\mathcal{R}$ as $\mathbf{z}_{\text{raw}}$, by vectorizing their attributes such as: variable shape, sparsity, date type, synchronizer types, reduction destination, merge group, and compressor, and designates trainable weights ϕ to each feature. Formally, it can be written as

$$\hat{u} = Q(\mathbf{z}_{\text{predefined}} \oplus \mathbf{z}_{\text{raw}}, \phi), \qquad (3)$$

where Q represents an ML model, instantiated as a variable-length RNN to model $\mathcal{G}$ and $\mathcal{R}$ of different shapes (see Appendix D). We train Q with a *ranking loss* using collected real runtime data. The states of Q (i.e., ϕ) persist throughout Algorithm 1, and is updated whenever more runtime data are collected.

Filtering. Since Q is trained with a ranking loss, depending on how many trial data the simulator has seen, its ranking ability exhibits uneven trustworthiness. At the initial phase, the simulator can only distinguish strategies with sharp performance differences (e.g., left buckets in Figure 4). Seeing more data, it gradually improves and learns the subtle differences between closer strategies (right buckets). Figure 4 presents this phenomenon, which we refer as the simulator in-

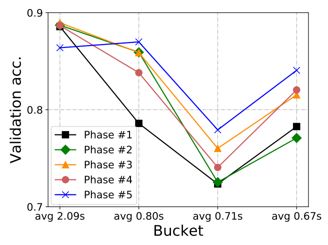


Figure 4. Explored (strategy, runtime) during auto-optimization (Algorithm 1) are spited into 4 performance buckets (x-axis) and 5 chronological phases based on when it is visited. For each phase, we train Q using data up to the current phase, and report the ranking accuracy (y-axis) on a conserved validation set of each bucket. The figure shows the ranking accuracy in high-performance buckets improves for low to high, meaning the trustworthiness of its predictions on these buckets increases.

creases its *fidelity*. We tailor the search to cope with this property. The search first identifies the current fidelity level, which is a simulator score (higher is better), below which the simulator's prediction is highly-confident – simply set as the highest score below which the simulator maintains a desired ranking accuracy (say 80%) by testing the simulator against the most recent trial data. Then, it rejects proposals with scores below the fidelity level, resulting in a set of "elite" strategies, whose scores $\hat{u}$ may not accurately reflect their qualities due to simulator's uncertainty. To identity the final set of candidates for trial, it selects a subset that trade-off between the simulator scores and their diversity, following (Chen et al., 2018b). This prevents overfitting with the simulator's inaccurate prediction and balances exploitation and exploration.

3 Evaluation

In this section, we evaluate AutoDist, aiming to answer: (1) Is the proposed design feasible, how does the system perform when composing *de facto* strategies? (2) Is a composable design and auto-optimization in AutoDist necessary? How do multiple aspects of synchronization effect on different $\mathcal{G}$ and $\mathcal{R}$? can simple rules or heuristics to capture them? (3) Is strategy composition effective? Can AutoDist auto-optimization find better strategies within budget?

Testbeds. We focus on GPU clusters which is the main setup for distributed DL. We conduct experiments on two setups: *Cluster A* includes maximally 16 nodes, each equipped with a GeForce TITAN X GPU interconnected via a 40GbE switch; *Cluster B* is an AWS cluster containing different types of instances: g3.4xlarge (1x Telsa M60 GPU and 10GbE Ethernet), g3.16xlarge (4x M60, 25GbE), g4dn.2xlarge (1x T4, 25GbE), g4dn.12xlarge (4x T4, 50GbE) instance types. Due to AWS constraints, they all have 10GbE single-flow bandwidth. On top of A and B, we generate a diverse set of heterogeneous resource specifications ($\mathcal{R}$) shown in Table 1. We rely on TF 2.0 for dataflow graph distributed execution, which uses CUDA10.0, cuDNN 7.1, NCCL 2.4.7, and gRPC for network communication. We *do not alter* any runtime or

Table 1. Experimented cluster specifications ($\mathcal{R}$). GPU dist: distribution of GPUs. bw spec: bandwidth specifications.

$\mathcal{R}$	setup	GPU dist	bw spec
A1	16x Cluster A nodes	[1] x 16	40GbE
A2	11x Cluster A nodes	[1] x 11	40GbE
B1	2x g3.16	[4] x 2	25 GbE
B2	3x g3.16	[4] x 3	25 GbE
B3	4x g3.16	[4] x 4	25 GbE
B4	1x g4dn.12	[4] x 1	50 GbE
B5	2x g4dn.12	[4] x 2	50 GbE
B6	3x g4dn.12	[4] x 3	50 GbE
B7	4x g4dn.12	[4] x 4	50 GbE
B8	8x g4dn.12	[4] x 8	50 GbE
B9	1x g3.4, 1x g3.16	[1, 4]	10/25 GbE
B10	1x g3.16, 1x g4dn.12	[4] x 2	25/50 GbE
B11	2x g3.16, 2x g4dn.12	[4] x 4	25/50 GbE
B12	1x g4dn.2, 1x g4dn.12	[1, 4]	25/50 GbE

communication libraries of native TF, and choose baselines with the same runtime for fair comparisons.

Benchmark models and metrics. We choose diverse benchmark ML models following MLPerf (Mattson et al., 2019), listed in Appendix E. They cover: multiple CNNs, enlarged neural collaborative filtering of sparse and dense versions (NCF-large), LM1B language models, Transformers, BERT of different layers and sparse/dense versions, etc. We focus on evaluating the *system throughput*, and report *per-iteration training time*, instead of statistical convergence to evaluate the system. We conduct fully synchronous training, and exclude synchronization aspects that would change the nature or results of the ML training as in the original single-node program. We manage to train all benchmark models to the reported accuracy in MLPerf – hence skip this comparison in later sections.

3.1 Oceanus-composed Strategy Performance

In this section, we compose existing notable strategies as *fixed strategy builders* using AutoDist' composable backend, and compare to their original implementations in their corresponded specialized systems. We experiments on clusters B4-B8. We consider the following strategies: (1) `MirroredStrategy` from native *tf.distribute* – which is based on ring allreduce[1]; (2) `TF-PS-LB`: we improve the native TF parameter server, which is reported with major drawbacks in load balancing (Zhang et al., 2017; Peng et al., 2019): we track the loads of each node of $\mathcal{R}$, and dynamically place each variable of $\mathcal{G}$ to the next node with least loads. With this enhancement, we observed much stronger performance of TF2.0 distributed runtime – which we will use as a PS baseline; (3) `Horovod`: We deploy Horovod 0.19.0 with NCCL 2.4.7. It is a strong collective communication baseline specialized in using AllReduce/AllGather for parameter synchronization, with BO-based autotuned hypterparameters; (4) `AllReduce` builder: we compose an `AllReduce` builder using AutoDist kernels, which exactly reproduces the strategy of Horovod. Related hyperparameters for each model (e.g., collective fusion) is hand-tuned and the best performance is reported; (5) `PS` builder: we compose a parameter server using AutoDist kernels, with many known *rule-based optimizations*: it performs hierarchical Reduce/Broadcast (Li et al., 2014) on nodes with >1 GPUs to reduce network traffic; it *partitions* large variables (upon a hand-tuned threshold) into shards equal to number of nodes in $\mathcal{S}$, and evenly distributes shards, suggested by many PS systems (Li et al., 2014; Peng et al., 2019). We use it as a strong hand-optimized PS baseline.

[1]Among all strategies in tf.distribute, we only managed to setup `MirroredStrategy` on ≤4 GPUs within a single node.

Basedline performance. We compare these methods on BERT-large (340M), DenseNet121, ResNet101, InceptionV3, all with *dense gradients*, in Figure 5. We observe that collective-based strategies outperform the rest by a large margin, on all single node settings (1, 2, 4 GPUs), in which NCCL is advantageous. The improved TF PS, `TF-PS-LB`, scales well on CNNs (DenseNet121, InceptionV3, and ResNet101), outperforms `Horovod` occasionally in several settings (unlike reported (Kim et al., 2018; Zhang et al., 2017)). This verifies TF distributed runtime is a powerful and valid testbed. All methods scale sublinearly on BERT-large, among which `PS` builder performs best.

Composed vs. specialized. The composed `AllReduce` builder shows nearly the same performance with `Horovod` on all CNNs, but is slightly worse on BERT-large, perhaps attributing to Horovod autotuning (Sergeev & Del Balso) collective fusion. The hand-composed rule-based `PS` builder outperforms all other methods in distributed settings (>4 GPUs), particularly substantially on BERT-large, though BERT-large has no sparse gradients, for which collective strategies is suggested to work better per previous work (Kim et al., 2018). We will reinspect collective communication vs. PS in the next section.

To summarize, using distributed TF as runtime, hand-composed strategies using AutoDist backend exhibit at least matched performance with specialized systems, and offers the best *all-round* performance on all models *in one system* – which is a critical advantage to reduce system/strategy switching overhead and trial-and-error.

3.2 Study of Synchronization Aspects

In this section, we re-examine several synchronization aspects commonly chosen based on rules or heuristics.

Synchronization architecture. Different conclusions have been drawn in recent literature on advising how to choose synchronization architecture. Parallax (Kim et al., 2018) shows constant advantages of using NCCL AllReduce to synchronize dense gradients over PS. ByteScheduler (Peng et al., 2019), with self-implemented runtime, demonstrates

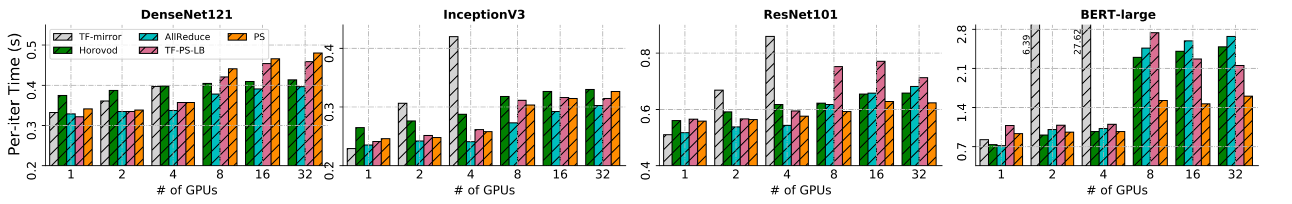


Figure 5. Based on TF distributed runtime, we comparing `MirrorStrategy`, `TF-PS-LB`, `Horovod` with manually implemented `AllReduce` and `PS` builders on AutoDist. See §3.1 for a detailed analysis.

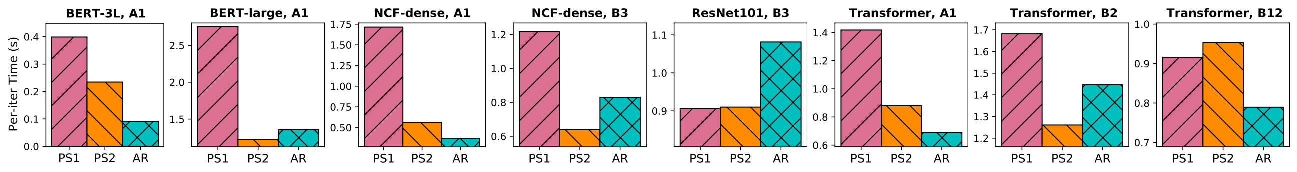


Figure 6. Comparing synchronization architectures across diverse models and cluster specifications on `TF-PS-LB` (PS1 in x-axis), `PS` (PS2), and `AllReduce` (AR), Per-iter training time is reported.

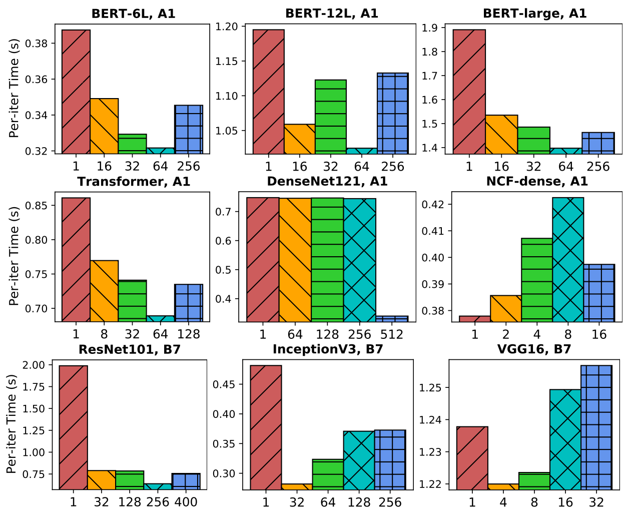


Figure 7. Using a `AllReduce(chunk)` builder, we study how the performance is impacted by the collective merge scheme on different $\mathcal{G}$ and $\mathcal{R}$. X-axis lists the value of `chunk`, the per-iter training time (lower is better) is reported.

PS with tensor partitioning is empirically better.

Using TF distributed runtime and NCCL, we reinspect the two architectures on a series of *dense* models across a list of heterogeneous clusters. We compare the three methods `TF-PS-LB` (PS1 in Figure 6), `AllReduce` (AR), and `PS` (PS2) defined in §3.1 in Figure 6. Note that `TF-PS-LB` differs from `PS` in that it does not partition tensors.

Should one use AR other than PS for dense messages (Kim et al., 2018)? On BERT-3L (11M), AllReduce demonstrates outstanding advantage (>2x faster) over both PS1 and PS2; but when increasing the model size to BERT-large, partitioned PS outperforms AR by a small margin. On NCF-dense, the ranks between PS2 and AR inconstantly change depending on the cluster. These observations obviously invalidate the statement, but comparing PS2 and PS1 seem to suggest that PS with variable partitioning is better than without partitioning (Peng et al., 2019) – as intuitively the former makes it easier to balance loads and schedule communication. However, on ResNet101-B3 and Tranformer-B12, PS1 shows noticeable advantage over PS2 – the two models have many variables, causing partitioning to incur significant stitching overheads that cancel out its gain.

Another common belief is – NCCL-based allreduce is likely to outperform PS when the majority of messages are transferred between colocated GPUs[2], which are observed in §3.1. Looking at Transformer, we however observe AR underperforms PS1/PS2 substantially on B2 (0.3s increase on per-iter time), a 3-node cluster each with 4 GPUs (hence good locality to leverage), but outperforms them on B12. This advantage counter-intuitively grows on A1, a cluster with no co-colocated GPUs at all.

We also experiment with NCF-sparse and LM1B (Chelba et al., 2013) with sparse gradients. The results echo the advantages of PS over AllGather on synchronizing gradients with sparsity above a certain level – which is however invalidated in an example shown later in §3.3.

In summary, we see that different synchronization architectures seem to demonstrate uncertain pros and cons – only depending on models and resources, and often violating rules-based heuristics.

Collective communication merge scheme. A critical optimization in collective-based systems is to merge multiple collective operations into one, to reduce the min-cost at the invocation of each collective. While adopted broadly (Sergeev & Del Balso, 2018; Larsen & Shpeisman, 2019), its effects and tuning against different $\mathcal{G}$,$\mathcal{R}$ remain largely unknown. To study it, we implement a `AllReduce(chunk: int)` builder with a *chunk*-based merging scheme. Under this scheme, starting from the first variable, the synchronization of every $n =$ `chunk` adjacent variables will be merged into one collective. Based on this builder, we experiment with different values of `chunk` on diverse model-resource combinations, shown in Figure 7.

[2]All our clusters have GPUDirect enabled when applicable.

We observe the chunk effects differently on different models – among examined values, the performance improvement due to merging can go up to >2x (ResNet101), or have no impact (VGG16), or even undermine the performance (NCF-dense). We hypothesize this is because NCF-dense uniquely has two adjacent large embedding variables; communicating them in one collective might be bandwidth-bounded – in this situation, partitioning them might be a better choice. Constantly, the performance drops when merging *all* variables into one collective – perhaps because it prevents opportunities of pipelining communication with computation of model parts, which is a key optimization adopted nowadays (Zhang et al., 2017; Peng et al., 2019). In choosing the optimal value of `chunk`, we cannot draw a constant conclusion, e.g., while on BERT-6L, BERT-12L, Transformer, 3 transformer-based architectures, the value approaches 64, it shifts toward 128 on BERT-large. How to create the optimal merge scheme that copes with structures in $\mathcal{G}$ and $\mathcal{R}$ is an NP problem, but has critical performance implications.

To conclude the results, understanding and interpreting such intrinsic subtleties in and between aspects are nontrivial. We do not think a typical ML practitioner should be reasonably expected to have this knowledge, and we argue this demonstrates the need for strategy composition, and corresponding auto-optimization methods.

3.3 End-to-end Strategy Optimization

In this section, we perform auto-optimization on three challenging models NCF-dense (122M), VGG16 (138M), and BERT-large (340M) and two large setups A2 and B7. We compare the following methods: (1) `Random+G`: we perform **g**uided search following Algorithm 1, but *without* a simulator. We set a larger search budget and allows it to explore 400 trials. (2) `AutoDist`: we use the full pipeline described in §2.4 with limited budget of 200 trials. (3) `ARBuilder`: the better one between `AllReduce` builder and `Horovod` described in §3.1. (4) `PSBuilder`: the better one between hand-optimized `PS` builder and `TF-PS-LB` in §3.1 – a strong PS representative based on TF runtime. For each trial, we train the model for 10 warm-up iterations, then collect the average per-iter time of another 40 iterations (hence on trial has 50 iterations).

End-to-end results. Figure 8 reports the performance of the *best* strategy found by `Random+G` and `AutoDist`. On all $(\mathcal{G}, \mathcal{R})$, `AutoDist` manages to find strategies better than both hand-optimized builders – at least 20% reduction of training time for all settings, and up to 60% on (VGG16, B7). On BERT-large, a challenging-to-parallelize model, `Oceanus` is 1.25x faster than the best hand-optimized baseline on A2, and 1.2x on B7. On VGG16, with less variables (hence easier for auto-optimization), we observe >1.6x speedup over the stronger baseline (`ARBuilder`) on B7. On NCF-large, a 10-variable model (hence little space for strategy composition), we observe 15% reduction on per-iteration time. Comparing `Random+G` to `AutoDist`: except on NCF-large, `Random+G` successfully finds strategies (slightly) better than hand-optimized in 400 trials. This suggests strategy composition, guided by a few simple constraints, can yield improvement over fix-formed strategies.

Auto-optimization net gain. BERT-large on B7 (batch size = 8x16 GPUs) normally needs 2M training steps to its reported accuracy. A 20% reduction in training time, if considering that g4dn.12x costs \$3.912/h/node and the budget for auto-optimization (20 trials = 1K steps), can save $\approx$\$2166 per training job, compared to hand-optimized strategies. But note auto-optimization removes development burden in strategy/system selection. The searched strategy is explicitly represented and might be reused across different training jobs to amplify the net gain.

Search space investigation. One way to validate the effectiveness of the suggested strategy composition is to see how many randomly composed strategies are above hand-optimized strategies during auto-optimization. We define *hit rate* as the percentage of strategies above the baseline among all strategies explored. Figure 9 (right axis) plots the hit rate vs. trials conducted. On the smaller model VGG16, throughout 200 trials, `Random-G` maintains 60% hit rate on A2 and 42% hit rate on B7. For BERT-large, the hit rates drop to 10% on A2 and 30% on B7, though. The BERT-large results in a much more challenging space with significantly more variables. The full `Oceanus` maintains hit rates above 80% in 200 trials, on 3 experiment settings, nearly 70% on VGG+A2. Despite the search algorithm, strategy composition depending on model and resource specification clearly offers promising performance improvement.

Budget vs. quality. Figure 9 (left axis) also visualizes the auto-optimization progress, in terms of the best strategy found vs. the number of trials conducted, comparing `Random+G` and `AutoDist`. Surprisingly, constrained random search can quickly hit strategies slightly better (1.1x) than heavily-optimized ones – in less than 20 trials on VGG, and in 80 trials on BERT-Large B7.

The full `Oceanus` discovers strategies better than hand-optimized shortly in 20 trials – in particular, the best strategy discovered for BERT-large on A2 only uses 50 trials on A2 and 30 trials on B7, out of 200 budget. For VGG16, AutoDist takes 20 trials to find strategies 1.2x better on A2, and 1.5x better on B7. This suggests that the cost model effectively narrows down space with few training data – for medium-sized models such as VGG16, using the auto-optimizer in place of hand-optimized strategies for training, even with a small allowed budget, can have substantial gain.

Strategy case studies. Besides settings in Figure 9, we perform auto-optimization on LM1B, BERT-sparse with

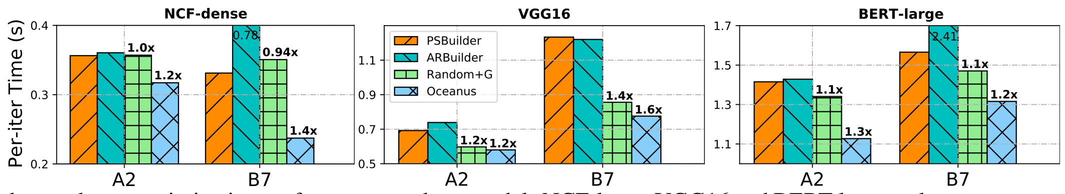


Figure 8. End-to-end auto-optimization performance on three models NCF-large, VGG16 and BERT-large and two resource specifications A2 and B7. The per-iteration (lower is better) time is reported.

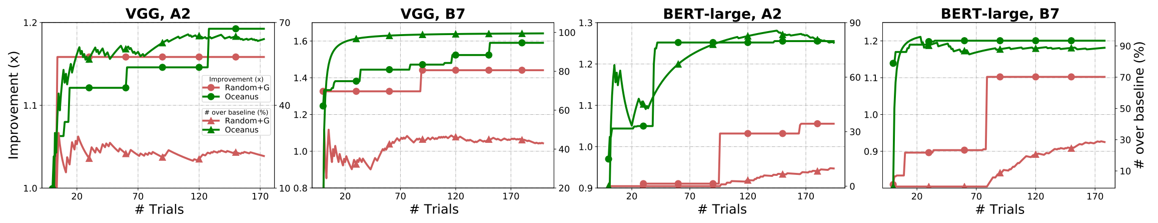


Figure 9. Auto-optimization progress, shown as performance improvement of the best strategy found so far (left y-axis) over baseline and the hit rate (right y-axis) vs. the number of trials conducted (x-axis). Baseline (1x) is the best among all methods used in §3.1.

sparse gradients, and ResNet101. We inspect the best strategies found in these experiments and reveal a few interesting findings: (1) PS is dominant on dense BERT-large; other settings have mixed PS/collective synchronizers. On ResNet, hybrid strategies outperform collective-only or PS-only significantly. (2) Deviated from Parallax, sparse gradients on cluster B often prefer being synchronized via AllGather than PS on LM1B and BERT-sparse. (3) We observe *partition-then-allreduce* patterns, contrast to the merge-then-allreduce pattern widely adopted in existing systems, in BERT-large and NCF-dense on cluster B, probably due to AWS 10Gbps single-flow bandwidth limitation.

In general, best-performing strategies have variable-specific aspect choices, and differ case-by-case, which are non-trivial to manually determine and implement.

4 Discussion and Related Work

Data-parallel synchronization systems. Poseidon (Zhang et al., 2017) and Parallax (Kim et al., 2018) use rule-based hybrid communication architectures to synchronize gradients. Horovod (Sergeev & Del Balso, 2018) brings in collective communication to reduce gradients for distributed DL. These systems are specialized based on one or two synchronization strategy, and exhibit varying performance when models or cluster setups change. Oceanus makes different aspect choices for different models/clusters, limited only by the available graph writing kernel library.

Synchronization performance can also be optimized at lower levels, such as scheduling or pipelining computation and communication (Harlap et al., 2018; Hashemi et al., 2018; Peng et al., 2019). While AutoDist does not include these optimizations as it assumes each aspect to map with a composable dataflow graph rewriting function, it can be integrated with and benefit from these system runtimes.

A few synchronization systems offer limited autotuning capabilities on certain parameters. Parallax (Kim et al., 2018) decides the partitioning of sparse variables and Horovod (Sergeev & Del Balso, 2018) tunes the collective merge scheme, both via trial runs. ByteScheduler (Peng et al., 2019) adjusts the credit size using Bayesian optimization. In contrast, AutoDist support strategy composition, and develops and optimizes a complete strategy expression.

DL graph rewriting and optimization. TVM (Chen et al., 2018b;a), TASO (Jia et al., 2019), XLA (Google TensorFlow XLA) perform automatic operator optimization and graph rewriting on a single device, mostly for inference graphs. AutoDist draws insights from these work, particularly AutoTVM (Chen et al., 2018b), but faces different challenges: the training needs to handle states, and the synchronization has unique semantics and challenges.

Representation of parallelisms. Device placement (Mirhoseini et al., 2017) automatically places nodes of a graph using a trained NN. Mesh-TensorFlow (Shazeer et al., 2018) defines “meshes” to describe how operations are partitioned and dispatched across devices. FlexFlow (Jia et al., 2018b;a) proposes the “SOAP” representation, and uses a stochastic MCMC-based algorithm to search for the optimal partitioning strategies for coarse-grained NN layers. Tofu (Wang et al., 2019) searches the optimal partitioning configruation for each operator. These techniques focus on partitioning or placement, and partially intersect with AutoDist as variables in the dataflow graphs are also represented as nodes (hence can be partitioned or placed using their techniques).

5 Conclusion

We present AutoDist, a composable and automated synchronization system for data-parallel distributed DL. AutoDist develops a unified representation for synchronization, and a corresponded composable backend to support strategy composition. AutoDist matches or outperforms specialized systems when representing their fixed-form strategies. We perform auto-optimization of synchronization strategies given model and resource specifications – AutoDist can find strategies better than manually optimized across multiple models and clusters, improving usability of parallel ML systems.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., and Isard, M. Tensorflow: A system for large-scale machine learning. *arXiv preprint arXiv:1605.08695*, 2016.

Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., Koehn, P., and Robinson, T. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 578–594, 2018a.

Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pp. 3389–3400, 2018b.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Google TensorFlow XLA. https://www.tensorflow.org/performance/xla/.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

Hashemi, S. H., Jyothi, S. A., and Campbell, R. H. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

Jia, Z., Lin, S., Qi, C. R., and Aiken, A. Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924*, 2018a.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018b.

Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 47–62, 2019.

Kim, S., Yu, G.-I., Park, H., Cho, S., Jeong, E., Ha, H., Lee, S., Jeong, J. S., and Chun, B.-G. Parallax: Automatic data-parallel training of deep neural networks. *arXiv preprint arXiv:1808.02621*, 2018.

Larsen, R. M. and Shpeisman, T. Tensorflow graph optimizations. *TensorFlow*, 2019.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 583–598, 2014.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, W. J. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

Mattson, P., Cheng, C., Coleman, C., Diamos, G., Micikevicius, P., Patterson, D., Tang, H., Wei, G.-Y., Bailis, P., Bittorf, V., et al. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 2430–2439. JMLR. org, 2017.

MPI, O. https://www.open-mpi.org/.

Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455. ACM, 2013.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., and Antiga, L. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pp. 8024–8035, 2019.

Peng, Y., Zhu, Y., Chen, Y., Bao, Y., Yi, B., Lan, C., Wu, C., and Guo, C. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 16–29, 2019.

Sergeev, A. and Del Balso, M. https://github.com/horovod/horovod/blob/master/docs/autotune.rst.

Sergeev, A. and Del Balso, M. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., and Young, C. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pp. 10414–10423, 2018.

Wang, M., Huang, C.-c., and Li, J. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 26. ACM, 2019.

Xie, P., Kim, J. K., Ho, Q., Yu, Y., and Xing, E. Orpheus: Efficient distributed machine learning via system and algorithm co-design. In *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–13, 2018.

Zhang, H., Zheng, Z., Xu, S., Dai, W., Ho, Q., Liang, X., Hu, Z., Wei, J., Xie, P., and Xing, E. P. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pp. 181–193, 2017.