SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2024, Vol. 67, Iss. 9, 192104:1–192104:21
https://doi.org/10.1007/s11432-023-3894-4

SDCC: software-defined collective communication for distributed training

Xin JIN[1*], Zhen ZHANG[2], Yunshan JIA[1], Yun MA[1] & Xuanzhe LIU[1]

[1]*School of Computer Science, Peking University, Beijing* 100871*, China;*
[2]*Department of Computer Science and Technology, Johns Hopkins University, Baltimore* 21218*, USA*

Received 5 January 2023/Revised 9 August 2023/Accepted 31 October 2023/Published online 31 July 2024

Abstract Communication is crucial to the performance of distributed training. Today's solutions tightly couple the control and data planes and lack flexibility, generality, and performance. In this study, we present SDCC, a software-defined collective communication framework for distributed training. SDCC is based on the principle of modern systems design to effectively decouple the control plane from the data plane. SDCC abstracts the operations for collective communication in distributed training with dataflow operations and unifies computing and communication with a single dataflow graph. The abstraction, together with the unification, is powerful: it enables users to easily express new and existing collective communication algorithms and optimizations, simplifies the integration with different computing engines (e.g., PyTorch and TensorFlow) and network transports (e.g., Linux TCP and kernel bypass), and allows the system to improve performance by exploiting parallelism exposed by the dataflow graph. We further demonstrate the benefits of SDCC in four use cases.

Keywords machine learning systems, distributed training, deep learning, collective communication, software-defined networking

1 Introduction

Deep learning is enabling a new breed of intelligent applications. These applications rely on deep neural network (DNN) models to achieve human-level or even super-human performance in many difficult tasks in computer vision and natural language processing [1–4]. To deploy a DNN model in an application, the model must first be trained on a dataset to achieve a certain prediction accuracy or loss.

DNN training is becoming increasingly distributed for two reasons. First, DNN models are becoming increasingly complex. A recent study from OpenAI shows that the amount of computing power needed to train state-of-the-art models grows faster than that prescribed by Moore's law [5]. Second, the demand for short training time is growing. An increasing number of applications are deploying DNN models, and training time is crucial for the development efficiency of these applications. Some emerging use cases continuously train models based on user runtime data and update the models at intervals as short as 10 min to enhance the user experience [6, 7].

Communication is crucial to distributed training. In distributed training, workers rely on collective communication to exchange gradients for updating model parameters. If not optimized, communication can account for as much as 90% of the training time [8, 9]. This issue is becoming even more severe with the trend toward larger models and training these models with more workers. Thus, there is a surge of research on improving communication efficiency for distributed training, which proposes a wide range of techniques from gradient compression and topology-aware aggregation to gradient scheduling and tensor fusion [8, 10–12].

Despite the vast amount of optimization techniques proposed in the literature, communication remains a key performance bottleneck for distributed training in production environments [13]. The root cause is not that these techniques are not effective but that the design of today's collective communication framework hinders the development, integration, and deployment of these techniques in practice.

* Corresponding author (email: xinjinpku@pku.edu.cn)

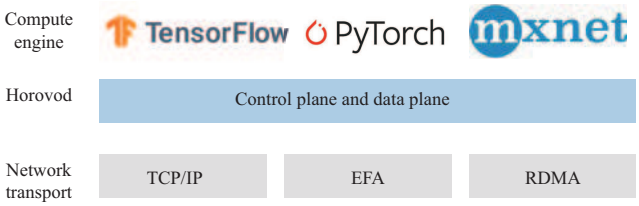


Figure 1 (Color online) Existing distributed training frameworks.

Distributed training today relies on native packages included in DNN computing engines (e.g., TensorFlow DDP and PyTorch DDP) or separately distributed training frameworks (e.g., Horovod and ONNX-runtime). They adopt a three-layer approach for collective communication (Figure 1). The middle layer (which is part of the native packages in computing engines or distributed training frameworks) interfaces with computing engines to receive gradient tensors and interacts with the bottom layer (i.e., network transports) to transport data among workers. The functionalities of the control plane and the data plane are tightly coupled in the middle layer. This approach has the following three limitations.

• **Flexibility limitation.** The tight coupling of the data plane and the control plane makes it difficult to add new optimization techniques or dynamically adapt existing ones based on runtime conditions. Implementing optimizations requires familiarity with the codebase of the middle layer, and often necessitates modifying multiple components, which is very time-consuming.

• **Generality limitation.** There exist multiple computing engines (e.g., TensorFlow, PyTorch, and MXNet) and network transports (e.g., Linux TCP, kernel-bypass TCP, and RDMA). A specific deployment often has few optimization techniques because of the considerable time and effort required to implement these techniques correctly and efficiently. For example, a new topology-aware aggregation algorithm for all-reduce would require two different implementations for PyTorch DDP and TensorFlow DDP, even though this algorithm has the same core logic.

• **Performance limitation.** The coupling makes it difficult to develop and deploy effective optimization techniques and tune performance for a specific scenario. Moreover, the coupling in the middle layer introduces nontrivial performance overhead. To reconfigure part of a system, the middle layer requires a complete shutdown and restart. Some systems even require restarting the upper layer.

In this study, we propose SDCC, a software-defined collective communication framework for distributed training. SDCC is designed based on the principle of modern systems design to cleanly decouple the control plane from the data plane. We redesign the collective communication stack for distributed training by dividing the functionalities into a control plane and a data plane and exposing the programmable components of the control plane to users.

SDCC abstracts communication operations in distributed training with dataflow operations, enabling users to easily express and implement new and existing collective communication algorithms and optimizations. SDCC exposes an event-driven application programming interface (API) for users to inject user-defined programs at appropriate points in the communication phase. SDCC unifies DNN computing and communication in distributed training with a single dataflow graph. The unified dataflow graph allows SDCC to exploit inter-operation parallelism among communication operations and compute operations when their dependencies are satisfied. Existing studies [14, 15] abstracted the communication as edges on the computation graph and did not provide programmable interfaces to users for developing optimization algorithms. SDCC provides an abstract data operation API for users to ease the development of communication algorithms and optimizations. It allows users to focus on the core logic of algorithms and leave other implementation details, such as runtime initialization and temporal buffer allocation, to SDCC. Moreover, the abstractions let user-defined programs generalize to different compute engines and network transports. With the clean separation of data and control planes and the modular abstractions, SDCC provides the following benefits.

• **Better flexibility.** SDCC simplifies the development, integration, and deployment of new and existing communication algorithms and optimizations for distributed training (Section 6).

• **Better generality.** SDCC provides "write-once, run-anywhere", to enable an optimization technique to support different computing engines and network transports (Section 5).

• **Better performance.** SDCC improves the training performance by allowing users to explicitly control and customize the gradient synchronization (Section 6) and reduces the control overhead by clean

separation (Section 5).

In summary, we make the following contributions.

• We propose SDCC, a software-defined collective communication framework for distributed training that cleanly decouples the control plane from the data plane.

• We abstract the communication phase with dataflow representation and unify DNN computing and communication in a single dataflow graph.

• We propose an abstract data operation API to provide flexibility and generality to user-defined communication algorithms and optimizations.

• We implement an SDCC prototype and integrate it with multiple computing engines and network transports. Experimental results show that SDCC is flexible, general, and has high performance. We demonstrate the benefits of SDCC with four use cases.

2 Background and motivation

In this section, we provide background on distributed training and existing frameworks, and motivate the design of SDCC.

2.1 Distributed training

DNN training on a single device usually consists of two phases, i.e., forward and backward. In the forward phase, the model takes a batch of data as input and computes the loss with respect to the labels of the input data. At the backward phase, the loss is used to compute the gradients for the model. The gradients are applied to update the model with an optimizer. The forward and backward phases are performed for several iterations until the model converges.

Because the size of DNN models and the amount of training data keep growing rapidly, the training task on a single device becomes time-consuming and even hardly practical. As a result, distributed training becomes a natural choice to reduce the training time. There are two major paradigms for distributed training, i.e., data parallelism and model parallelism. In this paper, we focus on data parallelism, as it is the most commonly used distributed training paradigm. Compared to single-device training, distributed training has an additional communication phase to synchronize the gradients of the model from different workers. Gradient synchronization can be done by either collective communication or parameter server (PS). Both of these two paradigms can be supported by the general architecture and the design of SDCC. To support PS structure, we can follow the design principles in Section 3 to restructure PS-based systems. In this paper, we choose collective communication as the implementation target, because it is widely adopted by popular DNN computing engines like PyTorch [16], TensorFlow [17], and those specialized for distributed training like Horovod [10] and DeepSpeed [18].

2.2 Distributed training frameworks

Is not collective communication well studied? Collective communication has been extensively studied, particularly in the high-performance computing (HPC) community. In the HPC setup, collective communication focuses on optimizing a single operation to distribute the exact data without loss in a homogeneous environment. However, distributed training is largely different from HPC in three aspects, which pose both research opportunities and challenges.

• **Error tolerant.** DNN training is tolerant of small errors. This gives rise to many lossy gradient compression techniques to reduce communication overhead [11, 19, 20]. Certain transformation and compression operations are applied before and after gradient synchronization, and the exact set of operations depends on both the values of the gradients and the runtime network condition.

• **Structured computing.** DNN training has a structured computing pattern with forward and backward passes. The data that need to be synchronized are spread over multiple DNN layers, and multiple synchronization operations are needed. This provides opportunities for domain-specific batching (i.e., tensor fusion) and scheduling (i.e., ordering gradient synchronization of different layers) to improve performance [10, 12, 21, 22].

• **Heterogeneous environment.** DNN training is often performed in heterogeneous environments that are shared by multiple tenants (e.g., public clouds or shared private clusters). The workers in a

Table 1 Comparison of distributed training frameworks

		Native	Horovod	SDCC
Flexibility	Scheduling	○	○	●
	Compression	◐	◐	●
	Comm. pattern	◐	◐	●
Generality	Optimization	○	○	●
	Network	○	●	●
	Engine	○	●	●

single training job can have disparate peer-to-peer bandwidth, depending on whether they are in the same server, rack, or pod, and the bandwidth is also affected by other jobs [9, 23–25].

Limitations of existing solutions. Despite a large amount of domain-specific algorithms and optimizations proposed for collective communication in distributed training, current distributed training frameworks still lack flexibility and generality (see Table 1), which leads to suboptimal performance in certain use cases.

Popular DNN computing engines have native packages to support distributed training, e.g., `nn.parallel` package in PyTorch, and `distribute` package in TensorFlow [17]. These packages are dedicated to their engines and thus have different designs. For instance, PyTorch provides hooks for users to manipulate gradients, while TensorFlow requires users to modify the computation graph to manipulate gradients. Indeed, such differences require non-trivial and even tedious engineering efforts to adapt new communication optimizations for different engines. Moreover, the control plane and the data plane are tightly coupled in these packages. They wrap the communication component with high-level APIs, e.g., `torch.nn.parallel.DistributedDataParallel` in PyTorch and `tf.distribute.Strategy` in TensorFlow, which is hard to modify or customize.

Several frameworks have been proposed specifically for distributed training, e.g., Horovod [10] and ONNX-runtime. They support different computing engines and different network transports. Figure 1 shows an overview of Horovod [10]. The distributed training is realized by a three-layer structure. Horovod itself does not handle data transmission. Instead, it relies on the underlying network transport such as TCP/IP, EFA, or RDMA to transmit data. Horovod mainly handles the interaction with the computing engines, e.g., receiving gradient tensors from the computing engines and notifying computing engines when gradient synchronization is done.

This three-layer approach lacks flexibility and performance. First, the control plane and data plane of the middle layer are tightly coupled, without exposing any interfaces for control and customization. Such coupling makes it hard to develop new collective communication algorithms and optimizations. For example, when adapting hierarchical all-reduce algorithms to network topologies [8] to avoid slow links, we need to do all-reduce multiple times with different participants. This optimization requires modifications on the code path of calling collective communication functions and creating multiple communication runtime contexts for different workers. The modification spreads across multiple components in Horovod and takes significant engineering efforts. Second, the coupling of the control plane and data plane also introduces non-trivial performance overheads for system reconfigurations. When the workers join or leave the system (e.g., for elastic training), Horovod requires a complete restart to rebuild the worker group for distributed training. But, if the control and data planes are cleanly separated, reconfiguring the network connections is sufficient, which is faster.

3 SDCC design

SDCC is a software-defined collective communication framework for distributed training. Compared with native PyTorch/TensorFlow packages and existing distributed training frameworks, SDCC provides flexibility, generality, and high performance (Table 1). It is designed with three principles.

- **Decouple control plane from data plane.** SDCC consolidates the communication-related functionalities in existing distributed training frameworks (Figure 1), and decouples them into a control plane and a data plane (Figure 2).
- **Abstract and unify DNN communication.** SDCC abstracts the communication phase in distributed training using a dataflow graph, and unifies it with the dataflow graph for computation. SDCC provides an event-driven API for users to customize the dataflow graph.

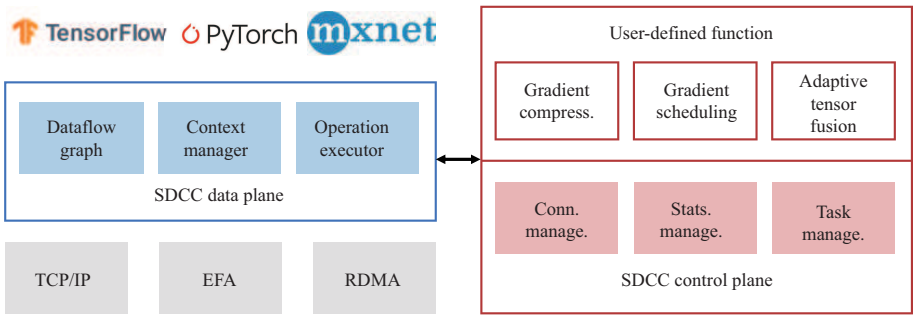


Figure 2 (Color online) SDCC overview.

• **Abstract data operation.** SDCC provides an abstract data operation API for users to construct data processing and communication algorithms. The abstraction provides the benefit of "write-once, run-anywhere".

In the remainder of this section, we describe the design of SDCC based on the three principles, and show how SDCC achieves flexibility, generality, and high performance.

3.1 Decouple control plane from data plane

SDCC modularizes the functionalities of collective communication and decouples them into a control plane and a data plane. This architecture enhances flexibility, offering the potential to customize user-defined collective communication algorithms. In the following subsections, we elaborate on the challenges to exploit the principle of decoupling within the context of distributed training and provide detailed explanations of the specific functions of the control plane and the data plane.

Decoupling the functionalities of collective communication. The principle of decoupling the control plane from the data plane has seen wide application and significant progress in software-defined networking. Nonetheless, applying decoupling to distributed training frameworks is challenging. On one hand, the expressiveness of the control plane must be capable of encompassing various existing and new collective communication algorithms. On the other hand, the data plane must be efficient enough to ensure training performance. To address these challenges, we (i) abstract the distributed training process into a unified dataflow graph to harness parallelism and extract a series of events for user logic injection (Subsection 3.2), and (ii) design a data-operation API from the computing engine and communication libraries to reduce the complexity of user development (Subsection 3.3). Figure 2 provides an overview of SDCC. The control plane manages the data plane, and provides an API for users to write user-defined functions to control the data plane behavior. The data plane handles the actual data processing and signals the control plane when certain events happen.

Control plane. The control plane contains a connection management module, a statistics management module, and a task management module. The connection management module manages the connections between workers. Developers can get the updated worker information from the connection management module, e.g., next and previous workers in a ring when a worker drops the connection. The statistics management module maintains the runtime information and periodically reports the runtime statistics to user-defined functions. Developers can get runtime information like the iteration time and react (e.g., change tensor fusion size) based on the information in user-defined functions. The task management module is responsible for enqueuing and applying the operations in the data plane based on user-defined functions.

Data plane. The data plane includes a dataflow graph module, a context manager module, and an operation executor module. The dataflow graph module is responsible for dynamically constructing the dataflow graph based on user-defined functions in the control plane. The context manager module maintains runtime contexts including contexts for communication libraries, e.g., communicator for NCCL, CUDA streams. The operation executor module executes data processing and communication operations from the user-defined function body. It provides the necessary API adaptation for underlying computing engines and communication libraries. The module is also responsible for creating temporal buffers and providing required runtime contexts for execution. Compared to the structure of Horovod [10] or ByteScheduler [12], SDCC provides a communication control API for users, which is not provided by existing studies. Besides, SDCC abstracts communication processes and data operations.

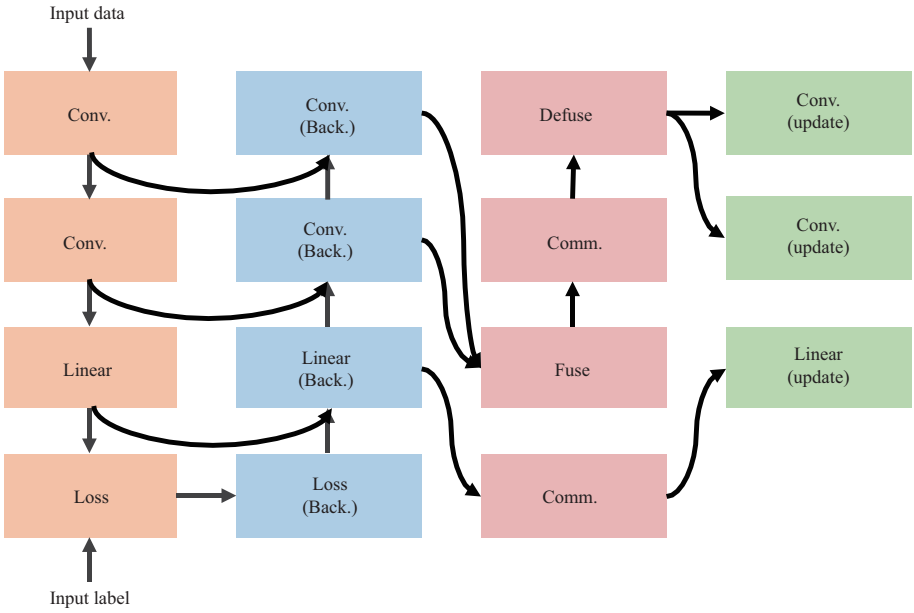


Figure 3 (Color online) Unify DNN computing and communication with dataflow graphs.

3.2 Abstract and unify DNN communication

SDCC abstracts DNN communication and unifies it with DNN computation. In the following, we first explain why we use a dataflow graph to represent DNN communication phase. Then we present how and what a user-defined function can do with our API. Finally, we illustrate how we realize the unified computing and communication graph.

Dataflow representation for DNN communication. SDCC abstracts the communication phase in distributed training as a dataflow graph. It is a common practice for DNN computing engines like TensorFlow and PyTorch to build static or dynamic dataflow graphs for forward and backward phases [17, 26–28]. These systems provide a dataflow abstraction for users to simplify application development, and allow developers to optimize system performance underneath. In terms of our problem, representing DNN communication as a dataflow graph provides the flexibility for users to develop collective communication algorithms and optimizations. New operations can be easily added to the dataflow graph, and different optimizations can be composed. In addition to the flexibility, the dataflow representation also allows SDCC to track the dependency between the operations and efficiently parallelize the operations to improve performance.

Unified dataflow graph representation. As we abstract the communication phase as a dataflow graph, we can unify DNN computing and communication in a single dataflow graph for execution. Figure 3 provides an example to illustrate how dataflow graphs of computing and communication can be unified. The left two columns of the dataflow graph depict the forward and backward phases. Each backward operation depends on the activation outputs from the corresponding forward operation, as well as the gradient outputs from the previous backward operation. The right two columns of the graph are the communication and weight update operations. To show how optimizations for communication can be integrated, we have one communication operation that depends on a backward operation (Linear Back.), and another communication operation that depends on a tensor fusion operation (Fuse). The fusion operation fuses two backward gradients of convolution layers (Conv. Back.), and the successive communication operation operates on the fused buffer. After the communication operation is completed for a fused buffer, the content will be copied (Defuse) to the original gradient buffer. As the DNN optimizer requires synchronizing gradients for model weight updates, each update operation depends on the corresponding communication operation. The communication nodes (Comm.) are dataflow graphs as well, and they are programmable to users.

The unified dataflow graph provides opportunities to exploit both intra-communication parallelism and inter-phase parallelism. Multiple operations can be executed in parallel when their dependencies are all satisfied. The unified graph further enables the communication operations to overlap with the backward

Table 2 Event-driven API of SDCC

Event	Description
`onTensorReady`	Gradient tensor enqueues
`onBufferReady`	Tensor buffer ready for sync.
`onTimerExpired`	Certain time interval reached
`onWorkerAdded`	Some worker joins training
`onWorkerRemoved`	Some worker drops

phase. For example, the communication operation (Comm.) for the linear layer at the bottom in Figure 3 can be executed in parallel with the backward computation of the first two convolution layers.

Injecting user-defined programs. SDCC provides an event-driven API for users to inject user-defined programs in the dataflow graph of gradient communication. Table 2 summarizes the API. For brevity, we omit the full function signatures in the table. To use the API, the user program first inherits the class containing the event-driven API, and then implements custom code logic in each event function body.

We provide the gist of what users can do with the API. An `onTensorReady` event is called with the input of the ready tensor object. Users can perform ordering control and data processing on the tensor. An `onBufferReady` event indicates the buffer of multiple ready tensors is copied into the buffer, and the buffer object is passed to the function. Users can perform collective communication or compression on the buffer. When an `onTimerExpired` event is called, the statistics management module is passed to the user-define program, which can collect runtime statistics like backward time and communication time. Users can change system parameters periodically with this API, e.g., changing threshold for fusion buffer and status checking frequency. `onWorkerRemoved` and `onWorkerAdded` events provide control points when the set of workers changes. The connection management module is passed into these two event functions.

Realizing the unified dataflow graph. SDCC links the communication graph with the computation graph, by triggering the events in the event-driven API at runtime. Specifically, SDCC inserts hooks to the backward computation component of the computing engine. When the computing engine finishes the backward computation of a tensor, the inserted hooks are triggered. Then SDCC saves the gradient tensor into its own task queues for the successive communication phase. In this way, SDCC opens the entry point for injecting the communication dataflow graph to the computing engine. To dynamically expand the dataflow graph from the entry point, SDCC systematically triggers a list of pre-defined events in the gradient communication phase. These task queues are exposed to user-defined programs for ordering control.

Figure 4 shows how SDCC achieves preemptive communication by dynamically triggering two events in the gradient communication phase. The gradient tensors from the computing engine are first pushed to the readiness check queue. After the readiness checking, the `onTensorReady` event is triggered. Users can proactively set the priority for each tensor inside the function body. The `getBackwardOrder` function is a built-in utility function. After that, the tensor is passed into a tensor scheduling queue. The tensor with the highest priority is scheduled for coordination among all workers. The coordination guarantees all workers have the same set of tensors for communication. Tensors in the set are fused into a fusion buffer. Tensors that are scheduled but not in the set wait for the next coordination. Once tensors are fused into the fusion buffer, the `onBufferReady` event is triggered. In the event handling function, users can implement communication optimizations using the abstract data operation API in Subsection 3.3. We give concrete examples in Section 6.

Other events allow users to actively adapting runtime conditions. SDCC periodically triggers the `onTimerExpired` event for users to configure runtime variables, e.g., tensor fusion threshold. The interval of the event is configurable. The connection management module monitors the connections of the workers and triggers `onWorkerAdded` and `onWorkerRemoved` events.

3.3 Abstract data operation

In principle, users can inject any custom code logic for communication, by manipulating the unified computing and communication graph. But programming communication or data processing logic from scratch is tedious. Users have to take care of connection initialization, computation stream initialization, etc. To lower the barrier of the development for collective communication, SDCC provides a set of abstract data operations for users so that users can focus on the algorithmic logic in the development.

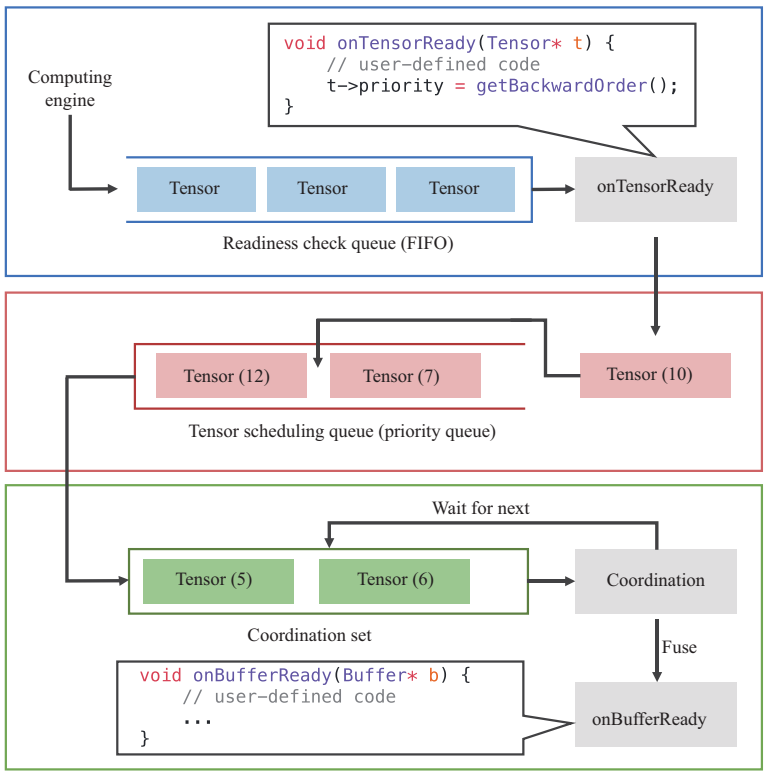


Figure 4 (Color online) Realizing the unified dataflow graph.

Table 3 Abstract data operation API of SDCC

Operation	Description
`astype`	Convert a tensor/memory to the given type
`apply`	Apply a function on a memory buffer
`alloc`	Allocate a memory buffer on accelerator
...	...
`allreduce`	Do all-reduce on the buffer
`allgather`	Do all-gather on the buffer
...	...

We summarize the key operations in Table 3. As the user-defined program focuses on the algorithm rather than implementation details, the abstract operation API brings the benefit of "write-once, run-anywhere" to the users. In the following, we provide the design choices we made to realize abstract operations for different computing engines and communication libraries.

Abstract data operations from computing engines. Existing distributed training frameworks require users to provide engine-specific implementations for tensor operations, e.g., type conversion and data selection. This imposes an extra development burden for compression algorithm developers to support multiple computing engines. And this is also true for the general-purpose framework Horovod.

SDCC introduces a proxy layer for tensor-level operations and a set of functions for memory-level operations. The proxy layer performs operations on the tensor level. It invokes corresponding functions in different computing engines, which is known via the meta-data of the tensor object. On the other hand, the API calls like `apply` and `alloc` are introduced for operations on the memory level to provide flexibility and generality. As such SDCC can support all kinds of data processing algorithms, even the unconventional tensor manipulations like 1-bit [19] and 4-bit [29] compressions. User-defined logic must explicitly provide meta-data for input and output buffers when using memory-level operations. SDCC launches the user-defined data processing program with necessary contexts, e.g., launching a CUDA kernel function on a CUDA stream and creating an output buffer with specified meta-data. Thus users can focus on the data processing algorithms. Operating on the memory level is general to different computing engines because different computing engines eventually operate on memory buffers.

We can further extend the abstraction to make user-defined data processing logic general to different accelerators, e.g., AMD ROCm. This can be achieved by wrapping the accelerator-dependent APIs and providing abstract operations for users to program data processing algorithms. We leave this extension as future work. The takeaway, here is the abstraction for data operations, is generally applicable in the design of distributed training frameworks.

Abstract data operations from communication libraries. There are many communication libraries developed for efficient communication, e.g., oneCCL, Gloo, NCCL, and MPI [30]. They are optimized for different scenarios and have different designs, which results in different initialization procedures and APIs. However, the algorithmic optimizations for the communication should not be affected, because these algorithmic optimizations are built on top of common communication primitives provided by communication libraries. For instance, the topology-aware [8] all-reduce can be implemented by calling the `allreduce` primitive for multiple times with different workers in the cluster. DGC [11] and 1-bit compression [19] are built with the `allgather` primitive. Thus, we need to decouple the data operation for algorithmic optimizations from the implementation details of underlying communication libraries.

SDCC introduces an intermediate layer to free users from library-specific nuance. It maintains library-specific contexts and translates communication operations to corresponding underlying libraries. The collective communication operation of the communication library must be bound to a specific communication context, which includes information such as communication topology and parallel strategies. Collective communications based on various communication topologies are bound to different communication contexts. To acquire the context, library-specific initialization is required before the actual communication. To avoid unnecessary repetitive initialization during each round of gradient aggregation, the intermediate layer continuously records the communication library contexts along with their corresponding topologies. When calling the communication operations in Table 3, users only need to specify the communication buffers and topology. The intermediate layer translates communication operations to corresponding underlying libraries. For instance, assuming NCCL is the target library and `allreduce` is the target operation, the intermediate layer will (i) determine the target function in NCCL library, which is ncclAllReduce(sendbuff, recvbuff, count, datatype, op, comm, stream); (ii) populate parameters, including pointers to sending and receiving buffer (`sendbuff` and `recvbuff`), communication data volume (`count`), communication data type (`datatype`), communication context-related parameters (NCCl communicator `comm` and CUDA stream `stream`), and operation type (`op`); and (iii) call `ncclAllReduce` and handle potential errors. Eventually, users will obtain aggregated communication results at the specified location.

4 Implementation

We have implemented a prototype of SDCC with ∼3000 lines of code (LOC) in C++ and ∼1000 LOC in Python.

Task management module. SDCC leverages Horovod for interfacing with different computing engines but redesigns the control logic for gradient synchronization. When gradient tensors are returned from a compute engine, it triggers a gradient synchronization event, which calls `EnqueueTensorAllreduces` function for synchronization in the end. We thus intercept that function of Horovod, so that the following all-reduce processing is handled by our system. We have two helper threads for handling readiness checking and communication scheduling, respectively. The readiness checking ensures data readiness, as the computation for gradient tensors is asynchronous. Once the data of a tensor is ready, the tensor is passed into the communication scheduling thread. The scheduling thread picks the top prioritized gradient tensors for communication. By default, all gradient tensors have the same priority. Though the gradient computation mainly uses CUDA C++, the code logic for launching these computations is on the Python side. Thus, for better overlapping gradient computation and communication, our helper threads are implemented in C++ to avoid the Python Global Interpreter Lock issue.

In SDCC, we wrap gradient tensors from different computing engines with a unified buffer operation interface and a priority attribute for ordering control. For users to inject custom logic, the `onTensorReady` event is triggered once the gradient tensor passes readiness checking. To efficiently handle buffer manipulations, e.g., type converting, we maintain an additional CUDA stream. All buffer operations via our interface are launched on the additional stream so that these operations can be executed in parallel with gradient communication and computation. To allow overlapping forward with backward and communi-

cation, we follow the design of ByteScheduler [12] to cross the global barrier, which blocks the forward and backward from efficient pipelining. We add locks for all layers. Once the forward computation of a certain layer at iteration t is completed, we lock that layer. The forward computation of that layer at iteration $t+1$ cannot start until the gradient communication of that layer is completed and the lock is unlocked.

Statistics management module. We implement this module with in-memory key-value storage. SDCC periodically calls `record` functions of this module to store runtime information. Recording a value with an existing key appends the value to the old ones. For the current implementation, to gather batch processing time, we trigger the function at the beginning of the backward computation of each iteration. The first gradient tensor is used as the anchor tensor to determine the boundary for a new iteration. The iteration counter is maintained for periodically triggering the `onTimerExpired` event, in which users call `get` functions of the module to consume runtime statistics. To ease the development, auxiliary functions for summarizing statistics, like `getAvgIterTime`, are provided. The interval of the `onTimerExpired` event is configurable with the environment variable.

We do not intend to cover all runtime statistics in our prototype but rather give a demonstration of the benefits of this modular design. We can easily extend SDCC with more runtime statistics by placing the record functions on the code path of gradient communication. Besides, users can leverage the record function and event-driven API to record the preferred information, because the module is accessible to users in the control plane. For example, users can record the communication time inside the implementation of `onBufferReady` function for each buffer.

Connection management module. This module actively checks membership status at the beginning of each backward and communication procedure. When a membership change (e.g., adding or removing workers from the job) is detected, the control plane sets the reconfiguration flag to be true to block the following communication operations. The time to detect a membership change is different on each node. If we start reconfiguration immediately, it could result in a deadlock or even a crash, because some workers might have ongoing communication operations, which requires the old runtime context. To avoid this issue in reconfigurations, all workers must agree on the reconfiguration status before proceeding. The control plane of each worker starts gathering the reconfiguration status from other workers until all reconfiguration flags are true. The control plane reconfiguration includes changing node rank and communication world size. The data plane reconfiguration includes resetting the communicators of underlying data communication libraries like NCCL or Gloo. To further improve the reconfiguration performance, we reimplement the data plane initialization phase in Horovod. We use multiple threads to overlap the rendezvous process of the connection initialization.

5 Evaluation

We evaluate the following aspects of SDCC.

- **Generality (Subsection 5.1).** Can SDCC support different training frameworks and network transports?
- **Fidelity (Subsection 5.2).** Do algorithms implemented in SDCC achieve high fidelity?
- **Efficiency (Subsection 5.3).** Does SDCC introduce performance overheads by decoupling the control plane from the data plane?

In this section, we validate whether SDCC can achieve comparable generality and fidelity to the distributed training framework Horovod while incurring negligible system overhead. Furthermore, we also explore whether the architecture of decoupling the control plane from the data plane enhances the efficiency of membership reconfiguration during elastic training. To maintain controlled variables, the performance results presented for SDCC in Section 5 are obtained based on the same collective communication algorithms as in Horovod. In Section 6, we show four cases to demonstrate how users can implement different optimizations to improve performance with SDCC.

Hardware and software. All experiments are conducted on AWS. The instance type is p3dn.24xlarge, which is configured with 96 vCPUs (Intel Xeon 8175M), 8 GPUs (NVIDIA V100 with 32 GB GPU memory), 768 GB memory, and 100 Gbps network. Linux `tc` command is used for network bandwidth control to simulate different network conditions. The software environment includes Horovod-0.21.0, PyTorch-1.5, torchvision-0.6, TorchElastic-0.2.0, TensorFlow-gpu-2.1, NCCL-2.8.3, and CUDA-10.1.

Models and datasets. We use four models in the experiments, which are ResNet50 [31], ResNet101 [31], VGG16 [32], and BERT (base-cased) [33]. These models are widely used in computer vision and natural language processing and are often used for distributed training benchmarks. Their model sizes are 97, 170, 527, and 440 MB, respectively, which allow the experiments to evaluate small, medium, and large models. The models also have different characteristics. VGG16 has a layer with 400 MB parameters, while the parameters in ResNet50, ResNet101, and BERT models are distributed more evenly in the layers. We use the ImageNet [34] dataset and MultiNLI [35] dataset for training, and set the batch size as 32. For the BERT (base-cased) model, we set the maximum sequence length as 128.

Evaluation metric. Unless otherwise specified, we use the training throughput as the evaluation metric. The throughput of a training job is measured by the number of images processed per second or the sequences processed per second, i.e., img/s or seq/s. Each number is reported as an average of at least 300 iterations. We use unmodified Horovod as the baseline. All data marked with Horovod in Figures 5–13 use NCCL as the collective communication library. Unless particularly specified, we use PyTorch as the training framework, TCP as the network transport, and ring all-reduce as the collective communication algorithm. By default, we use 128 MB as the maximum tensor fusion size.

5.1 Generality

To demonstrate the generality of SDCC, we show that SDCC can support different training frameworks and network transports, and provide throughput that matches or outperforms Horovod. Specifically, we integrate SDCC with widely-used training frameworks including PyTorch and TensorFlow, and representative network transports including TCP and elastic fabric adapter (EFA). EFA is a kernel-bypass network interface provided for AWS EC2 instances that enables high-performance inter-node communication on AWS. Currently, AWS does not support RDMA. Instead, AWS provides EFA for high-performance networking as a kernel-bypass transport solution. However, it is straightforward and easy to support RDMA in SDCC as well. On top of basic ring all-reduce, we add half-precision compression to the collective communication implementation to verify the generality of SDCC by applying a user-defined compression function on the data at the memory level. The experiments are performed on 32 GPUs. We normalize the throughput of SDCC to that of Horovod.

Different training frameworks. Figure 5 evaluates the generality of SDCC in terms of supporting different training frameworks. The experiment uses TCP as the network transport. For TensorFlow, the average difference in throughput on the four models between SDCC and Horovod is no worse than 1%. The performance of SDCC on ResNet models outperforms Horovod with TensorFlow by up to 13.4%. The average difference in throughput between SDCC and Horovod is no worse than 1.2% (observed under the VGG16 model). This experiment demonstrates that SDCC supports different training frameworks, and its performance matches or is better than Horovod.

The performance improvement of SDCC with TensorFlow is mainly due to better pipelining of backward computation and gradient communication. The batching procedure in Horovod happens on potentially unready tensors, which is because of the asynchronous nature of current computing engines. Returning a gradient tensor from the compute engine does not mean the data of the tensor is ready. In Horovod, batching unready tensors causes the gradient communication to be blocked by waiting for tensor data. SDCC, on the other hand, starts the batching on ready tensors only, which avoids blocking gradient communication.

Different network transports. Figure 6 evaluates the generality of SDCC in terms of supporting different network transports. The experiment uses PyTorch as the training framework. The TCP results on PyTorch are the same as those in the previous experiment. For EFA, SDCC also performs very close to Horovod. The average difference in throughput between SDCC and Horovod is no more than 1%. As such, we show that SDCC supports different network transports without compromising performance.

5.2 Fidelity

To evaluate the fidelity of SDCC, we measure the convergence speed of SDCC against Horovod. Because the training objective is to minimize the loss value on the training data, we choose training loss as the metric for measuring the model convergence. We plot the training loss to the number of elapsed iterations. The results of both Horovod and SDCC are evaluated on four V100 GPUs. Figure 7 shows the training loss of the four models for 30000 iterations. The results demonstrate that SDCC can provide almost

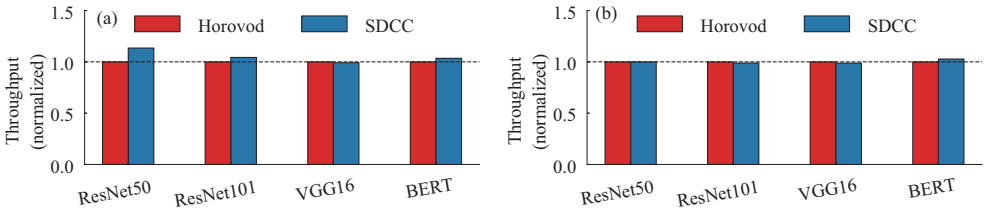


Figure 5 Ring all-reduce with half-precision compression on (a) TensorFlow and (b) PyTorch. Throughput is normalized to that of Horovod. The experiment uses 32 GPUs.

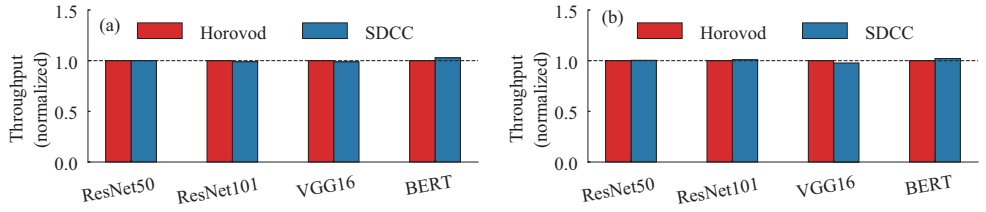


Figure 6 Ring all-reduce with half-precision compression on (a) TCP and (b) EFA. Throughput is normalized to that of Horovod. The experiment uses 32 GPUs.

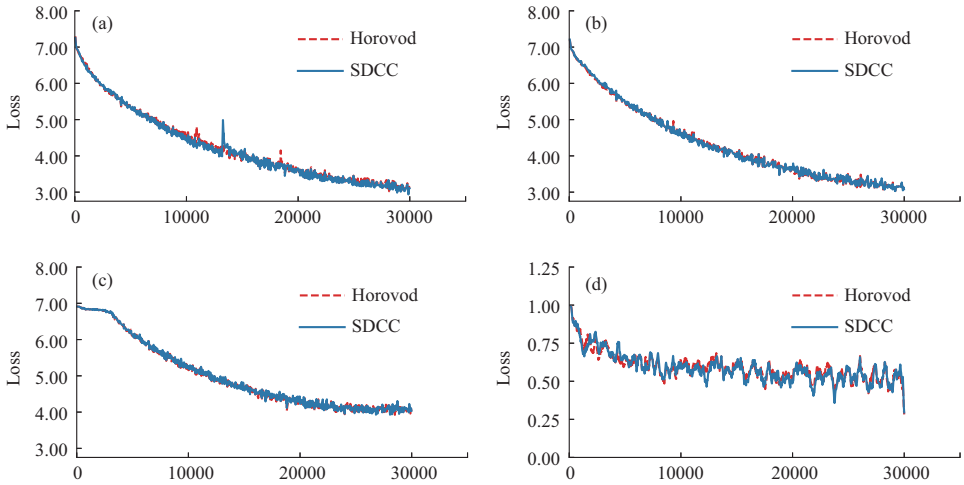


Figure 7 In the distributed training running on SDCC and Horovod, the training loss follows the same trend. (a) ResNet50; (b) ResNet101; (c) VGG16; (d) BERT.

the same convergence curve as Horovod. This means that SDCC can correctly complete the collective communication algorithm and provide the same convergence rate as Horovod in training.

5.3 Efficiency

Overhead. This experiment shows that the separation of the control plane and data plane brings low overheads compared to existing solutions. We compare the throughput of SDCC and Horovod with the number of GPUs growing up to 64. Here we measure the throughput with and without half-precision compression on each setup because only float16 compression is available to Horovod by default. Figure 8 shows the training throughput of SDCC and Horovod with different numbers of GPU workers on four DNN models. The results show that SDCC can obtain a similar throughput as Horovod in all cases. In the worst case, SDCC throughput is 4.7% lower than that of Horovod (VGG16, 64 GPUs, float16). In general, SDCC can provide training throughput similar to Horovod and therefore has low overhead. SDCC is slightly better than Horovod for deeper models like ResNets and BERT, mostly because of the improved pipelining of communication and computation, as explained in Subsection 5.1. In the float16 compression mode, SDCC is slightly worse than Horovod in some cases. It is due to the insufficient pipelining between compression and communication, as they perform sequentially. This is not a fundamental problem, and we leave the pipelining improvement as future work.

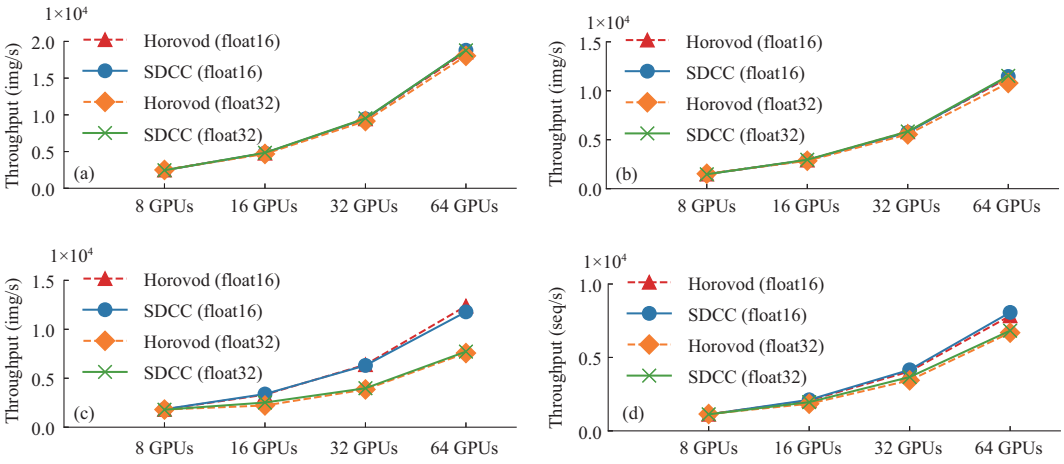


Figure 8 In distributed training with 8–64 GPUs and 100 Gbps bandwidth, SDCC and Horovod achieve similar throughput. (a) ResNet50; (b) ResNet101; (c) VGG16; (d) BERT.

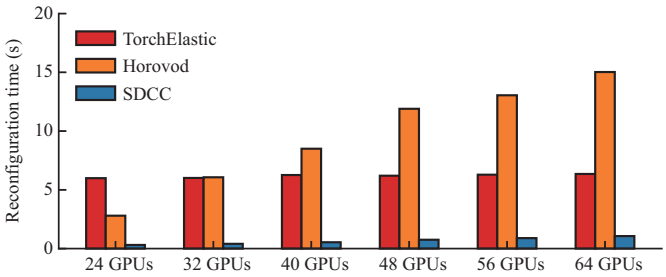


Figure 9 Response time when a worker drops randomly in elastic training. SDCC can rebuild the communication connections and resume training faster.

Performance benefits. We show that the separation of the control plane and data plane has performance benefits compared to existing solutions in this experiment. We measure the latency for membership reconfiguration in the control plane. Membership reconfiguration is a primary control plane operation in elastic training when the workers join (e.g., when more GPU resources are available) or leave (e.g., when a worker fails) a training job dynamically. Figure 9 compares the reconfiguration time of TorchElastic, Horovod, and SDCC when a worker exits training. The x-axis represents the number of GPUs before the worker leaves, and the y-axis represents the reconfiguration time. In TorchElastic, when reconfiguration happens, the training script on each worker is restarted completely. The time cost of CUDA initialization and model loading dominates the reconfiguration time. Horovod only restarts itself and costs less time to reconfigure as compared to TorchElastic, when the number of GPUs is small. However, the reconfiguration time grows rapidly with the number of GPUs. This is mainly due to the inefficient implementation of the rendezvous server in Horovod. Benefiting from the decoupling, SDCC only needs to rebuild the connections for the collective communication. In the case of 64 GPUs, the reconfiguration time of SDCC is about 6× and 15× faster than TorchElastic and Horovod, respectively.

In elastic training scenarios, the joining and dropping of workers lead to frequent membership reconfigurations. During these reconfigurations, training workers occupy computational resources without performing model computations or communication, resulting in resource and energy wastes. By speeding up the reconfiguration, SDCC not only reduces the idle time for servers and accelerators, minimizing energy wastes but also increases the proportion of effective model training time, thereby accelerating training efficiency.

6 Case study

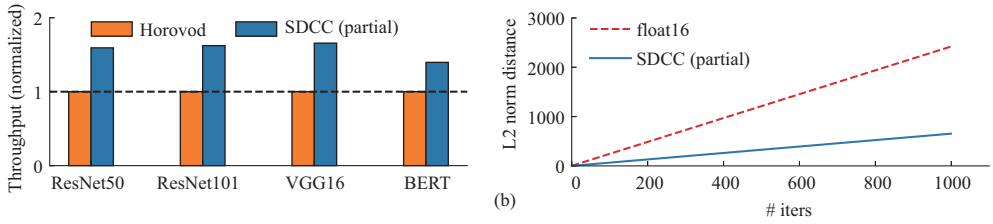
In this section, we provide four use cases to show how to implement popular communication optimizations with SDCC. The selected user cases include partial compression [11,19,20], gradient scheduling [12,21,22], topology-aware ring all-reduce [8,36], and adaptive tensor fusion [10]. These four cases involve data oper-

Table 4 Module overview for case study

Module	LOC	Characteristic
Partial compression (Subsection 6.1)	247	Data operation
Gradient scheduling (Subsection 6.2)	10	Scheduling operation
Topology-aware ring all-reduce (Subsection 6.3)	50	Topology optimization
Adaptive tensor fusion (Subsection 6.4)	43	Dynamic communication

```
__global__ static void add_half_float(
    float* dst, __half* src, uint64_t nelem) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  if (index >= nelem) return;
  dst[index] = __half2float(src[index]) + dst[index];
}
void onBufferReady(Buffer* buff) {
  std::vector<Buffer*> chunks;
  // ... split buff into chunks
  // reduce-scatter procedure with partial compression
  for (int i = 0; i < nranks - 1; ++i) {
    int _send_idx = (my_rank + nranks - i) % nranks;
    int _recv_idx = (my_rank + nranks - i - 1) % nranks;
    Buffer* _send_chunk = chunks[_send_idx];
    if (is_slow_link(my_rank, next_worker)) {
      Buffer* _compressed = astype(_send_chunk,
          dtype=SDCC::DataType::float16);
      send(_compressed, async=True);
    } else { send(_send_chunk, async=True); }
    // prepare temporal buffer, _recv_temp, for receive
    recv(_recv_temp, pre_worker);
    // sum received data into recv chunk
    void* reduce_func = is_slow_link(my_rank, pre_worker)?
        (void*)add_half_float : (void*)float_add;
    apply(reduce_func, chunks[_recv_idx], _recv_temp);
  }
  // all-gather procedure ...
}
```

(a)



(b)

Figure 10 Use case of partial compression. (a) Code for partial compression; (b) effectiveness of partial compression. Evaluated on 32 GPUs.

ations, communication scheduling, communication topology optimization, and dynamic communication strategies, covering the majority of features in existing collective communication algorithms. We demonstrate the flexibility that developers can control and optimize the communication with SDCC and show the effectiveness of our implementation. Table 4 summarizes the characteristics of different use cases and the LOC required to implement the use cases with SDCC.

6.1 Case 1: partial compression

In this use case, we assume distributed training happens on machines equipped with different types of network interfaces. It happens due to the compute cluster consisting of machines of different generations, or due to insufficient cloud capacity for certain types of cloud instances. In such a usage scenario, workers with slower network interfaces bottleneck the training throughput. Existing gradient compression techniques [11, 19, 20, 29] improve the performance by performing data compression among all workers. When workers communicate data with high-speed network interfaces, then compression does not necessarily improve performance and at the same time result in data precision loss. Thus, we propose a novel compression scheme, partial compression, to only compress data on low-speed network interfaces, which preserves gradient information as much as possible.

Figure 10(a) shows the code snippet for implementing the partial compression algorithm. The implementation mainly consists of two parts. The `add_half_float` function is a user-defined data processing

function, which is a CUDA kernel function. It fuses the decompression and summation operations. The `onBufferReady` function implements the ring all-reduce algorithm by composing reduce-scatter and all-gather procedures. When sending data through slow links, we compress float32 to float16 with the built-in `astype` function. For data received in compressed format, we use the custom `add_half_float` function for reduction. For brevity, we omit the implementation details of buffer preparation, slow link detection, and all-gather procedure. The takeaway here is that users can easily implement a custom gradient all-reduce algorithm.

The evaluation results on the left of Figure 10(b) show the effectiveness of the partial compression in terms of system throughput. We use four instances for evaluation. Each instance has 8 GPUs. We assume that three of them have 100 Gbps network cards and another one has a 10 Gbps network card. The setup emulates the scenario where users use compute clusters provisioned with different network interfaces. The throughput of SDCC with the custom partial compression outperforms Horovod by up to 65.6%. On the right of Figure 10(b), we show the accumulated $L2$ norm distances of partial compression and conventional float16 compression. The $L2$ norm of each iteration is computed from the difference of all-reduced results with and without compressions. The smaller accumulated norm distance means better data fidelity.

From Figure 10(b), we observe that when compared to Horovod, partial compression implemented based on SDCC achieves higher training throughput and demonstrates higher fidelity. By enabling users to deploy more efficient collective communication algorithms, SDCC not only improves training efficiency but also conserves energy usage, as distributed training often consumes substantial computational resources. Compared to other models, the optimization effect of the partial compression algorithm on BERT is relatively modest. This is because the forward and backward processes of BERT are relatively time-consuming compared to gradient communication. This characteristic allows the communication phase to effectively overlap with the computation phase, resulting in limited optimization space. The effectiveness of the partial compression algorithm depends on workload and hardware settings, such as accelerator computing capacity and network bandwidth. When communication time significantly outweighs computation time, there is more optimization opportunity for the partial compression algorithm.

6.2 Case 2: gradient scheduling

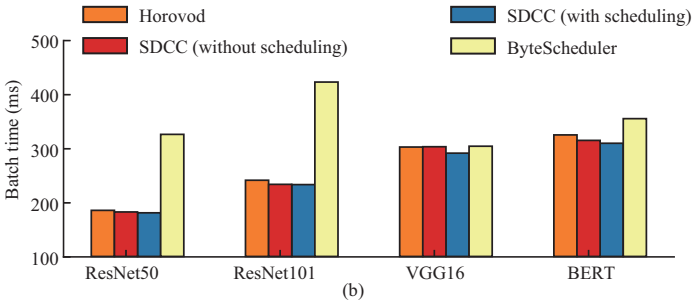
Several recent studies such as ByteScheduler [12], P3 [21], and TicTac [22] proposed gradient scheduling to improve the efficiency of distributed training. The main idea of gradient scheduling is to order the transmissions of the gradients in different layers to better overlap the forward phase and the backward phase. To implement the prioritized gradient scheduling policy, developers have to hack the existing distributed training frameworks. It typically takes hundreds to thousands of LOC [12]. We argue that the heavy engineering effort for implementing gradient scheduling is mainly due to the bad abstraction.

As shown in Figure 11(a), with the event-driven dataflow API in SDCC, developers can easily control the order of gradient synchronization by assigning priorities to tensors. The `getBackwardOrder` is a built-in utility function. SDCC can easily record the backward order of the layers in training, based on the tensor ready time. The later the tensor is ready, the larger the order is. Thus, the first few layers of the model have the largest order number. For the queueing order, we assume that the larger number means higher priority. SDCC supports tensor partitioning [12]. Users can split a large tensor into small tensors at the onTensorReady event, and then enqueue small tensors into the scheduling queue.

Figure 11(b) shows the impact of gradient scheduling with SDCC. We evaluate SDCC and Horovod with 32 GPUs on four p3dn.24xlarge instances. We record the batch time of each iteration. The lower the batch time is, the earlier the first layer starts its forwarding phase, which implies better performance. Overall, the figure demonstrates the effectiveness of the priority control by SDCC. The improvement of VGG16 is larger than that of other models. The reason is that the gradient communication for other models mostly completes in backward computation, which results in less optimization room with overlapping forward and communication [12]. Besides, there is an additional scheduling overhead. Thus, it is natural to see less performance gain. The ByteScheduler implementation performs much worse in our setups. It is mainly due to inefficient scheduler implementation at Python side that causes fragmented communication operations. In SDCC, the training throughput is improved by up to 5.9% with gradient scheduling.

```
void onTensorReady(Tensor* t) {
    t->priority = getBackwardOrder();
}
```

(a)



(b)

Figure 11 Use case of gradient scheduling. (a) Code for gradient scheduling; (b) effectiveness of gradient scheduling. Evaluated on 32 GPUs.

6.3 Case 3: topology-aware ring all-reduce

With the growing demand for DNN training, large-scale clusters in private and public clouds have been built to run training workloads. At the same time, the hierarchical topology of the datacenter network and the dynamic traffic load inevitably pose new challenges to distributed training jobs [8, 37]. The communication bandwidth between the workers in a training job is affected by their physical locations. As a result, a distributed training job with multiple workers typically has heterogeneous peer-to-peer bandwidth, depending on whether two workers are on the same server, same rack, or same pod. The heterogeneous network condition is mainly due to the network topology, which is different from Subsection 6.1. Such bandwidth imbalance makes the simple ring all-reduce algorithm sub-optimal.

In this use case, we show how to use SDCC to implement a two-level topology-aware ring all-reduce algorithm based on PLink [8]. For simplicity, we assume the network topology is available (which can be obtained by topology probing as in PLink) and the cloud instances are organized as a two-level hierarchy as in PLink. In the two-level hierarchy, the cloud instances are first grouped into several local groups (level one). One instance of each local group is selected to form the global group (level two). The two-level all-reduce algorithm first runs a ring all-reduce inside each local group. Then with the aggregated result of each local group, the algorithm performs a ring all-reduce at the global group. Finally, the selected node of each local group broadcasts the final results to its local group members.

Figure 12(a) shows the code of the two-level ring all-reduce algorithm. It applies the all-reduce algorithm inside the local group. Then, if the rank of the current worker is the selected one for the global ring all-reduce, it performs an additional all-reduce operation within the workers of the global group. At the end, we have a broadcast operation if the local group contains more than one rank. For those selected ranks in each local group, they are the sources of the broadcast. Other ranks receive the data from the broadcast sources. The `local_group` and the `global_group` are two vectors containing ranks of participants. For brevity, we omit code snippets for obtaining the local group and the global group. The `allreduce` function is an abstract wrapper for the underlying communication library, as referring to Subsection 3.3.

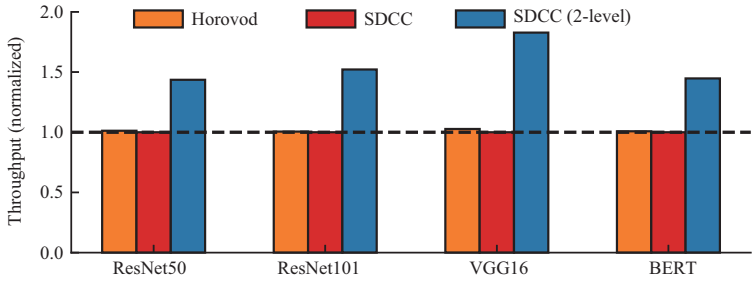
To demonstrate the effectiveness of topology-aware all-reduce, we evaluate our implementation with 32 GPUs on four p3dn.24xlarge instances. We evenly partition them into two local groups (level one). Within each local group, the network bandwidths to 100 Gbps. The inter-connection bandwidth of members in two local groups is 10 Gbps, except the selected ones, which have 100 Gbps network bandwidth with other selected ones in the global group (level two). Figure 12(b) shows the normalized training throughput of the two-level all-reduce algorithm (SDCC 2-level) versus the ring all-reduce algorithm (Horovod and SDCC). The topology-aware all-reduce algorithm performs better in four cases by up to 82.8%. The hierarchical all-reduce provided by Horovod does not use the topology, and thus it is not comparable.

```
void onBufferReady(Buffer* buff) {
    allreduce(buff, local_group);
    Buffer local_reduced = *buff;
    // the allreduce on next level should
    // be done on the output buffer
    local_reduced.input_buffer =
            local_reduced.output_buffer;

    if (rank == local_selected) {
        allreduce(&local_reduced, global_group);
    }

    if (local_group.size() > 1) {
        bcast(&local_reduced, local_group);
    }
}
```

(a)

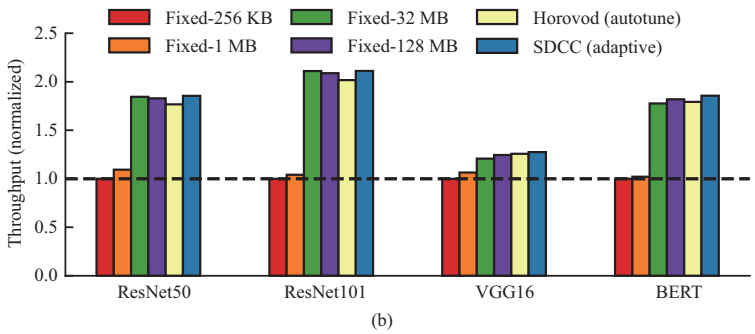


(b)

Figure 12 Use case of topology-aware ring all-reduce. (a) Code for topology-aware ring all-reduce; (b) effectiveness of topology-aware ring all-reduce. Throughput is normalized to that of SDCC ring all-reduce. Evaluated on 32 GPUs.

```
void onTimerExpired(StatModule* m) {
    double recent_t = m->getAvgIterTime();
    if (recent_t > last_t) {
        cur_step = -last_step;
    } else {
        cur_step = last_step;
    }
    fusion_threshold += cur_step;
    m->negotiate("fusion_size", fusion_threshold);
    last_step = cur_step;
    last_t = recent_t;
}
```

(a)



(b)

Figure 13 Use case of adaptive tensor fusion. (a) Code for adaptive tensor fusion; (b) effectiveness of adaptive tensor fusion. Throughput is normalized to that of fixed 256 KB strategy. Evaluated on 32 GPUs.

6.4 Case 4: adaptive tensor fusion

Batching is another well-known technique to improve the performance of network transmission. Tensor fusion batches multiple gradient tensors to a large tensor for efficient collective communication. But on the other hand, batching a large number of tensors also means that the transmission has to wait for all the batched tensors to be ready, which delays the transmission. Finding a suitable batch size for tensor fusion is important to the training performance.

The best fusion size is determined by the network environment, the computation power of the accelerator, and the structure of the DNN model [10]. In this use case, we show how to implement a hill-climbing algorithm to demonstrate how easy it is to write an adaptive tensor fusion algorithm with the `onTimerExpired` API of SDCC. Figure 13(a) shows the code snippet of the algorithm. The algorithm gathers the statistics of the iteration time from the statistics management module via `getAvgIterTime`. If the most recent iteration time is shorter, the algorithm follows the previous search direction; otherwise, the algorithm turns to the opposite direction. The `negotiate` function gathers all thresholds among all ranks, and picks the mode.

Figure 13(b) shows the benefits of the hill-climbing algorithm against fixed tensor fusion sizes. We evaluated our implementation on 32 GPUs. The hill-climbing algorithm can beat the fixed 256 KB and fixed 128 MB fusion sizes by up to 111.6% and 2.4%, respectively. SDCC outperforms Horovod with autotuning by up to 5.9%. We note that this algorithm is for illustration purposes, and thus can use simple control logic and limited runtime statistics. By wrapping the runtime information into the statistics management module and passing it to `onTimerExpired` API, SDCC can easily provide more runtime statistics, e.g., forward computation time and network bandwidth. SDCC brings flexibility to the users to interact with the system dynamically.

7 Related work

Software-defined networking (SDN). SDN decouples the control plane from the data plane. It has been deployed in production networks and has enabled several new use cases [38–45]. Many programming languages [46–49] and frameworks [47,50–57] have been proposed for simplifying the development of SDN and P4 [46] platforms. Other research efforts have applied the principle of decoupling the control plane from the data plane to individual components in the network stack [58–60]. We apply this principle to SDCC for flexibility, generality, and performance benefits.

Distributed training. Distributed training takes two popular approaches to distributed training: data parallelism and model parallelism [61–63]. Most widely used distributed training frameworks use data parallelism, e.g., PyTorch DDP, TensorFlow DDP, Horovod [10], MXNet PS, Microsoft ZeRO [64], and BytePS [65]. Our work SDCC focuses on data parallelism because of its popularity. Due to the tightly coupled architecture of currently distributed training frameworks [10,66], collective communication optimization algorithms [8, 11, 12, 19–22, 36] cannot be easily integrated with these frameworks. SDCC is a software-defined collective communication framework that extends the decoupling principle to the domain of distributed training. Furthermore, framework selection and job scheduling in training are emerging popular topics that focus on allocating GPU resources more efficiently [9, 23–25, 67]. These efforts are orthogonal to SDCC.

Collective communication. Communication libraries such as MPI [30], NCCL, and Gloo are widely employed in distributed training, providing collective communication primitives, e.g., all-gather and all-reduce. There are also industry efforts on collective communication libraries [68–71]. Benefiting from the decoupling design principle, SDCC can integrate with the aforementioned communication libraries, relieving the efforts of developers in implementing collective communication algorithms for different libraries. Apart from optimizing the underlying communication libraries, directions for improving communication performance include optimizing the transport layer [8, 72–74], reducing data transfer volume [11, 19, 20], synchronizing computation and communication [12, 21, 22], and optimizing communication with programmable switches [75, 76]. These solutions are implementable in SDCC as in Section 6 or complement SDCC for better performance.

Conventionally, the first instance of an abbreviated term needs to be spelled out once in the abstract and again in the main text, along with its abbreviation given in parentheses. Subsequent instances are to be abbreviated throughout the document. Conversely, if the term has been used only once in the entire document, then the abbreviation need not be mentioned. Kindly review all such instances in your document.

8 Conclusion

We proposed SDCC, a system that decouples the control plane from the data plane in collective communication for distributed training. SDCC abstracts the gradient communication phase as dataflow

operations and unifies DNN computing and communication in a single dataflow graph. SDCC provides abstract data operations to reduce the development burden for users. We demonstrated the flexibility, generality, and performance of SDCC via a range of experiments. We described four cases of developing important and popular algorithms for improving the performance of distributed training.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant No. 62172008) and National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas).

References

1 He K M, Zhang X Y, Ren S Q, et al. Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. In: Proceedings of IEEE International Conference on Computer Vision (ICCV), Santiago, 2015. 1026–1034
2 Geirhos R, Janssen D H J, Schütt H H, et al. Comparing deep neural networks against humans: object recognition when the signal gets weaker. 2018. ArXiv:1706.06969
3 Xiong W, Wu L, Alleva F, et al. The Microsoft 2017 conversational speech recognition system. In: Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Calgary, 2018. 5934–5938
4 Xiong W, Droppo J, Huang X, et al. Achieving human parity in conversational speech recognition. 2016. ArXiv:1610.05256
5 Hernandez D, Brown T B. Measuring the algorithmic efficiency of neural networks. 2020. ArXiv:2005.04305
6 You Y, Zhang Z, Hsieh C J, et al. Imagenet training in minutes. In: Proceedings of the 47th International Conference on Parallel Processing (ICPP), Eugene, 2018. 1–10
7 Jia X Y, Song S T, He W, et al. Highly scalable deep learning training system with mixed-precision: training imageNet in four minutes. In: Proceedings of Workshop on Systems for ML and Open Source Software at the Annual Conference on Neural Information Processing Systems, Montréal, 2018. 1–8
8 Luo L, West P, Krishnamurthy A, et al. PLink: discovering and exploiting datacenter network locality for efficient cloud-based distributed training. In: Proceedings of Conference on Machine Learning and Systems (MLSys), Austin, 2020. 82–97
9 Xiao W C, Ren S R, Li Y, et al. AntMan: dynamic scaling on GPU clusters for deep learning. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Banff, 2020. 533–548
10 Sergeev A, Balso M D. Horovod: fast and easy distributed deep learning in TensorFlow. 2018. ArXiv:1802.05799
11 Lin Y J, Han S, Mao H Z, et al. Deep gradient compression: reducing the communication bandwidth for distributed training. In: Proceedings of the 6th International Conference on Learning Representations (ICLR), Vancouver, 2018. 1–14
12 Peng Y H, Zhu Y B, Chen Y R. A generic communication scheduler for distributed DNN training acceleration. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles Organizers (SOSP), Ontario, 2019. 16–29
13 Zhang Z, Chang C K, Lin H B, et al. Is network the bottleneck of distributed training? In: Proceedings of the ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI), 2020. 8–13
14 Jia Z H, Zaharia M, Aiken A. Beyond data and model parallelism for deep neural networks. In: Proceedings of Conference on Machine Learning and Systems (MLSys), Stanford, 2019. 1–13
15 Jia Z H, Lin S N, Qi C R, et al. Exploring hidden dimensions in parallelizing convolutional neural networks. In: Proceedings of International Conference on Machine Learning (ICML), Stockholm, 2018. 2279–2288
16 Paszke A, Gross S, Massa F, et al. PyTorch: an imperative style, high-performance deep learning library. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems, Vancouver, 2019. 8026–8037
17 Abadi M, Barham P, Chen J M, et al. TensorFlow: a system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, Savannah, 2016. 265–283
18 Rasley J, Rajbhandari S, Ruwase O, et al. DeepSpeed: system optimizations enable training deep learning models with over 100 billion parameters. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2020. 3505–3506
19 Seide F, Fu H, Droppo J, et al. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In: Proceedings of the 15th Annual Conference of the International Speech Communication Association (ISCA), Singapore, 2014. 1058–1062
20 Lim H, Andersen D G, Kaminsky M. 3LC: lightweight and effective traffic compression for distributed machine learning. In: Proceedings of Conference on Machine Learning and Systems (MLSys), Stanford, 2019. 53–64
21 Jayarajan A, Wei J L, Gibson G, et al. Priority-based parameter propagation for distributed DNN training. In: Proceedings of Conference on Machine Learning and Systems (MLSys), Stanford, 2019. 132–145
22 Hashemi S H, Jyothi S A, Campbell R. TicTac: accelerating distributed deep learning with communication scheduling. In: Proceedings of Conference on Machine Learning and Systems (MLSys), Stanford, 2019. 418–430
23 Gu J C, Chowdhury M, Shin K G, et al. Tiresias: a GPU cluster manager for distributed deep learning. In: Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, 2019. 485–500
24 Bai Z H, Zhang Z, Zhu Y B, et al. PipeSwitch: fast pipelined context switching for deep learning applications. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Banff, 2020. 499–514
25 Yu P F, Chowdhury M. Fine-grained GPU sharing primitives for deep learning applications. In: Proceedings of Conference on Machine Learning and Systems (MLSys), Austin, 2020. 98–111
26 Zaharia M, Xin R S, Wendell P, et al. Apache spark: a unified engine for big data processing. Commun ACM, 2016, 59: 56–65
27 Gonzalez J E, Xin R S, Dave A, et al. GraphX: graph processing in a distributed dataflow framework. In: Proceedings of the 11th USENIX symposium on operating systems design and implementation (OSDI), Broomfield, 2014. 599–613
28 Murray D G, McSherry F, Isaacs R, et al. Naiad: a timely dataflow system. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP), Farmington, 2013. 439–455
29 Aji A F, Heafield K. Compressing neural machine translation models with 4-bit precision. In: Proceedings of the 4th Workshop on Neural Generation and Translation, 2020. 35–42
30 Gabriel E, Fagg G E, Bosilca G, et al. Open MPI: goals, concept, and design of a next generation MPI implementation. In: Proceedings of European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Budapest, 2004. 97–104
31 He K M, Zhang X Y, Ren S Q, et al. Deep residual learning for image recognition. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, 2016. 770–778

32 Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. In: Proceedings of the 3rd International Conference on Learning Representations (ICLR), San Diego, 2015. 1–14
33 Devlin J, Chang M W, Lee K, et al. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), Minneapolis, 2019. 4171–4186
34 Deng J, Dong W, Socher R, et al. ImageNet: a large-scale hierarchical image database. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Miami, 2009. 248–255
35 Williams A, Nangia N, Bowman S R. A broad-coverage challenge corpus for sentence understanding through inference. In: Proceedings of Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), New Orleans, 2018. 1112–1122
36 Cho M, Finkler U, Kung D. BlueConnect: novel hierarchical all-reduce on multi-tired network for deep learning. In: Proceedings of Conference on Machine Learning and Systems (MLSys), Stanford, 2019. 241–251
37 Farley B, Juels A, Varadarajan V, et al. More for your money: exploiting performance heterogeneity in public clouds. In: Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC), New York, 2012. 1–14
38 McKeown N, Anderson T, Balakrishnan H, et al. OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Comput Commun Rev, 2008, 38: 69–74
39 Jain S, Kumar A, Mandal S, et al. B4: experience with a globally-deployed software defined WAN. ACM SIGCOMM Comput Commun Rev, 2013, 43: 3–14
40 Hong C Y, Mandal S, Al-Fares M, et al. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In: Proceedings of Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), Budapest, 2018. 74–87
41 Hong C Y, Kandula S, Mahajan R, et al. Achieving high utilization with software-driven WAN. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), Hong Kong, 2013. 15–26
42 Gupta A, MacDavid R, Birkner R, et al. An industrial-scale software defined internet exchange point. In: Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, 2016. 1–14
43 Jin X, Li X Z, Zhang H Y, et al. NetChain: scale-free sub-RTT coordination. In: Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Renton, 2018. 35–49
44 Yu Z L, Zhang Y W, Braverman V, et al. NetLock: fast, centralized lock management using programmable switches. In: Proceedings of Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), 2020. 126–138
45 Zhu H, Bai Z, Li J, et al. Harmonia: near-linear scalability for replicated storage with in-network conflict detection. Proc VLDB Endow, 2019, 13: 376–389
46 Bosshart P, Daly D, Gibb G, et al. P4: programming protocol-independent packet processors. ACM SIGCOMM Comput Commun Rev, 2014, 44: 87–95
47 Shukla A, Hudemann K, Vági Z, et al. Fix with P6: verifying programmable switches at runtime. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), 2021. 1–10
48 Narayana S, Sivaraman A, Nathan V, et al. Language-directed hardware design for network performance monitoring. In: Proceedings of Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), Los Angeles, 2017. 85–98
49 Shah R, Shirke A, Trehan A, et al. pcube: primitives for network data plane programming. In: Proceedings of the 26th International Conference on Network Protocols (ICNP), Cambridge, 2018. 430–435
50 Monsanto C, Reich J, Foster N, et al. Composing software defined networks. In: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Lombard, 2013. 1–14
51 Koponen T, Casado M, Gude N, et al. Onix: a distributed control platform for large-scale production networks. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Vancouver, 2010. 351–364
52 Yeganeh S H, Ganjali Y. Beehive: simple distributed programming in software-defined networks. In: Proceedings of Symposium on SDN Research (SOSR), Santa Clare, 2016. 1–12
53 Hogan M, Landau-Feibish S, Arashloo M T, et al. Elastic switch programming with P4All. In: Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets), Chicago, 2020. 168–174
54 Wintermeyer P, Apostolaki M, Dietmüller A, et al. P2GO: P4 profile-guided optimizations. In: Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets), Chicago, 2020. 146–152
55 Doenges R, Arashloo M T, Bautista S, et al. Petr4: formal foundations for p4 data planes. In: Proceedings of ACM on Programming Languages (POPL), 2021. 1–32
56 Yu L C, Sonchack J, Liu V. Mantis: reactive programmable switches. In: Proceedings of Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), 2020. 296–309
57 Natesh V, Kannan P G, Sivaraman A, et al. Sluice: network-wide data plane programming. In: Proceedings of ACM SIGCOMM 2019 Conference Posters and Demos, Beijing, 2019. 156–158
58 Hsu K F, Beckett R, Chen A, et al. Contra: a programmable system for performance-aware routing. In: Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Santa Clare, 2020. 701–722
59 Narayan A, Cangialosi F, Raghavan D, et al. Restructuring endpoint congestion control. In: Proceedings of Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), Budapest, 2018. 30–43
60 Arashloo M T, Lavrov A, Ghobadi M, et al. Enabling programmable transport protocols in high-speed NICs. In: Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Santa Clare, 2020. 93–109
61 Narayanan D, Harlap A, Phanishayee A, et al. PipeDream: generalized pipeline parallelism for DNN training. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles Organizers (SOSP), Ontario, 2019. 1–15
62 Huang Y P, Cheng Y L, Bapna A, et al. GPipe: efficient training of giant neural networks using pipeline parallelism. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS), Vancouver, 2019. 103–112
63 Liu Y R, Hu Y Q, Qian H, et al. ZOOpt: a toolbox for derivative-free optimization. Sci China Inf Sci, 2022, 65: 207101
64 Rajbhandari S, Rasley J, Ruwase O, et al. ZeRO: memory optimizations toward training trillion parameter models. In: Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis, 2020. 1–16
65 Jiang Y M, Zhu Y B, Lan C, et al. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In: Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI), Banff, 2020. 463–479
66 Moritz P, Nishihara R, Wang S, et al. Ray: a distributed framework for emerging AI applications. In: Proceedings of the

13th USENIX Conference on Operating Systems Design and Implementation (OSDI), Carlsbad, 2018. 561–577
67 Dai H L, Peng X, Shi X H, et al. Reveal training performance mystery between TensorFlow and PyTorch in the single GPU environment. Sci China Inf Sci, 2022, 65: 112103
68 Prisacari B, Rodriguez G, Garcia M, et al. Performance implications of remote-only load balancing under adversarial traffic in dragonflies. In: Proceedings of the 8th International Workshop on Interconnection Network Architecture-On-Chip, Multi-Chip, Vienna, 2014. 1–4
69 Cowan M, Maleki S, Musuvathi M, et al. MSCCL: microsoft collective communication library. 2022. ArXiv:2201.11840
70 Shah A, Chidambaram V, Cowan M, et al. TACCL: guiding collective algorithm synthesis using communication sketches. In: Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, 2023. 593–612
71 Cai Z X, Liu Z Y, Maleki S, et al. Synthesizing optimal collective algorithms. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Virtual Event, 2021. 62–75
72 Wang G H, Venkataraman S, Phanishayee A, et al. Blink: fast and generic collectives for distributed ML. In: Proceedings of Conference on Machine Learning and Systems (MLSys), Austin, 2020. 172–186
73 Wan X C, Zhang H, Wang H, et al. Rat-resilient allreduce tree for distributed machine learning. In: Proceedings of the 4th Asia-Pacific Workshop on Networking (APNet), 2020. 52–57
74 Chen Y R, Peng Y H, Bao Y X, et al. Elastic parameter server load distribution in deep learning clusters. In: Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC), 2020. 507–521
75 Sapio A, Canini M, Ho C Y, et al. Scaling distributed machine learning with in-network aggregation. In: Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, 2021. 785–808
76 Lao C L, Le Y, Mahajan K, et al. ATP: in-network aggregation for multi-tenant learning. In: Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, 2021. 741–761