Cluster Computing (2020) 23:2287–2300
https://doi.org/10.1007/s10586-020-03144-9

Towards an optimized distributed deep learning framework for a heterogeneous multi-GPU cluster

Youngrang Kim[1] · Hyeonseong Choi[1] · Jaehwan Lee[1] · Jik-Soo Kim[2] · Hyunseung Jei[3] · Hongchan Roh[3]

Received: 29 November 2019 / Revised: 3 May 2020 / Accepted: 21 June 2020 / Published online: 11 July 2020

Abstract
This paper presents a novel "Distributed Deep Learning Framework" for a *heterogeneous* multi-GPU cluster that can effectively improve overall resource utilization without sacrificing training accuracy. Specifically, we employ a hybrid aggregation approach using a parameter-server and all-reduce schemes in order to address potential performance degradation problems in running deep learning applications on a heterogeneous computing system. In addition, we design and implement an asynchronous large mini-batch training mechanism to maintain training accuracy for asynchronous data-paralleled deep learning processing with enhanced collective communication capability based on MPI. We successfully implement our proposed framework on TensorFlow and perform extensive experiments in both of homogeneous and heterogeneous computing systems. Evaluation results show that our proposed framework can improve computing performance by decreasing I/O bottlenecks, and effectively increasing the resource utilization in the heterogeneous multi-GPU cluster.

Keywords Data parallel · Distributed deep learning · Heterogeneous cluster · Large-scale deep learning

1 Introduction

Recently, *distributed deep learning* frameworks have been proposed [1] to accelerate overall deep learning computations by exploiting multiple GPUs and multiple computing nodes. Typically, distributed deep learning mechanisms can be classified into asynchronous and synchronous aggregations based on the execution timing of the operations. Also, it can be further categorized into parameter-server [2] and all-reduce [3] schemes depending on the methods of exchanging data for the aggregation among training workers. However, employing combinations of these distributed deep learning mechanisms on top of a *heterogeneous* multi-GPU cluster may result in lower computing resource utilization.

In the case of synchronous training, other training workers may have to wait a substantial amount of time due to relatively slow workers (*stragglers*) which results in lower computing performance. To address such problems, Ho et. al. proposed a *Stale-Synchronous Parallel Parameter Server* [4], which worked by specifying the staleness threshold. Each worker maintained its difference in the number of training iterations compared to the slowest worker below the staleness threshold. However, even with this approach, the total computing performance would still degrade because workers had to delay the computation for slower workers.

✉ Jaehwan Lee
jlee@kau.ac.kr

Youngrang Kim
kimyr207@gmail.com

Hyeonseong Choi
chyon794@gmail.com

Jik-Soo Kim
jiksoo@mju.ac.kr

Hyunseung Jei
hsjei@sk.com

Hongchan Roh
hongchan.roh@sk.com

[1] Korea Aerospace University, Goyang-si, Republic of Korea

[2] Myongji University, Yongin-si, Republic of Korea

[3] SK Telecom ML Infra Lab., Seongnam-si, Republic of Korea

To address this problem, we have designed a novel distributed deep learning framework that can efficiently increase the usage of computing resources on the heterogeneous multi-GPU cluster without degrading training accuracy. In addition, we have implemented our proposed design on a well-known distributed deep learning framework, Google's TensorFlow [5]. Specifically, the main contributions of our paper are as follows:

- First, we propose a *hybrid* design with parameter-server and all-reduce schemes. Our proposed design effectively utilizes asynchronous parameter-server aggregation for inter-worker node communications, and synchronous all-reduce method for local aggregation among intra-node GPUs. Also, we add a local parameter server process between the worker and parameter server to reduce network communication overhead. With this structure, the overall computational resource usage rate and performance on the heterogeneous cluster can be improved.
- Second, we propose an *asynchronous* large mini-batch training mechanism to guarantee the learning accuracy. Large mini-batch training [6] updates the parameter when the training of a specific sized mini-batch is completed. We distribute parameter update cycles according to the performance ratio of each worker. When the training is finished for the cycle, the worker receives the updated parameter from the parameter-server and continues to the next large mini-batch training process.
- Third, we exploit the *MPI (Message Passing Interface)* [7] to support the effective communication of the proposed system, most of which is composed of collective communication patterns. Because the TensorFlow API cannot efficiently support the MPI communications, we convert the MPI API to the operation of TensorFlow using `py_func`.

To evaluate our proposed framework, we use the ResNet-50 model [8] to train ImageNet data sets [9], and we employ four different types of GPUs. First, we use only a single GPU of each type to investigate the performance trends based on the number of images processed per second. Secondly, we have performed cluster-level experiments using homogeneous and heterogeneous multi-GPU clusters with distributed TensorFlow and our proposed framework. Our experimental results demonstrate that the proposed design improves the overall performance by up to 18% and 10% in homogeneous and heterogeneous computing clusters respectively, compared to the existing Distributed TensorFlow. In addition, unlike Distributed TensorFlow, our proposed framework can achieve average GPU performance similar to that of a single GPU which indicates the improved utilization of computing resources in the cluster.

2 Background

2.1 Distributed deep learning

To accelerate deep learning computation, developers and researchers have generally used multiple GPUs per node and a large-scale multi-GPU cluster that can reduce the overall training time. The most popular method of distributing the deep learning workload is *data-paralleled* deep learning [1]. In data parallelization, additional aggregation processing is executed to reflect the contents learned by each GPU in the whole training. Data-parallel deep learning can be classified into asynchronous and synchronous ways depending on the aggregation execution timing. After summing or averaging gradients computed at each GPU, an aggregation operation is performed, and all training parameters are subsequently adjusted. Data-parallel deep learning can also be divided into parameter-server [2] and all-reduce [3] schemes depending on the method of exchanging data for the aggregation between GPUs. Distributed TensorFlow [10] is one of the most popular distributed deep learning frameworks that can support the parameter-server-based data-parallel mechanisms (as depicted in Fig. 1). In the parameter-server scheme, overall processes are divided into two groups-*parameter servers* and *workers*. Parameter servers basically update global model parameters with gradients computed by workers and send the latest model parameters to workers. Workers then receive model parameters from parameter servers and compute their local gradients. After successfully completing the computation of gradients, workers send the computed gradients to parameter servers.

On the other hand, in the case of the *all-reduce* scheme, workers send and receive parameters and gradients with each other and perform aggregations [3] without any centralized control (e.g., parameter servers) as shown in Fig. 2.

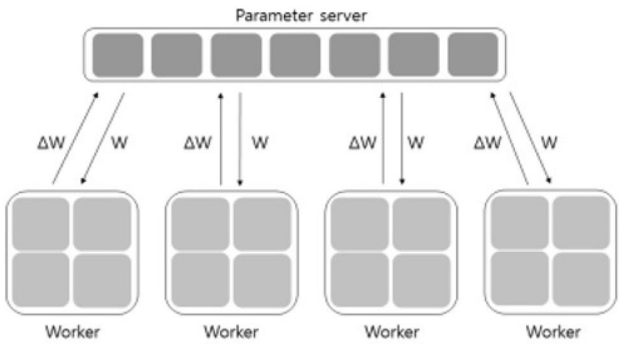


Fig. 1 Architecture of parameter-server method

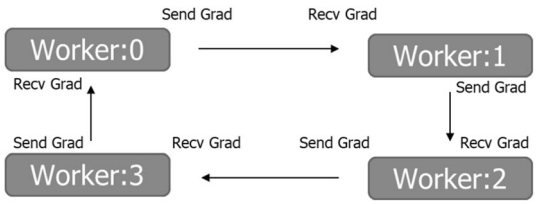


Fig. 2 Architecture of ring all-reduce method

2.2 Parameter update state

Due to the nature of the distributed deep learning process, an aggregation operation must be performed (either synchronously or asynchronously). In a synchronous aggregation mechanism, the aggregation operation is performed for each iteration, and after updating parameters such as weights, synchronization is executed by broadcasting the parameters to all workers. However, with the asynchronous method, the parameter server aggregates the learning results through sequential receipts from workers, updating the parameters, and immediately sending updates to the workers. Therefore, unlike the synchronous aggregation method, all workers may learn with different parameters with each iteration. Because of these characteristics, for a heterogeneous computing system, workers whose computational performance is relatively lower may have older parameters that can affect the overall training accuracy.

To solve such problems, Q. Ho proposed a Stale-Synchronous Parallel Parameter Server [4]. It works by specifying the staleness threshold, and if the difference of the learning clock with the slowest worker is greater than the staleness threshold, the worker pauses training. The paused worker resumes training again when the clock difference with the slowest worker become less than the staleness threshold. Using this method can lower the differences between the training clocks of the workers. However, a worker with higher computing performance must still delay its own training, so that it is impossible to fully utilize the computing resources such as the GPUs of all workers.

In the case of Facebook, a large mini-batch training experiment was performed to analyze learning accuracy according to different batch sizes during distributed deep learning [6]. The large mini-batch training carried out by Facebook worked synchronously and cluster trained an 8k batch for a single iteration using 128 GPUs that trained 64 mini-batches for each GPU. When the training used batch sizes less than 8k, the change of loss per epochs was almost identical, but the experiment demonstrated that the rate of change of loss decreased when using batches larger than 8k. However, this experiment was synchronous, which may have led to poor resource utilization in the heterogeneous computing cluster.

2.3 Message passing interface (MPI)

MPI is a standardized data communication library for message passing parallel programs. It has a high portability of different systems, and a manager that directly manages processes to internally join in communication operations [7]. MPI allocates jobs on a per process basis and the processes send and receive messages with other processes in a set called a *communicator*. The communication methods used by MPI are classified into point-to-point and collective communications. Point-to-point communication refers to a method in which one reception process corresponds to a single transmission process. Collective communication is performed by all processes of the *communicator*. For example, point-to-point communication includes MPI_Send and MPI_Recv, whereas MPI_Bcast and MPI_Reduce are included in collective communication. As the message size increases, bandwidth can be used more efficiently. Therefore, it is important to be able to send the largest possible message when using MPI. For distributed deep learning, without optimizing the data transfer for gradients and parameters, the computation performance of the training will be reduced because of the network bottlenecks. These data transfer processes can perform efficient distributed deep learning by using the MPI communication routines.

2.4 Horovod

Distributed TensorFlow has problems with scalability due to communication overhead. For example, when training is performed using 128 GPUs, about half of the computing resources were not available (i.e. 50% resource utilization). To address this problem, Uber developed Horovod, which can reduce this communication overhead and achieve high scalability [3]. Specifically, Horovod utilizes the ring-type all-reduce method, and the collective communication of MPI and NCCL to optimize the network usage. All-reduce can efficiently use the network when the data used for communication is sufficiently large. However, it is not optimized for small sized data. Therefore, Horovod applies an algorithm called *Tensor Fusion* that can integrate several tensors before Horovod's all-reduce is called. This can result in improved network usage efficiency. The Tensor Fusion provides 65% better performance than unoptimized network, and Uber has achieved 88% resource utilization when training Inception V3 and ResNet-101 models using 128 GPUs.

2.5 Python asyncio

When using multiple threads with Python, only a single thread can access the Python object due to "Global Interpreter Lock (GIL)". This is a type of mutex used for the memory management in Python programs. Therefore, only a single task can actually work even if the program itself is implemented with multi-threading. In the case of distributed deep learning, many workers can communicate with a single global parameter server simultaneously. However, because of GIL, the global parameter server can receive the gradient from a specific single worker and execute training using the gradient, which prevents it from immediately continuing to receive the next worker's gradient. This can cause delays in other workers' I/O operations, which can result in a significant increase to the overall training time. To address this problem, Python supports a library called *asyncio*.

Asyncio is a Python library supported from Python 3.4 [11]. By using asyncio, CPU computations and I/O operations can be effectively overlapped which can reduce the overall I/O bottleneck. Python has added two functions, async and await, for using asyncio. Async is a function for declaring an asynchronous co-routine. Await is a function for waiting for the completion of the next asynchronous co-routine and the shift to other asynchronous co-routines. Asynchronous co-routines declared by async are scheduled by the event loop obtained using `get_event_loop()`. The asynchronous co-routines waiting for completion through await are stored in a queue. The event loop assigns the CPU to a completed asynchronous co-routine that requires CPU computations. In this way, asyncio can parallelize CPU tasks and I/O operations.

However, most Python functions are based on blocking I/O operations rather than asynchronous co-routines. To execute these blocking I/O functions asynchronously, Python supports the `run_in_executor` function of the event loop. The `run_in_executor` function allows blocking I/O operations to be run in parallel with a current asynchronous co-routine by executing a blocking I/O task through another thread or process. The network I/O overhead caused by the transmission of parameters in a distributed deep learning system can also be a substantial performance degradation problem. However, by leveraging the asyncio functionality in Python, we can also effectively overlap the training operations and network I/O tasks in the distributed deep learning which can potentially contribute to the improved scalability and throughput.

3 Motivations

As we discussed in Sect. 2.1, data-parallel distributed deep learning frameworks can be divided into parameter-server and all-reduce schemes.

The parameter-server method collects gradients computed in each GPU from the CPU or GPU memory, executes an aggregation operation, and then broadcasts the aggregated gradients to all GPUs. As a result, all GPUs are periodically synchronized. This creates an advantage in that all GPUs share the same parameters when using the parameter-server scheme synchronously, and therefore training accuracy can be guaranteed. However, when using a heterogeneous computing system where the performance of each GPU might differ, the overall computing performance may be degraded due to stragglers (i.e. GPUs with relatively lower processing performance). With asynchronous aggregation, this potential performance degradation problem may be addressed since faster GPUs do not have to wait for slower GPUs by skipping the synchronization step for every iteration. However, this can still cause network bottlenecks by concentrating I/O operations on specific devices that perform aggregations such as parameter servers.

In the all-reduce scheme, it is possible to avoid network bottlenecks by distributing I/O operations to multiple devices to execute peer-to-peer communications without centralized communication as in the parameter-server method. However, because all GPUs can perform aggregations after computing gradients, only synchronous operations are possible. Therefore, overall computing performance can be degraded by a worker with the lowest computing performance in a heterogeneous computing cluster.

To address the problems inherent in the existing data-parallel distributed deep learning schemes, we have designed a novel distributed deep learning framework that can effectively increase the computing resource utilization on a heterogeneous multi-GPU cluster without degrading the training accuracy.

4 Architecture and implementation

In this section, we propose a method to perform efficient distributed deep learning when using a heterogeneous multi-GPU cluster. Our proposed solution uses a hybrid approach that employs parameter-server and all-reduce methods, and it executes aggregation through collective communication of MPI. In addition, we make full use of computational resources in the heterogeneous cluster

without lowering accuracy by using asynchronous large mini-batch training.

4.1 Hybrid-aggregation for heterogeneous capability

Distributed TensorFlow uses the parameter server method, which is the most traditional distributed deep learning method. For multi-GPU distributed processing, Distributed TensorFlow creates worker processes for each worker node. The worker aggregates the gradients computed by GPUs on that worker node. The local aggregation is usually performed by a CPU. After the local aggregation, a worker sends aggregated gradients to the parameter server for global aggregation, and receives the updated parameter to broadcast to all GPUs. Another method is to create worker processes for each GPU on the worker node. Without local aggregation, each worker process sends computed gradients directly to the parameter server. With Uber's Horovod, which uses ring-type all-reduce, worker processes are created for each GPU. All worker processes in a cluster communicate in a peer-to-peer method between other worker processes without the parameter server. However, when we use distributed deep learning mechanisms with a heterogeneous computing cluster, the following performance degradation problems can occur.

- First, with the asynchronous parameter- server method, a network bottleneck can occur because I/O operations from workers are concentrated on the parameter server.
- Second, I/O bottlenecks can be solved when performing local aggregation after a single process uses a multi-GPU in each worker node. However, GPU computations are delayed because aggregation is computed on the CPU.
- Third, when using the all-reduce method, network I/O can be solved, but workers have to wait for the worker possessing the relatively lower computing performance.

To address these problems, we propose a hybrid structure that uses the all-reduce and parameter-server methods together, as shown in Fig. 3. With this architecture, each worker node performs local aggregation via the intra all-reduce method. Computing the aggregation with all-reduce uses GPUs rather than CPUs, so that it can perform computations faster than the local aggregation of Distributed TensorFlow. To investigate the effects of our local aggregation using the all-reduce method, we conduct an experiment comparing the throughput of the gather and all-reduce methods. When using four TITAN Vs, the worker using the gather method processes 117 images/s. However, the worker using the all-reduce method processes 145 images/s (representing 24% improvement in image processing). This result demonstrates that using the all-reduce method can effectively reduce the communication overhead. If the size of the gradient data is very large or the bandwidth of the network is low, in one iteration, the I/O ratio increases and the operation of the next iteration may be significantly delayed. Therefore, we propose to overlap the I/O and computation to avoid performance degradation by the network. We add the local parameter server process to our Hybrid-Aggregation system. The local parameter server is created on each worker node. When the worker processes complete the gradient aggregation, the local parameter server sends it to the global parameter server. While the local parameter server executes I/O with the global parameter server, the worker process can perform the operation of the next iteration without waiting, so the total computation time can be reduced. In addition, instead of directly sending the updated parameter to the worker process from the global parameter server, after broadcasting the parameters to each local parameter server, the local parameter servers broadcast it to the sub worker processes. With this architecture, we can effectively reduce network usage.

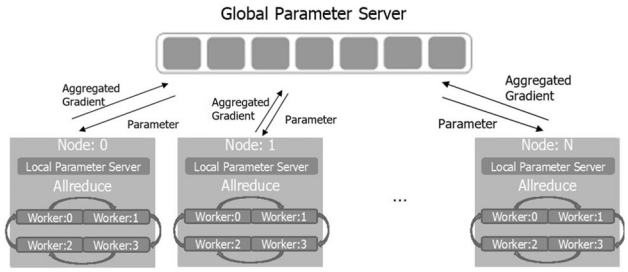


Fig. 3 System architecture of proposed Hybrid-Aggregation method

4.2 Asynchronous large mini-batch training

To address the potential training accuracy degradation problem, we propose an effective asynchronous large mini-batch training strategy as follows. The parameter server executes only executes aggregation with gradients received from workers. After aggregation execution, unlike the existing asynchronous method, the parameter server does not forward updated parameters to workers. Also, workers do not update parameters, and they just train with new mini-batches. When aggregation is executed as much as the large mini-batch of the specified size, the parameter server updates the parameters and simultaneously broadcasts all the updated parameters to the workers. Figure 4 demonstrates this process. Each worker receives the updated parameters from the parameter server after training for an iteration cycle computed based on the computation performance ratio. After that, workers continue to train for the next large mini-batch. The iteration cycle executed by each

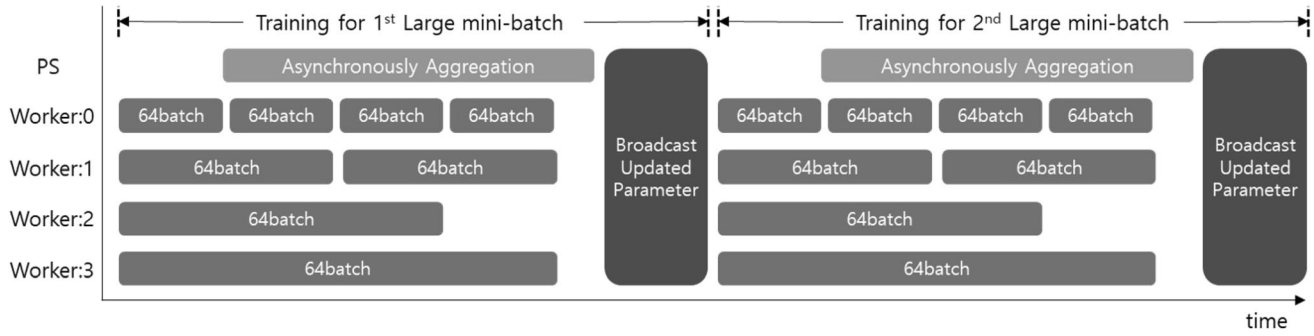


Fig. 4 Timeline of asynchronously large mini-batch training

worker is calculated as follows at the initialization before training.

- Step 1. The parameter server calculates the necessary iterations by dividing large mini-batch into the mini-batch size that will be used for the single iteration.
- Step 2. Each worker node multiplies the computation performance of the GPU to be used and the number of installed GPUs to calculate the computing performance of that node and then sends it to the parameter server.
- Step 3. After receiving the computation performance of each node, the parameter server distributes the total number of iterations needed to calculate the large mini-batch by the ratio of each worker's performance, and it then broadcasts this information to all worker nodes.

Figure 4 shows a timeline for the execution of training with four workers with different performances capabilities. In this case, the large mini-batch is set to 512, and each worker trains with a mini-batch of 64 for a single iteration. Therefore, a total of eight iterations is required to train with the large mini-batch. When the computing performance ratio of the workers is *4:2:1.5:1*, each worker's iteration cycle to update the parameters will be set to *4, 2, 1, 1*, as shown in Fig. 4. Here, workers receive updated parameters from the parameter-server at almost the same time. In this way, it is possible to minimize worker waiting time compared to synchronous training. In addition, all workers share the same parameters, so the reduction of accuracy can be prevented.

4.3 Collective-communication implementation via MPI

To implement our proposed system for actual distributed deep learning training, we use Google's TensorFlow. TensorFlow employs gRPC communication to support Distributed TensorFlow[10], but in this paper we use MPI instead of gRPC to implement communication. In the proposed design, most communication processing is composed of collective communication patterns. Typically, with local aggregation, the all-reduce method is used, and sending updated parameters to all workers is performed via broadcasting. TensorFlow operates using a "define-and-run" method in which a developer predefines the graph at the Python level [5]. To define the graph, only operations implemented in the TensorFlow APIs can be used, however MPI is not implemented as TensorFlow APIs. Therefore, implementing MPI in TensorFlow with existing Python implementation method, communication and computation cannot be overlapped as shown in the first timeline of Fig. 5. As a result, the computation can be delayed by the networking time.

To solve this problem, we switch MPI API to TensorFlow operation API by using `py_func` [12] to overlap computation and communication. `py_func` is a function provided by TensorFlow to convert another Python libraries to TensorFlow operations. By converting the MPI operations to TensorFlow's operation, MPI communications can overlap with computation, as shown in the second timeline graph of Fig. 5.

4.4 Parallel I/O via asyncio

In this paper, we propose a hybrid aggregation method that effectively combines the "parameter-server" and "all-

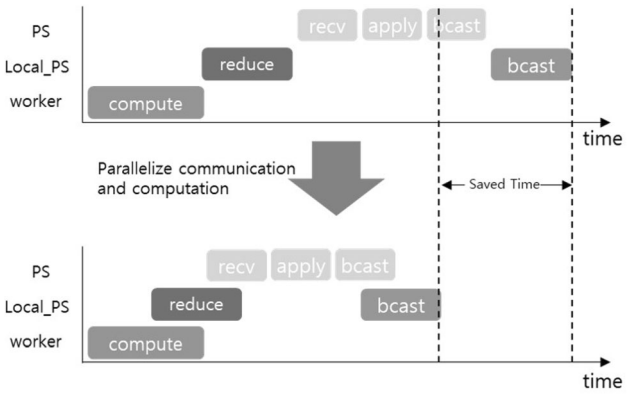


Fig. 5 Operation timeline of each process depending on whether overlapping is applied

reduce" approaches to maximize the overall utilization of computing resources in a heterogeneous cluster. In our proposed scheme, the global parameter server asynchronously receives local aggregated gradients from each worker and updates the training parameters with received gradients. However, when multiple workers are simultaneously sending gradients to the parameter server, I/O bottlenecks can occur, which can result in the overall performance degradation of training process.

To address this problem, we parallelize the communicating tasks (that receive gradients data from workers), and the training tasks (of parameters on the global parameter server). For this purpose, we implement *separate* processes to perform the communication and training parameters. The communication process receives the local aggregated gradient from each worker and pushes it to the data queue. The training process uses the aggregated gradient from the data queue to train the parameters. Similarly, workers are also implemented for training and communication tasks in different processes.

However, when performing actual training operations, we found that the overall training time increases significantly compared to the conventional Distributed TensorFlow. In order to analyze the results, we have checked the latency of data communication and computation of training parameters. Figure 6 shows the data flow and latency of each operation. For the worker, the time for training computation is 0.55 seconds, which is the same as the training computation for TensorFlow. However, we confirm that it takes almost the same time as the training computation latency for exchanging data between computation process and communication process in both workers and the global parameter server. Since each process allocates and uses different memory space, the slowdown occurs when writing data to another memory space.

Therefore, we attempt to parallelize computations and communications using multi-threading instead of multi-processing with the global parameter server and worker. However, when using multi-threading in Python, all threads are only able to use a single CPU core due to GIL, so the threads are not truly able to operate in parallel.

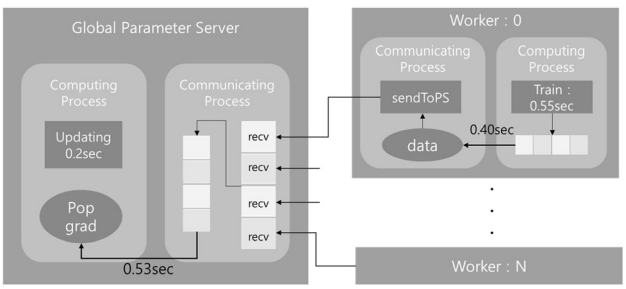


Fig. 6 System flow of data transfer using multi-processing

To address this problem of parallelizing computations and I/O operations in distributed deep learning, we effectively leverage *asyncio*, which is a Python module for asynchronous programming that can enable I/O operations and CPU tasks to be run in parallel. Asyncio allows the parallel execution of I/O with CPU work via asynchronous co-routine context switches. With asyncio, asynchronous co-routines are executed within the event-loop. The asynchronous co-routine executing I/O operations inside the event loop does not use the CPU cores. On the other hand, co-routines that need CPU computations can use the CPU cores. Thus, similar to multi-threading, threads that do not actually need the CPU cores to execute the I/O no longer occupy the cores, and only asynchronous co-routines that execute CPU tasks utilize the CPU (while other co-routines are executing I/O).

In this paper, we design a communication I/O using MPI to be executed asynchronously with TensorFlow's session, and asyncio is used to run the CPU processing and I/O operations in parallel. We perform the same analysis as seen in Fig. 6 to determine if the problems with multi-processing are resolved by using asyncio. The results indicate that the data exchange between the existing communication process and computing process originally took 0.5 seconds, but it can be reduced into 0.4 milliseconds with asyncio. With multi-processing, each process uses other memory space, so it takes time to send and receive data. However, with asyncio, because the co-routines can share the same memory space, it does not take additional time to send and receive data.

5 Evaluation

To evaluate our proposed scheme, we check the computation performance of our system by training ImageNet [9] data using the ResNet-50 [8] model. In addition, we compare the performance of Distributed TensorFlow's synchronous and asynchronous schemes with our proposed framework in terms of resource usage on heterogeneous and homogeneous computing systems. We conduct experiments using four kinds of GPUs: NVIDIA's Tesla K80, Tesla P100, TITAN Xp and TITAN V. The API used for training is TensorFlow r1.12, and the versions of CUDA driver and cuDNN are 9.0 and v7.

First, we confirm the computational performance when learning is performed using only a single GPU. The size of the mini-batch used for training ResNet-50 model is 64. Table 1 shows computing performance as the number of images processed per second. Tesla K80 has the lowest performance with 52 images/s. TITAN Xp and Tesla P100 has approximately triple the performance of Tesla K80, and TITAN V has four times as much performance. For

Table 1 Multi-GPU construction of each Worker Nodes and computation performance of single GPU

Worker #	Multi-GPU construction	Computation performance of single GPU
*Worker*0	*Tesla P*100 4*ea*	152 image/s
*Worker*1	*TITAN Xp* 4*ea*	148 image/s
*Worker*2	*Tesla K*80 4*ea*	52 image/s
*Worker*3	*TITAN V* 4*ea*	207 image/s

subsequent experiments, to evaluate the cluster performance (Figs. 7 and 8), we compare based on the above results and determine whether it is identical to the sum of the number of GPUs. For the distributed processing experiment, four worker nodes are used, and the GPU configuration of each worker node is as shown in Table 1.

5.1 Computational performance using distributed tensorflow in homogeneous and heterogeneous systems

Before evaluating the performance of our proposed framework, we check the computational performance of Distributed TensorFlow. First, we examine the performance when Worker 0 and Worker 1 are used to evaluate the distributed computing performance in a homogeneous system. Tesla P100 and TITAN Xp are different models, but the results in Table 1 reveal similar throughputs in terms of images per second, indicating that they constituted a homogeneous system. In addition, we confirm the performance when using a total of 16 GPUs of four worker nodes, as shown in Table 1, to confirm the performance in a heterogeneous system. The experiment is divided into synchronous and asynchronous, and 10 Gigabit Ethernet is used as a network. Throughout these experiments, we compare the performance of the entire system and each GPU based on the number of images processed per second.

The results of the synchronous training are presented in Table 2. They demonstrate that the average throughput for each GPU is 140.80 for Tesla P100 and 134.71 for TITAN Xp when using a homogeneous environment with Worker 0

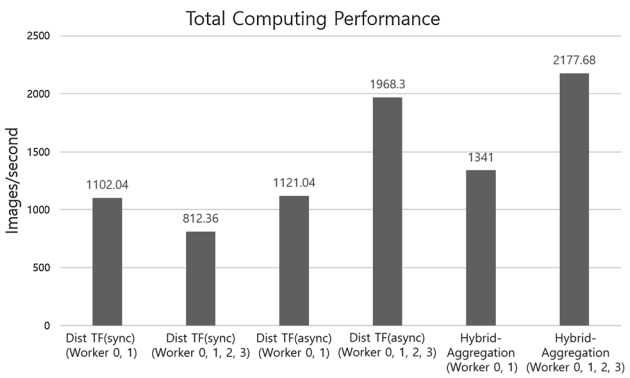


Fig. 7 Total computing performances with each distributed strategy

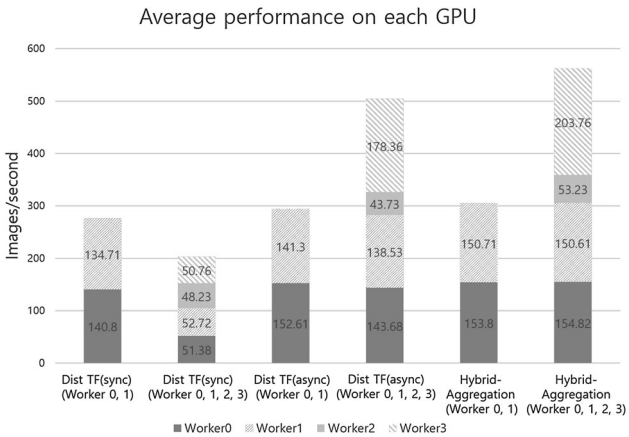


Fig. 8 Average performance on each GPU with each distributed strategy

and Worker 1. Performance is slightly lower than that shown in Table 1, but this is because of the network I/O, which was an existing problem while performing distributed processing. The performance of entire cluster is similar to the sum of the image throughputs for all GPUs. The results of the heterogeneous system using four workers shows that all GPUs have similar computing performance. This is because the total computing time has increased as other workers wait for the computing time of Worker 2, which has the slowest performance. Therefore, although the number of GPUs has increased more than in the homogeneous system, the performance of the system has decreased significantly.

Table 3 shows the results of asynchronously executing the same experiment as that shown in Table 2. When only Worker 0 and Worker 1 are used, the results are similar with those found in the synchronous training. when using the heterogeneous system, the results show much higher computing performance than in the synchronous training. When training asynchronously, unlike synchronously, workers do not have to wait for the computations of a slower worker. Accordingly, delays are reduced and the computing performance of the cluster increases significantly. However, computing performance of each GPU shows lower performance than the performance in Table 1. This is because all GPUs independently perform the I/O with the parameter server asynchronously, the total network usage increase, and a performance bottleneck occurs.

Table 2 ResNet-50 training computation performance using synchronous distributed TensorFlow

System construction	Total performance	Average performance on each GPUs
*Worker*0	$1102.04_{\text{image/s}}$	*Worker*0 : $140.80_{\text{image/s}}$
*Worker*1		*Worker*1 : $134.71_{\text{image/s}}$
*Worker*0	$812.36_{\text{image/s}}$	*Worker*0 : $51.38_{\text{image/s}}$
*Worker*1		*Worker*1 : $52.72_{\text{image/s}}$
*Worker*2		*Worker*2 : $48.23_{\text{image/s}}$
*Worker*3		*Worker*3 : $50.76_{\text{image/s}}$

Table 3 ResNet-50 training computation performance using asynchronous Distributed TensorFlow

System construction	Total performance	Average performance on each GPUs
*Worker*0	$1121.04_{\text{image/s}}$	*Worker*0 : $152.61_{\text{image/s}}$
*Worker*1		*Worker*1 : $141.30_{\text{image/s}}$
*Worker*0	$1968.3_{\text{image/s}}$	*Worker*0 : $143.68_{\text{image/s}}$
*Worker*1		*Worker*1 : $138.53_{\text{image/s}}$
*Worker*2		*Worker*2 : $43.73_{\text{image/s}}$
*Worker*3		*Worker*3 : $178.36_{\text{image/s}}$

Due to network bottleneck, each GPU waits to execute communication, the next iteration is delayed, and the total operation time increases.

To summarize, Distributed TensorFlow with a homogeneous system shows no significant performance difference between synchronous and asynchronous methods. With the heterogeneous system, the waiting time for each worker decreases and the overall performance of the cluster increases significantly with asynchronous training. However, due to the network communication bottleneck, it is very difficult to fully utilize available computing resources.

5.2 Computational performance using the proposed hybrid-aggregation method

The proposed Hybrid-Aggregation scheme computes the parameter update period of each worker for asynchronously large mini-batch training using the computation performance value. Each worker waits for the global parameter server to complete the aggregation after performing the training for the calculated period. When the updated parameters are broadcasted after aggregation, the next iterations are executed.

To evaluate the performance of the proposed design, the large mini-batch size is set to 8192, and the same experiment as that with Distributed TensorFlow is performed. In addition, we confirm that the parameter update cycle of each worker is set correctly. The experiment results are presented in the Table 4. From the results of training with a homogeneous system using Worker 0 and Worker 1, we can confirm that the system archives approximately 18% higher cluster system performance than Distributed TensorFlow. In the proposed system, unlike with Distributed TensorFlow, computing operation and communication are overlapped to minimize the delayed time due to communication, which results in decreased training time. Also, it is confirmed that similar cycles (17 and 15 respectively) are used for the parameter updates for Worker 0 and Worker 1. The results of the heterogeneous experiments using all workers demonstrate about a 2.5 times higher performance than the synchronous Distributed TensorFlow and 10% higher than asynchronous Distributed TensorFlow. In addition, the cycles for the parameter update of each worker are set to the same ratio as the computation performance ratio in Table 1. Moreover, the average performances on each GPU are almost same as those in Table 1.

To summarize, with our proposed distributed deep learning framework, we can effectively reduce the overhead of the network I/O by exploiting asynchronous large mini-batch training. Since communication and learning operations are performed in parallel, we have minimized the computation waiting time from the communication, and as a result, we are able to confirm that the overall training time and the computing performance can actually be improved compared to the performance of Distributed TensorFlow on heterogeneous computing systems.

Table 4 ResNet-50 training computation performance using proposed Hybrid-Aggregation way

System construction	Parameter update cycle(clock)	Total performance (image/s)	Average performance on each GPU (image/s)
*Worker*0	*Worker*0 : 17	1341	*Worker*0 : 153.80
*Worker*1	*Worker*1 : 15		*Worker*1 : 150.71
*Worker*0	*Worker*0 : 8	2177.68	*Worker*0 : 154.82
*Worker*1	*Worker*1 : 8		*Worker*1 : 150.61
*Worker*2	*Worker*2 : 4		*Worker*2 : 53.23
*Worker*3	*Worker*3 : 12		*Worker*3 : 203.76

5.3 ImageNet training time and accuracy with the ResNet-50 model

We have proposed a scheme to complete training jobs faster than the existing distributed training method with a heterogeneous computing cluster. We have also designed an asynchronous large mini-batch training method to ensure the same accuracy. To verify the effectiveness of our proposed schemes, we perform the training of ImageNet data with the ResNet-50 model, and we analyze the overall training time and accuracy. For comparison models, we use our proposed scheme and the conventional Distributed TensorFlow with synchronous and asynchronous training. We use the benchmark program provided by TensorFlow and perform training operations during 90 epochs for both comparison models in a heterogeneous cluster. The system used for training consists of a total three workers and a single parameter-server. Two of the three workers use eight of NVIDIA's TITAN V GPUs, and the remaining worker uses eight TITAN Xp. We perform four different types of experiments, including our proposed method and the conventional Distributed TensorFlow. The first experiment uses the synchronous Distributed TensorFlow, and the second uses the asynchronous Distributed TensorFlow. The third and fourth experiments are for performing training using our proposed system based on 8k and 16k batches.

Table 5 shows the summary of our performance evaluation results. In terms of training time, the synchronous Distributed TensorFlow takes 12 hours, while the asynchronous system takes 9.5 hours (25% more overhead than the asynchronous training). This result shows that the synchronous training scheme can result in performance degradation problems in a heterogeneous computing cluster. This is mainly because with the synchronous Distributed TensorFlow, the servers using TITAN V have to wait until the servers using TITAN Xp complete the tasks (in a heterogeneous computing cluster). Without this delay caused by the heterogeneity in the computing cluster, the asynchronous Distributed TensorFlow could complete the training job faster than the synchronous system. The proposed Hybrid-Aggregation method takes 8.5 hours and 7 hours with 8k and 16k batches, respectively. With the 16k large mini-batch, it is possible to reduce the training time by 1.5 hours because the cycle used to broadcast the trained parameters by the parameter server to the workers is longer than that used with the 8k batches. In addition, the proposed method completes training faster than the asynchronous Distributed TensorFlow. This is because we can reduce the communication bottleneck that occurs in the

Table 5 Training accuracy and latency of using each distributed training

	Distributed TensorFlow Synchronously	Distributed TensorFlow Asynchronously	Hybrid aggregation using 8k batch	Hybrid aggregation using 16k batch
Training time(hours)	12	9.5	8.5	7
Training loss	1.583	1.577	1.0258	1.1296
$Top-1$ *accuracy*	0.766	0.764	0.7616	0.7409
$Top-5$ *accuracy*	0.924	0.922	0.9266	0.9097

vanilla Distributed TensorFlow while executing computation and communication asynchronously with asyncio.

The final training accuracy and loss show similar results across all experiments as we can see from Figs. 9, 10 and 11. Although the minimum value of training loss is similar, it can be confirmed that this loss is reduced more quickly when using our proposed Hybrid-Aggregation than with the conventional Distributed TensorFlow. This is because the parameters are updated using the large mini-batch training method proposed in this paper, and all workers perform synchronization periodically. As seen from the results in Figs. 10 and 11, the training accuracy of the asynchronous Distributed TensorFlow is higher than that of the synchronous Distributed TensorFlow. This is because workers using an asynchronous training mechanism process more mini-batches per unit time than those using synchronous mechanism. Our proposed scheme shows faster accuracy improvements than the Distributed TensorFlow. Despite the asynchronous training mechanism, we can achieve higher training accuracy than the vanilla Distributed TensorFlow.

To summarize, our Hybrid-Aggregation method can substantially reduce the overall training time compared to the conventional Distributed TensorFlow without losing the training accuracy for the ImageNet data with the ResNet-50 model. In particular, when we are using the 16k large mini-batches, we are able to achieve almost the same training accuracy within only about a half the amount of time.

6 Related works

To improve the deep learning computation performance in a distributed system, various methods have been proposed. CosmoFlow [13] utilized Intel Xeon Phi processors to accelerate the deep learning based cosmology data training. CosmoFlow used a TensorFlow framework with MPI communication through the Cray PE Machine Learning Plugin. The main difference between Cosmoflow and our proposed design is that Cosmoflow used fully-synchronous data paralleled training. To improve the computing resource utilization of a heterogeneous cluster, we designed a Hybrid-Aggregation with all-reduce and parameter-server mechanisms. CosmoFlow demonstrated distributed training on 8192 Cori nodes with 77% parallel efficiency. However, if the training is performed on a heterogeneous cluster using Cosmoflow, the parallel efficiency can be lower.

S. Kim et.al. presented Parallax to optimize data parallel training by utilizing the sparsity of model parameters [14]. Parallax also combined parameter-server and all-reduce to optimize the I/O bottleneck on the distributed training. Parallax achieved 2.8x and 6.02x speed-ups for NLP models compared to TensorFlow and Horovod, respectively. However, the evaluation was performed on a homogeneous multi-GPU cluster. When using a

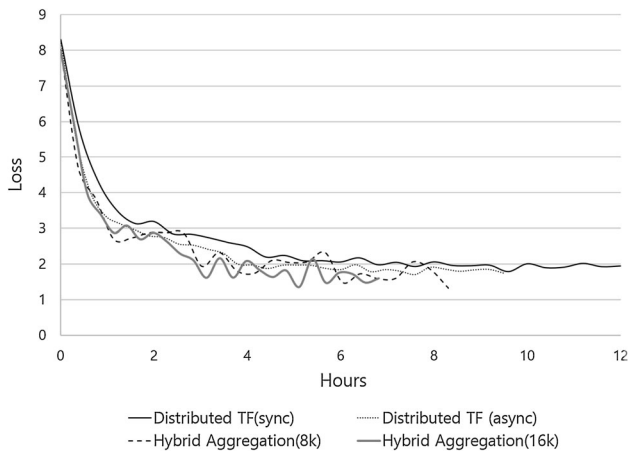


Fig. 9 Training loss of each distributed training

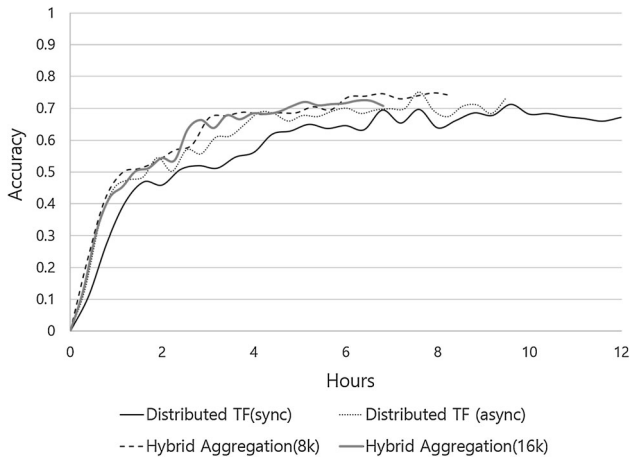


Fig. 10 Top-1 accuracy of each distributed training

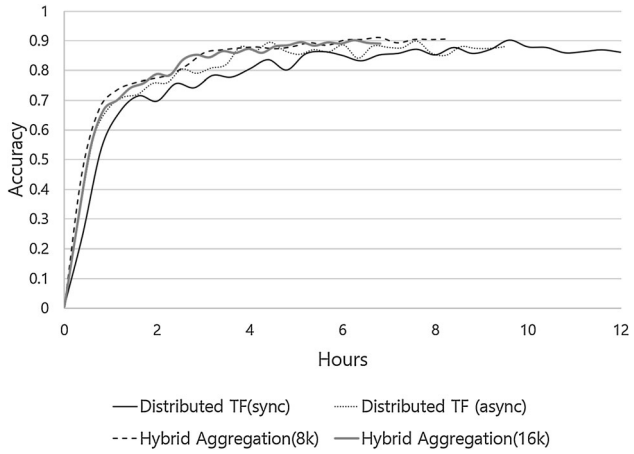


Fig. 11 Top-5 accuracy of each distributed training

heterogeneous computing cluster, the training accuracy and throughput were similar tottereretersdfasd TensorFlow's results.

X. Lian et.al. proposed an asynchronous decentralized parallel stochastic gradient decent (AD-PSGD) to mitigate the performance degradation caused by communication bottlenecks in an asynchronous distributed deep learning system [15]. In AD-PSGD, each worker updated its model parameters by averaging the model parameters from randomly selected neighbors. AD-PSGD achieved similar accuracy as all-reduce, while it converged faster than all-reduce. The difference between AD-PSGD and our method is that AD-PSGD reduces the communication bottleneck via decentralized training, while we are leveraging the local parameter server.

To improve the performance of distributed learning in heterogeneous computing environments, a heterogeneity-aware decentralized training protocol called Hop was proposed [16]. Hop used a queue-based synchronization mechanism for synchronizing model parameters in distributed environments. Hop also proposed skipping iterations to mitigate the degradation of training performance due to slow workers. Hop achieved 3.262x speed-ups for the CNN model compared to the parameter server method. Hop also converged 1.37 times faster than the parameter server method when randomly slowing down every worker by a factor of six. Unlike Hop, we have proposed a novel distributed deep learning architecture to improve training performance.

Ammar Ahmad et.al. proposed new MPI designs to improve the performance of applications in multi-GPU environments [17]. They proposed two new designs. First, a pipelined chain (PC) of MPI Bcast, which provides efficient intra- and inter-node GPU communications, was proposed. Second, they proposed a Topology-Aware pipelined chain (TA-PC) for multi-GPU sys-tems to efficiently utilize PCIe links in a node. They compared the proposed MPI Bcast with NCCL-based MPI Bcast (MPI+NCCL1) and ncclBroadcast (NCCL2)to evaluate the performance of the proposed MPI Bcast. The proposed MPI Bcast showed up to 16.6x better performance than MPI+NCCL1 in inter-node communication. In addition, the proposed MPI Bcast showed up to 10x better performance than NCCL2. They efficiently communicate between GPUs by modifying the existing MPI. Unlike them, we efficiently communicate using the existing MPI and hybrid aggregation architecture.

Recently NVIDA has release a report titled "SONY Breaks ResNet - 50 Training Record with NVIDIA V100 Tensor Core GPUs" [18]. In NVIDIA's report, several studies are stated to have trained ImageNet with the ResNet-50 model. Each study showed the computing performance of training while increasing the size of the large mini-batch to be trained. According to the results of the studies described in this report, Facebook used 256 NVIDIA Tesla P100s to train an 8K large mini-batch. Also, Sony increased the size of the large mini-batch further and trained a 68K large mini-batches using 2176 NVIDIA Tesla V100s. Both of these studies have improved the computing performance of deep learning by using larger sizes of large mini-batches at the cluster using large numbers of GPUs. However, these experiments were performed in a homogeneous environments constructed with GPUs with the same computing performance. Therefore, with a heterogeneous multi-GPU cluster, using the same methods from the above research would lower the computing performance compared to the proposed Hybrid-Aggregation method in this paper.

7 Conclusion

In this paper, we proposed an efficient distributed deep learning framework for a heterogeneous multi-GPU cluster. Our proposed design was able to address the disadvantages of existing all-reduce and parameter-server methods through a hybrid structure of the two schemes. In addition, we performed large mini-batch training asynchronously to increase the overall utilization of available computing power in the entire cluster. To evaluate the performance of our proposed design, we implemented it using MPI and TensorFlow, and we demonstrated a performance improvement in homogeneous and heterogeneous computing systems through ResNet-50 training with the ImageNet dataset. Further experiments on the training accuracy in large mini-batch training will be carried out in the future.

Acknowledgements This research was supported by Basic Science Research Program and Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (2020R1F1A1072696, 2015M3C4A7065646), Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT)(No. : 2020- 0-01305, Development of AI Deep-Learning Processor and Module for 2,000 TFLOPS Server), GRRC program of Gyeong-gi province (No. GRRC-KAU-2020-B01, "Study on the Video and Space Convergence Platform for 360VR Services") and ITRC (Information Technology Research Center) support program (IITP-2020-2018-0-01423)

References

1. Zinkevich, M., Weimer, M., Li, L., Smola, A.J.: Parallelized stochastic gradient descent. In: Advances in Neural Information Processing Systems 23
2. Heigold, G., McDermott, E., Vanhoucke, V., Senior, A., Bacchiani, M.: Asynchronous stochastic optimization for sequence

training of deep neural networks. In: 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)
3. Sergeev, A., Balso, M.D..: Horovod: fast and easy distributed deep learning in tensorflow. In: arxiv.org, Feb 2018
4. Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J.K., Gibbons, P.B., Gibson, G.A., Ganger, G., Xing, E.P.: More effective distributed ml via a stale synchronous parallel parameter server. In: Advances in Neural Information Processing Systems 26 (NIPS 2013)
5. TensorFlow: an open source machine learning library for research and production. https://www.tensorflow.org/
6. Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., He, K.: Accurate, large minibatch sgd: Training imagenet in 1 hour. In: arxiv.org, April 2018
7. MPICH: high-performance portable MPI, https://www.mpich.org/
8. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: arxiv.org, 2015
9. ImageNet, https://image-net.org, 2017
10. Distributed TensorFlow, https://www.tensorflow.org/deploy/distributed/
11. asyncio—Asynchronous I/O, http://docs.python.org/3/library/asyncio.html
12. py_func, https://www.tensorflow.org/ api_docs/python/tf/py_func
13. Mathuriya, A., Bard, A., Mendygral, P., Meadows, L., Arnemann, J., Shao, L., He, S., Karna, t., Moise, D., Pennycook, S.J., Maschoff, K., Sewall, J., Kumar, N., Ho, S., Ringenburg, M., Prabhat, Lee, V.: Cosmoflow: using deep learning to learn the universe at scale. In: arxiv.org, Aug 2018
14. Kim, S., Yu, G.-I., Park, H., Cho, S., Jeong, E., Ha, H., Lee, S., Jeong, J.S., Chun, B.-G. Parallax: sparsity-aware data parallel training of deep neural networks. In: EuroSys 2019, March 2019
15. Lian, X., Zhang, W., Zhang, C., Liu, J.: Asynchronous decentralized parallel stochastic gradient descent. In: Dy, J.G., Krause, A., Eds., Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10–15, 2018, series Proceedings of Machine Learning Research, vol. 80. PMLR, 2018, pp. 3049–3058. http://proceedings.mlr.press/v80/lian18a.html
16. Luo, Q., Lin, J., Zhuo, Y., Qian, X.: Hop: Heterogeneity-aware decentralized training. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 893–907
17. Awan, A.A., Manian, K.V., Chu, C.-H., Subramoni, H., Panda, D.K.: Optimized large-message broadcast for deep learning workloads: MPI, MPI+NCCL, or NCCL2? Parallel Comput. **85**, 141–152 (2019)
18. SONY Breaks ResNet-50 Training Record with NVIDIA V100 Tensor Core GPUs. http://news.developer.nvidia.com/sony-breaks-resnet-50-training-record-with-nvidia-v100-tensor-core-gpus/

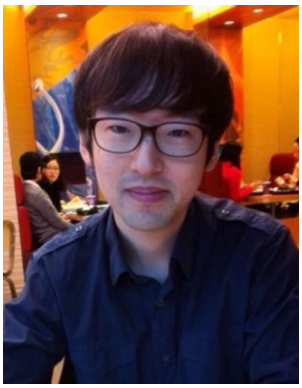
Youngrang Kim received his B.S. and M.S. in Electronics and Information Engineering at Korea Aerospace University (KAU). His primary research interests include distributed computing, high-performance computing, and multi-gpu based distributed deep learning framework.

Hyeonseong Choi is a M.S. candidate in Electronics and Information Engineering at Korea Aerospace University (KAU). He receieved his B.S. in Telecommunication and Information engineering from Korea Aerospace University. His primary research interests include distributed computing, high-performance computing, and multi-gpu based distributed deep learning framework.

Jaehwan Lee is an Associate Professor at the Department of Electronics and Information Engineering of Korea Aerospace University. He received his B.S. and M.S. in Electrical Engineering from Seoul National University, and Ph.D. in Computer Science from University of Maryland at College Park. He has several industry research experiences; Korea Telecom (KT) as a senior researcher (2000–2005), NEC labs in America and Bell labs, Alcatel-lucent as a research intern, and Samsung System Architecture lab in US as a Research Staff Engineer. His research interests include distributed computing, high-performance computing, and Big-data infrastructures to support data intelligence. He was a recipient of the General Electric (GE) Scholarship and the Korean Government Scholarship for Electric Power Industry.

Jik-Soo Kim received his B.S. & M.S. in Computer Science and Statistics from Seoul National University in Korea, and Ph.D. in Computer Science from University of Maryland at College Park, USA. He is currently an Assistant Professor at the Department of Computer Engineering of Myongji University. His primary research interests are in the design and analysis of distributed computing infrastructures to support Many-Task Computing, Cloud Computing and Data-intensive Computing.

Hyunseung Jei received the BS degree in 2008 from Soongsil University, Seoul, Korea. He is currently a SW engineer for SK Telecom ML infra group. His current research interests include distributed deep learning, system sotfware for flash memory, and SSD.



Hongchan Roh received the BS degree in 2006, the MS degree in 2008, and the Ph.D. degree in 2014, all from Yonsei University, Seoul, Korea. He is currently a research fellow for SK Telecom. His current research interests include distributed deep learning, network processors, database systems, flash memory, and SSD.