Journal of Parallel and Distributed Computing 198 (2025) 105027

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

OWL: Worker-assisted server bandwidth optimization for efficient communication federated learning

Xiaoming Han [a], Boan Liu [b], Chuang Hu [a,*], Dazhao Cheng [a,*]

[a] Wuhan University, Luojiashan Road, Wuhan, Hubei Province, China
[b] Hong Kong Polytechnic University, Hong Kong, China

ARTICLE INFO

Keywords:
Federated learning
Model distribution
Worker-assisted overlap

ABSTRACT

Edge computing in federated learning based on centralized architecture often faces communication constraints in large clusters. Although there have been some efforts like computation-communication overlapping and fine-granularity flow scheduling towards how to reduce the communication cost, this is still a matter of ongoing research. Motivated by the underutilization of bandwidth among workers (edge devices) and the replication of deep neural network (DNN) model distributions in data-parallel federated learning, we propose OWL, a novel worker-assisted server bandwidth optimization method. OWL partitions numerous computation branches into groups based on the model's network topology, allowing for overlapping model distribution and computation among workers, thereby leveraging idle communication resources on the workers to compensate for server bandwidth. To address the issue of model distribution congestion on the server, we formulate group partition as an optimization problem, which proves to be NP-hard. We tackle this problem through a divide-and-conquer approach employing an approximation grouping algorithm and a deploying algorithm. Finally, we evaluate the performance of OWL through simulations and a comprehensive real-world case study involving model training on OWL and deployment on edge systems. Experimental results demonstrate that OWL reduces overall training time by up to 20%-69% and improves scalability by over 9.5% compared to state-of-the-art overlapping approaches.

1. Introduction

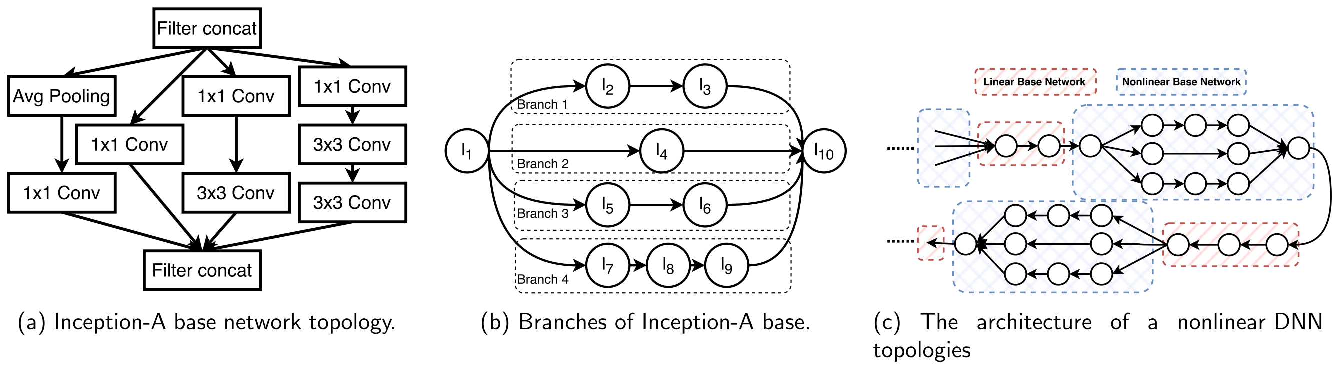
Deep learning (DL), a subfield of machine learning (ML), has demonstrated remarkable success across various applications, including super-resolution [1], object detection [2,3], recommendation system [4], Internet of Things [5,6] and Large Language Model (LLM) [7,8]. In DL, deep neural network (DNN) [9] models achieve high accuracy through the use of deeply layered structures with many parameters [10]. Training large DNN models typically requires numerous accelerators (such as GPUs) and a substantial amount of training data, along with prolonged runtime [11], which poses a significant cost barrier for researchers. For example, training a GPT-3 model [12] on 1024 A100 GPUs is estimated to take more than one month [13]. Meanwhile, the increasing regulations on data privacy [14] restrict data from leaving the source. However, as the storage and computational capabilities of the "workers" i.e. edge devices within distributed networks grow, it is possible to leverage enhanced local resources on each worker. This has led to a growing interest in federated learning, which explores training statistical models directly on remote workers.

In recent years, federated learning methods have been increasingly adopted by major service providers [15] and play a critical role in supporting privacy-sensitive applications where the training data are distributed among workers. One widely adopted framework for federated learning is the conventional centralized architecture [16]. In this architecture, a group of workers independently trains locally collected datasets and exchanges model parameter updates, typically represented as gradients, through a centralized server. The conventional centralized architecture effectively decentralizes the computational burden of model training to the workers, resulting in substantial reductions in training time. However, communication has become a critical bottleneck over the issue of *model distribution congestion* within the server [17].

Researchers have dedicated considerable effort to mitigating the consequences of congestion in model distribution. On one hand, several techniques have been developed to address this issue by reducing the communication overhead through methods such as Gradient Quantization [18–20] and Sparsification [21–23], which aim to make DNN models smaller. However, these approaches often result in compromised model accuracy as they involve a fundamental trade-off between

* Principal corresponding authors.
E-mail addresses: hanxiaoming@whu.edu.cn (X. Han), bo-an.liu@connect.polyu.hk (B. Liu), handc@whu.edu.cn (C. Hu), dcheng@whu.edu.cn (D. Cheng).

https://doi.org/10.1016/j.jpdc.2024.105027
Received 29 July 2024; Received in revised form 31 October 2024; Accepted 12 December 2024
Available online 16 December 2024
0743-7315/© 2024 Elsevier Inc. All rights reserved, including those for text and data mining, AI training, and similar technologies.



(a) Inception-A base network topology. (b) Branches of Inception-A base. (c) The architecture of a nonlinear DNN topologies

Fig. 1. Nonlinear DNN base and network architecture.

model size and accuracy [24,25]. On the other hand, the introduction of computation-communication overlapping has been proposed to regulate model updates and alleviate model distribution congestion. Unfortunately, due to the linear increase in latency with the cluster size, model distribution becomes the bottleneck in distributed learning.

It is evident that recent advancements in DNNs have moved beyond linear topology, with a growing preference for nonlinear architectures [26]. Prominent examples, such as transformer-based language models GPT-4 [27] and PaLM [28], and convolution-based visual models ResNet [29], and DeepLabv2 [30], all exhibit nonlinear characteristics. Nonlinear DNN topology encompasses layers where the forward computation is independent of certain other layers. This means that the computation of some layers and the parameter communication of others can occur simultaneously, capitalizing on the unused physical network bandwidth among workers. Fig. 1b provides an illustration of this concept, where the communication between l_2 and l_3 is unrelated to the computation of l_4. This overlapping capability presents an opportunity to alleviate communication bottlenecks by leveraging the idle physical network resources of workers.

In this paper, we introduce OWL, a novel worker-assisted server bandwidth optimization method for federated learning. In OWL, our approach revolves around dividing the nonlinear DNN topology into distinct segments and managing their deployment. By partitioning the parameters and distributing them among workers, OWL enables concurrent forward computation and parameter pulling from other workers. This overlapping procedure creates an efficient overlap between different workers' tasks. Notably, this optimization is predicated on the assumption that the computation graph remains unchanged. Compared to conventional centralized federated learning architecture, OWL offloads the responsibility of model distribution to specific workers, effectively mitigating model distribution congestion.

Within this new architecture, an essential question arises: *how can we schedule parameter communication and computation effectively to accelerate training while maintaining training accuracy?* This question presents two key challenges: 1) how to allocate model parameters in a manner that balances the communication overhead between the server and workers, and 2) how to schedule layer computation to expedite training. Addressing the first challenge, OWL employs an approximation grouping algorithm that efficiently groups parameters of similar sizes. As for the second challenge, OWL utilizes a deploying algorithm that optimally schedules forward computation and parameter communication to minimize training time while preserving accuracy. By tackling these challenges, OWL aims to enhance training efficiency without compromising the quality of the training process.

In summary, this paper makes three contributions:

- We demonstrate the issue of model distribution congestion in traditional centralized federated learning (§2.2). Through careful measurement, we found that model distribution significantly increases communication latency, even with overlapping communication and computation between the server and workers. We use examples to demonstrate that the idle bandwidth between workers, coupled with the nonlinear characteristics of DNN models in federated learning, offers a wider space for a qualitative leap in overlapping strategies (§2.3).
- We propose OWL (§3), a novel worker-assisted server bandwidth optimization method for federated learning. OWL decentralizes model distribution to specific workers and achieves communication-computation overlap among workers. We formulate a model distribution scheduling problem (§4.1) that is proven to be NP-hard (§4.2). To address this, we design an approximation Grouping algorithm (§4.3) and a Deploying algorithm (§4.4) that minimizes overall training time. We also provide implementation details of OWL (§5).
- We conduct experiments using real traces (§6). We compare OWL against the naive federated learning architecture and two state-of-the-art overlapping DNN training systems. The experimental results showcase the superior performance of OWL, with training speed improvements of up to 20%-69%. Additionally, we present a real-world case study involving the implementation of a face-recognized punch clock trained using OWL (§7).

2. Background and motivation

2.1. Background on DNNs training in federated learning

DNN Topology. Modern DNNs have evolved to consist of up to hundreds of layers, including input layers, hidden layers, and output layers. Recent advances show that DNNs are usually composed together in a sophisticated non-linear neural network. Programming frameworks such as TensorFlow/MXNet express DNN computation as a dataflow graph, as the layers no longer form a chain, but contain branches, joins and unrolled loops [26,31,32]. As a result, there are many different potential schedules for executing operators. If a DNN is organized in a directed graph, where each node represents one layer of the neural network, and the edges represent dependency among layers, nonlinear topologies include *branches* shown as Fig. 1.

Definition 1 *(Branch).* Given a set of layers $\mathcal{L} = \{l_i\}, i = 1...T$ (specially, l_1 and l_T represent the input layer and the output layer respectively), $\mathcal{N}$ is the set of links. A link $(l_i, l_j) \in \mathcal{N}$ represents that l_i has to be processed before l_j, and l_i feeds its output to l_j. A branch of $\mathcal{L}$ is defined as $B = (l_m, \ldots, l_n)$, where $m \leq n$. A branch of $\mathcal{L}$ and $\mathcal{N}$ should meet the conditions: 1) $(l_i, l_{i+1}), \forall i \in [m, n)$; 2) $(l_1, l_m) \in \mathcal{N}$; 3) $(l_n, l_T) \in \mathcal{N}$; 4) $(l_i, l_j) \notin \mathcal{N}, \forall i \in (m, n), j \notin (m, n)$

Fig. 1b shows the directed graph of the nonlinear Inception base network [33] in Fig. 1a. The Inception network includes four branches: (l_2, l_3), (l_4), (l_5, l_6), and (l_7, l_8, l_9).

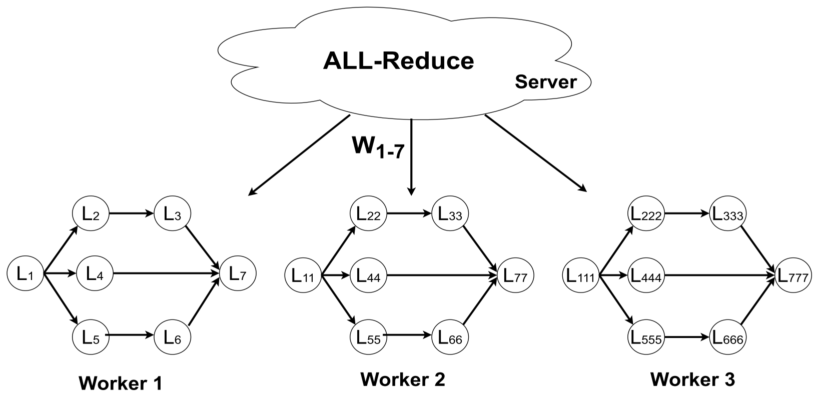


Fig. 2. An example application of federated learning. The number of workers are 3, and the load is a partial dataflow graph of a model.

Based on the definition of the branch, we can formally define *Nonlinear Base* as follows:

Definition 2 *(Nonlinear Base).* Given a set of layers $\mathcal{L} = \{l_i\}, i = 1...T$ (specially, l_1 and l_T represent the input layer and the output layer respectively) and the set of links $\mathcal{N}$. We define the set of branches as $\mathcal{B} = \{B_1, B_2, \dots, B_M\}$, and b_j is the number of layers in branch B_j.

Let $\mathcal{L}_j = \{l_1^j, l_2^j, \dots, l_{b_j}^j\}$ be the set of layers of branch B_j. Each layer must belong to only one branch, thus we have

$$\bigcup_{j=1}^{m} \mathcal{L}_j = \mathcal{L} \tag{1}$$

$$\mathcal{L}_i \cap \mathcal{L}_j = \emptyset, \forall i \neq j \wedge i = 1, 2, \dots, m \wedge j = 1, 2, \dots, m \tag{2}$$

Then we can formally define *Nonlinear neural network* as follows:

Definition 3 *(Nonlinear neural network).* Given a base set $Base = \{base_i\}, i = 1...Z$, a nonlinear neural network is defined as:

$$Nonlinear = (base_1, base_2, \dots, base_Z)$$

where the output of $base_i$ is the input of the $base_{i+1}$. $Nonlinear$ includes at least one nonlinear base network.

Fig. 1c shows a neural network including 4 linear or nonlinear bases. Different kinds of base networks are linked with each other, making the architecture complex.

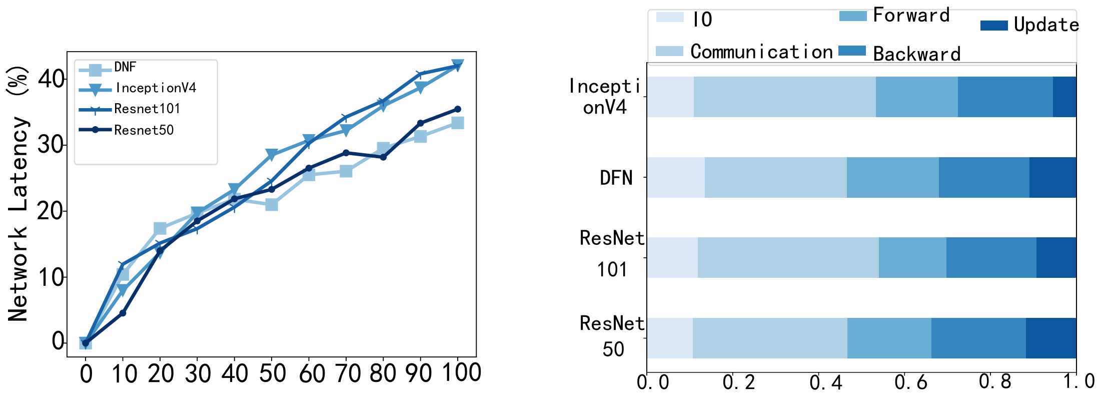
An Overview of Federated Learning. The key idea of federated learning is to allow distributed workers to collaborate and train an DNN model without sharing their private data, therefore preserving the data privacy of participating parties. Federated Learning emerged as a solution to the challenges posed by both centralized and distributed learning models. As shown in Fig. 2, the server maintains the latest parameter updates. Once the server collects gradients from all workers, it proceeds to update the model accordingly. Subsequently, the workers retrieve the updated parameters from the server and engage in forward and backward computations. The training process involves iterative execution of the following three steps until the model achieves convergence: 1) *Global Model Distribution:* The parameters residing in the server are distributed across all workers. This step ensures that each worker possesses the latest parameters before the next iteration. 2) *Local Model Update:* Within this step, each worker with the same DNN load independently computes gradients based on local data and allows for personalized model updates. 3) *Global Model Aggregation:* The server aggregates the gradients calculated by all workers and updates the model parameters through ALL-Reduce.

Expensive Communication. The primary challenge of federated learning is the communication bottleneck, which arises from the limited bandwidth available for communication. When the server receives a substantial number of communication requests from workers, it can severely impact system performance, potentially resulting in reduced efficiency during training. Several existing studies [34,35] have aimed to optimize the current architecture by exploring different approaches. One such method is communication and computation overlapping, which helps alleviate the communication bottleneck [36][37]. The two key independencies present during the training process pave the way for two types of overlap: 1) The parameter communication W_i is independent of the forward computation L_i, where i denotes the layer indices. 2) The gradient communication G_i is independent of the backward computation B_i. In this paper, we focus on the overlap between parameter communication and forward computation. However, with the explosive growth of DNN model parameters, the potential proliferation of workers, and the enhanced capabilities of GPUs/TPUs, network communication relying solely on server bandwidth may be several orders of magnitude slower than local computation, even with overlapping methods. In the following, we present a measurement study to illustrate the problem that needs to be addressed and to validate the feasibility of our proposed solution.

2.2. Measurement study

Measurement Setup. Each worker was equipped with a GeForce RTX-3080 GPU and a 1Gbps Ethernet connection. The workers were interconnected using a HUAWEI Ethernet switch S1700-24GR and operated on Ubuntu Server 20.04 with Linux kernel 4.4.6. We trained four different models using a method that overlaps communication and computation between the server and workers: Inception V4 [33], DFN [38], Resnet50, Resnet 101 [39]. All experiments were conducted using Pytorch 1.9.0.

Impact of Communication. The delay time percentage of the full training time is depicted in Fig. 3a as the number of workers increases. The presence of homogeneous hardware results in similar training progress across workers, and before each new epoch, the server must replicate and send the updated parameters to each worker, leading to physical network congestion on the server and creating performance bottlenecks. The impact of communication on training time is illustrated in Fig. 3b, where the average communication time and other time components per training step are shown for the Inception V4, DFN, Resnet101 and Resnet50 models. Note that the communication time here does not include the overlap time. It can be observed that communication time significantly contributes to the overall training time, particularly with the Inception V4 and Resnet101 load. In the case of DFN, although the communication time is relatively shorter, it still accounts for a substantial portion of the total training time, representing approximately 32% of the effective time used for model training. This indicates that overlapping computation and communication solely between the server and workers cannot cover the expensive communication overheads.



(a) The trend of physical network latency time with the number of workers.

(b) The percentage of time spent in one minibatch.

Fig. 3. Communication bottleneck in federated learning.

Time	T_1	T_2	T_3	T_4	T_5	T_6	$T_{7\text{-}19}$	T_{20}	T_{21}	T_{22}
Computation		L_1	L_{11}	L_{111}	L_2	L_{22}	L_{222}......L_{666}	L_7	L_{77}	L_{777}
Server->Worker (S->Wk)	$W_{1(S\text{->}Wk1)}$	$W_{1(S\text{->}Wk2)}$	$W_{1(S\text{->}Wk3)}$	$W_{2(S\text{->}Wk1)}$	$W_{2(S\text{->}Wk2)}$	$W_{2(S\text{->}Wk3)}$	$W_{3(S\text{->}Wk1)}$ .... $W_{7(S\text{->}Wk1)}$	$W_{7(S\text{->}Wk2)}$	$W_{7(S\text{->}Wk3)}$	
Worker->Worker										

(a) Computation and communication overlap strategy without worker assistance and scheduling branches

Time	T_1	T_2	T_3	T_4	T_5	T_6	T_7	$T_{8\text{-}12}$	T_{13}	T_{14}
Computation		L_1	L_{11}, L_{111}	L_2	L_{22} , L_{222}	L_3	L_{33}, L_{333}	L_4.......L_{66},L_{666}	L_7	L_{77},L_{777}
Server->Worker (S->Wk)	$W_{1(S\text{->}Wk1)}$	$W_{1(S\text{->}Wk2)}$	$W_{2(S\text{->}Wk1)}$	$W_{2(S\text{->}Wk2)}$	$W_{3(S\text{->}Wk1)}$	$W_{3(S\text{->}Wk2)}$	$W_{4(S\text{->}Wk1)}$	$W_{4(S\text{->}Wk2)}$ .... $W_{7(S\text{->}Wk1)}$	$W_{7(S\text{->}Wk2)}$	
Worker->Worker		$W_{1(Wk1\text{->}Wk3)}$		$W_{2(Wk1\text{->}Wk3)}$		$W_{3(Wk1\text{->}Wk3)}$		$W_{4(Wk1\text{->}Wk3)}$	$W_{7(Wk1\text{->}Wk3)}$	

(b) Computation and communication overlap strategy with worker assistance but no scheduling branches

Time	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
Computation		L_1	L_{111}	L_{11},L_{444} ,L_2	L_5, L_{22}, L_{222}	L_6,L_{44},L_{555}	L_3,L_{333},L_{55}	L_{33},L_{666},L_7	$L_{66}$$L_{777}$	L_{77}
Server->Worker (S->Wk)	$W_{1(S\text{->}Wk1)}$	$W_{2(S\text{->}Wk2)}$	$W_{4(S\text{->}Wk3)}$	$W_{5(S\text{->}Wk1)}$	$W_{6(S\text{->}Wk1)}$	$W_{3(S\text{->}Wk1)}$	$W_{7(S\text{->}Wk1)}$			
Worker->Worker		$W_{1(Wk1\text{->}Wk3)}$	$W_{1(Wk1\text{->}Wk2)}$ $W_{2(Wk2\text{->}Wk1)}$	$W_{2(Wk2\text{->}Wk3)}$ $W_{4(Wk3\text{->}Wk2)}$	$W_{5(Wk1\text{->}Wk3)}$	$W_{3(Wk1\text{->}Wk3)}$ $W_{5(Wk3\text{->}Wk2)}$	$W_{6(Wk1\text{->}Wk3)}$ $W_{3(Wk3\text{->}Wk2)}$	$W_{7(Wk1\text{->}Wk3)}$ $W_{6(Wk3\text{->}Wk2)}$	$W_{7(Wk3\text{->}Wk2)}$	

(c) Computation and communication overlap strategy with worker assistance and scheduling branches

Fig. 4. Different schedule base on computation and communication overlap strategy for a dataflow graph.

Discussion. It is clear that the issue lies in the limited server bandwidth and the need to infinitely replicate and send parameters. Fortunately, the idle bandwidth between workers, coupled with the nonlinear characteristics of DNN models in federated learning, can turn this situation around, offering a wider space for a qualitative leap in overlapping strategies.

2.3. Potential approach

Example. We use an example to demonstrate how the strategy of overlapping computation and communication, enabled by worker-assisted scheduling, affects the utilization of physical network bandwidth in federated learning, thereby influencing training speed. This example is a simple federated learning application, with its load derived from a portion of the data flow graph of a toy neural network, as illustrated in Fig. 2. This branching structure is common in modern CNNs (see Fig. 1). We only shows the forward computation and omits the backward part. The computation performed in the layer is labeled as L_{xxx}, and the parameters transmitted for L_{xxx} are labeled as W_x, where the value of x represents the layer index, and the quantity of x indicates the corresponding worker. To elucidate the issue more clearly, assume the server and all workers have equal network bandwidth (reasonable in scenarios with an explosive increase in workers), executing an L_{xxx} or an W_x requires one unit of time (T). The latest updated parameters initially reside on the server and must be transmitted to the workers before being used.

There are many ways to allocate parameters and schedule layer computation for Fig. 2. Fig. 4(a) shows an optimal example where communication solely relies on server bandwidth, with no assistance from workers, overlapping with the forward computation of load. We show two examples enabled by worker-assisted scheduling: in Fig. 4(b), all workers perform forward computation following the same layer execution sequence, from L_1, L_{11}, L_{111} to L_7, L_{77}, L_{777}; in Fig. 4(c), the workers utilize the nonlinear characteristics of DNNs to perform forward computation with different layer execution sequences. As communica-

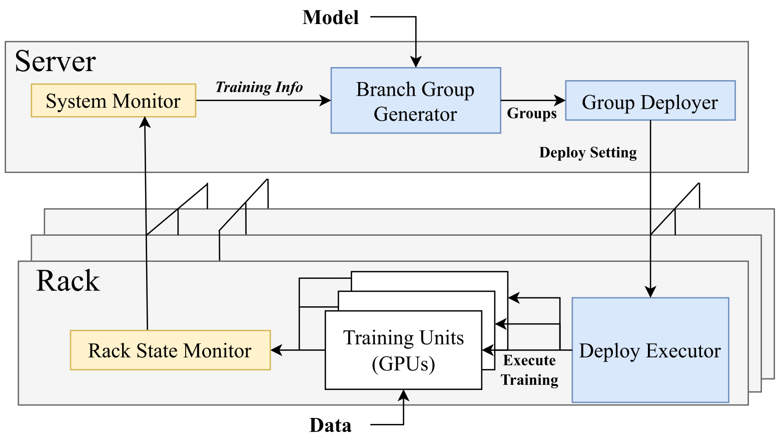


Fig. 5. The OWL system.

tion is duplex and concurrent with GPU execution, each table's 3 rows give the timeline of actions for GPU computation, Transmission parameters from server to worker, and Transmission parameters from worker to worker respectively, where $T_i = T$, for $T_i = T_1, \dots, T_n$.

Let's contrast Fig. 4(a) and Fig. 4(b) to see why parameters allocation affects training speed. Both (a) and (b) have the same execution sequences. However the total execution time of (a) is 8 units time (T) longer than that of (b). It can be observed that the idle bandwidth of the workers compensates for part of the server's bandwidth and overlaps with its communication as seen during the T_2 interval.

We contrast Fig. 4(b) and Fig. 4(c) to see why layer computation scheduling affects training speed. Both (b) and (c) use the idle bandwidth of the workers to compensate for the server. However, due to further communication overlap in (c), it takes 4 units of time (T) shorter than (b). Certain time intervals in (c), such as T_3, T_4, T_6, and T_7, exhibit not only the overlap of worker-to-server and server-to-worker communication as in (b) but also direct communication overlap among workers themselves. This significantly improves communication efficiency. For instance, during the T_3 interval, W_1, W_2, and W_4 are transmitted to their respective workers simultaneously. Subsequently, during the T_4 interval, L_{11}, L_2, and L_{444} can be computed simultaneously, benefiting from the nonlinear characteristics of modern DNNs.

Our approach. Since parameters allocation and layer computation scheduling critically affect training speed, we derive a training plan (aka which parameters are transmitted and when via leverage the features of nonlinear DNN load and available bandwidth between workers) assuming a given dataflow graph as well as a corresponding worker-assisted scheduling (§4). Specifically, the training plan optimizes computation and communication overlap by obtaining parameters of each layer from the server as early as possible and having the worker share the obtained parameters as much as possible.

We search the space of possible parameters allocation and layer computation scheduling to find a combination with the best training speed. Instead of using manual heuristics to constrain and guide the search, we adopt the Grouping Algorithm to search for a good combination of parameter allocation and layer computation scheduling. Grouping Algorithm has been used for NP-hard combinatorial problems and scheduling in parallel systems. We chose Grouping Algorithm among other search heuristics (e.g. colony systems, genetic algorithms and M-GRASP) because it is fast: Grouping Algorithm can be parallelized and computed efficiently on multicore CPUs. However, real-world model configurations and physical deployments can introduce additional complexities. For example, the number of branches may not match the number of workers, and GPUs might be deployed in rack servers where intra-rack bandwidth is typically larger than inter-rack bandwidth. When workers and servers span multiple racks in cloud-based training, significant delays can occur. Several studies have focused on designing parameter servers that are aware of the physical network topology. For instance, SwitchML [40] designed a system that supports multiple racks by hierarchically composing several instances of switch logic. However, no prior research has fully leveraged available bandwidth to achieve communication overlap among workers. We utilize deploying algorithm that optimally schedules forward computation and parameter communication to minimize training time while preserving accuracy.

3. Design overview

The OWL system (see Fig. 5) follows a centralized architecture. It consists of a server and multiple racks (a standardized frame or cabinet used for mounting and organizing computing equipment) each of which contains homogeneous GPUs (i.e., workers). On each rack, it has a **Rack State Monitor** to estimate the forward/backward computation time of each layer on the GPU. It also has a **Deploy Executor** to execute the deploy setting.

On the server-side, OWL has a **System Monitor** to estimate *Training Information* including the inter-rack/intra-rack bandwidths and rack information like GPU numbers and rack numbers. The cores of OWL are a **Branch Group Generator** and a **Group Deployer**. The branch group generator collects the training information and runs the grouping algorithm (details in §4.3) to partition each nonlinear base of the DNN into groups with the branch as a unit. The group deployer takes the groups information as input and runs deploying algorithm (details in §4.4) to compute *deploy setting* that determines how to distribute the generated groups to workers, with the objective of minimizing the overall training time.

The workflow of OWL is as follows: when a training task arrives in the OWL system, the rack state monitor estimates the forward/backward computation time of each layer by running the forward/backward on GPUs. Note that layer-wise training time is the main constraint which is supposed to be smaller than the communication time in deployment. After that, the system monitor estimates the training information and sends it to the branch group generator. Then the branch group generator computes groups. The group deployer computes the deploy setting taking the groups as input. Finally, deploy executors in workers execute parameter communication and training in GPU.

4. Model distribution optimization

4.1. Problem formulation

4.1.1. Nonlinear neural network modeling

We consider a nonlinear neural network $\mathcal{N}_{nl} = (\mathcal{N}_1.\mathcal{N}_2, \dots, \mathcal{N}_A)$. For every base network N_i, M_i is the branch number of N_i and $\mathcal{B}_i = \{B^i_1, B^i_2, \dots, B^i_{M_i}\}$ denotes the branches of N_i. Noted that linear base network can be considered as a one branch nonlinear base network. $\mathcal{B}_i$ should meet the condition:

$$\bigcup_{j=1}^{M_i} \mathcal{B}_j^i = \mathcal{B}_i \tag{3}$$

Correspondingly, the data size of branch B_j^i is d_j^i.

4.1.2. Cluster modeling

We focus on the GPU rack server in this paper. A rack is a kind of GPU server. The difference between the conventional GPU server and the rack server is that the bandwidth between racks and in racks is different. Bandwidth in racks often goes with PCIe or an independent switch that has a higher speed. To formulate the communication time in rack architecture, we consider a rack cluster $\mathcal{R} = \{\mathcal{R}_1, \dots, \mathcal{R}_C\}$. For each rack $\mathcal{R}_i$, $\mathcal{R}_i = \{g_1^i, \dots, g_{n_i}^i\}$ denotes GPUs in rack $\mathcal{R}_i$. A GPU represents a worker.

Bandwidth between racks and inside a rack is different in the rack architecture. We notate bandwidth between racks as $\mathcal{B}_{Inter} = \{B_{Inter}^{i,j}\}, i, j \in \mathcal{R}$, where $B_{Inter}^{i,j}$ denotes the bandwidth between rack i and rack j. Similarly, $\mathcal{B}_{Intra} = \{B_{Intra}^1, \dots, B_{Intra}^W\}$.

4.1.3. Load constraints

In this section, we will focus on one nonlinear base network N_r. To simplify the representation, we eliminate the subscript r. In our proposed potential method for federated learning, a complete training process includes 6 phases: pulling part of the parameters from the server, pulling the remaining parameters from the worker, forward computation, backward computation, pushing gradients, and model update. In our proposed method, gradients pushing, the backward computation, and model update for the nonlinear neural network is the same as the conventional linear neural networks. So we notate the time over the three phases as E_0. To formulate the full training time, we denote the communication time of group c as T_{tp}^c and the forward computation time of group i as T_{fw}^i. We focus on overlapping forward computation with communication in nonlinear neural networks.

Our idea is to group branches. We can consider the original approach as a particular case, where each branch forms a group. In this way, the overlap of branch communication and computation is difficult to control. The imbalance of proportions and the large number of time-sharing will increase the control cost greatly, which is obviously meaningless. However, due to the often unbalanced branch partition in DNN, this grouping strategy is usually suboptimal. We then prove that this is NP-hard and propose our algorithm.

The grouped branches can be further represented as:

$$\mathcal{G} = \{\mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_D\} \tag{4}$$

Thus we have the constraint:

$$\bigcup_{i=1}^{D} \mathcal{G}_i = \mathcal{B} \tag{5}$$

$$\mathcal{G}_i \cap \mathcal{G}_j = \emptyset, \forall i \neq j \wedge i = 1, 2, \dots, D \wedge j = 1, 2, \dots, D \tag{6}$$

Firstly we need to represent the communication time. The communication time for node a in rack R_a to node b in rack R_a is formulated as:

$$T_{tp}^i(a,b) = \frac{d_i}{B_{a,b}} = \begin{cases} \frac{d_i}{B_{Inter}^{a,b}} & R_a \neq R_b \\ \frac{d_i}{B_{Intra}^{R_a}} & R_a = R_b \end{cases} \tag{7}$$

where d_i is the model size of the group i and $B_{a,b}$ is the bandwidth between node a and node b. In our architecture, $B_{a,b}$ can be bandwidth of inter-racks or bandwidth of intra-racks, depending on which rack node a and b belongs to. Given a t layers nonlinear neural network N_{nl}, a federated learning deployment setting can be formulated as a set of deployment ways for all groups. We analyze the potential approach illustrated in the example in Section §2.3. One group is pulled from the server while others are pulled from other workers. So we can notate deploy set $\mathcal{D}$ for all w workers as $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_W\}$. And $D_i = ((u_1^i, v_1^i), (u_2^i, v_2^i), \dots, (u_W^i, v_W^i))$, where u_j^i represents which worker or server that worker i communicates with and v_j^i denotes which group that worker i pulls from worker u_j^i. Noted that although worker i can pull parameters of group v_j^i from worker u_j^i, worker u_j^i must pull parameters of group v_j^i from server first. So we have 2 constraints:

$$\bigcup_{i=1}^{W} \{u_j^i\} = \mathcal{G}, \forall \mathcal{R}_k \in \mathcal{R} \wedge \forall j \in \mathcal{R}_k \tag{8}$$

$$j < m, \forall v_j^i = v_m^l \tag{9}$$

Eq. (8) makes sure that a valid deploy covers all parameters in one model. Eq. (9) makes sure that a worker can pull parameters from other workers if and only if the worker pulls parameters from the server first.

Based on the notation for groups and clusters, we can represent the full training time in OWL. The full training time consists of three parts. The first part is the parameter communication time in the first group (e.g., the time interval T_1 in Fig. 4(c)). The communication in this part is between servers and workers. This part cannot be overlapped with computation since there is no parameter for any computation. The second part contains several groups. The time is the parameter communication time from the second group to the last group (e.g., the time interval T_2 to T_9 in Fig. 4(c)). In this part, the communication is between workers. The communication in one group is independent of the computation in its previous group so that the communication can be overlapped with the computation. The third part is the forward computation time in the last group (e.g., the time interval T_{10} in Fig. 4(c)). The computation in this part cannot be overlapped with any communication since there is no communication in the part. The three parts of the full training time are notated as E_1^i, E_2^i and E_3^i. Based on the notations. The three parts of worker i are formulated as:

$$\begin{aligned} E_1^i &= T_{tp}^1 = \frac{P_1}{B_{i,server}} \\ E_2^i &= \sum_{j=1}^{d-1} T_{tp}^j = \sum_{j=1}^{d-1} \frac{P_j}{B_{i,k}}, k \neq server \\ E_3^i &= T_{fw}^d \end{aligned} \tag{10}$$

We also have

$$T_{fw}^i \leq T_{tp}^i \quad 1 < i < g \tag{11}$$

Constraint Eq. (11) ensures the right execution sequences of the parameter communication in partition $k+1$ and the forward computation in partition k.

We have the overall training time E:

$$\begin{aligned} E &= \sum_{i=1}^{W} E_i \\ &= \sum_{i=1}^{W} (E_1^i + E_2^i + E_3^i + E_0^i) \end{aligned} \tag{12}$$

4.1.4. The problem

The goal of the problem in this paper is to minimize the overall training time of nonlinear neural networks, given a nonlinear neural network N_{nl}'s model architecture, the layer-wise computation time of N_{nl}, the rack architecture, and the inter-rack and intra-rack bandwidth. Hence, the overall training time E is the objective function. The decision variables are the packed set of branches of all nonlinear base networks $\mathcal{B}^i$, and the deployment setting set $\mathcal{D}$. In summary, we have the following **Model Distribution Scheduling (MDS)** problem:

Problem 1 *(MDS).* Given N_{nl}, R, n, $\mathcal{B}$, B_{Intra} and B_{Inter}, determine $\mathcal{D}$ and $\mathcal{G}$, subject to constraints (6)(8)(9)(11), to minimize E.

4.2. Problem analysis

Theorem 1. *MDS Problem is NP-hard.*

Proof 1. We provide the sketch of the proof. We prove it by reducing the minimum weighted set cover problem, which is known to be NP-complete. By minimizing the training time E, we must find a group partition where the weighted subsets of E have a relatively close summation, which will make the overlap fully. As a result, the MDS problem is equivalent to an optimal weighted set cover problem: to select a minimum number of sets from $\{E_1, E_2, \dots, E_A\}$ that covers all elements in the E.

MDS is NP-hard, it is unrealistic to find a globally optimal solution within polynomial time. We can divide the MDS problem into two sub-problems: the Grouping Problem and the Deploy problem:

Problem 2 *(Grouping).* Given N_{nl}, R, n, B, B_{Intra} and B_{Inter}, group the different branches of $N_{nonlinear}$, making the size of different groups as equal as possible.

Problem 3 *(Deploying).* Given $\mathcal{G}$, R, n, B, B_{Intra} and B_{Inter}, generate $\mathcal{D}$ to minimize E.

Accordingly, we develop two sub-algorithms, specifically, a Grouping Algorithm and a Deploying Algorithm, to solve the above two sub-problems, respectively.

4.3. Branches grouping operation

We consider one of the nonlinear base networks $\mathcal{N}_r$. The Grouping Algorithm must find a proper group set $\mathcal{G}_r$ that every element of $\mathcal{G}_r$ takes a similar communication time, hence different workers can get the same communication time to minimize the synchronous waiting time. In other words, for a nonlinear base networks $\mathcal{N}_r$ and its parameter set $\mathcal{D}^r = \{d_1^r, d_2^r, \dots, d_{M_r}^r\}$, we should find a group set $\mathcal{G}_r = \{\mathcal{G}_1^r, \mathcal{G}_2^r, \dots, \mathcal{G}_{D_r}^r\}$. Every group in $\mathcal{G}_r$ will have a similar size. We will describe how to decide the number of groups and deploy models in different workers to accelerate training next.

Algorithm 1 The Grouping Algorithm.

```
1: Input: a nonlinear base networks $\mathcal{N}_r$; The parameter set of $\mathcal{N}_r$: $\mathcal{D}^r$, the size
   of $\mathcal{D}^r$: $M_r$.
2: Output: All possible grouped branches set $\mathcal{G}$
3: Sort($\mathcal{D}^r$)                                   ▷ sort $\mathcal{D}^r$ in descending order
4: for $i = 1, 2, ..., M_r$ do
5:     Initialize $\mathcal{G}$
6:     for $j = 1, 2, ..., M_r$ do
7:         Put the $j^{th}$ element in $\mathcal{D}^r$: $d_j^r$ into the smallest group in $\mathcal{G}$
8:         Recompute the size of elements in $\mathcal{G}$
```

The grouping algorithm is a greedy algorithm that always puts the largest branch into the smallest group. In the MDS scenario, we consider it to be more intuitive and efficient compared to other search algorithms. For instance, ant colony systems have high complexity, genetic algorithms require more prior knowledge, and M-GRASP incurs greater overhead due to its iterative nature.

Given the parameter set $\mathcal{D}^r$ and the desirable number of groups D_r, it first sorts the parameter set D^r (line 3). Sort the parameter set D^r, which has a time complexity of $O(M_r \log M_r)$. Then, for every possible D_r, we initialize the $\mathcal{G}$ as D_r empty sets (line 5). The inner loop (line 6) iterates through the current groups (at most M_r groups) including the external loop, the total time complexity is $O(M_r^2)$. We try to put the largest elements in D^r into the set with the smallest sum of elements in $\mathcal{G}$ (line 7) and recompute the size of the element in $\mathcal{G}$ for the next iteration (line 8). The algorithm ends when all branches are grouped. Thus, the overall time complexity of Algorithm 1 is $O(M_r^2 + M_r \log M_r)$ dominated by $O(M_r^2)$. Moreover, the memory required to store the group G is proportional to the number of parameters M_r, so the space complexity is $O(M_r)$.

Theorem 2. *The approximation ratio of the grouping algorithm is* $\frac{3}{2} - \frac{1}{2D_r}$.

Proof 2. We denote OPT as the maximal group size of the optimal grouped set. Based on pigeonhole principle that all M_r branches must belong to D_r groups, we have two claims that: 1) $OPT \geq \sum_{i=1}^{M_r} D_i^r / D_r$, which indicates that all grouping algorithms are not better than OPT; and 2) Suppose sorted set D^r and $i > j$, which indicates that if set is sorted, at least two branches will be grouped in one group. Then, $OPT \geq d_{D_r}^r + d_{D_r+1}^r$. Let L be the largest group size in $\mathcal{G}$ and d_k^r be the last branch assigned to the largest group $\mathcal{G}_{k*}^r$. Since the grouping algorithm assigns d_k^r to $\mathcal{G}_{k*}^r$, all groups must be assigned at least $L - d_k^r$ parameters. Hence, we have $(\sum_{i=1}^{M_r} d_i^r) - d_k^r \geq D^r(L - d_k^r)$, which implies $L - d_i^r \leq \frac{(\sum_{i=1}^{M_r} d_i^r) - d_k^r}{D_r} = \frac{(\sum_{i=1}^{M_r} d_i^r)}{D_r} + d_k^r(1 - \frac{1}{D_r}) \geq OPT + \frac{1}{2}OPT(1 - \frac{1}{D_r}) = OPT(\frac{3}{2} - \frac{1}{2D_r})$.

4.4. OWL deploying algorithm

We need to decide the group size and deploy the model parameters on workers. We consider the nonlinear base networks $\mathcal{N}_r$. To calculate the full training time of different group sizes D_r, we should first decide on the deployment setting. Decentralized and centralized deployment are shown in Fig. 6. Centralized deployment allows peer-to-peer communication to accelerate parameter communication. The process will be: (1) Central workers pull parameters from the server. In one rack, different Central workers pull different parts of parameters. (2) Central workers do *Broadcast* that pushes parameters to workers in their communication group. (3) Different communication groups do *AllGather* that share their updated parameters with other communication groups. At the same time, workers can do forward computation based on updated parameters from the server. During this process, workers overlap computation and communication with other workers. (4) Workers do forward computation based on parameters from other workers. Centralized deployment utilizes the intra-rack bandwidth to accelerate parameter communication. Decentralized deployment preserves every link between workers and servers. Each worker pulls parameters from the server and overlaps communication and computation with other workers. To calculate the training time of different deployment settings, we model the communication process.

We first consider the centralized setting. The size of parameters to be pulled from the server will be the full parameters: $\sum_{i=1}^{M_r} d_i^r$. After that, the remaining parameters $n \cdot \frac{\sum_{i=1}^{M_r} d_i^r}{D_r} - \sum_{i=1}^{M_r} d_i^r$ will be pulled from other workers. To simplify the calculation, we make an assumption that the bandwidths between racks are the same which is notated as B_{inter}. Based on the notation proposed in section 4.1, we can formulate the training time in a centralized setting:

$$t_{ce} = \frac{\sum_{i=1}^{M_r} d_i^r}{B_{Inter}} + \frac{n \cdot \frac{\sum_{i=1}^{M_r} d_i^r}{D_r} - \sum_{i=1}^{M_r} d_i^r}{B_{Intra}}, M_r \leq n \tag{13}$$

As for the decentralized setting, each worker will pull a group from the server. So $n \cdot \sum_{i=1}^{M_r} d_i^r / M_r$ parameters are transferred. We can formulate the training time in the decentralized setting:

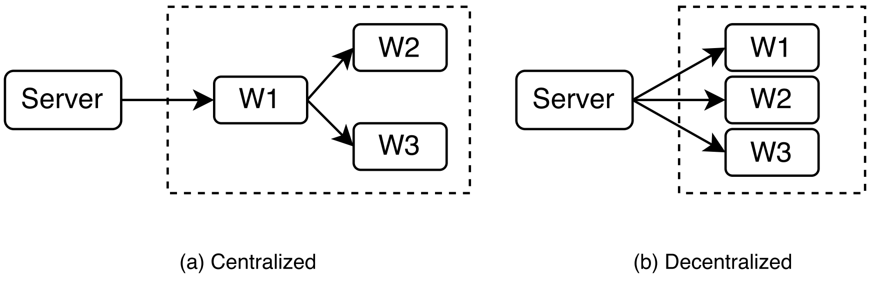


Fig. 6. Centralized and decentralized parameter communication over rack server.

$$t_{de} = \frac{n \cdot \sum_{i=1}^{M_r} d_i^r}{B_{inter} \cdot M_r} \tag{14}$$

To find the fastest way for parameter communication, we compute $\frac{t_{ce}}{t_{de}}$:

$$\frac{t_{ce}}{t_{de}} = \frac{B_{Inter}}{B_{Intra}} + \frac{D_r}{n} \cdot (1 - \frac{B_{Inter}}{B_{Intra}}) \tag{15}$$

We denote $\frac{B_{Inter}}{B_{Intra}}$ as α to represent the bandwidth decay inter racks. Note that in our assumption, $B_{Inter} \leq B_{Intra}$. So it is easy to find that $t_{ce}/t_{de} \leq 1$. Based on the analysis above, we will choose the centralized setting because it is faster. To decide the size of the group D_r and the deployment settings, we design a Deploying Algorithm.

Algorithm 2 The Deploying Algorithm.

```
1: Input: the number of GPUs $n$, parameter set of $\mathcal{N}_r$: $\mathcal{D}^r$;
2: Output: deploy setting
3: Grouping
4: for $i = 1, 2, ..., n$ do
5:     Part $n$ GPUs in $i$ communication groups equally.
6:     Train one minibatch                    ▷ Train one iteration
7:     Profile                    ▷ Training time for every possible $D_r$
8: Set $D_r$ as the value that has the smallest training time
9: Generate deploy setting
```

The deploying algorithm (see Algorithm 2) tries to find the best deployment setting for workers. At the beginning of training, workers need to run a grouping algorithm for all possible values of D_r (line 3). Then for every possible group, we part the GPUs inside a rack (line 5). For every possible D_r, the algorithm runs once (line 6) and profiles the training time (line 6). Then we set D_r as the value with the smallest training time (line 7) and generate the deployment setting. Note that the model size in the training process is not changed, and bandwidth changes can be ignored in our private cloud environment. Therefore, the algorithm runs only once at the beginning of the training process. The training and profiling operations involve running one iteration of model training, and the complexity depends on the specific model structure and size. In a simplified case, assume that each possible D_r undergoes one training and profiling. Thus, the time complexity of Algorithm 2 is approximately $O(n \cdot M_r^2)$, since the grouping algorithm is called in each iteration. Space Complexity which the memory usage includes storing the parameters for each GPU and the different deployment settings. The memory consumption mainly depends on the parameter set size and the number of GPUs, so the space complexity is $O(n \cdot M_r)$

5. Implementation

We implement the OWL system. The system is deployed on a one-hundred-node GPU cluster. We extend Pytorch RPC APIs to link workers and accelerate communication.

Communication Backbone: We implement OWL using TensorPipe backbone which addresses some of the limitations of Gloo. Compared with Gloo, asynchronous communication and different transports like TCP, NVLink and InfiniBand are supported. Distributed RPC framework provides lots of APIs for us to extend. The machine only transmits references (*called RRef*) to remote objects without copying the real data around. By doing that, system construction will be easy, and memory can be greatly saved.

Servers: Servers are responsible for reducing gradients from workers and broadcasting updated parameters. So **Group Generator** in server holds all *RPC RRef* of workers. Every time Servers need to broadcast parameters, servers try to make a non-blocking RPC call by *torch.distributed.rpc.rpc_async()*. The asynchronous call will generate threads background, and RPC messages are sent in parallel to different workers. We implement *The Grouping Algorithm* in **Branch Group Generator**. **System Monitor** extends *torch.profiler* library to profile *Training Information*. The background thread created by **System Monitor** is blocked until **Rack State Monitor** sends messages of the full system.

Workers: We design two threads to implement the overlap on workers in **Deploy Executor**. One thread is responsible for computation based on partial parameters. Another thread is responsible for communication, which makes RPC calls to run the collect function on the server. The alternating access of the model makes overlap between nodes. Besides, a semaphore locks the parameters to avoid parameters being accessed when updates and communication tokens of deployment decisions are needed. When the communication thread created by **Deploy Executor** begins to receive part of parameters from **Group Deployer**, semaphore locks the model, lest a crash happen. **Rack State Monitor** also extends *torch.profiler* library to profile the exact training time of each layer.

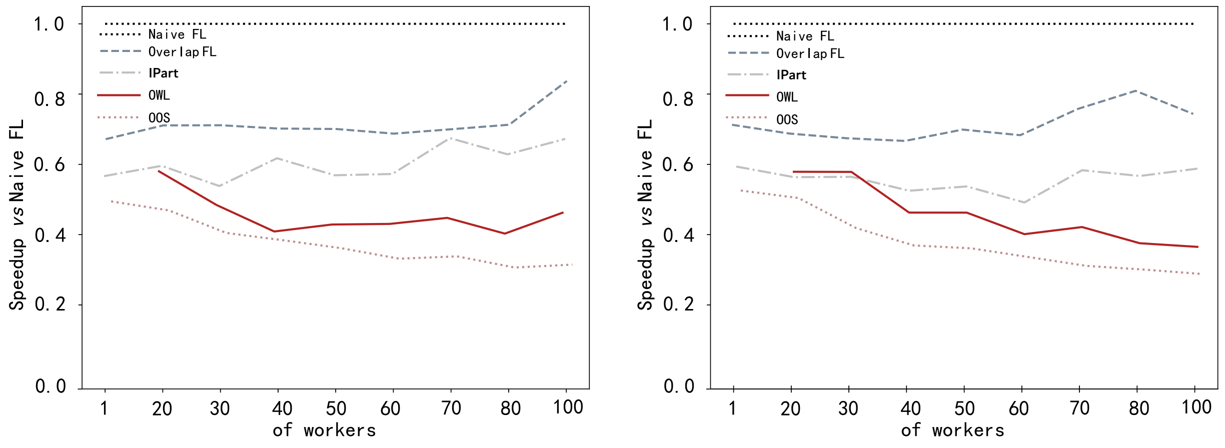
6. Evaluation

In this section, we evaluate the performance of OWL with the aim to answer the following questions:

- How does OWL compare to existing federated learning systems for modern DNNs with nonlinear characteristics? (§6.2)
- How do the external factors, e.g., the value of α and the device settings, affect the performance? (§6.3)
- How do the internal factors, e.g., the grouping algorithm and the deploying algorithm, affect the performance? (§6.4)
- To what extent can OWL scale? (§6.5)

6.1. Methodology

Testbed. We conduct experiments on a GPU cluster. The cluster contains 100 worker candidates, equipped with one GeForce RTX-3080 GPU with 10GB VRAM, 3.70 GHz i9-10900X Intel CPU, 64GB RAM, and 10Gbps Ethernet. We set up an additional one with the same configuration as the server. All machines are connected with one HUAWEI Ethernet switch S1700-24GR and run Ubuntu Server 20.04 with Linux kernel 4.4.6. To achieve high performance, machines use Cuda 11.1 library in the forward and backward computation. By default, the device setting contains 2 GPU*50 racks, and α is 0.5.



(a) Normalized training time for **DFN** (b) Normalized training time for **Inception V4**

Fig. 7. Comparing OWL with existing schemes on training time.

Neural Network Models and Dataset. For the neural network models, we employ two well-known neural networks: 1) DFN [38], a simple convolutional neural network designed to test different fusion of networks with 25.41M parameters; and 2) Inception V4 [33], an improved version of GoogLeNet [41] designed to extract information more efficiently with 42.68M parameters. Our experiments focus on the image classification applications where deep learning is most successfully applied. We adopt a well-known image classification dataset CIFAR10 [20], which contains 50K training images of 10 classes.

Baseline. We compare OWL against three state-of-the-art distributed neural network training systems and an offline optimal scheme serving as the performance upper bound: 1) **Naive** [16], naive federated learning(FL) architecture introduces a federated learning architecture that makes a naive data-parallel implement; 2) **Overlap**, default overlap FL [42] optimizes Naive by introducing a layer by layer overlap with communication and computation; 3) **IPart** [43], a layer packed overlap FL takes up set up time into consideration and packs some layers to make a full overlap; and 4) **OOS**, the offline optimal scheme is computed using dynamic programming with complete workload and bandwidth information. It outputs the optimal overlapping strategy. The offline optimal serves as an upper bound on the performance of training speed of an omniscient policy with complete knowledge of the future workload.

Evaluation Metrics. We evaluate the performance of the five communication approaches mentioned above on training time. We define a normalized training time (denoted as $\overline{\mathrm{T}}(x)$) by computing the ratio between the measured training time (denoted as $\mathrm{T}(x)$) to the measured training time of Naive FL (denoted as $\widehat{\mathrm{T}}$), i.e., $\overline{\mathrm{T}}(x) = \frac{\mathrm{T}(x)}{\widehat{\mathrm{T}}}$. This metric indicates how close the system x approximates the optimal performance. By normalizing the training time, we can enhance comparability and draw conclusions easily.

6.2. Overall performance

In order to answer the question of how OWL performs compared with the baselines (Naive, Overlap, and IPart), we investigate the overall performance that aggregates a combination of different factors. Then we zoom in on the individual factors to explain the performance improvements in detail.

Fig. 7 shows the speedup of DFN and Inception v4 network workloads running on different numbers of workers relative to the Naive FL baseline (set to 1 always). OOS represents the complete overlap of communication and computation (without communication delay, one is almost a subset of the other) to illustrate the ideal situation. This setup makes the performance of our OWL more intuitive. Please note that our comparison of speedup is conducted under consistent training throughput, without affecting accuracy, to ensure clearer results. We can see that the OWL outperforms Naive FL, Overlap FL and IPart methods on speedup by 63-69%, 27-30%, and 20-24% when the number of workers is 100, respectively. We can also observe that OWL is closer to the ideal situation which the gap is within 10% of the OOS. Recall that the performance of OOS cannot be achieved in practice because the full overlap is achieved and the setup time of communication is hidden. This reveals little room exists for OWL algorithms without future knowledge to improve. These results validate the superiority of OWL on training speed compared with baseline. Interestingly, the gap between OWL and the baseline widens as the number of worker nodes increases and closer to OOS, indicating that the workers in OWL compensate for the server's communication pressure, thereby improving performance. Notably, this increasing trend is more pronounced in the Inception V4 workload shown in Fig. 7b compared to the DNF workload in Fig. 7a. This is due to the more branches of neural network topology of Inception V4 workload, which further highlights the superiority of our algorithm.

Fig. 8a shows the training time breakdown of baselines and OWL when the number of workers is 100. We can observe that the average communication time percentage of OWL is 12% which is significantly smaller than that of baseline. Fig. 8b shows the GPU memory utilization. We can observe that the average percentage of GPU memory used per unit time of OWL is 83% which is significantly higher than that of baseline. This illustrates where the performance gains of OWL are from, i.e., worker-assisted compensation for server communication delays and better achieving communication and computation overlap. At the same time, overlapping communication between workers further improves speed. Our experiment with the other models shows similar results.

6.3. Parameter sensitivity analysis

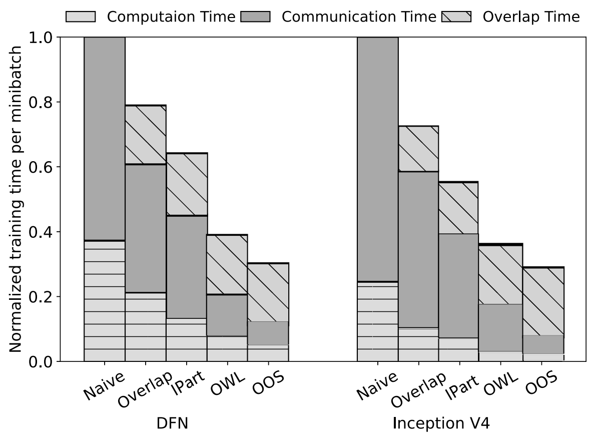
In this section, we explore OWL's external factors such as device setting and the α to better understand their impact on the performance of the system.

6.3.1. α sensitivity

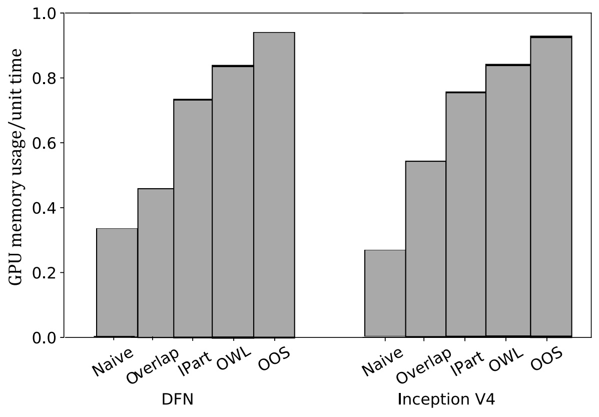
In this experiment, we evaluate the performance of OWL with different values of α. Fig. 9a and Fig. 9b show the normalized training speed of OWL running over different values of α compared with Ipart. There are two key takeaways from these results.

First, we see that OWL outperforms Ipart across all values of α, with a 16.7%-29% training time reduction, respectively. These results illustrate the steady high performance of OWL compared to Ipart methods.

Second, we can find that the gap in the training time between OWL and Ipart widens as the value of α decreases. For instance, the perfor-

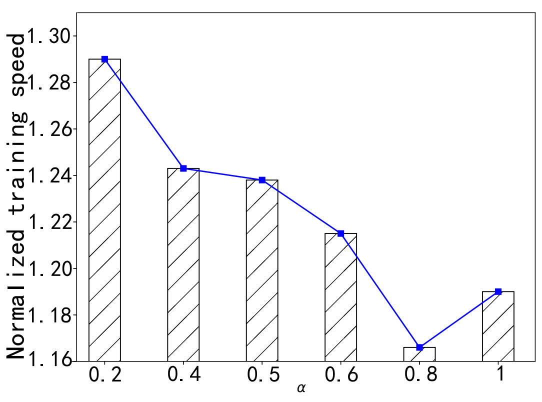


(a) Training time breakdown.

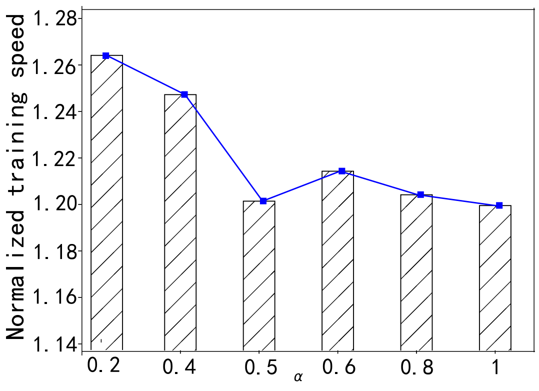


(b) GPU memory utilization.

Fig. 8. Time and memory usage analysis.



(a) α analysis over DFN.



(b) α analysis over Inception V4.

Fig. 9. The normalized training time over different values of α.

mance gap of Ipart is within 19% of our OWL when α equals 1, while the gap becomes 29% when α equals 0.2 shown in Fig. 9b. Fig. 9a show similar trends with a relatively smaller gap. This is because when the value of α decreases, the gap between intra-rack bandwidth and inter-rack bandwidth becomes larger. Our method utilizes the fast intra-rack bandwidth so has a better performance.

6.3.2. *Device setting sensitivity*

Next, we evaluate the performance of Naive FL, Overlap FL, IPart, and OWL under different device settings.

Fig. 10a shows the normalized training speedup of four methods under different device settings. The experiment results validate the superiority of OWL in terms of training speed again. We also can see that, as the number of devices increases, the training speedup of OWL increases. This is because the large size of the cluster makes the parameter communication a bottleneck while our method greatly utilizes spare bandwidth.

One more interesting observation is that when the size of the cluster is roughly the same, the larger number of racks, the smaller the training speed up. For example, compared to Naive FL, the average training speedup is 58% when there are 8 GPU * 10 racks and it becomes 43% when there are 4 GPU * 20 racks. This is because the total bandwidth inside a rack is fixed. If there are more GPUs inside a rack, the average communication time will be larger.

6.4. *Component analysis study*

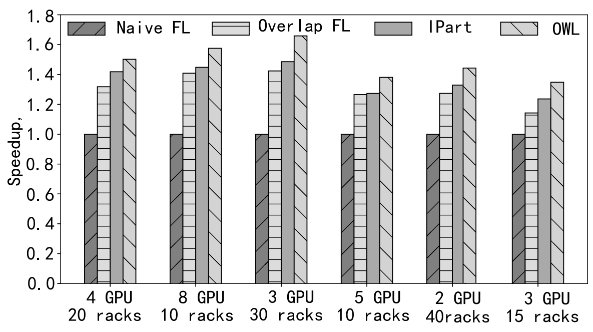
We further explore OWL's internal components to understand their contributions to system performance. We implemented two breakdown versions of OWL to take a closer look at the contribution of each component: 1) OWL-A has a grouping algorithm but does not enable a deploying algorithm, an unbalanced deploy setting is replaced; and 2) OWL-B has the deploying algorithm but does not enable grouping algorithm, a random grouping is replaced.

Fig. 10b shows the comparison results on inception V4 and DFS networks. As shown, the training time reduction brought by both OWL-A and OWL-B is significant. For example, in the inception V4 network, the average training time per minibatch of OWL, OWL-A, and OWL-B are 1, 1.09, and 1.14, respectively. It means that more than 14% of training time will be incurred by disabling the grouping algorithm, and 9% of training time will be incurred by disabling the deploying algorithm. We can observe a similar training time increase in DFS networks. These results indicate the importance of both the grouping algorithm and the deploying algorithm. OWL is able to combine the advantages to achieve better performance than using only one of them.

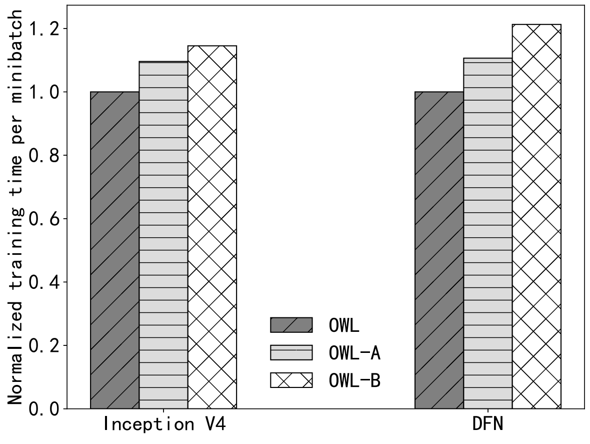
6.5. *Scalability*

Fig. 11a and Fig. 11b show the average training time per minibatch of inception V4 and DFN networks by experiments run over the different numbers of workers without any racks, respectively. We have the following two observations.

First, we can observe that, for both two neural network topologies, our proposed OWL outperforms all three baselines with a training time and the gap becomes larger as the worker number increases. Furthermore, OWL demonstrates a performance advantage over the suboptimal

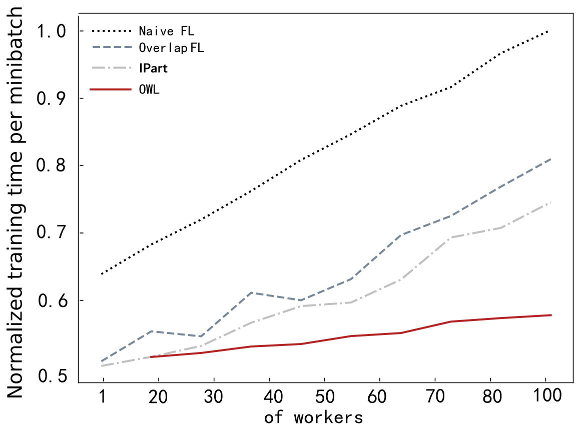


(a) Scalability analysis over Inception V4

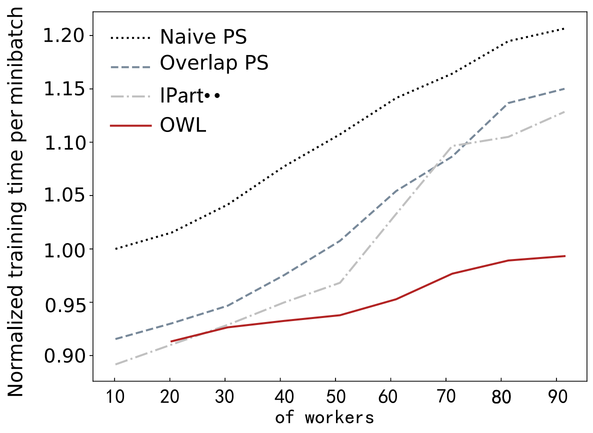


(b) Training time comparison for Component Analysis.

Fig. 10. Parameter sensitivity analysis.



(a) Scalability analysis over DFN



(b) Scalability analysis over Inception V4

Fig. 11. Training time comparison for scalability analysis.

IPart in complex neural network topologies. For instance, OWL outperforms IPart by 21% in the Inception V4 workload, while 15% in DFN workload when the number of workers is 100. Second, we can also see that the average training time per minibatch increases slowly as the number of workers increases. Specifically, in the Inception V4 workload, the normalized training time per minibatch is 0.60 when the number of workers is 50, and it becomes 0.63 when the number of workers is 100 with only 3% training time increasing. This indicates that our method is applicable to larger models and a greater number of workers, demonstrating excellent scalability.

7. Case study

We have developed a face-recognized punch clock for efficient time tracking in our laboratory. This clock utilizes advanced techniques to accurately recognize and localize individuals captured by the camera. The deployment process of this clock is illustrated in Fig. 12. Initially, we train the necessary models on our OWL system, optimizing them for high performance. Subsequently, these trained models are deployed on edge systems such as laptops and mobile phones. Once deployed, individuals can conveniently punch in and out using the face recognition capabilities of these models, ensuring accurate and efficient time tracking.

To enhance both the usability and accuracy of our model, we have chosen Inception V3 as the feature extractor and classifier for the collected face samples. To ensure compatibility and optimize training performance, we resize the pictures to a resolution of 224x224 and train the model on our 10-machine cluster. Both the Grouping Algorithm and Deploying Algorithm employed in OWL have linear time complexity, enabling rapid deployment decision-making in a matter of milliseconds, as demonstrated in the case study. Throughout the entire training process, these algorithms are executed only once, resulting in minimal additional overhead introduced by OWL. In comparison to the significant communication and training times, the overhead of OWL is negligible and can be disregarded. Moreover, our measurements indicate an average bandwidth usage of over 64.7%, indicating effective utilization of available bandwidth and substantial improvement in overall training efficiency. While the current training duration is relatively short due to limited data, the results presented in §6 suggest that OWL maintains an acceptable acceleration ratio as the data size increases, as depicted in Fig. 12. We successfully deploy the trained model on a laptop within our laboratory, and the rapid training system ensures ease of future updates and maintenance.

8. Related work and discussion

Federated Learning Towards the Communication Perspective. Federated Learning faces a significant challenge: communication inefficiency [17]. Many studies [44–46] have focused on data-parallel federated learning to improve training efficiency, such as by reducing the number of communication rounds and participating clients, or by re-

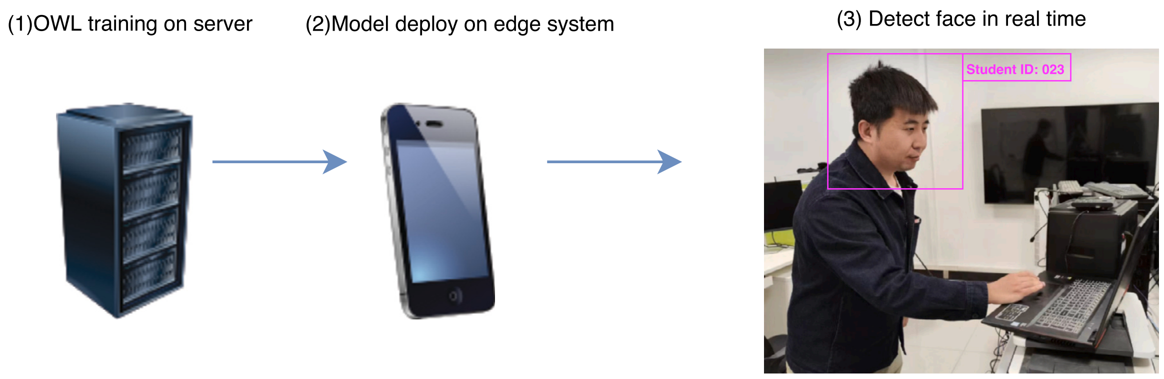

(1)OWL training on server
(2)Model deploy on edge system
(3) Detect face in real time
Student ID: 023

Fig. 12. The process of case study. (Photo has been informed and approved by the people in it and erased private information.)

ducing physical network burdens through techniques like sparsification and quantization.

APF [47] adaptively freezes and unfreezes parameters to reduce communication volume while preserving convergence. This method directly manipulates individual model parameters, which negatively affect the scaling efforts. FedRolex [48] uses a creative rolling extraction scheme to address the above issue more efficiently. Although demonstrating promising efficiency, width-based splitting schemes tend to result in very slim client models, which can significantly lose basic features, resulting in a drastic drop in model quality. FedPAQ [49] designs a periodic averaging and quantization method to address the communication bottleneck caused by a large number of devices uploading their local updates to server. Following that, Fedscale [50] enhances the theoretical guarantee of FedPAQ with a tuned server learning rate. However, this approach suffers from a trade-off between reducing the communication overhead and sacrificing the model performance. DFedAvgM [51] extends the centralized FedAvg [52] algorithm to a decentralized setting, where workers are connected by an undirected graph and communicate with their neighbors instead of a central server. This approach avoids the communication congestion caused by using the ALL-Reduce method on the server side, but it results in slower model convergence due to local gradient aggregation at the worker. OWL successfully combines the advantages of the aforementioned approaches by leveraging both worker-to-worker and worker-to-server communications to accelerate learning. Maoqiang Wu [53] proposed a differential privacy defense mechanism that injects noise into shared intermediate data to prevent attackers from recovering original data or inferring labels. The effectiveness of this approach was validated through experiments on a real satellite dataset. The results demonstrate that, with proper adjustment of the noise scale, a balance can be achieved between privacy protection and training performance. Future work will focus on enhancing the practicality of OWL by addressing privacy protection for parameters exchanged between workers.

Overlaps Computation and Communication. In distributed training, asynchronous background communication across stages and forward/backward computation make overlap between minibatches possible. Many works focus on the overlap computation and communication in PS [42,54] and other distributed architectures [55,36]. However, these works overlap forward or backward computation with parameter communication in one node, while our work focuses on the overlap between different nodes.

Pipeline Parallelism. Pipeline parallelism is also a popular way to accelerate training. Both forward [56] and backward [57] processes can be pipelined. Naïve pipelining can harm model convergence due to inconsistent weight versions between the forward and backward passes of a particular input [58]. To avoid this, existing techniques trade off memory footprint and throughput in different ways. Based on how a pipeline parallel system handles the synchronization of DNN parameters among input batches, the system falls into two categories: barrier synchronous parallel (BSP) systems and asynchronous parallel (ASP) systems. BSP systems (e.g., GPipe [59]) let a set of training input batches work on the same version of the model parameters, aggregate gradients computed by these iterations and enforce a barrier that stops the pipeline to apply the gradients to the model parameter. Gpipe splits minibatches into microbatches and performs forward passes followed by backward passes for these microbatches. GPipe maintains a single-weight version, but has periodic pipeline flushes where the pipeline is drained of inputs to update weights; these flushes limit overall throughput as resources are idle. ASP systems (e.g., PipeDream [60], XPipe [61] and PipeMare [62]) remove the sync barrier and let each input batch directly update the model parameters. PipeDream is another implementation of the model parallelism pipeline. PipeDream does not periodically flush the pipeline but stores multiple weight versions, which increases throughput but also increases the bandwidth bottleneck. Our approach is similar to the currently popular large-scale training techniques known as 3D parallelism [63,64], which combines data parallelism and pipeline parallelism in a hybrid training model. However, in our case, the pipeline transmits parameters.

Design for the specific DNN workload. Split-CNN [65] splits CNN layers into small patches and processes them independently before entering subsequent stages, which reduces data communication. It also includes a heterogeneous memory management system (HMMS) to utilize the memory-friendly properties of Split-CNN. Andrei et al. [66] target operation fusion between tensor operators and element-wise operators. They proposed a data movement algorithm designed for BERT. [67] accelerates the training of Transformer models by sub-graph parallelism that provides a significant performance improvement over pure data parallelism with a fixed number of resources. MPMoE [68] accelerates Mixture-of-Experts (MoE) training by designing a pipeline parallelism method that reduces communication latency through overlap with computation operations. These training optimizations all focus on performance bottlenecks for specific models. Sven [69] designs redundancy-free data organization and load-balancing partitioning strategies that mitigate redundant data communication for Temporal Graph Neural Network (TGNN).

9. Conclusion

This paper presents OWL, a novel design that exploits worker-assisted bandwidth in data-parallel federated learning to compensate for limited server bandwidth. OWL achieves superior performance by optimizing two dimensions: grouping branches of DNN topologies and deploying the generated groups on rack GPUs. We achieve 20%-69% speed up on different workloads. Moreover, we analyze the impact of the inter-rack bandwidth decay ratio on system performance. The result shows that both the grouping and the deploying algorithm are efficient. Compared to state-of-the-art approaches, extensive evaluations show that OWL significantly enhances scalability and accelerates training without affecting training throughput or model accuracy. Of course, our method relies on the nonlinear topological structure of modern DNNs. The more complex the DNN topology, the more effective our method is. Next, we will study the application of large models in this regard.

CRediT authorship contribution statement

Xiaoming Han: Writing – review & editing, Writing – original draft, Validation. **Boan Liu:** Writing – original draft, Methodology. **Chuang Hu:** Writing – review & editing, Supervision, Formal analysis. **Dazhao Cheng:** Visualization, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (2023YFE0205700), National Natural Science Foundation of China (62341410, 62302348), Science and Technology Development Fund, Macao S.A.R. (FDCT) project (0078/2023/AMJ), and General Program of Hubei Provincial Natural Science Foundation of China (2023AFB831).

Data availability

Data will be made available on request.

References

[1] J. Wang, A.F. Heimann, M. Tannast, G. Zheng, Ct-guided, unsupervised super-resolution reconstruction of single 3d magnetic resonance image, in: International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer, 2023, pp. 497–507.

[2] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, M. Pietikäinen, Deep learning for generic object detection: a survey, Int. J. Comput. Vis. 128 (2) (2020) 261–318.

[3] H. Zhang, H. Li, X. Liao, F. Li, S. Liu, L.M. Ni, L. Zhang, Da-bev: Depth aware bev transformer for 3d object detection, arXiv e-prints, arXiv:2302.13002, 2023.

[4] J. Yang, C. Fu, X.-Y. Liu, A. Walid, Recommendations in smart devices using federated tensor learning, IEEE Internet Things J. (2021).

[5] A. Nassar, Y. Yilmaz, Deep reinforcement learning for adaptive network slicing in 5 g for intelligent vehicular systems and smart cities, IEEE Internet Things J. 9 (1) (2021) 222–235.

[6] G. Abdelmoumin, D.B. Rawat, A. Rahman, On the performance of machine learning models for anomaly-based intelligent intrusion detection systems for the Internet of things, IEEE Internet Things J. 9 (6) (2021) 4280–4290.

[7] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A.M. Dai, A. Hauth, et al., Gemini: a family of highly capable multimodal models, arXiv preprint, arXiv:2312.11805, 2023.

[8] C. Li, J. Wang, Y. Zhang, K. Zhu, W. Hou, J. Lian, F. Luo, Q. Yang, X. Xie, Large language models understand and can be enhanced by emotional stimuli, arXiv preprint, arXiv:2307.11760, 2023.

[9] V. Sze, Y.-H. Chen, T.-J. Yang, J.S. Emer, Efficient processing of deep neural networks: a tutorial and survey, Proc. IEEE 105 (12) (2017) 2295–2329.

[10] Y. Li, B. Hu, H. Shi, W. Wang, L. Wang, M. Zhang, Visiongraph: leveraging large multimodal models for graph theory problems in visual context, arXiv preprint, arXiv:2405.04950, 2024.

[11] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, F. Yang, Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads, in: 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019, pp. 947–960.

[12] T. Brown, B. Mann, N. Ryder, M. Subbiah, J.D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Adv. Neural Inf. Process. Syst. 33 (2020) 1877–1901.

[13] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, et al., Efficient large-scale language model training on gpu clusters using megatron-lm, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–15.

[14] G. Greenleaf, Global convergence of data privacy standards and laws: Speaking notes for the European commission events on the launch of the general data protection regulation (gdpr) in Brussels & New Delhi, 25 May 2018, UNSW Law Research Paper 2018, pp. 18–56.

[15] Q. Yang, A. Huang, L. Fan, C.S. Chan, J.H. Lim, K.W. Ng, D.S. Ong, B. Li, Federated learning with privacy-preserving and model ip-right-protection, Mach. Intell. Res. 20 (1) (2023) 19–37.

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

[17] P. García Santaclara, A. Fernández Vilas, R.P. Díaz Redondo, Prototype of deployment of federated learning with iot devices, in: Proceedings of the 19th ACM International Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks, 2022, pp. 9–16.

[18] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, Y. Zou, Dorefa-net: training low bitwidth convolutional neural networks with low bitwidth gradients, arXiv preprint, arXiv:1606.06160, 2016.

[19] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., {TensorFlow}: a system for {Large-Scale} machine learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 265–283.

[20] A. Krizhevsky, G. Hinton, et al., Learning multiple layers of features from tiny images, 2009.

[21] A.F. Aji, K. Heafield, Sparse communication for distributed gradient descent, arXiv preprint, arXiv:1704.05021, 2017.

[22] Y. Lin, S. Han, H. Mao, Y. Wang, W.J. Dally, Deep gradient compression: reducing the communication bandwidth for distributed training, arXiv preprint, arXiv:1712.01887, 2017.

[23] C.-Y. Chen, J. Choi, D. Brand, A. Agrawal, W. Zhang, K. Gopalakrishnan, Adacomp: Adaptive residual gradient compression for data-parallel distributed training, in: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32, 2018.

[24] C. Wang, G. Zhang, R. Grosse, Picking winning tickets before training by preserving gradient flow, arXiv preprint, arXiv:2002.07376, 2020.

[25] N. Lee, T. Ajanthan, P.H. Torr, Snip: single-shot network pruning based on connection sensitivity, arXiv preprint, arXiv:1810.02340, 2018.

[26] C.-C. Huang, G. Jin, J. Li, Swapadvisor: pushing deep learning beyond the gpu memory limit via smart swapping, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 1341–1355.

[27] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F.L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al., Gpt-4 technical report, arXiv preprint arXiv:2303.08774, 2023.

[28] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X.V. Lin, et al., Opt: open pre-trained transformer language models, arXiv preprint, arXiv:2205.01068, 2022.

[29] J. Zhang, Y. Zheng, D. Qi, Deep spatio-temporal residual networks for citywide crowd flows prediction, in: Thirty-First AAAI Conference on Artificial Intelligence, 2017.

[30] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, A.L. Yuille, Deeplab: semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs, IEEE Trans. Pattern Anal. Mach. Intell. 40 (4) (2017) 834–848.

[31] Z. Zhang, C. Wu, Z. Li, Near-optimal topology-adaptive parameter synchronization in distributed dnn training, in: IEEE INFOCOM 2021 – IEEE Conference on Computer Communications, IEEE, 2021, pp. 1–10.

[32] S. Zhang, X. Yi, L. Diao, C. Wu, S. Wang, W. Lin, Expediting distributed dnn training with device topology-aware graph deployment, IEEE Trans. Parallel Distrib. Syst. 34 (4) (2023) 1281–1293.

[33] F. Chen, J. Wei, B. Xue, M. Zhang, Feature fusion and kernel selective in inception-v4 network, Appl. Soft Comput. 119 (2022) 108582.

[34] D. Yang, D. Cheng, Efficient gpu memory management for nonlinear dnns, in: Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, 2020, pp. 185–196.

[35] J. Sun, Z. Xu, D. Yang, V. Nath, W. Li, C. Zhao, D. Xu, Y. Chen, H.R. Roth, Communication-efficient vertical federated learning with limited overlapping samples, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2023, pp. 5203–5212.

[36] C. Chen, X. Li, Q. Zhu, J. Duan, P. Sun, X. Zhang, C. Yang, Centauri: enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 3, 2024, pp. 178–191.

[37] C. Jiang, Y. Tian, Z. Jia, S. Zheng, C. Wu, Y. Wang, Lancet: accelerating mixture-of-experts training via whole graph computation-communication overlapping, arXiv preprint, arXiv:2404.19429, 2024.

[38] C. Li, Y. Hou, W. Li, Z. Ding, P. Wang, Dfn: a deep fusion network for flexible single and multi-modal action recognition, Expert Syst. Appl. 245 (2024) 123145.

[39] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[40] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, P. Richtárik, Scaling distributed machine learning with {In-Network} aggregation, in: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), 2021, pp. 785–808.

[41] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1–9.

[42] J. Wang, H. Liang, G. Joshi, Overlap local-sgd: an algorithmic approach to hide communication delays in distributed sgd, in: ICASSP 2020 – 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2020, pp. 8871–8875.

[43] S. Wang, A. Pi, X. Zhou, J. Wang, C.-Z. Xu, Overlapping communication with computation in parameter server for scalable dl training, IEEE Trans. Parallel Distrib. Syst. 32 (9) (2021) 2144–2159.

[44] D. Jhunjhunwala, A. Gadhikar, G. Joshi, Y.C. Eldar, Adaptive quantization of model updates for communication-efficient federated learning, in: ICASSP 2021 – 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2021, pp. 3110–3114.

[45] G. Cheng, Z. Charles, Z. Garrett, K. Rush, Does federated dropout actually work?, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 3387–3395.

[46] M. Kim, S. Yu, S. Kim, S.-M. Moon, Depthfl: depthwise federated learning for heterogeneous clients, in: The Eleventh International Conference on Learning Representations, 2023.

[47] C. Chen, H. Xu, W. Wang, B. Li, B. Li, L. Chen, G. Zhang, Communication-efficient federated learning with adaptive parameter freezing, in: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), IEEE, 2021, pp. 1–11.

[48] S. Alam, L. Liu, M. Yan, M. Zhang, Fedrolex: model-heterogeneous federated learning with rolling sub-model extraction, Adv. Neural Inf. Process. Syst. 35 (2022) 29677–29690.

[49] A. Reisizadeh, A. Mokhtari, H. Hassani, A. Jadbabaie, R. Pedarsani, Fedpaq: a communication-efficient federated learning method with periodic averaging and quantization, in: International Conference on Artificial Intelligence and Statistics, PMLR, 2020, pp. 2021–2031.

[50] F. Haddadpour, M.M. Kamani, A. Mokhtari, M. Mahdavi, Federated learning with compression: unified analysis and sharp guarantees, in: International Conference on Artificial Intelligence and Statistics, PMLR, 2021, pp. 2350–2358.

[51] T. Sun, D. Li, B. Wang, Decentralized federated averaging, IEEE Trans. Pattern Anal. Mach. Intell. 45 (4) (2022) 4289–4301.

[52] B. McMahan, E. Moore, D. Ramage, S. Hampson, B.A. y Arcas, Communication-efficient learning of deep networks from decentralized data, in: Artificial Intelligence and Statistics, PMLR, 2017, pp. 1273–1282.

[53] M. Wu, G. Cheng, P. Li, R. Yu, Y. Wu, M. Pan, R. Lu, Split learning with differential privacy for integrated terrestrial and non-terrestrial networks, IEEE Wirel. Commun. 31 (3) (2024) 177–184, https://doi.org/10.1109/MWC.015.2200462.

[54] Y. Zhou, Q. Ye, J. Lv, Communication-efficient federated learning with compensated overlap-fedavg, IEEE Trans. Parallel Distrib. Syst. 33 (1) (2021) 192–205.

[55] S. Wang, J. Wei, A. Sabne, A. Davis, B. Ilbeyi, B. Hechtman, D. Chen, K.S. Murthy, M. Maggioni, Q. Zhang, et al., Overlap communication with dependent computation via decomposition in large deep learning models, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 1, 2022, pp. 93–106.

[56] X. Chen, A. Eversole, G. Li, D. Yu, F. Seide, Pipelined back-propagation for context-dependent deep neural networks, in: Thirteenth Annual Conference of the International Speech Communication Association, 2012.

[57] Z. Huo, B. Gu, H. Huang, et al., Decoupled parallel backpropagation with convergence guarantee, in: International Conference on Machine Learning, PMLR, 2018, pp. 2098–2106.

[58] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, M. Zaharia, Memory-efficient pipeline-parallel dnn training, in: International Conference on Machine Learning, PMLR, 2021, pp. 7937–7947.

[59] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q.V. Le, Y. Wu, et al., Gpipe: efficient training of giant neural networks using pipeline parallelism, Adv. Neural Inf. Process. Syst. 32 (2019) 103–112.

[60] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N.R. Devanur, G.R. Ganger, P.B. Gibbons, M. Zaharia, Pipedream: generalized pipeline parallelism for dnn training, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019, pp. 1–15.

[61] L. Guan, W. Yin, D. Li, X. Lu, Xpipe: efficient pipeline model parallelism for multi-gpu dnn training, arXiv preprint, arXiv:1911.04610, 2019.

[62] B. Yang, J. Zhang, J. Li, C. Ré, C. Aberger, C. De Sa, Pipemare: asynchronous pipeline parallel dnn training, Proc. Mach. Learn. Syst. 3 (2021).

[63] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, et al., Dapple: a pipelined data parallel approach for training large models, in: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2021, pp. 431–445.

[64] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E.P. Xing, et al., Alpa: automating inter- and {intra-operator} parallelism for distributed deep learning, in: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), 2022, pp. 559–578.

[65] T. Jin, S. Hong, Split-cnn: splitting window-based operations in convolutional neural networks for memory system optimization, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 835–847.

[66] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, T. Hoefler, Data movement is all you need: a case study on optimizing transformers, Proc. Mach. Learn. Syst. 3 (2021).

[67] A. Jain, T. Moon, T. Benson, H. Subramoni, S.A. Jacobs, D.K. Panda, B. Van Essen, Super: sub-graph parallelism for transformers, in: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2021, pp. 629–638.

[68] Z. Zhang, Y. Xia, H. Wang, D. Yang, C. Hu, X. Zhou, D. Cheng, Mpmoe: memory efficient moe for pre-trained models with adaptive pipeline parallelism, IEEE Trans. Parallel Distrib. Syst. (2024).

[69] H. Wang, D. Yang, X. Zhou, Y. Xia, Z. Zhang, D. Cheng, Redundancy-free high-performance dynamic gnn training with hierarchical pipeline parallelism, in: Proc. 32nd Int. Symp. High-Perform. Parallel Distrib. Comput, IEEE, 2023, pp. 17–30.

Xiaoming Han received the B.S. degree in software engineering from Inner Mongolia University in 2013 and the M.S. degree in computer technology from the Xiamen University in 2019. He is currently pursuing the Ph.D. degree with the Wuhan University. His research interests distributed/parallel computing, federated learning, and AI system.

Boan Liu received his B.S. degree from the South China University of Technology in 2020 and the M.S. degree in computer technology from the Wuhan University in 2023. He is currently pursuing the Ph.D. degree with the Hong Kong Polytechnic University. His research interests including the distributed machine learning system and large model training acceleration.

Chuang Hu received his B.S. and M.S. degrees from Wuhan University in 2013 and 2016. He received his PhD degree from the Hong Kong Polytechnic University in 2019. He is currently an Associate Researcher in the School of Computer Science at Wuhan University. His research interests include edge/federated learning, distributed/parallel computing, and Network.

Dazhao Cheng (Senior Member, IEEE) received his B.S. and M.S. degrees in electrical engineering from the Hefei University of Technology in 2006 and the University of Science and Technology of China in 2009. He received his PhD from the University of Colorado at Colorado Springs in 2016. He was an AP at the University of North Carolina at Charlotte in 2016-2020. He is currently a professor in the School of Computer Science at Wuhan University. His research interests include big data and cloud computing.