Vol.2019-OS-145 No.8
2019/2/28

A Hybrid Simulator to Analyze Gradient Staleness Effect

Duo ZHANG[2,1] Yusuke TANIMURA[1,2] Hidemoto NAKADA[1,2]

Abstract: One of the obstacles for parallel execution of Deep Learning is the Gradient exchange overhead, and there are numerous exchange methods are proposed to mitigate the overhead. Investigating these methods with real machines requires a lot of resources. Furthermore, it is impossible to investigate the behavior under other circumstances, such as different network latency. We propose a hybrid simulator that combines gradient computation on real machine and virtual time management using discrete event simulator, that enables to accurately reproduce the behavior under arbitrary gradient exchange methods and arbitrary setup. We implemented this simulator using Python coroutine. We confirmed that we can reproduce the behavior of asynchronous gradient exchange, and it can handle 64 nodes with single node.

1. Introduction

Various methods for parallel SGD have been proposed at present. The limitation is that to analyze the behavior in the case of execution with some number of nodes, we actually have to have the number of nodes. This prevents us from investigating the behavior with a lot of nodes. In [1], we proposed an environment where we can control communication delay. However, it still require the same number of nodes as the target environment.

On the other hand, discrete event simulator can handle the arbitrary number of nodes. However, since the execution portion of the calculation is abstracted, no calculation is actually performed, and therefore the influence on the convergence of the gradient arrival delay in the parallel SGD cannot be analyzed [2].

In this paper, we propose a 'hybrid' simulator that actually perform the graident computation while managing the virtual time as the discrete event simulators. The purpose of this simulator is to manage the calculation time and communication time by using the discrete event simulator while actually performing the gradient calculation, and make it possible to simulate any number of nodes on a few nodes to perform the simulation while accurately grasping the effect of the gradient arrival delay on convergence.

The next section of this paper gives the overview of distributed machine learning systems focusing on the parameter exchange methods, the introduction of stochastic gradient descent. Section 3 presents how we implement the hybrid simulator that enables to accurately reproduce the behavior under arbitrary gradient exchange methods and arbitrary setup. Section 4 describes experiment setup and the results of the experiments. Section 5 gives summary of the paper and the future work.

[1] 産業技術総合研究所
National Institute of Advanced Science and Technology
[2] 筑波大学
Uniersity of Tsukuba

2. Background

2.1 Parameter exchange methods for large scale machine learning systems

To parallelize machine learning systems, there are two methods; **Data Parallel** and **Model Parallel**. While data parallel method simultaneously trains multiple machine learning models synchronizing each other, model parallel parallelize inside a single machine learning model. While these two methods are not exclusive each other and often used complementarily, we focus on data parallel in this paper.

To implement data parallel machine learning systems, there are two methods: parameter server based methods and direct communication method. In this paper we focus on the former, where central parameter servers manage the graident communication among the worker nodes [3][4].

2.2 Stochastic Gradient Descent(SGD)

Both statistical estimation and machine learning consider the problem of minimizing an objective function with summation form:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J_i(\theta)$$

where the parameter θ which minimizes $J(\theta)$ is to be estimated. Each sum function J_i is typically associated with the i-th observation value(for training) in the dataset.

In the machine learning algorithm, when the loss function is minimized, it can be iteratively solved step by step

through the gradient descent method to obtain a minimized loss function and model parameter values. Gradient Descent Optimization is the most commonly used optimization algorithm for neural network model training. For the deep learning model, the gradient descent algorithm is basically used for optimization training[5].

When used to minimize the above function, a standard (or "batch") gradient descent method would perform the following iterations :

$$\theta := \theta - \eta \nabla J(\theta) = \theta - \eta \sum_{i=1}^{n} \nabla J_i(\theta)/n$$

Where $\nabla J(\theta)$ is the gradient of the parameter, according to the difference in the amount of data used to calculate the objective function $J(\theta)$, the gradient descent algorithm can be divided into **Batch Gradient Descent**, **Stochastic Gradient Descent** and **Mini-batch Stochastic Gradient Descent**[6]. For the batch gradient descent algorithm, the $J(\theta)$ is calculated over the entire training set. If the data set is large, it may face the problem of insufficient memory, and its convergence speed is generally slow. The stochastic gradient descent algorithm is another case. $J(\theta)$ is calculated for a training sample in a training set. That is, a sample is obtained and a parameter update can be performed. Therefore, the convergence speed will be faster, but there may be fluctuations in the value of the objective function because high-frequency parameter updates result in high variance. The Mini-batch stochastic gradient descent algorithm is a compromise solution. Selecting a small batch of samples in the training set to calculate $J(\theta)$ can ensure that the training process is more stable, and that the batch training method can also use the advantage of matrix calculations. This is the most commonly used gradient descent algorithm. We focus on **Mini-batch stochastic gradient descent** in this paper.

2.2.1 Synchronous Stochastic Gradient Descent(SSGD)

As we mentioned in the 2.1 section, data parallel machine learning methods could be categorized into two types; synchronous methods and asynchronous methods. Synchronous SGD means that each computer used for parallel computing calculates the gradient value after calculating its own batch, and sends the gradient value to parameter server. The parameter server obtains the gradient average and updates the parameters on the parameter server.

As shown in Figure 1, it can be seen as four computers. The first computer is used to store parameters, share parameters, and share calculations. It can be simply understood as a memory and computing shared area, that is the parameter server job; The other three computers are used for parallel computing to calculate the gradient value, which is worker task.

The disadvantage of this method of calculation is that each gradient update must be waited until all workers A, B, and C have been calculated before updating the parameters. That is, the speed of iterative update depends on the slowest worker among the three A, B, and C workers. Therefore, the method of simultaneous update is recommended to have the same computing power.

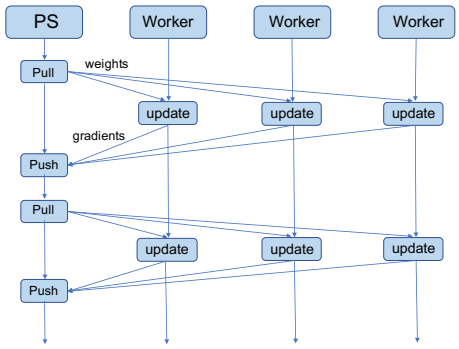

Fig. 1: Synchronous Stochastic Gradient Descent.

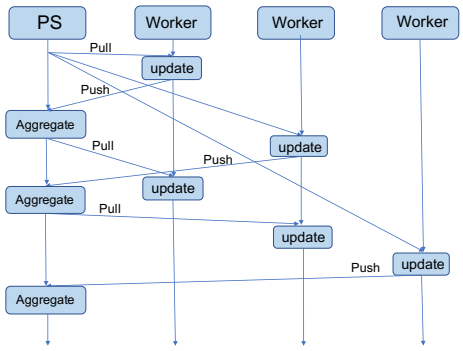

Fig. 2: Asynchronous Stochastic Gradient Descent.

2.2.2 Asynchronous Stochastic Gradient Descent(ASGD)

As long as the parameter server receives the gradient value of a machine, it updates the parameters directly without waiting for other machines. This iterative method is relatively unstable, and the convergence curve is more severe, because when the worker A updates the parameters in the parameter server, it may be that the worker B is still using the old parameter values of the previous iteration.

3. Simulator

3.1 Overview

This simulator uses Python language and implemented using Python's coroutine function. We choose Python because many neural network frameworks are currently implemented in Python. The reason for using the coroutine is to facilitate the description of the algorithm by the user.

3.2 Network Simulation Model

Since the purpose of this simulator is not precise simulation of network communication, we do not need to express congestion on switches and network paths. However, the overlapping of network and communication, and the independence of sending and receiving are important characteristics in today's computer system, so we need to express them.

Therefore, the network model in this simulator is as follows:

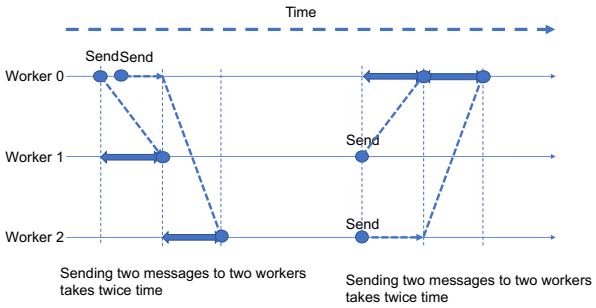

Fig. 3: The Basic Model in Simulator

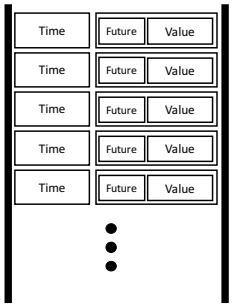

Fig. 4: The Structure of Scheduler

- Send, receive, and calculate can be performed independently.
- Transfer to multiple targets is serialized. That is to say, if you want to send information to multiple targets, the time spent is proportional to the number of objects received.
- Similarly, receiving data from multiple nodes is also serialized. That is if you want to receive information from multiple sources, the time spent is proportional to the number of objects sent.

This basic model is shown in Figure 3.

3.3 The Configuration of Simulator

The simulator consists of a Scheduler and multiple Walkers. The Scheduler is implemented as a coroutine, and there is only one Scheduler in the simulator. Walker means everything scheduled by the Scheduler. Each Walker is also implemented as a coroutine.

3.3.1 Scheduler

The Scheduler has the role of managing the virtual time and sequentially executing operable Walker. The key component of the Scheduler is the Sorted Dictionary called Event Queue. This data structure is used to reserve an event to occur at a certain virtual time.

This structure is shown in Figure 4.

Events are implemented by binding values in the Future. The key is the virtual time, the value is the value to be bound to a Future and the Future. Since it is a Sorted Dictionary, it is automatically sorted in ascending order of virtual time, so the first one is an event to occur next.

The operation of Scheduler is shown in Figure 5. Basically, the Scheduler works only when all the walkers are blocking. If all the walkers are blocking, in order to cause

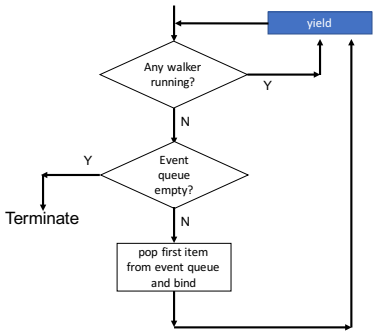

Fig. 5: The Operation of Scheduler

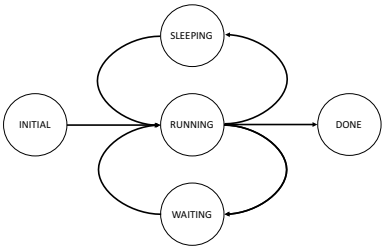

Fig. 6: The State Transition Diagram of Walker

Table 1: The method of Walker.

wait	Blocks until someone call notify
notify	Release the waiting Walker
sleep	Sleep for the specified simulation time

the next event, take out the value to be bound with the Future from the beginning of the Event Queue and execute the binding. As the result, it will unblock any of the walkers and make it proceed the operation.

3.3.2 Walker

A state transition diagram of Walker is shown in Figure 6. Walker has five states: INITIAL, RUNNING, SLEEPING, WAITING, and DONE.

When Walker starts, the state is INITIAL. After initialization, it became RUNNING. Walker implements wait and notify. The meaning of these methods is similar to wait and notify of Java objects, and when the wait is executed, it blocks until notify is called. In addition, the sleep method is implemented in Walker. When this method is executed, the Walker sleeps within the specified simulation time.

Methods of Walker are summarized in Table 1.

3.3.3 Worker

The Worker is an object representing a computer in the simulator, and it is implemented as a subclass of Walker. The Worker has two Walkers in addition to the Worker itself: a Sender Walker that expresses a send buffer, and a Receiver Walker that expresses a receive buffer.

The Worker has a send method for sending data and a recv method for receiving data. When sending, deliver the data to Sender Walker. When receiving, call a method of Receive Walker to receive data. These will be described in detail later.

The Worker has an execute function that actually performs calculations. This function takes a reference to a function and a list of arguments, executes the function, and

Table 2: The Method of Worker.

send	Send data to other Workers
recv	Receive data from other Workers
recvNB	Receive data in non blocking fashion
execute	Execute the function and sleep for the actual time spent

Table 3: The Method of Simulator.

addWorker	Add Worker. At that time, specify the Worker name, the function to create a Worker, and its argument.
start	Execute the simulator

returns the result. In doing so, measure the time taken to execute the function, and proceed the simulation time using the sleep method. Walker methods are summarized in Table 2.

3.3.4 Sender Walker

The Sender Walker represents a send buffer. Sender Walker has a queue to hold send requests. Send requests received from the Worker are queued.

The Sender Walker scans the send request queue from the beginning and checks the status of the Receive Walker of the destination. If the destination is available, send the data and sleep for the time it takes for sending. If the destination is not available, it will send a "request for notification when available" and proceed to the next entry.

3.3.5 Receive Walker

The Receive Walker represents the receive buffer. Receive Walker is WAITING when not in communication and becomes RUNNING during communication. When it receives from Sender Walker, it sleeps only for the time it takes. As a result, when receiving data from a certain node, it indicates that it can not communicate with other nodes. When receiving is completed, notify is sent to the requested source to indicate that it is available.

Receive Walker must also communicate with the Worker main body. When the Worker receives the data and the data has not yet arrived, a Future representing the receive status is newly created and the Worker blocks by waiting for the value of that Future. When data is received, bind the received value to that Future. As a result, the received worker returns to the RUNNING state.

3.4 Simulator

To use the Simulator, each node is implemented as a subclass of Worker, and the object is registered in the simulator. Programmers need to implement the "run" method of the Worker only. Simulator automatically calls the run method of Worker on startup. Methods of the class Simulator are summarized in Table 3.

Program samples are shown in Figure 7 as an example. This program defines two workers that communicate with each other. These perform transmission and reception repetitively.

```
class Ping(Worker):        # Define a worker inheriting Worker
    async def run(self):
        data = 'dummy'
        for i in range(3):
            self.send('pong', data, delay=0.1)  # send data
            data = await self.recv()            # recv data

class Pong(Worker):        # Define another worker inheriting Worker
    async def run(self):
        for i in range(3):
            data = await self.recv()            # recv data
            self.send('ping', data, delay=0.1)  # send data

sim = Simulator(log=True)     # create a Simulator
sim.addWorker('ping', Ping)   # create Ping Worker, named 'ping'
sim.addWorker('pong', Pong)   # create Pong Worker, named 'pong'
sim.start()                   # start the simulation
```

Fig. 7: The Description of Ping Pong by Simulator

4. Experiments

For this simulator, what we want to know is whether the simulator can reproduce gradient updates as in real multi-worker scenarios, and we want to know the scalability of the simulator, that is, how many workers can be simulated at most. To evaluate the performance of the simulator, we performed experiments with several latencies settings and several workers settings.

Table 4: Experiment Configurations.

	#workers	#latency
Sequential mode(baseline)	1	0
Synchronous mode	2/4	0, 0.001, 0.01, 0.1
Asynchronous mode	2/4	0, 0.001, 0.01, 0.1

Table 5: The Experimental Environment.

CPU	Intel(R) Core i7-7700 3.60GHz * 8
GPU	Nvidia GeForce GTX 1080Ti 11GB
Memory	16 GB
Operating System	Ubuntu 16.04

4.1 Network architecture and Datasets

We chose TensorFlow code implementation on CIFAR-10[7] for our neural network architecture; specifically, we trained CIFAR-10 dataset. We run the image classification problem of CIFAR-10[8], which consists of 60,000 32*32 RGB color pictures for a total of 10 categories. There are 50000 training images and 10000 test images(cross-validation). We fixed the minibatch size per worker as 100. We have implemented parameter server based data-parallel machine learning on the simulator, in synchronized and asynchronized fashion.

4.2 Experiments Settings

We tested two modes, synchronous mode, and asynchronous mode. Each mode has two options, one is the number of workers, and the other is the latencies.

In summary, the experiments settings are these: First, We tested 1 worker's synchronous mode with the latency of 0 as the baseline. And 2 worker's and 4 worker's synchronous mode and asynchronous mode with the latencies of 0, 0.001, 0.01 and 0.1 seconds. We also tested the case where the worker is 8, 16, 32 and 64 with the latencies of 0.001 seconds in asynchronous mode. Table 4 shows the experiments setups. Table 5 shows the experimental environment.

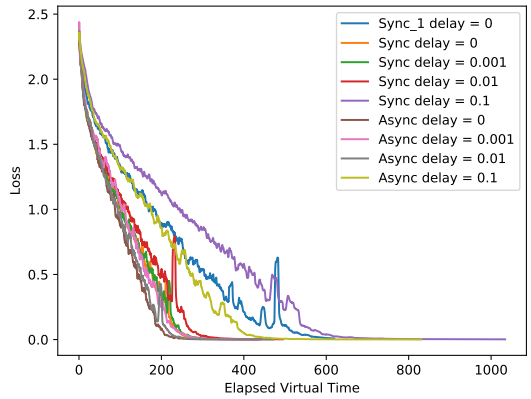

Fig. 8: Comparison of Synchronous and Asynchronous Experimental Results of 2 Workers, X-axis is the Elapsed Virtual Time.

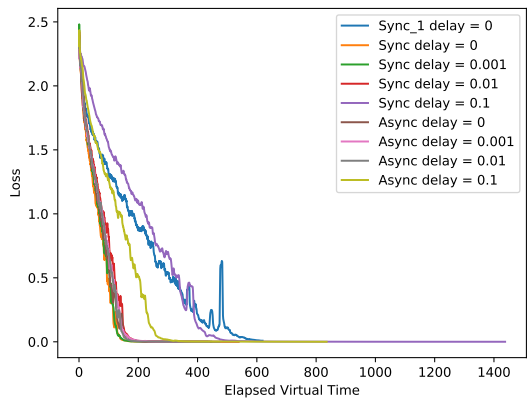

Fig. 9: Comparison of Synchronous and Asynchronous Experimental Results of 4 Workers, X-axis is the Elapsed Virtual Time.

4.3 Results of Experiments

Figure 10 shows the comparison of synchronous and asynchronous experimental results of 2 workers, Figure 11 shows the comparison of synchronous and asynchronous experimental results of 4 workers. The x-axis is the **iterations**, the y-axis is the loss to perform gradient exchange.

Also, Figure 8 shows the comparison of synchronous and asynchronous experimental results of 2 workers, Figure 9 shows the comparison of synchronous and asynchronous experimental results of 4 workers. The x-axis is the **elapsed virtual time**, the y-axis is the loss to perform gradient exchange.

Finally, Figure 12 and Figure 13 show the asynchronous experimental results of 8, 16, 32 and 64 workers with the latency of 0.001 seconds.

4.4 Discussion on the Results

From these results, it can be seen that: Firstly, compared with elapsed virtual time in synchronous mode, it can be seen that four workers converge faster than two workers, and

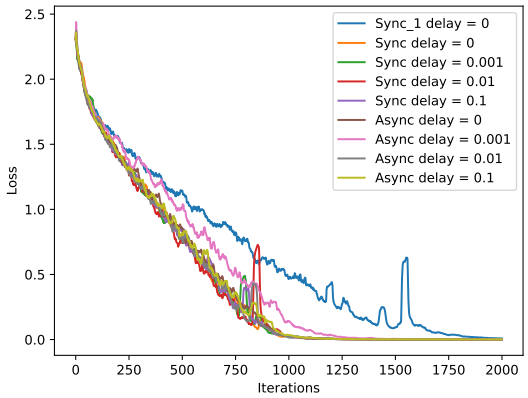

Fig. 10: Comparison of Synchronous and Asynchronous Experimental Results of 2 Workers, X-axis is the Iterations.

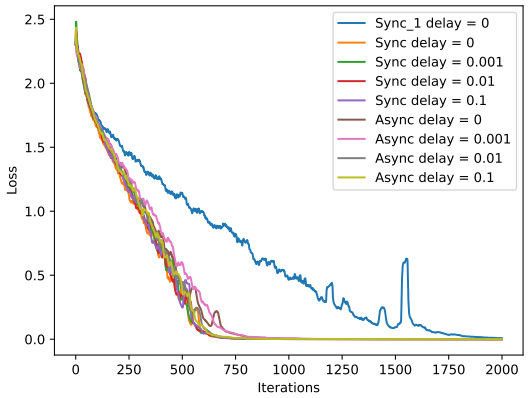

Fig. 11: Comparison of Synchronous and Asynchronous Experimental Results of 4 Workers, X-axis is the Iterations.

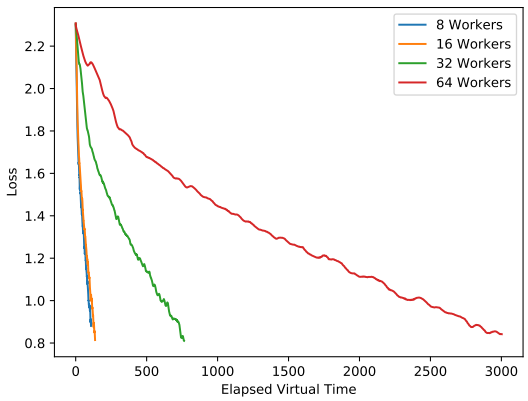

Fig. 12: The Experimental Results of 8, 16, 32 or 64 Workers in an Asynchronous Mode with latency = 0.001, X-axis is the Elapsed Virtual Time.

three kinds of latency: 0, 0.01, 0.001 curves of two workers and four workers basically coincide. The latency of 0.1 is not particularly desirable, probably because of the latency of 0.1 seconds is really a little bit big.

Comparing with elapsed virtual time in asynchronous

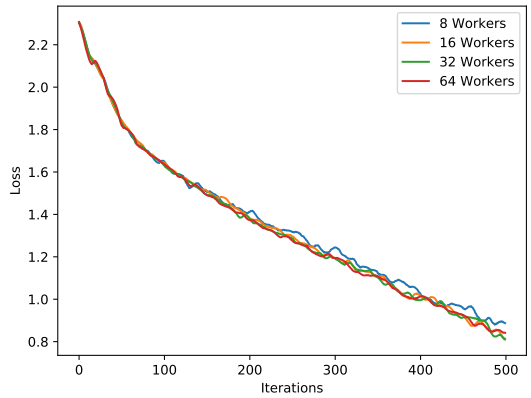

Fig. 13: The Experimental Results of 8, 16, 32 or 64 Workers in an Asynchronous Mode with latency = 0.001, X-axis is the Iterations.

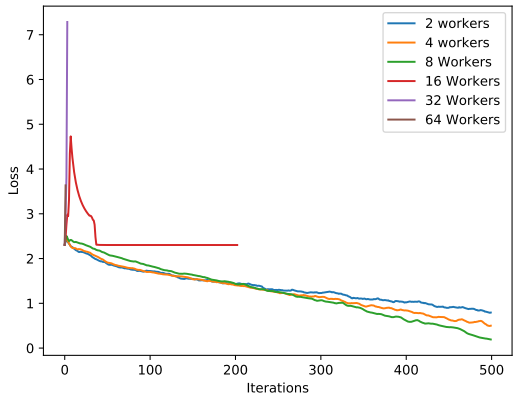

Fig. 14: A finding of gradient update strategy from experiments.

mode, the situation is basically the same as that in synchronous mode. In the three cases of latency 0, 0.001 and 0.01, the convergence speed of the synchronous mode and asynchronous mode is the same, but compared with the latency of 0.1, the asynchronous mode can converge faster.

Comparing with iterations in synchronous mode, it can be seen that four workers converge faster than two workers. Unlike iterations, the curves of four latencies of two workers and four workers basically coincide. The performance of these two workers in asynchronous mode is basically the same as that in synchronization mode.

Finally, compared with the iterations of 8, 16, 32, and 64 workers, we can see that the convergence curve is basically the same, so we can know that the number of workers added in the asynchronous mode has no effect on the loss update. Secondly, we can prove that the simulator can operate at least 64 workers, and more workers need to be verified later.

It should be noted that the updating gradient method used in the network model is modified when 8, 16, 32, and 64 workers are tested. When the network model is used before, it works well at 8 workers, but for 16 workers, it does not improve loss, and when 32 and 64 workers are tested, the gradient explosion will occur. Figure 14 shows the results. We could find that this strategy is not scalable, without actually testing the strategy with large cluster. However, in order to verify whether the simulator can run more than 64 workers, we changed the updating gradient method in the network model.

5. Conclusion

In this paper, we propose a simulator, which combines gradient computation on the real machine and virtual time management using discrete event simulator that enables to accurately reproduce the behavior under arbitrary gradient exchange methods and arbitrary setup. Also, we have quantitatively measured the loss to perform gradient exchange with several settings. As a result, we have found the followings:

1) In this simulator, the number of workers that can be run can reach at least 64. That is, the scalability of the simulator seems well.

2) The convergence speed of four workers is generally better than two workers.

3) From the results of this experiment, the asynchronous mode is better than the synchronous mode.

Our future work include the followings:

- Test more advanced asynchronous data exchange methods, such as gossip algorithm.
- Modify the simulator it can simulate in parallel.

Acknowledgments This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO). This work was supported by JSPS KAKENHI Grant Number JP16K00116.

References

[1] Zhang, D., Tanimura, Y. and Nakada, H.: Asynchronous Deep Learning Test-bed to Analyze Gradient Staleness Effect, *IEICE technical report,vol. 118, no. 165, CPSY2017-29 , pp. 199-204* (2018).

[2] Zhang, D., Li, M., Tanimura, Y. and Nakada, H.: A study on Network Structure and Parameter Exchange Method in large-scale Cluster for Machine Learning, *IEICE technical report,vol. 117, no. 153, CPSY2017-29 , pp. 145-150* (2017).

[3] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J. and Su, B.-Y.: Scaling Distributed Machine Learning with the Parameter Server, *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Berkeley, CA, USA, USENIX Association, pp. 583–598 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2685048.2685095⟩ (2014).

[4] Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K. and Ng, A. Y.: Large Scale Distributed Deep Networks, *NIPS 2012: Neural Information Processing Systems* (2012).

[5] Stochastic Gradient Descent: `https://en.wikipedia.org/wiki/Stochastic_gradient_descent//`.

[6] Ruder, S.: An overview of gradient descent optimization algorithms, *arXiv:1609.04747* (2017).

[7] cifar10-model: `https://www.tensorflow.org/tutorials/deep_cnn//`.

[8] cifar-10: `https://www.cs.toronto.edu/~kriz/cifar.html/`.