Task allocation for decentralized training in heterogeneous environment

1st Yongyue Chao  
Institute of Automation, Chinese Academy of Sciences  
Beijing, China  
chaoyongyue2020@ia.ac.cn

3rd Jiaxin Gao  
Institute of Automation, Chinese Academy of Sciences  
Beijing, China  
jiaxin.gao@ia.ac.cn

2nd Mingxue Liao  
Institute of Automation, Chinese Academy of Sciences  
Beijing, China  
mimgxue.liao@ia.ac.cn

Abstract—The demand for large-scale deep learning is increasing, and distributed training is the current mainstream solution. Ring AllReduce is widely used as a data parallel decentralized algorithm. However, in a heterogeneous environment, each worker calculates the same amount of data, so that there is a lot of waiting time loss among different workers, which makes the algorithm unable to adapt well to heterogeneous clusters. Resources are not used as they should be. In this paper, we design an implementation of static allocation algorithm. The dataset is artificially allocated to each worker, and samples are drawn proportionally for training, thereby speeding up the training speed of the network in a heterogeneous environment. We verify the convergence and influence on training speed of the network model under this algorithm on one machine with multi-card and multi-machine with multi-card. On this basis of feasibility, we propose a self-adaptive allocation algorithm that allows each machine to find the data it needs to adapt to the current environment. The self-adaptive allocation algorithm can reduce the training time by nearly one-third to half compared to the same proportional allocation. In order to better show the applicability of the algorithm in heterogeneous clusters, We replace a poorly performing worker with a good performing worker or add a poorly performing worker to the heterogeneous cluster. Experimental results show that training time will decrease as the overall performance improves. Therefore, it means that resources are fully used. Further, this algorithm is not only suitable for straggler problems, but also for the most heterogeneous situations. It can be used as a plug-in for AllReduce and its variant algorithms.

Index Terms—distributed training, task allocation, Ring AllReduce, heterogeneity

I. INTRODUCTION

Artificial intelligence is widely applied in various fields as well as deep learning plays an important role in it. Establishing deep neural network model within big data is the main method in deep learning. With the development of technology, more complex model and larger dataset appear but it is hard to train by single machine. To speed up, distributed training steps in large scale deep learning. It is the idea that all data is distributed to nodes for calculating and then results are aggregated together to update parameters of model.

At present, small memory and long running time are two primary shortcomings of single machine. In distributed training, model parallelism[11] and data parallelism[1] are proposed to solve the problems. Owing to high throughput, data parallelism is extensively used to train large scaled neural network. most of data parallelism methods are constructed based on gradient aggregation. In 2013, Li et al. achieved parameter server (PS) which is one of common data parallelism framework as figure 1(a) shows. It consists of two functional nodes called worker and server. Workers pull latest parameters from servers to compute and then push local gradients to servers. Server aggregates all local gradients to update parameters of network model. However, communication bottleneck[15] exists in PS framework because of centralized communication. In 2017, All reduce is applied in deep learning by Ring allreduce algorithm as figure 1(b) shows. It eliminated communication bottleneck by letting each node only communicate with adjacent nodes. All nodes in Ring Allreduce has same amount of communication. Local gradients is passed to each node through Allreduce, then all nodes update parameters of model individually. Ring Allreduce is a synchronization algorithm. Aggregating local gradients happens after that all nodes have finished computing. Therefore, in heterogeneous environment, due to barrel effect, training velocity relies on the slowest node. The later algorithms focused on improving Ring Allreduce as an asynchronous algorithm. They modified the method of aggregating gradients to reduce training time.

Using Ring Allreduce to complete the training is divided into two steps. First, separate dataset equally to each node and train. In this step, all nodes obtain the same number of data. There is no interference between nodes. Each node just calculate the gradient of parameters. After computing, node will arrive at barrier to wait for other nodes together. Second, renew parameters. Through Allreduce, each node gathers global gradients from others and uses back propagation[7] to update model parameters. In homogeneous environment, due to equal performance and same amount data, all nodes reach barrier almost at the same time. Therefore, there is no time wasted by waiting. In heterogeneous environment, this problem can slow down training greatly. However, the second step received most attention from researchers. Most of
existing methods focusing on Ring Allreduce make progress in acceleration by asynchronous algorithm. Just few researchers have considered how to realize the partition of dataset based on Ring Allreduce and how to separate data set appropriately to nodes with different performance. Theoretically, through assigning reasonable task to different nodes, synchronization waiting time can be shortened and thus total training time can be shortened too.

In this paper, we proposed an implement of assigning tasks to nodes with different performance. It is called static allocation Allreduce algorithm. It could ensure that the dataset is allocated to each node under the premise of neural network convergence. To prove it, we train a simple convolutional neural network[2] model on MNIST dataset[4], some complex network such as ResNet18, ResNet50[18], VGG16 and VGG19[17] model on CIFAR10 dataset[10]. Experimental results show that changing the ratio of dataset or tasks on different nodes has little effect on the number of network convergence epochs but can adjust the training time of each epoch. On the basis of this implement, we further proposed an adaptive algorithm to compute how to distribute data to different nodes without manufacturer information of nodes. We set a series of experiments to prove the improvement of training velocity. We use two nodes equipped with v100 and RTX 2080ti respectively to confirm the training velocity will be increased along with the ratio of dataset on nodes and epochs changed. When the ratio of dataset stabilized, training time per epoch could be reduced 20 40 percentage than the same ratio. When we replace one of weak GPUs with strong GPU or add one GPU We also compared results. Under the same network bandwidth, the training time will be reduced along with the improvement of performance.

II. BACKGROUND AND MOTIVATION

A. Heterogeneity

From the perspective of heterogeneity, in data parallelism, whether PS or Ring Allreduce, although the training process seems to be accelerated through the way that each node computes local gradients, the occurrence of synchronization and communication among different nodes slows down the training time after that. Backup and asynchronization explored solution to ignore or put off straggler[19] in synchronization. They fully applied time difference between data computation doing something to make up. These methods still allocate tasks evenly to different nodes. However, they don’t work in some specific situations. For example, imagine that if there is one node faster than others, it must wait for aggregating gradients with another node at least. No one could complete with that fast node immediately. Actually, the effect of that faster node is equal with other nodes. It is obvious that computing resources are wasted. In a word, Only when the amount of data held by each node is different, can the maximum performance be exerted as much as possible.

B. Ring AllReduce

When Ring Allreduce is utilized in distributed deep learning by Baidu, Pytorch[8] and Horovod[16] successively designed their own distributed training framework based on the algorithm. However, the human understanding of this algorithm is not yet complete. Before starting our algorithm, let’s review Ring Allreduce in detail. Assuming there are \( n \) workers waiting for training, they are distributed on a ring and the total dataset is divided into \( n \) parts for \( n \) workers. After all workers finished calculate local gradients of subset, each worker cuts local gradients into \( n \) parts. There are two steps in communication. First, for the \( k \)-th worker, this worker will send the \( k \)-th data to the next worker, and at the same time receive the \( k-1 \)-th data from the previous worker. After looping \( N \) times, each worker will contain a copy of the final integration result. In the second stage, each worker sends the integrated part to the next worker. After the worker receives the data, it can update the corresponding part of its own data. There is no communication bottleneck among workers through Ring Allreduce. From the process of Ring Allreduce, there is a barrier[20] to synchronize all workers meanwhile distinguish two steps. Also, there are many synchronized operations during the second step. Everytime finishing changing data a round, the time consumed depends on the lowest worker. These synchronized operations are difficult to be reduced. Therefore, which we can extremely control is the first synchronization. We can make training time approach among workers to accelerate.

III. METHODS

In order to accelerate by balancing the number of tasks among different workers, we followed the idea of first experimental verification and then theoretical optimization. We proposed one implementation of static task allocation algorithm and one algorithm for adaptively adjusting the task volume. Static task allocation algorithm was achieved based on gradient accumulation to prove the feasibility of unequal tasks. After that, we proposed a self-adaptive algorithm to distribute tasks automatically without obtaining the external information of workers. We set \( w_1, w_2, \ldots, w_n \) as how many samples each worker need to train in one gradient aggregation.

A. static allocation implement

The purpose of static allocation implement is to verify that the ratio of \( w_i \) among workers is one of significant
velocity factors and has few influence on network convergence. First of all, we set the weight $w_1, w_2, \ldots, w_n$ to ensure that worker $i$ must wait for others after training $w_i$ samples in one gradient aggregation. Second, according to the ratio $\frac{w_1}{\sum_{i=1}^{n} w_i}, \frac{w_2}{\sum_{i=1}^{n} w_i}, \ldots, \frac{w_n}{\sum_{i=1}^{n} w_i}$, we assigned a corresponding proportion of training samples to each worker from the total dataset so that each worker holds unequal subdataset. Third, each worker computes independently. In one gradient aggregation, Worker $i$ draws $w_i$ samples from subdataset to calculate gradients. All workers need to accumulate gradients without back propagation. Specifically, (1) After transferring one sample to the network and getting prediction results, calculate the loss value according to the prediction result and label. (2) Use loss for back propagation and calculate parameter gradient. Accumulate the gradient instead of clearing up. (3) Repeat from (1) to (2) steps until $w_i$ samples are transferred to the network. Finally, when all workers finished computing and arrived at the synchronization barrier, they will join in Ring Allreduce to aggregate gradients with local accumulation gradients. Inevitably, total minibatch size are expanded to $\text{minibatch} = (\sum_{i=1}^{n} w_i)$. In figure 2, it is the flow chart showing the process of static allocation implement.

1) Acceleration Analysis: Worker $i$ draws $w_i$ samples from subdataset to accumulate gradients can be able to ensure that there are no remaining samples without training after one epoch. Also, it’s the most significant step to accelerate. Normally, whatever the worker is fast or slow, it must only train original minibatch samples and then wait for other workers to synchronize. It means that faster workers are stuck at synchronization barrier without any computing. The average allocation consumes resources and wastes time. However, accumulating gradient fully made use of time gap. As figure 3 shows, it delayed the faster worker entering synchronization by expanding batchsize. In the same time interval, Fast workers take it for granted. Each worker tries the best to avoid waiting for others. Ideally, the straggler accumulated least times while other workers accumulated more times. To some extent, though this procedure don’t break synchronization, it is approaching heterogeneous environment to homogeneous environment.

2) Convergence Analysis: Static allocation implement can guarantee model parameter dropping along the gradient direction and converge to stable loss. Figure 4 shows Ring AllReduce when changing allocating tasks among workers. Although the number of tasks was changed, the specific link of Ring AllReduce will not change. The only change of Ring AllReduce happens in back propagation. As formula (1) shows,

$$w_k = w_{k-1} - \eta \frac{1}{N} \sum_{i=1}^{N} \Delta f(w_{k-1})$$

(1)

$N$ represents total batchsize of all minibatch at different workers, when task allocation occurred, $N$ depends on $\sum_{i=1}^{n} \text{minibatch} + w_i$. As long as total tasks maintain abiding, $N$ and gradient direction of descent will maintain too. Therefore, the final convergence depends on model parameters’ initial weight and the whole process of training seems to be the same as equal task allocation. For some special situations, gradient aggregation SGD must be modified on the basis of the importance of samples. Setting the weighted sum of local gradients in SGD to highlight difference. That’s what we need to continue exploring later. In section 4, we set experiments to prove the convergence of static allocation.

B. Self-adaptive allocation algorithm

Static allocation laid the foundation for task allocation, but specific allocated ratio of tasks is completely adjusted based on experience. It is full of uncertainty. Besides, manufacturer information of worker or GPU generally can’t show accurate computing power. Load and network bandwidth during training will change slightly. They also influence the velocity of worker computing. Therefore, it’s difficult to allocate tasks in accordance with existing information. In this part, we proposed a self-adaptive algorithm to solve this problem based on the training information.

1) Notation:

- Gradient aggregation time: $t_1, t_2, \ldots, t_n$: In the $k$th epoch, the time to perform allreduce and update the global model parameters after the local gradient of each node is accumulated.
- Gradient computing time: $t_1^1, t_2^1, \ldots, t_n^1$: In the $k$th epoch, each node calculates the gradient of each sample before aggregation, including the sum of calculation and communication time.
- Synchronization waiting time: $t_1^1, t_2^1, \ldots, t_n^1$: In the $k$th epoch, the time that each node waits for other nodes to synchronize and aggregate.
- Total training time: $T_1, T_2, \ldots, T_n$: The total time for each node training to complete one aggregation in the $k$th epoch, and $T_i = t_i^1 + t_i^s + t_i^w$.
- Calculation speed: $v_1, v_2, \ldots, v_n$: the speed at which each node in the $k$th epoch calculates the sample gradient before aggregation. Moreover, $v_i = \frac{D}{t_i^c}$ where $D = \sum_{i=1}^{n} w_i$. $D$ represents the total number of samples.
- In the $k$th epoch, the samples computed by each node in one gradient aggregation: $w_1^{(k)}, w_2^{(k)}, \ldots, w_n^{(k)}$, the ratio of tasks obtained by each node: $\frac{w_1^{(k)}}{\sum_{i=1}^{n} w_i^{(k)}}, \frac{w_2^{(k)}}{\sum_{i=1}^{n} w_i^{(k)}}, \ldots, \frac{w_n^{(k)}}{\sum_{i=1}^{n} w_i^{(k)}}$.
- Difference in synchronization waiting time: $\Delta t_i^j = t_i^s - t_j^w$, in the $k$th epoch, the time to wait when worker $i$ synchronizes with worker $j$.
- The amount of increased samples required for the new proportion of deployment in one gradient aggregation: $u_1, u_2, \ldots, u_n$

2) Hypothesis: In order to describe the algorithm better, we made the following assumptions based on the real situation.

- Due to synchronizing before gradient aggregation and sending global model parameter gradients to all workers after aggregation, as well as there are many synchronization operations during AllReduce, so it can be approximated that all workers started and ended at the same
time in the process of AllReduce. Therefore, gradient aggregation time of all workers is equal. We set:

$$t_1^c = t_2^c = \cdots = t_n^c$$  \hspace{1cm} (2)

- Total training procedure includes three steps: computing, synchronization and AllReduce(update). All workers attain the first minibatch at the same time. After computing, they will be blocked at barrier. Integrating the first assumption, it can be approximated that the total time for each node training to complete one aggregation is equal. We set:

$$T_1 = T_2 = \cdots = T_n$$  \hspace{1cm} (3)

- To avoid modifying learning rate along with the ratio of task allocation changing, the total batchsize should be stable. Therefore, the sum of samples computed by each node in one gradient aggregation is set as a constant $C$. We set:

$$w_1^{(k)} + w_2^{(k)} + \cdots + w_n^{(k)} = C$$  \hspace{1cm} (4)

$$u_1 + u_2 + \cdots + u_n = 0$$  \hspace{1cm} (5)

3) Derivation: The self-adaptive allocation algorithm mainly relies on adjusting the calculation time of different workers, shortening the waiting time of fast workers, thereby saving overall training time. Before the start of each epoch, the system will recalculate $w$ for all workers based on the information in the previous epoch, and allocate the corresponding task amount. When calculating the gradient, the new $w$ is also used for accumulation. After 4-5 epochs of training, the ratio of allocated tasks is basically stable, that is, the amount of redistributed tasks is stopped, and the amount of tasks for subsequent training is fixed. The self-adaptation is reflected
in the amount of tasks that can be assigned by itself in each epoch. The specific idea is to use the available information in the previous epoch to get the amount of assigned tasks in the next epoch.

**Algorithm 1: self-adaptive allocation algorithm**

**Input:** Randomly specify the sample distribution ratio of each node $w_{1}^{(k)}, w_{2}^{(k)}, \cdots, w_{n}^{(k)}$, and the calculation sample gradient time on each worker $t_{s1}, t_{s2}, \cdots, t_{sn}$ is set to 0

```
for epoch do
  step 1: if node==i then
    The worker broadcasts its own statistics of the last round of calculation gradient time.
  else
    The worker accepts and updates the calculated gradient time broadcast by other workers.
  step 2: Calculate the new sample distribution ratio of all workers:
    $w_{i}^{(k+1)} = u_{i}^{(k)} = \frac{w_{i}^{(k)} / t_{i}}{\sum_{i=1}^{n} w_{i}^{(k)} / t_{i}} \sum_{i=1}^{n} w_{i}$
  step 3: Redistribute the subdataset of each worker according to the sample ratio.
    (Step 2 and step 3 could be cancelled when the ratio is not fluctuating.)
  while All data is unused do
    step 4: Proportionally draw samples from the sub-data set for training and accumulate gradients.
    step 5: Record the calculation time and enter synchronization to wait for other workers.
    step 6: Update network model parameters by AllReduce.
```

To explore which parameters are available information to accelerate, we analyzed from destination to what we chose to utilize. Our objective functions are reflected as following

$$
\min_{D} \sum_{i=1}^{n} \sum_{j=i}^{n} \Delta t_{ij} \quad \text{s.t. (2)(3)(4)}
$$

(6)

$$
T_{1}, T_{2}, \cdots, T_{n} \downarrow
$$

(7)

Formula (6) and (7) represent that our destination is minimizing time of synchronization and training. According to notations and hypotheses, difference of synchronization waiting time are expected equivalent to $\Delta t_{ij} = t_{i}^{t} - t_{j}^{t} = t_{i}^{s} - t_{j}^{s} = \frac{D_{j}}{v_{j}} - \frac{D_{i}}{v_{i}} = D \frac{w_{j}}{w_{i}} \cdot 1 - D \frac{w_{i}}{w_{i}} \cdot 1 = 0$ . Due to $\sum_{i=1}^{n} w_{i} = C$, we got the following formula:

$$
\frac{D_{uj}}{C_{v_{j}}} - \frac{D_{wi}}{C_{v_{i}}} = 0
$$

(8)

Simply, we can discover that the ratio of samples in one gradient aggregation is reciprocal to the velocity of calculating gradients between any two workers. In appendix A, we computed the solution of increased samples in one gradient aggregation. $u = [u_{1}, u_{2}, \cdots, u_{n}]$, where $u_{i}$ is:

$$
u_{i} = \frac{v_{i}}{\sum_{i=1}^{n} u_{i}} \sum_{i=1}^{n} w_{i}^{(k)} - w_{i}^{(k)}
$$

(9)

Coeccidently, the equation (8) reflected the final solution of $u_{i}^{(k+1)}$, because of $u_{i}^{(k+1)} = w_{i}^{(k)} + u_{i}$. Eventually, we can obtain the amount of change $u$ in $k+1$th epoch as the following equation according to the fifth notation:

$$
w_{i}^{(k+1)} = u_{i}^{(k)} = \frac{w_{i}^{(k)} / t_{i}}{\sum_{i=1}^{n} u_{i}^{(k)} / t_{i}} \sum_{i=1}^{n} w_{i}^{(k)}
$$

(10)

After derivation, it is found that the available information is the task amount $u_{i}$ and gradient calculation time $t_{si}$ of the previous epoch. In other words, only the task amount $u_{i}$ and gradient calculation time $t_{si}$ of the previous epoch need to be obtained to get the task amount of the next epoch. As figure 5 shows, we proposed an algorithm to achieve self-adaptive task allocation. The algorithm is improved from static allocation. The difference between them is that self-adaptive allocation algorithm computes $u_{i}$ by $u_{i}$ from previous epoch in next epoch and redistributes tasks by the ratio of $w_{i} / \sum_{i=1}^{n} w_{i}$ until $u_{i}$ stays still. The whole procedure is displayed in Algorithm 1, the most obvious modification is collecting gradient calculation time $t_{s}^{(i)}$. Each worker only get its own $t_{s}$, so they need to broadcast their own $t_{s}^{(i)}$ to other workers and receive others’ $t_{s}$. After collecting, worker $i$ calculates the ratio $w_{i}^{(k+1)}$ and updates all subsequent operations. The reason why rounding decimals of $u_{i}$ is that $w_{i}^{(k+1)}$ is integer.

The algorithm searched the most suitable ratio of sample distribution through numerical solution. Though repeating transferring samples, it will be steady in few epochs. Therefore, redistributing will stop in later epochs and worker computes based on specific $w_{i}$ in almost epochs. The algorithm will revert to static allocation. This algorithm is flexibly suitable for Ring Allreduce and its variants. The only thing should be taken care of depends on parameters in loss function. Most variants are designed on partial Allreduce or partial gradient aggregation. Therefore, changing tasks can’t influence the whole procedure.

**IV. EXPERIMENTS**

In this section, we developed a series of experiments on static allocation and self-adaptive allocation to prove that the idea on task allocation will accelerate network training. We first tested the influence of static allocation on convergence to ensure whatever the ratio of samples has been changed, models of network are unaffected in terms of convergence. It guarantees the feasibility of task allocation. Then, we manually adjusted the ratio of samples to prove training time changes with the ratio. It reflected ratio of samples is the factor of training speed indeed. Third, we demonstrated self-adaptive allocation can search the ratio to speed network training up. It shows that equal ratio is not the best ratio of samples on
Figure 5: the process of self-adaptive allocation algorithm

training speed. Existing the better ratio accelerate the convergence of models. Fourth, we added one node in cluster or changed one to display self-adaptive allocation can eliminate part of the impact to tolerant heterogeneity on synchronization. Finally, we compare self-adaption results to other algorithms in special situations. Experimental and analysis results show self-adaption is suitable for complex heterogeneous environment.

A. experiment setup

Due to the limitation of hardware resources, we used 2-3 machines in different experiments to illustrate the situation. We uses two machines configured with RTX2080ti and Tesla v100 respectively as well as one machine configured with RTX2080ti and GTX1080ti to do static allocation experiments. We uses three machines configured with 2*RTX2080ti and Tesla v100 respectively to get results of self-adaptive algorithm. In table 1, models, network and GPUs in these experiments are displayed.

B. Dataset And network

Experiments are conducted on some classical datasets and network models. We selected MNIST and CIFAR10 dataset to train network models. We train three-layer convolutional neural network called ConvNet on MNIST handwritten digit recognition dataset. The model contains two convolutional layers, two maxpooling layers and one fully connected layer. We train VGG11, VGG16, VGG19, ResNet18 and Resnet50 models on CIFAR10 to observe allocation algorithm. The configuration of the models is equal basically. The learning rate is taken to the negative 2 order of 10 and \( \text{weight decay} = 10^{-4} \). Total batchsize is controled from 2000 to 3000. Minibatchsize depends on the ratio of samples. For \( \sum_{i=1}^{m} w_i = C \), \( C \times \text{minibatch} = \text{total batchsize} \). To precisely control the ratio of samples \( w \), the value range of \( C \) can be from 20 to 30. In the later experiment, we set different value of \( w \). We will explain details at that time.

C. Effects of static ratio on convergence and velocity

For the purpose of testing different tasks allocation on different workers, we designed experiments to confirm smaller impacts on convergence and larger impacts on training speed. We did experiments with one machine with multiple cards and multiple machines with multiple cards.

Above all, we train ConvNet on MNIST dataset, ResNet18, ResNet50 and VGG11 on CIFAR10 dataset with one machine with multiple cards. The ratio of samples \( w_i \) was set to four groups including equal and unequal tasks. Except the ratio, other variables are the same. Minibatchsize is equivalent to 100, total batchsize is equivalent to 1000, learning rate is

| (a) group 1 |        |        |        |
|-------------|--------|--------|--------|
| Processor   | Intel(R) Xeon(R) Gold 5117 CPU @ 2.00GHz |
| GPUs        | Tesla V100 |
| Network     | Broadcom Inc. and subsidiaries NetXtreme BCM5720 Gigabit Ethernet PCIe |

| (b) group 2 (we have two same machines) |        |        |        |
|----------------------------------------|--------|--------|--------|
| Processor                            | Intel(R) Xeon(R) Gold 5117 CPU @ 2.00GHz |
| GPUs                                 | RTX2080ti |
| Network                              | Broadcom Inc. and subsidiaries NetXtreme BCM5720 Gigabit Ethernet PCIe |

| (c) group 3 |        |        |        |
|-------------|--------|--------|--------|
| Processor   | Intel(R) Xeon(R) Bronze 3104 CPU @ 1.70 GHz |
| GPUs        | GTX 1080ti and RTX2080ti |
| Network     | Intel Corporation Ethernet Connection (3) 1219-LM |
Figure 6: the convergence data of models results in 4 groups 5 : 5, 6 : 4, 3 : 7, 7 : 3 from one machine with GTX1080ti and RTX2080ti. The gray pillars in pictures are epochs and accuracy.

equivalent to $10^{-2}$, and weight decay is equivalent to $10^{-4}$. As figure 6 shows, the same network models with different ratio converge to same points approximately. No matter it is accuracy or training epochs, there will be no big ups and downs. Therefore, it is credible that convergence will never change due to the ratio changing.

Besides, we expand the sum of ratio and reduce the mini-batchsize to show the variety of training speed. We do two sets of experiments with one machine with multiple cards and multiple machines with multiple cards respectively. The first set of experiment is implemented on Intel(R) Xeon(R) Bronze 3104 CPU @ 1.70 GHz with RTX2080ti and GTX1080ti. The second set of experiment is implemented on two Intel(R) Xeon(R) Gold 5117 CPU @ 2.00GHz with Tesla V100 and RTX1080ti. The ratio was set to four groups too. As figure 7 and 8 shows, when the performance of GPUs and network is similar, the ratio approaches same. The larger difference of performance between faster worker and slower worker is, the larger ratio of samples is. The results show there are appropriate ratio existing in heterogeneous environment, but it is difficult to search. Therefore, it is necessary to self-adaption.

D. Speedup on self-adaptive allocation

Further, we do experiments on multiple machines with multiple cards to examine results of self-adaptive allocation.
algorithm. Multiple machines with multiple cards are obvious in computing and communication performance. We train ResNet18, ResNet50, VGG16, VGG19 and ConvNet models on two nodes with different cards respectively. We record the ratio of samples $w_1, w_2, \cdots, w_n$, Gradient computing time: $t_1^s, t_2^s, \cdots, t_n^s$ and training time $T_i$ in each epoch. In figure 9, training time of models on V100 and RTX2080ti is reduced along with the increasing of epoch. The gap between gradient computing time of two workers becomes smaller too. After 4 epochs, the ratio of samples becomes steady and the algorithm should be stopped. We make two sets of initial ratio of samples to confirm the convergence of the ratio to same point. In figure 10, we increased one machine with one card RTX2080ti to research more workers. We found it also became steady after several epochs. The speed of training increased along with epochs. From the above results, the calculating time of workers approaches to same points, the ratio of weights approaches stability and the training time approaches declining.

E. performance on heterogeneity

It is not enough to illustrate the importance of ratio $w_j$. We do the following experiments. We compared different group machines with little variety. For example, we compared V100+RTX2080ti with 2*RTX2080ti and V100+RTX2080ti with V100+2*RTX2080ti. In figure 11, under the same total batchsize, we can observe the velocity of training is increasing when adding a new card or replacing the weak card with strong card. It means that the performance of card is demonstrated to some extent.

F. Universality in complex heterogeneous environment

Through the combination of experiment and theory, we found that the existing AD-PSGD, AllReduce algorithm, etc., cannot guarantee the acceleration function in some heterogeneous environments. For example, when there are only two workers, AD-PSGD training speed is almost the same as AllReduce. As shown in Figure 12, we have drawn the convergence curve of each algorithm on GTX1080ti and RTX2080ti dual cards. It can be found that the allocation algorithm will have a significant acceleration effect. Another example, when a worker has a fast computing speed and other workers are relatively slow, the current asynchronous SGD algorithm cannot make full use of resources, which is equivalent to n slow nodes, but it can also be accelerated by...
Figure 11: the training time of models compared three groups, two machines with RTX1080ti and V100, three machines with two RTX1080ti and V100 as well as two machines with two RTX2080ti.

allocating data. In addition, there is another possibility that the allocation algorithm can be used as a plug-in of AllReduce and combined with other algorithms.

Figure 12: the training loss of models results in one machines with RTX1080ti and GTX1080ti.

Figure 13 shows the approximate speedup ratios of several algorithms. Because the allocation algorithm itself has the behavior of expanding batchsize and reducing gradient aggregation, the speedup ratio of this algorithm is based on the parameter server, when the straggler iteration time is twice that of other cards it can reach about 5.36X, and when the straggler iteration time is 5 times that of other cards, it can reach 2.75X, which is not very exaggerated, and its speedup is basically about 3.3X that of AllReduce.

Figure 13: the training speedup of models

V. RELATED WORK

Existing efforts on heterogeneous distributed deep learning algorithms can be classified into two types: algorithms based on task allocation and algorithms based on gradient aggregation.

For algorithms based on task allocation, Yang et al. [21] proposed a batch orchestration algorithm, which balances the amount of mini-batch data according to the speed of workers. FlexRR [6] addresses the straggler problem by integrating flexible consistency bounds with temporary peer-to-peer work reassignment. These methods have achieved certain results under the architecture of the parameter server [12], but there is communication bottleneck existing in parameter server. They are not very practical under the architecture of Ring Allreduce [5].

For algorithms based on gradient aggregation, DYNSGD [9] can dynamically adjust the local learning rate of training node according to the delay of the node. Zhang et al. [22] have a similar idea, it tracks the state of each gradient, and then adjusts the learning rate according to the state of the gradient. In addition to adjusting the learning rate during update, AD-PSGD [13] probabilistically reduces the effects of heterogeneity with randomized communication, by the way, this algorithm can also speed up training for some tasks in a homogeneous environment. Prague [14] uses Partial AllReduce mechanism to further accelerate training in heterogeneous environments.

Besides the work in the algorithm, it’s also possible to do some work in system to mitigate the straggler problem in heterogeneous environments. Chen et al. [3] introduce an approach of synchronous optimization with backup workers, which can avoid asynchronous noise while mitigating for the worst stragglers. Tandon et al. [19] use the method of Gradient Coding, which replicates some samples on each machine, so that each sample is repeatedly trained. In this way, the system only need to obtain the gradient of part of the training nodes to get the complete gradient. Although these methods are simple, they consume additional computing resources, so they are not a good choice.

VI. CONCLUSIONS

To cope with task allocation, we design an implementation of a static allocation algorithm based on gradient accumulation in this paper. The dataset is artificially allocated to each worker, and each worker draws samples in proportion to train to accelerate the training speed of the network in a heterogeneous environment. The static allocation training was completed on GTX 1080ti and RTX 2080ti, and the accuracy of the experiment, epoch and training duration were recorded to verify the convergence of the network model under this method and the impact on the training speed. Furthermore, in order to adapt to various training environments and workers, we proposed an adaptive allocation algorithm. We found that the proportion of task allocation is proportional to the training speed of each worker. We use this property to calculate the change in the amount of task allocation in each round. We have
verified the adaptive allocation process on multiple machines, and the training speed has changed from slow to faster. Finally, we also do experiments to verify that the speed of multi-card relative to fewer cards and the training speed of strong cards relative to weak cards are improved.

REFERENCES

[1] T. Ben-Nun and T. Hoefler. “Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis”. In: ACM Computing Surveys 52.4 (2018).

[2] J. Bouvrie. “Notes on Convolutional Neural Networks”. In: neural nets (2006).

[3] J. Chen et al. “Revisiting Distributed Synchronous SGD”. In: (2016).

[4] L. Deng. “The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]”. In: IEEE Signal Processing Magazine 29.6 (2012), pp. 141–142.

[5] P. Goyal et al. “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”. In: (2017).

[6] Aaron Harlap et al. “Addressing the straggler problem for iterative convergent parallel ML”. In: Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016. Ed. by Marcos K. Aguilera, Brian Cooper, and Yanlei Diao. ACM, 2016, pp. 98–111. DOI: 10.1145/2987550.2987554. URL: https://doi.org/10.1145/2987550.2987554.

[7] R. Hecht-Nielsen. “Theory of the Backpropagation Neural Network”. In: Neural Networks, 1989. IJCNN., International Joint Conference on. 1989.

[8] S. Imambi, K. B. Prakash, and G. R. Kanagadibamboresan. PyTorch. Programming with TensorFlow, 2021.

[9] J. Jiang et al. “Heterogeneity-aware Distributed Parameter Servers”. In: Acm International Conference. 2017, pp. 463–478.

[10] A. Krizhevsky and G. Hinton. “Learning multiple layers of features from tiny images”. In: Handbook of Systemic Autoimmune Diseases 1.4 (2009).

[11] S. Lee et al. “Primitives for Dynamic Big Model Parallelism”. In: Computer ence (2014).

[12] M. Li et al. “Scaling distributed machine learning with the parameter server”. In: ACM (2014).

[13] X. Lian et al. “Asynchronous Decentralized Parallel Stochastic Gradient Descent”. In: (2017).

[14] Q. Luo et al. “Heterogeneity-Aware Asynchronous Decentralized Training”. In: (2019).

[15] M. Roberts. “Overcoming the communication bottleneck”. In: (1992).

[16] A. Sergeev and M Del Balso. “Horovod: fast and easy distributed deep learning in TensorFlow”. In: (2018).

[17] K. Simonyan and A. Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: Computer Science (2014).

[18] C. Szegedy et al. “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning”. In: (2016).

[19] R. Tandon et al. “Gradient Coding: Avoiding Stragglers in Distributed Learning”. In: (2017).

[20] S. Xiao and W. C. Feng. “Inter-block GPU communication via fast barrier synchronization”. In: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on. 2010.

[21] E. Yang, D. K. Kang, and C. H. Youn. “BOA: batch orchestration algorithm for straggler mitigation of distributed DL training in heterogeneous GPU cluster”. In: The Journal of Supercomputing (2019).

[22] Wei Zhang et al. “Staleness-aware Async-SGD for Distributed Deep Learning”. In: Nov. 2015.

APPENDIX

Assuming that the \( k \)th epoch task allocation of workers is \( w_1^{(k)}, w_2^{(k)}, \ldots, w_n^{(k)} \), the \( k + 1 \)th epoch to be updated is \( w_1^{(k+1)}, w_2^{(k+1)}, \ldots, w_n^{(k+1)} \), and the increase is \( u_1, u_2, \ldots, u_n \). Therefore, the relationship of three variables can be expressed as

\[
\begin{align*}
    w_1^{(k+1)} &= w_1^{(k)} + u_1 \\
    w_2^{(k+1)} &= w_2^{(k)} + u_2 \\
    & \vdots \\
    w_n^{(k+1)} &= w_n^{(k)} + u_n \\
\end{align*}
\]

According to formula (8), the waiting time of \( n \) workers must be equal, and the pairwise constraint can be expressed as \( n(n-1)/2 \) equations:

\[
\begin{align*}
    \frac{Dw_1^{(k+1)}}{C_{v_1}} - \frac{Dw_1^{(k)}}{C_{v_1}} &= 0 \\
    \frac{Dw_2^{(k+1)}}{C_{v_2}} - \frac{Dw_2^{(k)}}{C_{v_2}} &= 0 \\
    & \vdots \\
    \frac{Dw_n^{(k+1)}}{C_{v_n}} - \frac{Dw_n^{(k)}}{C_{v_n}} &= 0 \\
\end{align*}
\]

However, the rank of the coefficient matrix of the linear equation system is \( n - 1 \), so the above equation system actually only has \( n - 1 \) effective equations. These \( n-1 \) equations
are extracted from the above equation system to form a new homogeneous equation system:

\[
\begin{align*}
\frac{Du_{1}^{(k+1)}}{C_{v1}} - \frac{Du_{2}^{(k+1)}}{C_{v2}} &= 0 \\
\frac{Du_{2}^{(k+1)}}{C_{v2}} - \frac{Du_{3}^{(k+1)}}{C_{v3}} &= 0 \\
\vdots & \quad \vdots \\
\frac{Du_{n-1}^{(k+1)}}{C_{vn-1}} - \frac{Du_{n}^{(k+1)}}{C_{vn}} &= 0 \\
\end{align*}
\]

(13)

Simplify to get:

\[
\begin{align*}
w_{1}^{(k)} + u_{1} &= w_{2}^{(k)} + u_{2} \\
w_{2}^{(k)} + u_{2} &= w_{3}^{(k)} + u_{3} \\
\vdots & \quad \vdots \\
w_{n-1}^{(k)} + u_{n-1} &= w_{n}^{(k)} + u_{n} \\
\end{align*}
\]

(14)

Extract the coefficient matrix to get

\[
A' = \begin{bmatrix}
\frac{1}{v_1} & -\frac{1}{v_2} & 0 & \cdots & 0 & 0 \\
0 & \frac{1}{v_2} & -\frac{1}{v_3} & \cdots & 0 & 0 \\
0 & 0 & \frac{1}{v_3} & -\frac{1}{v_4} & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & 0 & \frac{1}{v_{n-1}} & -\frac{1}{v_n}
\end{bmatrix}
\]

(15)

It can be seen that the rank of the coefficient matrix is \(n - 1\), so the system of equations has infinite solutions. According to the principle that the total number of batchsize remains unchanged, we set the total number of samples unchanged, which is the following equation

\[
w_1 + w_2 + \cdots + w_n = C = Const
\]

(16)

\[
u_1 + u_2 + \cdots + u_n = 0
\]

(17)

After the linear equations are added to the above equations, the rank of the coefficient matrix becomes \(n\), and there is a unique solution

\[
A = \begin{bmatrix}
1 & 1 & A & \cdots & 1 & 1
\end{bmatrix}
\]

(18)

\[
A = \begin{bmatrix}
\frac{1}{v_1} & -\frac{1}{v_2} & 0 & \cdots & 0 & 0 \\
0 & \frac{1}{v_2} & -\frac{1}{v_3} & \cdots & 0 & 0 \\
0 & 0 & \frac{1}{v_3} & -\frac{1}{v_4} & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & 0 & -\frac{1}{v_{n-2}} & \frac{1}{v_{n-1}} \\
0 & 0 & \cdots & 0 & \frac{1}{v_{n-1}} & -\frac{1}{v_n}
\end{bmatrix}
\]

(19)

The constant term is

\[
b = \begin{bmatrix}
\frac{w_2^{(k)}}{v_2} \\
\vdots \\
\frac{w_n^{(k)}}{v_n} - \frac{w_n^{(k)}}{v_{n-1}} \\
0
\end{bmatrix}
\]

(20)

Only need to solve the subordinate system of equations

\[
A \cdot u = b
\]

(21)

Therefore, solutions of equations: the increase \(u\) is represented:

\[
u = \begin{bmatrix}
\sum_{i=1}^{n} v_i \sum_{i=1}^{n} w_i - w_1^{(k)} \\
\vdots \\
\sum_{i=1}^{n} v_i \sum_{i=1}^{n} w_i - w_n^{(k)}
\end{bmatrix}
\]

(22)