Accelerating Distributed SGD With Group Hybrid Parallelism

KYUNG-NO JOO, (Member, IEEE), AND CHAN-HYUN YOUN, (Senior Member, IEEE)
School of Electrical Engineering, Korea Advanced Institute of Science and Technology, Daejeon 34141, South Korea

Corresponding author: Chan-Hyun Youn (chyoun@kaist.ac.kr)

This work was supported in part by the U.S. Department of Commerce under Grant BS123456, in part by the Defense Challengeable Future Technology Program of the Agency for Defense Development, South Korea, and in part by the Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean Government, (Core Technology Research for Self-Improving Integrated Artificial Intelligence System), under Grant 20ZS1100.

ABSTRACT The scale of model parameters and datasets is rapidly growing for high accuracy in various areas. To train a large-scale deep neural network (DNN) model, a huge amount of computation and memory is required; therefore, a parallelization technique for training large-scale DNN models has attracted attention. A number of approaches have been proposed to parallelize large-scale DNN models, but these schemes lack scalability because of their long communication time and limited worker memory. They often sacrifice accuracy to reduce communication time. In this work, we proposed an efficient parallelism strategy named group hybrid parallelism (GHP) to minimize the training time without any accuracy loss. Two key ideas inspired our approach. First, grouping workers and training them by groups reduces unnecessary communication overhead among workers. It saves a huge amount of network resources in the course of training large-scale networks. Second, mixing data and model parallelism can reduce communication time and mitigate the worker memory issue. Data and model parallelism are complementary to each other so the training time can be enhanced when they are combined. We analyzed the training time model of the data and model parallelism, and based on the training time model, we demonstrated the heuristics that determine the parallelization strategy for minimizing training time. We evaluated group hybrid parallelism in comparison with existing parallelism schemes, and our experimental results show that group hybrid parallelism outperforms them.

INDEX TERMS Distributed training, scheduling, data parallelism, model parallelism, hybrid parallelism.

I. INTRODUCTION

Recently the deep-learning technique has received considerable attention for application in various areas such as medical imaging, space imaging, and VR/AR imaging. These areas often require high accuracy, so the scale and complexity of deep-learning models and the amount of training data are becoming immense [1]–[3]. Accordingly, distributed stochastic gradient descent (SGD) should be considered for training large-scale DNN models. However, scaling distributed SGD is challenging [4]. Large-scale DNN models have huge model parameters and huge amount of activation data, which leads to long communication delays and lack of device memory to store them [5]. Thus, the key challenges of scaling distributed SGD for large-scale DNN is how to reduce the long communication time, and how to resolve the device memory limitation.

Two well-known parallelisms of training distributed SGD are data parallelism and model parallelism, but both parallelisms have limitations for training large-scale DNN models in terms of scalability and accuracy [4], [6]–[8]. Data parallelism is a parallelism method of replicating model parameters to workers, and distributing data batches equally to workers [3]. It is frequently used for distributed training due to its high worker utilization and simplicity. However, data parallelism shows low scalability when training large-scale DNN models [3], [9]. Training data and model parameters cannot be stored in fixed worker memory when training a large-scale DNN model [10], [11]. Also, model parameters should be synchronized among workers at the end of training iteration, and it is a major bottleneck of distributed training [12], [13]. When we apply allreduce operation
to $N$ compute nodes, the synchronization time can be represented by $O(\log N)$ or $O(N)$ contingent on the network topology, while the computation time can be represented by $O(1/N)$ [14]. In practice, the synchronization time takes more than 90% of the total training time as the number of workers increases over 32, which leads to poor scalability [15]. Moreover, existing schemes about shortening the synchronization delay often sacrifice accuracy. Model parallelism is the parallelism method of partitioning the entire training model into submodels [3]. Model parallelism does not require synchronization of parameters, and it resolves the worker memory limitation issue, but it also provides low scalability because of low worker utilization and communication time of exchanging activation data. Because activation data increases as batch size increases, communication time should be reduced during training of a large-scale DNN. Additionally, other challenging issues in model parallelism is how to partition a large model and allocate workers to minimize training time [3]. Large-scale DNN models should be partitioned into multiple submodels, but this is not a trivial task because the number of partitioning cases increases as the number of layers increases. Currently, model partitioning and device placement are done by human experts or through the reinforcement-learning approach [16], [17]. Therefore, automated model-partitioning and worker-allocation schemes are needed to minimize the training time [8].

In this paper, we propose a fast and scalable parallelism method of distributed SGD called group hybrid parallelism (GHP) for training large-scale DNN models. The key idea is that dividing workers into groups reduces activation size, so it mitigates both the communication time and device memory limitation, which arise during the training of a large-scale DNN model. In addition, we focused on the fact that the existing parallelism method lacks scalability for different reasons, and the methods are complementary to each other. To reduce the training time, we utilized both the data and model parallelism methods and balanced them. Moreover, we propose a scheduling scheme that automatically partitions a large-scale training model into submodels and allocates workers to each submodel to minimize the training time in a distributed environment. To do that, we propose a mathematical model to estimate the training time of distributed training. Our model has a limitation in that it does not accurately predict the training time as it does not consider all the parameters for estimating the training time, but it reflects the trend of training time for given configuration and shows acceptable error (within 15~20% of the error range) so it can be used for scheduling.

We expect that our work can be applied to cluster management for distributed training because our work minimizes training time with fixed resources.

We make the following contributions in this paper:

- We present an accurate mathematical model of training time when data, model, or hybrid parallelism is applied.
- We analyze the behaviors of distributed training of DNNs, and categorize the training time by computation, communication, and synchronization time. We prove that our proposed model clearly estimates the training time (about 15% of MAPE value) by experiment.
- We propose and evaluate parallelism scheme for fast and scalable training of large-scale DNNs. Our scheme optimally balances the data and model parallelism to minimize the training time and groups workers for scalability. We compare our work with other solutions and find out that our work outperforms others in terms of scalability and throughput.

II. RELATED WORKS AND PROBLEM DESCRIPTION

In this section, we overview parallelism methods for training distributed SGD, and discuss their problems for large-scale DNN models in terms of scalability and accuracy. We also review model partitioning schemes for model parallelism.

A. TRAINING LARGE-SCALE DNN MODELS

The trend of DNN models has two directions. In some studies, model parameters are becoming smaller for memory saving and power efficiency with acceptable performance. In other studies, model parameters are becoming deeper and wider to achieve high accuracy [18]. In these cases, the scale of datasets is also becoming immense. In this work, we are concerned with the second scenario with large batch sizes. When the batch size increases, the hardware utilization also increases, and the number of iterations for training decreases, so the training time is accelerated [19]. However, a large batch size reduces accuracy so it should be mitigated [3].

To train the large-scale DNN model, distributed training with multiple GPUs has gained attention [4]. Many DL frameworks support distributed training to handle large-scale DNN models. However, distributed training is a challenging issue because it includes many issues such as parallelization, model consistency, communication, optimization, hyperparameter tuning, and so on. Moreover, computing resources can be shared among multiple users to reduce costs. In multi-tenant case, we should consider communication, worker scheduling, worker placement, and fairness to enhance job completion time and GPU cluster utilization. Krizhevsky [20] suggested linear scaling heuristics to increase the learning rate linearly as the batch size increases. In other words, if the training rate is set to $\eta$ when the training batch size $B$, then the training rate will be $k \eta$ when the training batch size increases to $kB$, while other parameters, such as weight decay, are maintained. Goyal [21] proposed a warmup method that enhances linear scaling heuristics to linearly increase the learning rate from zero to the target value $k \eta$ after 5 epochs. With these heuristics, it is reported that 8K and 5K batches were successfully trained for RESNet50 and RESNet101, respectively, without any accuracy loss [21].

On the other hand, Google recently presented the concept of federated learning and it has become the emerging paradigm for distributed training which enables global model training across decentralized devices with local data such as mobile devices and sensors [22], [23].
McMahan [23] presented the concept of FedAvg by aggregating the model parameters from local devices. Federated learning is communication-effective and secure as it does not send the local data to the central server, instead it sends model parameters at each epoch. Also, it can utilize each device’s on-device performance for training [24].

**B. PARALLELISM METHODS FOR TRAINING LARGE-SCALE DNNs**

There have been various attempts to train DNNs in distributed computing environments due to the increase of data and computation requirements. The Google Brain Project proposed data parallelism and model parallelism techniques for distributed learning [25], [26]. Each parallelism method has trade-offs, so we should determine which parallelism scheme is better considering hardware constraints. We will discuss the problem of data and model parallelism when applied to train a large-scale DNN model and summarize existing works about that issue.

Data parallelism is mainly utilized as a parallelism method due to its simplicity and high worker utilization. However, data parallelism has limitations for application to large-scale DNN models due to the long synchronization time of model parameters and limitation of minibatch size [8], [10]. There are two methods to solve this problem, namely, relaxation of synchronization conditions and compression of model parameters.

Relaxing synchronization condition is a method of reducing synchronization overhead by training with stale parameters rather than training strictly synchronous parameters. The existing data parallelism method mainly utilizes the allreduce-based synchronous SGD (S-SGD) method in which all workers train with the same parameters [3], Dean [26] proposed a parameter server-based asynchronous SGD (A-SGD) method that reduces the synchronization time by allowing workers to train with different parameters. A-SGD greatly reduces communication delay and enables fast convergence, but the staleness problem occurs as a result of training with stale parameters [29]. To solve this, a stale-synchronous technique has been proposed which penalizes workers training with stale parameters to limit the staleness effect [27], [28].

Compressing model parameters is a method of pruning model parameters to reduce the synchronization delay. Alistarh [31] proposed a gradient quantization method for expressing parameters in only a few bits. Lin [32] proposed a gradient compression scheme based on sparsification to reduce the parameter size. Parameter compression greatly reduces the gradient size and improves the communication but it sacrifices accuracy as well [31]. Several works [32], [33] reduced model parameter size without sacrificing accuracy for some environments. However, whether they can support general cases has not been proved yet. Moreover, reducing parameter size does not fundamentally solve the problems due to model complexity and huge computations. Model parameter compressing schemes are orthogonal to the scheduling scheme that we are concerned about so they can be applied simultaneously.

**Model parallelism** is not frequently used as a parallelism method because of its low worker utilization, long communication time, and the need for a model-partitioning methodology. However, model parallelism has recently attracted attention to solve the problem of data parallelism as the scale of training models has increased [3], [4]. In particular, pipeline parallelism, an advanced form of the model parallelism method that resolves the problems of low utilization and long communication time through pipelining, has been proposed. Pipeline parallelism divides the entire training model into “stages” with the same number of workers and assigns workers on each stage. By pipelining data, the communication time is reduced and worker utilization is increased. However, pipeline parallelism also lacks scalability. Because the number of partitions (stages) is equal to the number of workers, pipeline parallelism has limited scalability as the number of workers increases. Moreover, the communication time cannot be hidden by overlapping as the batch size increases. Also, some pipeline parallelism schemes sacrifice accuracy [15].

Harlap [15] presented PipeDream as an asynchronous pipeline parallelism framework. PipeDream partitions layers into multiple stages, where each stage contains sequential layers. Each stage is mapped to a separate worker for training. PipeDream proposes a one-forward-one-backward (1F1B) scheme, which runs feed-forward, and backpropagation of the previous run alternately and stores multiple versions of gradients. It greatly increases worker utilization because workers do not have to wait for backpropagation, and this reduces activation communication time because communication can be overlapped with computation time. However, PipeDream is not scalable for two reasons. First, the communication time is long for the training of large-scale networks. PipeDream resolves the communication issue by overlapping communication and computation time, so the training time is the maximum value of the total computation time and total communication time. However, for a large-scale DNN, the communication time is long compared to the computation time; therefore it cannot be hidden by overlapping. Second, because PipeDream stores multiple versions of gradients, it wastes worker memory so it cannot handle large networks and batch sizes. Also, PipeDream reduces accuracy because it suffers from the staleness issue. It trains previous parameters so the accuracy is deteriorated.

Huang [34] provided a synchronous pipeline parallelism library named GPipe. GPipe focuses on maintaining high accuracy while partially solving the low-utilization issue of model parallelism. GPipe partitions a model into multiple stages in a way that divides the load of the entire model evenly. Then, GPipe assigns different partitions to different workers. To increase worker utilization, GPipe divides minibatches into microbatches and trains the microbatches in a sequential order to enable seamless training and increase worker utilization without any staleness problem. At the end
of the iteration, GPipe synchronizes the gradients of micro-batches. However, GPipe still has idle time in the training process, which lowers the throughput. Actually, the throughput of GPipe is only 29% compared to PipeDream [15]. For a large-scale network, GPipe cannot partition the entire model into stages because the solution space is too large. Moreover, GPipe has a load-balancing issue among partitions; therefore, GPipe is not scalable. However, GPipe does not reduce accuracy while increasing worker utilization.

**Hybrid parallelism** is a parallelization method that utilizes both data and model parallelism because they are complementary to each other. Data parallelization alleviates the low worker efficiency and activation communication time, which are characteristics of model parallelism, and model parallelization resolves the long synchronization time problem. Therefore, when the two parallelization methods are optimally mixed, the training time and scalability will be enhanced compared to that of pure data or model parallelism.

Gholami [9] proposed an effective parallelism scheme that utilizes model, data, and domain parallelism. The key idea is to minimize the communication time by mixing model, data, and domain parallelism, which is the main bottleneck of the training procedure. It considers every spectrum between pure data parallelism and pure model parallelism with a precise mathematical model. Gholami showed that balancing data and model parallelism can reduce the training time for even large-scale networks. However, there is no exact way of finding the optimum balance of data and model parallelism.

Jia [35] proposed layerwise parallelism. It analyzes the tensor structure of each layer and finds the parallelization scheme most suitable for a specific layer. A cost model and a graph-partitioning algorithm are applied to find the optimal solution.

**C. WORKER ALLOCATION PROBLEM IN DISTRIBUTED DEEP LEARNING**

The worker allocation problem (or device allocation problem) in distributed deep learning concerns how workers are allocated to a training model through scheduling [4]. When data parallelism is applied, a training model can be trained by simply dividing the entire batch evenly among workers with the entire model parameters. However, as we discussed above, model or hybrid parallelism should be considered to train a large-scale deep learning model. To apply model or hybrid parallelism, a methodology for partitioning a model into multiple submodels must be established. In other words, the worker allocation problem in distributed deep learning is a matter of determining how to partition the model into submodels and determining which workers to place in each submodel. We summarize the existing solutions about worker allocation and model partitioning schemes in distributed deep learning.

A reinforcement learning-based (RL-based) model partitioning scheme automatically obtains a near-optimal solution of model partitioning and device allocation regardless of model characteristics. Mirhoseini [16] proposed a device placement method to determine the optimal model partition and device allocation that minimizes computation time based on reinforcement learning and policy gradient method in heterogeneous environments. The RL-based scheme provides a faster training time compared to human experts’ placement. However, these schemes need a pre-training procedure for scheduling and it is time and resource intensive. It is reported that pre-training takes about 12 to 27 hours for each training model. Also, it does not consider data parallelism or pipelining for acceleration so the hardware utilization is low. Moreover, the RL-based model partitioning method cannot be applied to models with a lot of computation, such as RNNLM and Inception-v3. To overcome this issue, Mirhoseini [16] subsequently introduces a hierarchical method to train more complex models.

Heuristics-based model partitioning methods are greatly influenced by network characteristics and scheduling techniques. They are empirical so they are easy to understand and implement, and they can consider various parallelism methods. Krizhevsky [20] and Wu [37] proposed a layer separation scheme named one weird trick (OWT) based on the smart insight that the convolution layer has the most computation, while the fully-connected layer has the most parameters. Their method decouples the convolution and fully-connected layer and places workers on each submodel in a data-parallel manner. This method is intuitive and easy to implement; however, it cannot be generally applied to any models and datasets because the dependencies of the model, network, and system hardware should be considered. Moreover, these schemes are not scalable when the dataset becomes large so they are not suitable for training large-scale models.

**D. CHALLENGES IN HYBRID PARALLELISM FOR TRAINING LARGE-SCALE DNN**

We present three challenging issues involved in training a large-scale DNN model by applying hybrid parallelism and summarize our approach to tackle these issues.

1) **TRAINING ACCELERATION THROUGH WORKER ALLOCATION**

It is important to balance data and model parallelism optimally as the training time greatly varies by the system configuration [9], [38]. Currently, there is no specific methodology for balancing them due to its large solution space and need for a model partitioning scheme. Therefore, most existing systems use only data parallelism as a parallelism method due to its simplicity. A scheduling algorithm that finds efficient model partition and worker allocation that minimizes the training time of hybrid parallelism is challenging.

2) **GUARANTEEING SCALABILITY WITHOUT ACCURACY LOSS**

As mentioned in Section 2.A, large-scale DNN models are used to achieve high accuracy. However, these models are limited due to long training time and worker memory problems. The methodologies for overcoming this, as described in Section 2.B, have limitations for training large-scale models.
or datasets, or even result in accuracy loss. The accuracy drop fades the meaning of a large-scale training model to increase accuracy. Therefore, there is a need for a training methodology that is capable of training large-scale DNN models, without any loss of accuracy.

3) FULLY AUTOMATED TRAINING FRAMEWORK
Most of the existing frameworks support data parallelism, and model or hybrid parallelism is partially supported. In addition, to use the above framework, the system configuration such as model partition and worker allocation must be manually set, and data propagation or synchronization between specific workers must also be manually implemented. In addition, the existing worker scheduling schemes often require additional information, such as the number of partitions, because of the large solution space. Therefore, a framework that enables automated training when given basic information, such as training model, batch size, and network configuration is needed.

In this work, we proposed a novel parallelism method to solve the above three issues. Our work targets scalability and fast training time so it works faster than other schemes even when the number of workers, batch size, model size is large. The main idea of our scheme is to aggregate workers into $R$ groups for scalability. Through aggregation, the activation data size is reduced by $R$ times so it can work faster even when the batch size is large. Also, each group trains the network by utilizing both the data and model parallelism for faster training.

As described earlier, existing worker allocation schemes do not provide a systematic analysis of determining the parallelization strategy of minimizing the training time, but they focus on reducing the communication time or computation time itself. Even more, these schemes often sacrifice accuracy. However, our proposed scheme focuses on balancing computation, communication, and synchronization time to minimize training time through worker allocation. To optimally balance the data and model parallelism, we provide a precise mathematical model for estimating the training time when the data, model, or hybrid parallelism is applied. Also, we provide the heuristics to solve the optimization problem of minimizing the training time acquired by the math model in a near-optimal manner. Existing schemes failed to solve the problem due to its large solution space, but our approach can determine the solution by approximation.

III. GROUP HYBRID PARALLELISM METHOD AND ITS TRAINING TIME MODEL DESCRIPTION
In this section, we propose a group hybrid parallelism method, describe its operation mechanism, and we present the training time model when group hybrid parallelism is applied. Group hybrid parallelism is an improvement of the existing hybrid parallelism. It exploits both the data parallelism and model parallelism simultaneously and divides the training workers into multiple groups. We discuss how the proposed parallelism approach accelerates training time and grants scalability in detail.

A. SYSTEM OVERVIEW
Fig. 1 shows the overall architecture of the proposed group hybrid parallelism. As seen in Fig. 1, the group hybrid parallelization basically places workers (yellow rectangle in Fig. 1) in a 2D grid form that considers both data parallelism (vertical part in the figure; data parallelism is applied for workers associated with the same submodel) and model parallelism (horizontal part in the figure; the entire training model is partitioned through model partitioning process). In other words, the entire training model is divided into $K$ sequential submodels, and all of the workers are divided into $R$ groups. Workers belonging to a group are mapped to one submodel; thus, $\{w_1, \ldots, w_K\}$ workers are assigned to train $K$ submodels in a single group, and $Rw_k$ workers are assigned to train the submodel $G_k$. Training is performed in a unit of groups, and each group trains independently with equally divided batches. When a training iteration is over, workers belonging to all groups synchronize their model parameters to maintain accuracy.

The training procedure of group hybrid parallelism is similar to that of model parallelism. It is depicted in the right side of Fig. 1 with an example when the number of submodels $K = 2$ and the worker allocation $w = 1, 2$. When input data arrives at the submodel, it is equally distributed to all workers associated with the submodel, and the results obtained from all workers are collected to form activation data. The collected activation data is then propagated to the next submodel and the same process is repeated for all submodels. The process of distributing and collecting data within a group is implemented by MPI scatter and MPI gather, and the communication time occurring in this process is called the activation communication time (blue arrow in text in Fig. 1). When each group completes one training iteration, a local gradient update of the group is formed for each submodel. The local gradient updates from different groups will be accumulated to form a global gradient update. Model parameters are synchronized among workers and thus all workers assigned to the same submodel always have a constant parameter value regardless of the group. In this procedure, a communication delay occurs and it is called synchronization time (red arrow and text in Fig. 1).

Group hybrid parallelism consists of a model partitioning step that divides the learning model into $K$ submodels and a worker allocation step that divides workers into $R$ groups and assigns workers belonging to the group to submodels. We describe the role of each step and why it is necessary for the training of large-scale DNN models. Each step will be further analyzed in Sections 4.2 and 4.3 to determine the model partition and worker allocation that minimize the training time.

Step 1: Model Partitioning Step: Model partitioning is the process of partitioning the entire training model with $M$ layers layerwise to form $K$ sequential submodels. As seen in Fig. 2, partitioning model parameters reduces synchronization time because synchronization time can be overlapped by different submodels; and thus model partitioning process
must be considered for training a large-scale DNN model. Also, memory constraints are relaxed because the number of model parameters is reduced, allowing workers to train with a larger batch size.

**Step 2: Worker Allocation Step** Worker allocation is a process of mapping workers to submodels, it is applied after the model partitioning process is completed. The worker mapping procedure divides $NP$ workers into $R$ equivalent groups, and it assigns the workers into $K$ submodels to determine worker allocation $w = \{w_1, \ldots, w_K\}$. Then, $R(w_1 + \cdots + w_K) = NP$ holds. By dividing workers into groups, the activation communication time can be drastically reduced. For example, consider the process of training the first sub-model in Fig. 1. When workers are divided into groups, $w_1$ workers simultaneously train the same network and form activation of size $D_1/R$. Meanwhile, if the workers are not divided into groups, $Rw_1$ workers simultaneously train the same network and form activation data of size $D_1$. Therefore, worker grouping is essential for training large-scale deep learning models.

**B. TRAINING TIME MODEL**

In order to estimate the training time, we analyze the training procedure when group hybrid parallelism is applied. In this subsection, we design a mathematical model for training time of group hybrid parallelism when partitioned submodels $G = \{G_1, \ldots, G_K\}$, the number of workers allocated to each submodel $w = \{w_1, \ldots, w_K\}$, batch size $B$, and the number of groups $R$ are given. To model the training time, we should analyze the training behavior of data and model parallelism.

**1) COMPUTATION TIME**

The computation time is the time it takes for workers to perform computation operations (feed-forward, and back-propagation) for training. It is calculated by dividing the computation workloads of the submodel by the sum of workers’ FLOPs associated with the submodel. We assume that when the batch size is sufficient ($\geq 32$), computation workloads assigned to a worker are evenly distributed across cores and threads so that the worker performance is close to its FLOPS. Let $C_{F}^{G}$, $C_{B}^{G}$ be the computation FLOPs of feedforward and backpropagation process in $G_k$. Then, we can calculate the
FIGURE 2. The comparison of (a) data parallelism, (b) model parallelism, and (c) group hybrid parallelism procedure in a Gantt chart form.

TABLE 1. Variables used in this work.

| Symbol | Meaning |
|--------|---------|
| G      | Training graph comprised of layers |
| B      | Training batch size |
| b      | Minimal batch size for a single worker |
| N      | Number of cluster nodes |
| P      | Number of workers in a single node |
| f      | Compute capability of a worker |
| m      | Memory size of a worker |
| R      | The number of submodels |
| Gk     | The kth submodel of G |
| Ck     | Computation amount of Gk |
| Mw     | Memory requirements of Gk |
| kw     | Parameter size of Gk |
| wk     | The number of workers allocated to Gk |
| xk     | The number of nodes allocated to Gk |
| φk, ∇φk | Model parameters and its gradients of Gk |
| α, αp  | Setup time between nodes and workers |
| β, βp  | Bandwidth between nodes and workers |
| τ, τp  | Computation constant for MPI allreduce |

computation time of Gk by (1).

\[ T_{k}^{\text{comp}} = T_{k}^{\text{comp,F}} + T_{k}^{\text{comp,B}} \approx \frac{C_{F}}{w_{k}f} + \frac{C_{B}}{w_{k}f} \]  

(1)

2) COMMUNICATION TIME
The communication time is a delay generated by exchanging activation or gradient data between adjacent submodels. It includes scattering input data to workers associated to the submodel and exchanging the activation or gradient data between adjacent submodels. The communication time is determined by the network bandwidth and the data amount. Let input data with size \( D_{k-1}/R \) arrive to \( x_{k} = \lceil w_{k}/P \rceil \) compute nodes associated with submodel \( G_{k} \), and it generates activation of data size \( D_{k}/R \). When \( w_{k} > 1 \), a delay occurs while distributing input data to workers, and gathering output activation to next submodel.

We separated the communication time into inter-node communication time and intra-node communication time. Inter-node communication time is applied when node-to-node communication is occurred. The communication time is modeled by Hockney’s algorithm [40], and we assumed all nodes in the same group are connected to each other. For inter-node communication, MPI scatter is utilized among nodes to scatter data to workers and it is modeled by MST or BKT algorithm [14]. Each worker’s execution result is directly transferred to the next submodel. Also, direct communication via PCI-e link is applied for intra-node communication. Putting up altogether, we can formulate communication time \( T_{k}^{\text{comm}} \) as (2) where \( \delta_{k} = D_{k-1} + D_{k} \).

\[ T_{k}^{\text{comm}} = T_{k}^{\text{comm, inter}} + T_{k}^{\text{comm, intra}} \approx \left(2 - \frac{1}{x_{k}}\right)\frac{\delta_{k}}{R} + \frac{2\delta_{k}}{w_{k}R} + \frac{w_{k}R}{\beta_{p}} \]

(2)

3) SYNCHRONIZATION TIME
The synchronization time is the delay caused by collecting weight gradients from workers and updating new parameters. The synchronization time is also comprised of inter-node synchronization time and intra-node synchronization time. Intra-node synchronization time is modeled by direct communication using the PCI-e link. Inter-node synchronization process is modeled by MPI Allreduce, which is already
modeled considering various network topologies and algorithms [41]. Synchronization time based on MPI Allreduce is a function of the number of nodes to synchronize, denote by \( p_k(w_k) \), should be clarified. It can be obtained by multiplying the number of groups \( R \) by the number of nodes which are allocated to the submodel in a single group. Equation (3) formulates the number of nodes to synchronize the submodel \( G_k \).

\[
p_k(w_k) = \begin{cases} N \land R, & 0 \leq w_k < P \\ 2R, & P \leq w_k < 2P \\ \vdots & \vdots \\ x_k R, & (x_k - 1)P \leq w_k < x_k P \end{cases}
\]

(3)

where \( \land, \lor \) denotes min, max operations respectively. The synchronization time can be described as (4) when Ring-AllReduce algorithm [41] is applied.

\[
T_{\text{sync}}^k = T_{\text{sync, inter}}(p_k) + T_{\text{sync, intra}}^k \\
\approx \left[ 2(p_k - 1)\alpha + 2\frac{p_k - 1}{p_k} n\beta + \frac{p_k - 1}{p_k} n\gamma \right] + F_k p_p 
\]

(4)

Note that the synchronization time is a form of increasing step function of \( w_k \) by interval \( P \), and thus we can assume synchronization time as constant over \( w_k \) for the partitioned domain. Also, the synchronization time model can be adaptively changed according to applied interconnection network or algorithm [41].

Using the mathematical model of computation, communication, synchronization time, we can model training time when data, model, and group hybrid parallelism is applied. As described in Fig. 2, the training time of the data parallelism is simply acquired by adding the computation time and the synchronization time. The training time of the model parallelism is acquired by adding the computation time and the communication time of each submodel. When the group hybrid parallelism is applied, the synchronization time is hidden in the overall training process so the synchronization time is counted as the overall training time only once throughout all of the submodels. The overall training time is determined based on the submodel with the latest parameter update (Submodel #1 in Fig. 2). As submodel \( i \) starts synchronization when it finishes the backpropagation process, submodel \( i \) finishes its parameter synchronization process at:

\[
T_i = \sum_{k=1}^{i-1} (T_{k}^{\text{comp,F}} + T_{k}^{\text{comm}}/2) + \sum_{k=1}^{K} (T_{k}^{\text{comp}} + T_{k}^{\text{comm}}) + T_{i}^{\text{sync}} \\
= \frac{y_1}{w_1} + \cdots + \frac{y_K}{w_K} + T_{i}^{\text{sync}} + L(i) 
\]

(5)

where \( y_k = \begin{cases} \frac{C_f}{I} + \frac{\delta_k \beta_p}{R} - \frac{\delta_k \beta}{R} \cdot \frac{w_k}{|P|}, & k < k_p \\ \frac{C_f + C^B}{I} + \frac{2\delta_k \beta_p}{R} - \frac{3\delta_k \beta}{R} \cdot \frac{w_k}{|P|}, & k \geq k_p \end{cases} \)

and \( L(i) = \sum_{k=1}^{i-1} \frac{2\delta_k \beta_p}{R} - \sum_{k=1}^{i-1} \frac{\delta_k \beta}{R} \cdot \frac{w_k}{|P|} \) which is constant over \( w_k \). Finally, the training time in hybrid parallelism can be expressed as (6) when all submodels finish their parameter update. Note that \( \lor \) denotes the maximum element, \( k_p \) is the submodel index that finishes the training most late. The synchronization time of submodel \( G_{k_p} \) is accounted for training time and we call the submodel \( G_{k_p} \) by sync-submodel.

\[
T = \bigvee_{i=1}^{K} T_i = T_{k_p} \\
where k_p = \arg \max_i T_i
\]

(6)

C. CONVERGENCE ANALYSIS OF GROUP HYBRID PARALLELISM

As depicted in Fig.1, the global gradient update is calculated by the local gradients update acquired by each group. Let the parameter of the submodel \( G_k \) submodel be \( \phi_k \), and local gradient of \( r \)th group after training \( G_k \) be \( \nabla \phi_k \). Then the model parameter at \( r \)th iteration can be formulated by (7):

\[
\phi_k^{(r)} = \phi_k^{(r-1)} - \eta_k \nabla \phi_k \forall k \in (1, \ldots, K) \\
where \nabla \phi_k = \frac{\nabla \phi_{k,1} + \cdots + \nabla \phi_{k,R}}{R}
\]

(7)

With the update rule in (7), the convergence rate of group hybrid parallelism is the same as that of conventional minibatch SGD using full batch size \( B \), which has a convergence rate of \( O(1/\sqrt{Bt} + 1/t) \) at iteration \( t \) [39]. This is because the gradients obtained by group hybrid parallelism has the same value as the gradients acquired by minibatch SGD with full batch \( B \) and it is proved in Proposition 1.

Proposition 1 (Convergence Analysis of GHP): The update rule formulated in (7) is equivalent to that of minibatch SGD with full batches.

Proof: Let \( f(x, y) \) be the loss function obtained by \( r \)th test data, and \( \eta_k \) be the learning rate at iteration \( t \). Then, the gradient update of the minibatch SGD with batch size \( B \) can be formulated as:

\[
\nabla \phi_k = \frac{\partial \text{Loss}}{\partial \phi_k} = \frac{\partial}{\partial \phi_k} \left[ \sum_{i \in B} f(x_i, y_i) \right] 
\]

(8)

When group hybrid parallelism is applied, data batch \( B \) are equally distributed among \( R \) groups.

\[
\nabla \phi_k = \frac{\partial}{\partial \phi_k} \left[ \frac{\sum_{i \in B} f(x_i, y_i)}{R} \right] = \frac{1}{R} \sum_{i \in B} \frac{\partial f(x_i, y_i)}{\partial \phi_k} \\
= \frac{1}{R} \frac{\partial}{\partial \phi_k} \left[ \frac{\sum_{i=1}^{B/R} f(x_i, y_i)}{R} + \frac{\partial}{\partial \phi_k} \left[ \frac{\sum_{i=1}^{2B/R} f(x_i, y_i)}{R} \right] \\
+ \cdots + \frac{\partial}{\partial \phi_k} \left[ \frac{\sum_{i=1}^{B/R+1} f(x_i, y_i)}{R} \right] \right] \\
= \frac{1}{R} \left( \nabla \phi_{k,1} + \cdots + \nabla \phi_{k,R} \right)
\]

(9)

Through Proposition 1, we can conclude that the group hybrid parallelism does not sacrifice accuracy because it has the same update rule to synchronous SGD.
IV. DETERMINING SYSTEM CONFIGURATION FOR GROUP HYBRID PARALLELISM

In section 3, we explained the concept of group hybrid parallelism and provided a mathematical model of the training procedure. To get benefit from group hybrid parallelism, the model partitioning and the worker allocation technique should be developed to determine a system configuration that can minimize training time among solution spaces. This section deals with the worker allocation problem to minimize training time when group hybrid parallelization is applied.

A. WORKER ALLOCATION MODEL FOR GROUP HYBRID PARALLELISM

We focus on training a large-scale DNN model with a large dataset by a distributed computing infrastructure. We are trying to train a DNN model with $M$ layers $\{L_1, \cdots, L_M\}$ with $N$ homogeneous computing nodes where each node has $P$ workers in it. Each worker has compute capability per second $f$ and memory size $m$. In other words, an idle computing node has the computing capability of $P \cdot f$ and memory size $P \cdot m$. To increase the utility of computing resources, a batch size of $b$ or more should be assigned to each worker. We ignore any faults and inferences that may occurred in the training process.

The objective is to minimize the training time formulated in (6). As discussed in Section 3, the training time can be formulated by model partitioning $G = \{G_1, G_2, \cdots, G_K\}$, number of groups $K$, and worker allocation to submodel in a single group $w = \{w_1, w_2, \cdots, w_K\}$. Thus, the training time can be minimized by adjusting them. Also, as described in section 3, we have three constraints: (1) the number of workers is restricted by $NP/R$ (Worker constraint), (2) assigned activation data and model parameters $M_k/w_k$ should not exceed the memory size of the single worker $m$ (Memory constraint), and (3) batch size for a single worker $B/Rw_k$ should be allocated more than $b$ (Batch constraint). Equation (10) and Constraint (11-13) shows the linear programming form of minimizing training time when group hybrid parallelism is applied.

\[
\text{minimize } T = \sum_{k=1}^{K} \frac{y_k}{w_k} + T_{\text{sync}}^{\text{sync}} \tag{10}
\]

subject to
\[
\sum_{k=1}^{K} w_k \leq \frac{NP}{R} \tag{11}
\]
\[
w_k \cdot m \geq M_k \tag{12}
\]
\[
w_k \cdot b \leq \frac{B}{R} \tag{13}
\]

where $y_k = \begin{cases} 
\frac{c_f}{R} + \frac{\delta_k f_p}{R} - \frac{\delta_k \beta}{R} \cdot \frac{w_k}{w_p}, & k < k_p \\
\frac{c_f}{R} + \frac{2\delta_k f_p}{R} - \frac{\delta_k \beta}{R} \cdot \frac{w_k}{w_p}, & k \geq k_p
\end{cases}
$

The above problem is a form of the open-shop scheduling problem [42], which is proven to be NP-hard. Because model partitioning and worker allocation should be considered simultaneously to determine the system configuration, the solution space is too large to find an optimal solution; therefore, we cannot solve the problem directly. Therefore, we propose heuristics to determine the system configuration to minimize training time.

Our scheme have two steps: Model Partitioning and Worker allocation. Model partitioning step divides the entire training model into $K$ submodels $G = \{G_1, \cdots, G_K\}$ that minimize the training time. Worker allocation step determines the worker allocation strategy $w = \{w_1, \cdots, w_K\}$ and the number of groups $R$ when the submodel set is given. It is comprised of four substeps and the result obtained by worker allocation step is proven to be near-optimal. We will explain each procedure in detail.

B. MODEL PARTITIONING

The model partitioning step is a process of dividing the entire training model layer-wise to form a set of submodels. A submodel is a set of one or more subsequent layers, and all layers must be included in one submodel. Model partitioning determines the parameter size of the submodel, so it has a great influence on determining the synchronization time. Moreover, model partitioning also determines the activation size between adjacent submodels so it has an important effect on determining the communication time.

Because the training behavior changes as $K$ changes, determining the number of submodels $K$ is an important issue for model partitioning procedure.

When $K = 1$, the submodel is equivalent to the entire training model, so the training is performed in the same way as data parallelism. Therefore, all of the model parameters must be synchronized, but there is no communication time to pass the activation. This case is useful when the number of model parameters or the number of workers is small.

When $K \geq 2$, both data parallelism and model parallelism are considered together so that both parameter synchronization and activation communication occur. If $K$ is equal to the number of workers $NP$, this is the case when pure model parallelism or pipeline parallelism is applied because only one worker should be placed in the submodel. When a model is divided into submodels only one submodel with the slowest synchronization (sync-submodel) is accounted for training time because each submodel independently performs parameter synchronization. However, communication time occurs for each adjacent point of the submodels; thus, worker efficiency is lowered as $K$ increases. Therefore, when $K > 2$, training tends to be inefficient [16].

As model partitioning plays an important role in determining the synchronization time and communication time, the proposed model partitioning scheme basically considers all possible partitioning cases by increasing $K$ from 1 to $(NP \land M)$. Then, we obtain the system configuration that minimizes the training time for each available model partition (this will be described in Section 4.3). At this time, the proposed method analyzes the trend of the minimum training time by changing the number of submodels, and then stops training at the elbow point. In other words, when the min-
imized training time increases as the number of submodels increase, the algorithm stops and returns the submodel set and system configuration which minimizes the training time. The time complexity of model partitioning can thus be expressed as $O(M^K)$. Note that $K$ usually stops at 2-3 in most cases.

In summary, the model partitioning procedure returns the submodel set that minimizes the training time considering all partitioning cases until the elbow point is found while increasing the number of submodels $K$.

C. WORKER ALLOCATION

The worker allocation step returns the number of groups $R$ and the worker allocation of a single group $w$ that minimizes the training time when the submodel $G$ is given. Therefore, we deal with the problem of finding the solution of the primal problem formulated in (10-13) when the submodel set $G$ is specified. In this procedure, we only focus on the number of workers $w = \{w_k\}$ to minimize the objective function. We treat $R$ as a constant and execute worker allocation procedure repeatedly by changing the value of $R$. Also, we treat the synchronization time $T_{sync}$ as constant by fixing the domain range of $w_k$.

To find the solution through optimization technique, the objective function in (10) must be convex. To make the objective function convex, we eliminate non-promising solutions to reduce the solution space and approximates the objective function to make it convex. After then, we apply an optimization technique to minimize the objective function while satisfying all conditions.

In worker allocation step, we have four substeps to solve the problem. The first three substeps reduce the solution space through pruning or approximation of the worker allocation problem and finally make the objective function convex. The final substep then determines the optimal solution of the given submodel $G$ and the number of groups $R$. We will explain each substep in detail.

1) SUBSTEP 1: PRUNING NONPROMISING SOLUTIONS FOR $y_k < 0$

When $y_k < 0$, the term $y_k/w_k$ increases when $w_k$ increases. Therefore, if one or more $y_k$ has negative value, the objective function $T(w) = \sum_{k=1}^{K} y_k/w_k + T_{Sync}$ cannot be convex. In this substep, we eliminate the negative $y_k$ value to make the objective function convex. When $y_k < 0$, the increased effect of the training time due to communication is much larger than any other reduction effects of training time as the number of workers increases. Thus, the number of workers allocated to the $y_k < 0$ submodel should be minimized to reduce the communication. When $y_k > 0$, the training time decreases as $w_k$ increases. Thus, we should balance workers only for submodels with positive $y_k$. This is summarized in Proposition 2 and it is proved through contradiction (see Appendix.).

Proposition 2: Let $w^* = \{w_k^*\}$ minimizes $T(w) = \sum_{k=1}^{K} y_k/w_k + T_{Sync}$ while satisfying (11)-(13).

Then, $w_k^* = \frac{M_k}{m}$ for all $k$ if $y_k < 0$.

With Proposition 2, we can remove cases with $y_k < 0$, and the problem is equivalent to the same problem for additional constraint:

$$y_k \geq 0$$

2) SUBSTEP 2: FINDING SOLUTION FOR SYNC-SUBMODEL

In this substep, we determine the number of workers for sync-submodel $w_k$, when $y_k \geq 0$, to make the problem more simple. As described in Section 3.2, the synchronization time is a form of increasing step function of $w_k$, with interval $P$ and we treat $T_{Sync}$ as a constant by dividing the domain range of $w_k$. Let $w_k^*$ be the number of workers for sync-submodel that minimizes training time subject to (11)-(14). Then, we can determine the range of workers for sync-submodel and it is described in Proposition 3. Detailed proof of Proposition 3 is described in Appendix A.

Proposition 3: Let $w^* = \{w_k^*\}$ minimizes $T(w) = \sum_{k=1}^{K} y_k/w_k + T_{Sync}$ while satisfying (11)-(14).

Then, $\left[\frac{M_k}{m}\right] \leq w_k^* \leq \arg \min \left(\frac{y_k}{w_k} + T_{Sync}\right)$ holds.

With Proposition 3, we can find the possible candidates of $w_k^*$, and we choose one value among them which minimizes the training time.

3) SUBSTEP 3: DETERMINING THE NUMBER OF NODES

After Step 1 and 2, the minimization problem described in Section 4.1 is equivalent to minimizing $T(w) = \sum_{k=1}^{K} y_k/w_k$, while satisfying $y_k \geq 0$, $\sum_{k=1}^{K} w_k = W$, and $M_k/m \leq w_k \leq B/jbR$, where $W$ is the positive integer which is smaller than or equal to $NP/R$.

Instead, we can’t solve the optimization problem since $T(w)$ is not a convex function. $T(w)$ is convex when $f_1, \cdots, f_K$ are convex function. However, $f_k$ is not continuous when $w_k$ is an integer multiple of $P$. Therefore, we should clarify the domain region of $w_k$ to make $f_k$ continuous and convex. In this substep, we determine the number of nodes $y_k$ to make $f_k$ convex. To determine the region of $w_k$, we first approximate $y_k$ into constant and define $y_k'$ in (15).

$$y_k' = \left\{ \begin{array}{ll}
\frac{C_k^f}{C_k} + \frac{\delta_k \beta_p}{R} - \frac{P \delta_k \beta_k}{R}, & k < k_p \\
\frac{C_k^f}{f} + \frac{\delta_k \beta_p}{R} - \frac{P \delta_k \beta_k}{R}, & k \geq k_p
\end{array} \right.$$ (15)

Then, we solve the approximated primal problem of minimizing $T'(w) = \sum_{k=1}^{K} f'_k(w_k)$ while satisfying (11)-(14), where $f'_k(w_k) = \left\{ \begin{array}{ll}
\frac{y_k}{w_k} + T_{Sync}, & k = k_p \\
\frac{y_k}{w_k}, & k \neq k_p
\end{array} \right.$.

Note that $T : R^K \rightarrow R, f_k : R \rightarrow R$ are finite-valued and convex function, and (11)-(14) are closed convex set. Then, we can solve the approximated problem using duality. It is summarized in Lemma 1 (See Appendix for detailed proof).

Lemma 1: Let $\sum_{k=1}^{K} w_k = W, y_k > 0, \left[\frac{B}{bR}\right] \leq w_k' \leq \left[\frac{M_k}{m}\right]$, holds for all $k \in \{1, \cdots, K\}$. Then, $\sum_{k=1}^{K} \frac{y_k}{w_k}$ is
minimized when \( w'_k = \left( \left\lfloor \frac{\sqrt{K} + \sqrt{w'_k(W - S)}}{\sum_{i=1}^{K} \sqrt{y'_i}} \right\rfloor \wedge \frac{M_i}{m} \right) \vee \left\lfloor \frac{B}{Rb} \right\rfloor \), where \( S \) is the sum of \( w'_k \) which has \( \frac{M_i}{m} \) or \( \left\lfloor \frac{B}{Rb} \right\rfloor \).

From Lemma 1, we can simply determine the number of workers to minimize the objective function. We first calculate \( w_k = \left[ \frac{\sqrt{K} + \sqrt{W}}{\sum_{i=1}^{K} \sqrt{y_i}} \right] \) for all workers and adjust them to meet boundary conditions. If \( w_k < \left[ \frac{M_i}{m} \right] \), we set \( w_k \) to \( \left[ \frac{M_i}{m} \right] \), and if \( w_k > \left[ \frac{B}{Rb} \right] \), we set \( w_k \) to \( \left[ \frac{B}{Rb} \right] \). Finally, we add up all \( w_k \) which has value of \( \left[ \frac{M_i}{m} \right] \) or \( \left[ \frac{B}{Rb} \right] \) to \( S \), and calculate rest of \( w_k \) values to \( \left[ \frac{\sqrt{T} + \sqrt{w_k(W - S)}}{\sum_{i=1}^{K} \sqrt{y_i}} \right] \). If adjusted \( w_k \) value do not satisfy boundary condition, then we adjust \( w_k \) value again with the same procedure until all \( w_k \) value meet boundary condition.

However, the solution \( \{w'_k\} \) acquired by Lemma 1 is an approximation to the optimal solution since \( y_k \neq y'_k \). Theorem 1 analyzes the relationship between solutions of approximated problem and original problem. In conclusion, \( w'_k \) is an \((1 + \epsilon)\)-approximation to \( w_k^* \) and they have the same number of nodes \( y'_k \).

**Theorem 1 (Optimality of the approximated solution):**

Let \( w^* = \{w'_k\} \) minimizes \( \sum_{k=1}^{K} \frac{y_k}{w_k} \), \( w = \{w_k\} \) minimizes \( \sum_{k=1}^{K} \frac{y_k}{w_k} \). Then, \( w^* \) is an \((1 + \epsilon)\) approximation to the optimal solution \( w_k^* \), and \( (x'_k - 1)P \leq w_k^* \leq x'_kP \) holds.

**Proof:** Let \( y_k = y'_k + \Delta y'_k \leq y'_k(1 + \epsilon) \). By Lemma 1,

\[
w_k^* = \left( \left\lfloor \frac{\sqrt{y'_k + \Delta y'_k(W - S)}}{\sum_{i=1}^{K} \sqrt{y'_i + \Delta y'_i}} \right\rfloor \wedge \frac{M_i}{m} \right) \vee \left\lfloor \frac{B}{Rb} \right\rfloor \quad (16)
\]

It is obvious that \( w_k^* \) is minimized when \( y_i = y'_i(1 + \epsilon) \) except for \( i = k \), and \( w_k^* \) is maximized when \( y_i = y'_i(1 + \epsilon) \) only for \( i = k \). Thus, \( (17) \) holds, which means that \( w_k^* \) is \((1 + \epsilon)\)-approximation to the optimal solution \( w_k^* \).

\[
\frac{w_k^*}{1 + \epsilon} \leq w_k^* \leq (1 + \epsilon)w_k^* \quad (17)
\]

Also, \( \Delta y'_k \) diminishes to 0 when \( r'_k \) increases to \( P \). Thus, \( w_k^* \) will become larger compared to \( w_k^* \) when \( r'_k \) is small, and vice versa.

Therefore, \((x'_k - 1)P \leq w_k^* \leq x'_kP \) holds.

**Theorem 1** implies that the approximated solution \( w_k^* \) and the optimal solution \( w_k^* \) are similar to each other and they have the same number of nodes. Therefore, we can determine the range of the optimal solution \( w_k^* \) and the objective function becomes convex. Finally, we can solve the objective function at substep 4.

**4) SUBSTEP 4: FINDING THE SOLUTION**

Let \( w_k^* \) and \( x'_k \) be the number of workers and nodes to minimize the approximated problem in substep 3. Finally, through substep 1 to 3, the original problem in (10)-(13) is equivalent to minimizing

\[
T(w) = \sum_{k=1}^{K} \frac{y_k}{w_k} \quad (18)
\]

where \((x'_k - 1)P \vee \left\lfloor \frac{M_i}{m} \right\rfloor \leq w_k \leq (x'_kP \wedge \left\lfloor \frac{B}{Rb} \right\rfloor)\). \( \sum_{k=1}^{K} w_k \leq W \), and \( y_k \geq 0 \) holds.

As the range of \( w_k \) is given, the term \( \left\lfloor \frac{w_k^*}{m} \right\rfloor \) becomes constant. Thus, we re-formulate \( y_k \) as \( y'_k \) by:

\[
y'_k = \begin{cases} 
\frac{C_k^F + \delta_k^p}{f} + \frac{2\delta_k^p}{R}, & k < k_p \\
\frac{C_k^B + C_k^B}{f} + \frac{\delta_k^p}{R}, & k \geq k_p 
\end{cases} \quad (19)
\]

Therefore, based on Lemma 1, \( T(w) \) is minimized when

\[
w_k^* = \left[ \frac{\sqrt{y'_k(W - S)}}{\sum_{i=1}^{K} \sqrt{y'_i}} \right] \wedge \left( (x'_k - 1)P \vee \left\lfloor \frac{M_i}{m} \right\rfloor \right) \vee \left( x'_kP \wedge \left\lfloor \frac{B}{Rb} \right\rfloor \right) \quad (20)
\]

where \( S \) is the sum of \( w_k^* \) which has a value of \( \left\lfloor \frac{M_i}{m} \right\rfloor \), \( \left\lfloor \frac{B}{Rb} \right\rfloor \), \( x'_kP \), or \( (x'_k - 1)P \).

Through substep 4, we obtained the number of workers to minimize the objective function when \( w_k^* \) is fixed. By applying (20), we can obtain the optimal number of workers when \( R \) and \( G \) is given. We repeat Step 3-4 for every available candidate in Step 2, and find the solution which minimizes the training time.

To sum up, the worker allocation step has 4 substeps: pruning nonnegative solution of \( y_k < 0 \), determining a solution for sync-submodel to make the problem more simple, approximate the problem to figure out the number of nodes, and finding the solution. The worker allocation step is repeated for every model division acquired from Model Partition process and finds the global solution. The overall process of worker allocation is summarized in Algorithm 1.

The time complexity of the worker allocation scheme can be formulated as \( O(K\log N) \). Since the time complexity of the model partitioning scheme is formulated as \( O(M^K) \), the overall time complexity of the GHP can be formulated as \( O(M^K\log N) \). Because \( K \) is less or equal to 2 for most cases, we can figure out that our proposed heuristics is not a time-consuming job and the scheduling overhead can be ignored.

**V. EVALUATION**

In this section, we evaluate the performance of our proposed worker allocation schemes in distributed training with discussion.

**A. EXPERIMENTAL SETUP AND METRICS**

(Implementation) To verify the performance of the proposed group hybrid parallelism and worker allocation scheme,
Algorithm 1 Worker Allocation Scheme for Group Hybrid Parallelism

**Input**: $G$: Set of submodels

**Output**: $R$: Number of groups, $w$: worker allocation

for $R = 1$ to $NP$
do
    $W \leftarrow NP/R$
    for all $k$ where $y_k \leq 0$ /* Substep 1*/
do
        $w_k \leftarrow \left\lfloor \frac{M_k}{m} \right\rfloor$, $W \leftarrow W - w_k$
    for $w_{kp} = \left\lfloor \frac{M_{kp}}{m} \right\rfloor$ to argmin over $w_{kp}$
do
        /* Substep 2*/
        $W' \leftarrow W$
        if $y_{kp} \geq 0$ then
            $W' \leftarrow W - w_k$
        for all $k$ where $y_k > 0$ and $k \neq kp$
do
            $w'_{kp} \leftarrow \frac{w_{kp}}{P}$
            $x'_{kp} \leftarrow \frac{w'_{kp}}{B} \cdot \frac{R}{PR}$
            $W' \leftarrow W' - w_k$
        for all $k$ where $y_k > 0$ and $k \neq kp$
do
            $w_k \leftarrow \left( \left\lfloor \frac{\sqrt{y_k}W}{\sum_{i=1}^{N} \sqrt{y_i}} \right\rfloor \right) \lor \left\lfloor \frac{PR}{BR} \right\rfloor$
            $W \leftarrow W - w_k$
        /* Substep 3*/
    if $W < 0$ then
        return False
    return $(R^*, w^*)$

we implemented a scheduler and a distributed trainer based on Python and PyTorch. Fig. 3 depicts the architecture of the implemented scheduler and distributed trainer system. We briefly introduce how we implement each of them.

The scheduler determines the model partition and worker mapping that minimizes the training time when the node specification, network model, and batch size is given. The scheduler consists of the model partitioning function that splits the model into submodels, the worker grouping & mapping function that groups and maps the workers to the submodel, and the model profiler function that analyzes the computation FLOP of each layer. For convenience, the implemented model profiler does not predict computation FLOPs based on the measurement of real execution time; rather it calculates the FLOPs of direct calculation based on the layer structure [38]. Because deep-learning frameworks implement their own optimized operations instead of direct calculation, the estimated computation time may be different from the real execution value. However, our prediction of direct calculation is meaningful in that it can express the trends of computation FLOPs, even if it is not accurate.

The distributed trainer applies the node configuration obtained by the scheduler and executes training on the cluster nodes. The cluster manager manages the overall training process of cluster nodes. The metadata manager manages the communication process of collecting and distributing activation data among nodes or workers in the group. The gradient synchronization manages the process of forming a global gradient change by collecting local gradient changes generated by all workers, and distributing them to all training workers.

To manage the data communication among workers or nodes, our implementation uses two agents: group agents and node agents. Fig. 4 presents how agents manage data
communication among workers. A group agent exists for each group and manages data communication that occurs within the group. As depicted in Fig. 4 (b), multiple agents can exist for a single group when the group contains more workers than a node and they can communicate with each other through a node agent. Also, node agents exist for each node and they manage communication among nodes, including parameter synchronization among groups and data communication within a group. These processes were implemented through PyTorch and MPI.

(Testbed Specification and Training Model) In this study, we ran experiments on eight Nvidia RTX2080 workers. Each worker had 10.07 TFLOPS of performance with 368 tensor cores and 8GB GDDR6 memory. Each node contained a Xeon Phi CPU, 64GB memory, and 4 workers connected through a PCI-e channel. Nodes were connected to each other via a 1GBps Ethernet switch. We extended the node-to-node bandwidth to 10GBps and workers to 1024 for weak scaling test by emulation, based on the measured training time for laboratory scale. Cuda version 10 was installed in our testbed. We ran experiments for various network models and parallelism methods. We considered VGGNet16, VGGNet19, RESNet50, RESNet101, and RESNet152 models with the ImageNet-1K dataset.

(Baselines) We compare GHP with three types of parallelism methods: synchronous data parallelism [2], the one weird trick (OWT) method [20], and GPipe [34]. (i) Synchronous data parallelism is a data parallelism method of synchronizing all parameters for every iteration, and it is adopted for many frameworks due to its simplicity. (ii) OWT is an empirical hybrid parallelism method of separating conv and fc layers and applying data parallelism to each separation. (iii) GPipe is a pipeline parallelism method to divide the entire training model into stages and accelerate training by dividing batches into microbatches. Note that all of the parallelism methods listed above do not sacrifice accuracy so the accuracy remains the same.

(Performance Metrics) We measured the wallclock time of an iteration and the throughput for various configurations, such as batch size, network model, number of workers, and parallelism methods. The wallclock time and throughput were collected by averaging 100 iterations. In addition, we compared the real measured training time and the estimated training time to evaluate our time model for various configurations. To compare the real and estimated values for various cases, we used mean average percentage error (MAPE) for all test cases’ computation, communication, synchronization, and overall training time. MAPE can be calculated by (21), where $T_i$ is the number of test cases and $T_{\text{real}}$ and $T_{\text{est}}$ is the real value and the estimated value of $i^{\text{th}}$ test case.

$$\text{MAPE} = \frac{100\%}{TC} \sum_{i=1}^{TC} \frac{T_{\text{real}} - T_{\text{est}}}{T_{\text{real}}}$$

To measure the scalability, we used speedup, which is the throughput divided by throughput of a single worker [43].

$$\text{speedup}(n) = \frac{\text{throughput of } n \text{ workers}}{\text{throughput of 1 worker}}$$

In our experiment, we did not measure the accuracy of the training model because the parallelism methods used in our experiment (sync-DP, OWT, GPipe, and proposed model) do not sacrifice accuracy at all.

B. PERFORMANCE EVALUATION

We present three experiments to evaluate our proposed training time model and group hybrid parallelism.

1) TRAINING TIME MODEL VALIDATION

We first demonstrate the accuracy of the training time model by comparing the estimated execution time acquired by the math model with the real execution time measured by the distributed trainer. Fig. 5 compares the estimated training time and the actual execution result under various configurations (training model, batch size, and worker allocation). For more precise analysis, we analyzed the (a) computation time, (b) communication time, (c) synchronization time, and (d) overall training time for 46 different configurations.
As shown in Fig. 5, the proposed training time model predicted the training time at a significant level. For the 46 configurations, the mean average percentage error (MAPE) was calculated as about 35% for the computation time, and about 15% for the communication, synchronization, and total training time.

There are several sources for possible errors between estimated training time and real execution time. We observed that the computation time showed a larger error than other parts (see Fig. 5(a)) because we estimated the computation time based on the amount of calculation that occurs when the DNN operation is calculated directly. However, in a real environment, DNN operation is calculated through various methods such as GEMM or FFT instead of direct calculation. We expect a more precise estimation of computation time if we estimate the computation time based on profiling. Also, since each training framework is implemented in a different way, there are many other parameters that affect learning, such as a delay for each framework. Therefore, more parameters and variables should be considered for precise estimation. However, the training time model in this study does not aim to predict the exact level of training time, but the goal is to observe the trend of training time for given configurations (set of sub-models and worker assignment). Thus, we considered the above overheads beyond the scope in this paper. Despite these limitations, a low level of error occurred between the estimated training time and the actual training time, so that the estimated value through the mathematical model could be sufficiently utilized for scheduling.

In conclusion, our proposed math model can serve as a crucial tool for worker configuration, regardless of hardware configuration, such as training model, batch size, and worker allocation.

2) EFFECT OF GROUPING WORKERS
This experiment analyzed the effect of grouping workers in terms of the training time. We fixed the number of workers belonging to a single group and analyzed the minimum training time considering all possible configurations of the case of 8 workers. Fig. 6 is a graph representing the minimum training time with a variation of the number of workers from 1 to 8 in a single group.

Fig. 6 can be interpreted by dividing the domain by three regions. For each region, a different parallelism method is applied. When the number of workers in a group is 1 (\( \sum w_k = 1 \), grey region in Fig. 6), it means only one submodel exists for the training; thus, synchronous data parallelism is applied at this region. When the number of workers in a group is larger than the medium (4 < \( \sum w_k \leq 8 \), blue region in Fig. 6), it means only one group exists and more than two submodels can exist for the training; thus, model parallelism or hybrid parallelism is considered at this region. When the number of workers in a group is smaller than or equal to the medium (1 < \( \sum w_k \leq 4 \), green region in Fig. 6), more than two groups and more than two submodels can exist for the training; thus, group hybrid parallelism is considered at this region.

As the characteristics of each training model differ, the shapes of the graphs also differ and are unpredictable. However, as seen in Fig. 6, we observed that the minimal training time generally appears when the number of workers in a group is medium (1~4). This region cannot be discovered when we only consider data, model, or hybrid parallelism, so group hybrid parallelism finds out a better solution compared to that of data, models, and hybrid parallelism. Interestingly, large-scale models (VGGNet19, RESNet152) showed a great training time drop when the number of workers is close to 4. Because the proposed GHP scheme balances computation, communication, and synchronization time, it is especially effective when training large-scale models. So we can conclude that large-scale models can benefit from GHP because the synchronization time is greatly reduced by dividing model parameters into submodels. Moreover, when the number of workers in a group is multiplication or division of 4, workers in a node can be fully utilized because each node has 4 workers in it. As depicted in Figure 6, the training time tends to reduce when the number of workers is 1, 2, 4, 8.

In conclusion, the solution space of group hybrid parallelism includes that of data, model, and hybrid parallelism. Moreover, we should determine an efficient hardware configuration in order to benefit from group hybrid parallelism.

3) WEAK SCALING PERFORMANCE EVALUATION
Table 2 shows the weak-scaling performance of the proposed scheme. We measured the throughput when group hybrid parallelism is applied, and compared the throughput with that of a synchronous data parallelism scheme (Sync-DP),
TABLE 2. Weak scaling results on various models.

| Worker | Proposed | OWT | Sync-DP | GPipe |
|--------|----------|-----|--------|-------|
| 2      | 167.85 (63x) | 119.37 (45x) | 167.85 (63x) | 182.05 (69x) |
| 4      | 305.48 (1.2x) | 222.15 (84x) | 231.08 (87x) | 264.49 (1.0x) |
| 8      | 530.52 (2.0x) | 245.34 (93x) | 365.98 (1.4x) | 362.24 (1.0x) |
| 16     | 934.35 (3.5x) | 320.19 (1.2x) | 605.82 (2.3x) | - |
| 32     | 1640.86 (6.2x) | 394.95 (1.7x) | 1033.73 (3.9x) | - |
| 64     | 2922.21 (11x) | 459.57 (1.7x) | 1802.56 (6.8x) | - |
| 128    | 5144.35 (19x) | 508.62 (1.9x) | 3195.61 (12x) | - |
| 256    | 9421.93 (36x) | 541.95 (2.1x) | 5759.22 (22x) | - |
| 512    | 17248.65 (63x) | 635.75 (2.4x) | 10416.89 (38x) | - |
| 1024   | 31350.11 (18x) | 697.97 (2.6x) | 19068.72 (72x) | - |

(a) Throughput and speedup of RESNet50 [samples/sec]

| Worker | Proposed | OWT | Sync-DP | GPipe |
|--------|----------|-----|--------|-------|
| 2      | 86.63 (67x) | - | 86.63 (67x) | 76.90 (39x) |
| 4      | 169.21 (1.3x) | 119.16 (92x) | 119.49 (93x) | 178.81 (1.4x) |
| 8      | 290.87 (2.3x) | 150.74 (1.2x) | 195.68 (1.5x) | 198.72 (1.5x) |
| 16     | 506.91 (3.9x) | 216.44 (1.7x) | 319.38 (2.5x) | - |
| 32     | 899.09 (6.7x) | 294.53 (2.3x) | 547.64 (2.6x) | - |
| 64     | 1620.00 (13x) | 375.16 (2.9x) | 958.43 (7.4x) | - |
| 128    | 2898.62 (22x) | 446.38 (3.5x) | 1704.01 (13x) | - |
| 256    | 5335.94 (41x) | 500.75 (3.9x) | 3067.62 (24x) | - |
| 512    | 9877.57 (76x) | 603.77 (4.7x) | 5577.84 (43x) | - |
| 1024   | 18092.14 (100x) | 676.65 (5.2x) | 10226.79 (79x) | - |

(b) Throughput and speedup of RESNet101 [samples/sec]

| Worker | Proposed | OWT | Sync-DP | GPipe |
|--------|----------|-----|--------|-------|
| 2      | 59.78 (70x) | - | 59.78 (70x) | 65.06 (76x) |
| 4      | 118.98 (1.4x) | - | 82.79 (97x) | 120.14 (1.4x) |
| 8      | 198.72 (2.3x) | 111.64 (3.3x) | 133.83 (1.6x) | 146.10 (1.7x) |
| 16     | 330.82 (3.9x) | 186.72 (2.0x) | 224.39 (2.6x) | - |
| 32     | 566.66 (6.6x) | 240.53 (2.8x) | 346.94 (4.5x) | - |
| 64     | 994.37 (12x) | 323.17 (3.8x) | 679.68 (7.9x) | - |
| 128    | 1865.72 (22x) | 403.42 (4.7x) | 1211.9 (14x) | - |
| 256    | 3513.94 (41x) | 469.83 (5.5x) | 2186.47 (26x) | - |
| 512    | 6597.77 (77x) | 578.68 (6.8x) | 3983.14 (76x) | - |
| 1024   | 12320.14 (144x) | 689.19 (7.7x) | 7314.09 (85x) | - |

(c) Throughput and speedup of RESNet152 [samples/sec]

| Worker | Proposed | OWT | Sync-DP | GPipe |
|--------|----------|-----|--------|-------|
| 2      | 39.22 (65x) | - | 39.22 (65x) | 100.75 (1.7x) |
| 4      | 137.11 (2.3x) | 137.11 (2.3x) | 58.17 (96x) | 153.94 (2.5x) |
| 8      | 241.88 (4.0x) | 243.63 (4.0x) | 82.96 (14x) | 113.30 (1.9x) |
| 16     | 381.58 (6.3x) | 410.42 (6.8x) | 128.92 (2.1x) | - |
| 32     | 713.11 (12x) | 636.55 (11x) | 210.84 (3.5x) | - |
| 64     | 1177.05 (19x) | 904.28 (15x) | 356.65 (5.9x) | - |
| 128    | 1728.27 (29x) | 1171.81 (19x) | 618.01 (10x) | - |
| 256    | 2686.64 (47x) | 1395.72 (23x) | 1090.4 (18x) | - |
| 512    | 4492.47 (74x) | 1556.52 (26x) | 1950.8 (32x) | - |
| 1024   | 7383.12 (112x) | 2063.33 (34x) | 3552.9 (58x) | - |

(d) Throughput and speedup of VGGNet16 [samples/sec]

| Worker | Proposed | OWT | Sync-DP | GPipe |
|--------|----------|-----|--------|-------|
| 2      | 35.18 (74x) | - | 35.18 (74x) | 79.82 (1.7x) |
| 4      | 108.01 (2.3x) | 108.01 (2.3x) | 53.03 (1.1x) | 153.60 (3.2x) |
| 8      | 194.94 (4.1x) | 194.94 (4.1x) | 76.81 (1.6x) | 168.13 (3.5x) |
| 16     | 309.16 (6.5x) | 322.88 (7.0x) | 120.42 (2.5x) | - |
| 32     | 573.91 (12x) | 530.98 (11x) | 197.99 (4.2x) | - |
| 64     | 973.91 (20x) | 782.84 (16x) | 336.21 (7.0x) | - |
| 128    | 1697.03 (36x) | 1055.72 (22x) | 854.24 (12x) | - |
| 256    | 2831.65 (59x) | 1303.02 (27x) | 1035.01 (21x) | - |
| 512    | 4425.39 (93x) | 1492.31 (31x) | 1851.43 (39x) | - |
| 1024   | 7292.35 (153x) | 2002.45 (42x) | 3534.57 (70x) | - |

(e) Throughput and speedup of VGGNet19 [samples/sec]
performance compared to the data parallelism, and we should apply the optimized model-division scheme.

Furthermore, we found that the structure of the training model and its parameter size affect the performance. As seen in Table 2, OWT performs better than Sync-DP in the VGGNet model, and vice versa in the RESNet model. This means we should consider the model structure to determine the optimal parallelism strategy. However, group hybrid parallelism considers model structure in model partitioning process so it results in best performance regardless of the training model. Because model parameter size is directly related to the synchronization delay, it affects the performance. When the model parameter size is small, the gain obtained by model partitioning is small, so the effectiveness of the proposed scheme is not significant. For models with small parameter size, such as RESNet50, the performances of proposed scheme and Sync-DP did not show much difference. However, for models with large parameter sizes, such as VGGNet19, the proposed scheme outperformed the Sync-DP. This means that model partitioning is effective when the model size is large.

To sum up, we observed that our scheme greatly reduces the training time by utilizing group hybrid parallelism. We compared our scheme with data parallelism (sync-DP), hybrid parallelism (OWT), and model parallelism (GPipe) schemes and found that our scheme outperforms them in terms of throughput and speedup. Also, we observed that our scheme is useful for models with the large number of model parameters.

VI. CONCLUSION

In this paper, we addressed the limitations of existing parallelism schemes for training large-scale DNN models. We observed that existing parallelism schemes lack scalability due to the long communication time and worker memory limitation, and we proposed a group hybrid parallelism scheme to enhance scalability while maintaining accuracy. By analyzing the training behavior, we presented a training time model when group hybrid parallelism is applied. Based on the training time model, we proposed a worker allocation scheme to minimize the training time. Our solution is scalable as we minimize the communication time by grouping workers, and our solution resolves the worker memory limitation issue by partitioning the model parameters. Moreover, because we do not change model parameter size and parameter staleness, the accuracy remains high.

We implemented a scheduler and a distributed trainer to evaluate the performance of our proposed schemes, and we conducted experiments on a laboratory scale. First, we compared the estimated training time acquired by the training time model and real training time. In this experiment, the total training time was about 20% of the MAPE value, so we concluded that our model can represent the actual training time. Second, we measured the minimum training time by varying the number of workers in a single group to prove that the group hybrid parallelism outperforms the existing parallelism schemes. In most cases, we found that the training time has a minimum value when the number of workers in the group is not too small (data parallelism) or too large (model, hybrid parallelism). Finally, we conducted a weak scaling test by comparing the throughput of various parallelism schemes by increasing the batch size and the number of workers.

We concluded that our scheme greatly reduces the training time for large-scale DNN models. Based on experiments and simulations, we proved that the training time can be minimized by the group hybrid parallelism and worker allocation schemes.

APPENDIX A

PROOF OF PROPOSITIONS AND LEMMAS

Proposition 4: Let \( w^* = \{ w^*_k \} \) minimizes
\[
T(w) = \sum_{k=1}^{K} \frac{y_k}{w_k} + T_{\text{sync}} \quad \text{while satisfying (11)-(13)}.
\]
Then, \( w^*_k = \frac{M_k}{m} \) for all \( k \) if \( y_k < 0 \).

Proof: Suppose this proposition is false. Then, there exists \( w_k^* \) that satisfies
\[
T(w^*_1, \ldots, w^*_k, \ldots, w^*_K) \leq T(w^*_1, \ldots, M/m, \ldots, w^*_K) \quad \text{where} \quad M_m < w^*_k \leq \frac{B_k}{y_k} \quad \text{and} \quad y_k < 0.
\]
\( T(w) \) can be represented by
\[
f_k(w_k) = \frac{y_k}{w_k} + T_{\text{sync}}, \quad k = k_p
\]
Since \( T_{\text{sync}} \) is monotonic increasing function of \( w_k \), \( f_k(w_k^*) \geq f_k(M/m) \) always holds for \( y_k < 0 \) and this leads to a contradiction.

Therefore, \( w^*_k = \frac{M_k}{m} \) for all \( k \) if \( y_k < 0 \). \( \square \)

Proposition 5: Let \( w^* = \{ w^*_k \} \) minimizes
\[
T(w) = \sum_{k=1}^{K} \frac{y_k}{w_k} + T_{\text{sync}} \quad \text{while satisfying (11)-(14)}.
\]
Then, \( \left[ \frac{M_{k_p}}{m} \right] \leq w^*_k \leq \arg \min_{w_k} \left( \frac{y_k}{w_k} + T_{\text{sync}} \right) \) holds.

Proof: Let \( w_{k_p}^\min = \arg \min_{w_{k_p}} \left( \frac{y_{k_p}}{w_{k_p}} + T_{\text{sync}} \right) \), and \( \{ w_{k_p}^\min \} \) minimizes \( T(w) \) when \( w_{k_p} = w_{k_p}^\min \) with Condition (11)-(14).

By definition of \( w_{k_p}^\min \), Inequality (23) holds:
\[
\frac{y_{k_p}}{w_{k_p}^*} + T_{\text{sync}}(w_{k_p}^*) \geq \frac{y_{k_p}}{w_{k_p}^\min} + T_{\text{sync}}(w_{k_p}^\min)
\]
When \( w_{k_p}^\min \leq w^*_k \), \( \sum_{k \neq k_p} w^*_k \geq \sum_{k \neq k_p} w_{k_p}^\min \) holds by worker constraint, and thus Inequality (24) holds:
\[
\sum_{k \neq k_p} \frac{y_k}{w_k} \geq \sum_{k \neq k_p} \frac{y_k}{w_{k_p}} \quad \text{(24)}
\]
Combining Inequality (23, 24), for any \( w_{k_p}^* \geq w_{k_p}^\min \),
\[
T'(\{w_1, \ldots, w_K\}) \geq T'(\{w_1^\min, \ldots, w_K^\min\})
\]
\[
: \left[ \frac{M_{k_p}}{m} \right] \leq w_{k_p}^* \leq \arg \min_{w_{k_p}} \left( \frac{y_{k_p}}{w_{k_p}} + T_{\text{sync}} \right)
\]
\( \square \)
Lemma 2: Let \( \sum_{k=1}^{K} w_k = W \), \( y_k > 0 \), \( [\frac{B}{R}] \leq w_k \leq [\frac{M_k}{m}] \) holds for all \( k \in \{1, \ldots, K\} \). Then, \( \sum_{k=1}^{K} y_k w_k \) is minimized when \( w_k' = \left( \left( \sum_{k=1}^{K} \sqrt{y_k} \right) \left( W - S \right) \right) \wedge [\frac{M_k}{m}] \) \( \lor [\frac{B}{R}] \), where \( S \) is the sum of \( w_k' \) which has \( [\frac{M_k}{m}] \) or \( [\frac{B}{R}] \).

Proof: The corresponding dual problem is expressed as:

\[
L_w = \frac{y_k}{w_k} + \lambda_1 (\frac{B}{b}) + \lambda_2 (\frac{M_k}{m} - w_k) + \sum_{k=1}^{K} \lambda_k y_k w_k
\]

where

\[
\frac{\partial L_w}{\partial w_k} = -\frac{y_k}{w_k^2} + \lambda + \lambda_k - \lambda_{K+k} = 0
\]

\( \forall k \in \{1, \ldots, K\} \)

(27)

(28)

(29)

(30)

Since \( \lambda_k \) and \( \lambda_{K+k} \) cannot both be positive, we consider 3 cases:

- **(C1)** \( \lambda_k > 0 \), and \( \lambda_{K+k} = 0 \): By (31),

\[
w_k = \frac{B}{bR}
\]

(32)

- **(C2)** \( \lambda_k = 0 \), and \( \lambda_{K+k} > 0 \): By (31),

\[
w_k = \frac{M_k}{m}
\]

(33)

- **(C3)** \( \lambda_k = 0 \), and \( \lambda_{K+k} = 0 \): By (27),

\[
-\frac{y_k}{w_k} + \mu = 0
\]

(34)

Combining (32)-(34) to (28),

\[
\sum_{k=1}^{C_1} \frac{B}{bR} + \sum_{k \in C_2} \frac{M_k}{m} + \sum_{k \in C_3} \sqrt{y_k} = W - S
\]

(35)

where \( S = \sum_{k \in C_1} \frac{B}{bR} + \sum_{k \in C_2} \frac{M_k}{m} \) is the set of indices where \( \lambda_i = 0 \) and \( \lambda_{K+i} = 0 \).

Thus, \( \frac{1}{\sqrt{B}} \left( \frac{W - S}{\sum_{k \in C_3} \sqrt{y_k}} \right) \) and \( w_k = \sqrt{\frac{y_k}{\sum_{k \in C_3} y_k}} y_k \in C_3 \), and considering that \( w_k \) is an positive integer,

\[
w_k' = \left( \left( \sum_{k=1}^{K} \sqrt{y_k} \right) \left( W - S \right) \right) \wedge [\frac{M_k}{m}] \lor [\frac{B}{R}] \forall k
\]

(36)

REFERENCES

[1] R. Mikkilineni, A. Comparini, and G. Morana, “The turing O-machine and the DME network architecture: Injecting the architectural resiliency into distributed computing,” in *Turing-100* (EPIC Series, The Alan Turing Centenary), vol. 10, A. Voronkov, Ed. EasyChair Proceedings in Computing, Jun. 2012, pp. 239–251.

[2] D. Das, S. Avancha, D. Mudigere, K. Vaidyathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, “Distributed deep learning using synchronous stochastic gradient descent,” 2016, arXiv:1602.06709. [Online]. Available: http://arxiv.org/abs/1602.06709

[3] T. Ben-Nun and T. Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *ACM Comput. Surveys*, vol. 52, no. 4, pp. 1–43, Sep. 2019.

[4] P. Mayer and H.-A. Jacobsen, “Scalable deep learning on distributed infrastructures: Challenges, techniques and tools,” 2019, arXiv:1903.13114. [Online]. Available: http://arxiv.org/abs/1903.13114

[5] M. Rhu, N. Gimbelsheim, J. Clemens, A. Zulfiquar, and S. W. Keckler, “vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design,” in *Proc. 49th Annua. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Oct. 2016, pp. 1–13.

[6] F. Haddadpour, “Trading redundancy for communication: Speeding up distributed SGD for non-convex optimization,” in *Proc. 36th Int. Conf. Mach. Learn. (PMLR)*, Long Beach, CA, USA, 2019, pp. 2545–2554.

[7] J. Chen, K. Li, K. Bilal, X. Zhou, K. Li, and P. S. Yu, “A bi-layered parallel training architecture for large-scale convolutional neural networks,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 5, pp. 965–976, May 2019, doi: 10.1109/TPDS.2018.2877359.

[8] Z. Jia, M. Zaharia, and A. Aiken, “Beyond data and model parallelism for deep neural networks,” in *Proc. 2nd SystML Conf.*., Palo Alto, CA, USA, 2019, pp. 1–13.

[9] A. Gholami, A. Azad, P. Jin, K. Keutzer, and A. Buluc, “Integrated model, batch and domain parallelism in training neural networks,” 2017, arXiv:1712.04432. [Online]. Available: http://arxiv.org/abs/1712.04432

[10] J. Keuper and F.-J. Preundt, “Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability,” in *Proc. 2nd Work- shop Mach. Learn. HPC Environ. (MLHPC)*, Nov. 2016, pp. 19–26.

[11] H. Qi, E. R. Sparks, and A. Talwalkar, “Paleo: A performance model for distributed deep neural networks,” in *Proc. ICLR*, Apr. 2017, pp. 1–9.

[12] S. Shi, Q. Wang, X. Chu, and B. Li, “A DAG model of synchronous stochastic gradient descent in distributed deep learning,” in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2018, pp. 425–432.

[13] A. Castello, M. F. Dolz, E. S. Quintana-Orti, and J. Duato, “Theoretical scalability analysis of distributed deep convolutional neural networks,” in *Proc. 19th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2019, pp. 534–541.

[14] R. Thakur, R. Rabenseifner, and W. Gropp, “Optimization of collective communication operations in MPICH,” *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, pp. 49–66, 2005.

[15] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, “PipeDream: Generalized pipeline parallelism for DNN training,” in *Proc. 27th ACM Symp. Operating Syst. Princ.*, Oct. 2019, pp. 1–15.

[16] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, “Device placement optimization with reinforcement learning,” 2017, arXiv:1706.04972. [Online]. Available: http://arxiv.org/abs/1706.04972

[17] R. Addanki, S. B. Venkatakrishnan, S. Gupta, H. Mao, and M. Alizadeh, “Placerto: Learning generalizable device placement algorithms for distributed machine learning,” in *Proc. NIPS*, 2019, pp. 1–11.

[18] M. Tan and Q. V. Le, “EfficientNet: Rethinking model scaling for convolutional neural networks,” in *Proc. ICML*, 2019, pp. 6105–6114.

[19] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, “Fast deep neural network training on distributed systems and cloud TPUs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 11, pp. 2449–2462, Nov. 2019.

[20] A. Krizhevsky, “One weird trick for parallelizing convolutional neural networks,” 2014, arXiv:1404.5597. [Online]. Available: http://arxiv.org/abs/1404.5597

[21] A. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, and A. Tulloch, “Accurate, large minibatch SGD: Training ImageNet in 1 hour,” 2017, arXiv:1706.02677. [Online]. Available: http://arxiv.org/abs/1706.02677

[22] J. Konený, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, “Federated learning: Strategies for improving communication efficiency,” 2016, arXiv:1610.05492. [Online]. Available: http://arxiv.org/abs/1610.05492
[23] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Y. Arcas, “Communication-efficient learning of deep networks from decentralized data,” 2016, arXiv:1602.05629. [Online]. Available: http://arxiv.org/abs/1602.05629

[24] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecny, S. Mazzocchi, H. B. McMahan, T. Van Overveldt, D. Petrou, D. Ramage, and J. Roselander, “Towards federated learning at scale: System design,” 2019, arXiv:1902.01046. [Online]. Available: http://arxiv.org/abs/1902.01046

[25] Q. V. Le, “Building high-level features using large scale unsupervised learning,” in Proc. IEEE Int. Conf. Acoust., Speech Signal Process., May 2013, pp. 8595–8598.

[26] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, and A. Y. Ng, “Large scale distributed deep networks,” in Proc. Adv. Neural Inf. Process. Syst., 2012, pp. 1223–1231.

[27] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, “Revisiting distributed synchronous SGD,” 2016, arXiv:1604.00981. [Online]. Available: http://arxiv.org/abs/1604.00981

[28] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing, “Solving the straggler problem with bounded staleness,” in Proc. 14th Workshop Hot Topics Oper. Syst. (HotOS), 2013.

[29] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar, “Slow and stale gradients can win the race: Error-runtime trade-offs in distributed SGD,” in Proc. Int. Conf. Artif. Intell. Statist., 2018, pp. 803–812.

[30] P. H. Jin, Q. Yuan, F. Iandola, and K. Keutzer, “How to scale distributed deep learning?” in Proc. ML Syst. Workshop NIPS, 2016.

[31] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, “QSGD: Communication-efficient SGD via gradient quantization and encoding,” in Advances in Neural Information Processing Systems 30, Cambridge, MA, USA: MIT Press, 2017, pp. 1709–1720.

[32] Y. Lin, S. Han, H. Mao, Y. Wang, and W. Dally, “Deep gradient compression: Reducing the communication bandwidth for distributed training,” in Proc. Int. Conf. Learn. Represent. (ICLR), 2018.

[33] E. Azarkish, D. Rossi, I. Loi, and L. Benini, “Neurostream: Scalable and energy efficient deep learning with smart memory cubes,” IEEE Trans. Parallel Distrib. Syst., vol. 29, no. 2, pp. 420–434, Feb. 2018.

[34] Y. Huang, Y. Cheng, A. Babna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, “GPipe: Efficient training of giant neural networks using pipeline parallelism,” in Proc. Neural Inf. Process. Syst. (NeurIPS), 2019.

[35] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, “Exploring hidden dimensions in parallelizing convolutional neural networks,” 2018, arXiv:1802.04924. [Online]. Available: http://arxiv.org/abs/1802.04924

[36] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017, arXiv:1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347

[37] X. Wu, H. Xu, B. Li, and Y. Xiong, “Stanza: Layer separation for distributed training in deep learning,” 2018, arXiv:1812.10624. [Online]. Available: http://arxiv.org/abs/1812.10624

[38] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, “Performance modeling and scalability optimization of distributed deep learning systems,” in Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, Aug. 2015, pp. 1355–1364.

[39] M. Li, T. Zhang, Y. Chen, and A. J. Smola, “Efficient mini-batch training for stochastic optimization,” in Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, Aug. 2014, pp. 661–670.

[40] R. W. Hockney, “The communication challenge for MPP: Intel paragon and Meiko CS-2,” Parallel Comput., vol. 20, no. 3, pp. 389–398, Mar. 1994.

[41] R. Rabenseifner, “Optimization of Collective Reduction Operations,” in Proc. Int. Conf. Comput. Sci., Jun. 2004, pp. 1–9.

[42] T. Gonzalez and S. Sahni, “Open shop scheduling to minimize finish time,” J. ACM, vol. 23, no. 4, pp. 665–679, Oct. 1976.

[43] J. L. Gustafson, “Reevaluating Amdahl’s law,” Commun. ACM, vol. 31, no. 5, pp. 532–533, May 1988.

KYUNG-NO JOO (Member, IEEE) received the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2014, where he is currently pursuing the Ph.D. degree in electrical engineering. His research interests include graph scheduling, mobile edge computing, and distributed training acceleration.

CHAN-HYUN YOUN (Senior Member, IEEE) received the B.Sc. and M.Sc. degrees in electronics engineering from Kyungpook National University, Daegu, South Korea, in 1981 and 1985, respectively, and the Ph.D. degree in electrical and communications engineering from Tohoku University, Japan, in 1994. Before joining the University, from 1986 to 1997, he was the Leader of the High-Speed Networking Team, KT Telecommunications Network Research Laboratories, where he had been involved in the research and developments of centralized switching maintenance system, high-speed networking, and ATM network. Since 2009, he has been a Professor with the School of Electrical Engineering, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea. He was an Associate Vice-President of office of planning and budgets with KAIST, from 2013 to 2017. He is also the Director of the Grid Middleware Research Center and the XAI Acceleration Technology Research Center, KAIST, where he is developing core technologies that are in the areas of high performance computing, edge-cloud computing, AI acceleration system, and others. He wrote a book on Cloud Broker and Cloudlet for Workflow Scheduling, (Springer, 2017). He was a General Chair of the 6th EAI International Conference on Cloud Computing (Cloud Comp 2015), KAIST, in 2015, and a Guest Editor of IEEE Wireless Communications in 2016. He served as a TPC Member for many international conferences.