Copyright ⓒ 2021 KSII

Empirical Performance Evaluation of Communication Libraries for Multi-GPU based Distributed Deep Learning in a Container Environment

HyeonSeong Choi[1], Youngrang Kim[1], Jaehwan Lee[1], and Yoonhee Kim[2*]
[1] Korea Aerospace University
Goyang-City, Gyeonggi-do, Republic of Korea
[e-mail: chyon794@gmail.com, kimyr207@gmail.com, jlee@kau.ac.kr]
[2] Sookmyung Women's University
Seoul, Republic of Korea
[e-mail: yulan@sm.ac.kr]
*Corresponding author: Yoonhee Kim

Received November 11, 2019; revised May 19, 2020; accepted August 2, 2020; published March 31, 2021

Abstract

Recently, most cloud services use Docker container environment to provide their services. However, there are no researches to evaluate the performance of communication libraries for multi-GPU based distributed deep learning in a Docker container environment. In this paper, we propose an efficient communication architecture for multi-GPU based deep learning in a Docker container environment by evaluating the performances of various communication libraries. We compare the performances of the parameter server architecture and the All-reduce architecture, which are typical distributed deep learning architectures. Further, we analyze the performances of two separate multi-GPU resource allocation policies — allocating a single GPU to each Docker container and allocating multiple GPUs to each Docker container. We also experiment with the scalability of collective communication by increasing the number of GPUs from one to four. Through experiments, we compare OpenMPI and MPICH, which are representative open source MPI libraries, and NCCL, which is NVIDIA's collective communication library for the multi-GPU setting. In the parameter server architecture, we show that using CUDA-aware OpenMPI with multi-GPU per Docker container environment reduces communication latency by up to 75%. Also, we show that using NCCL in All-reduce architecture reduces communication latency by up to 93% compared to other libraries.

Keywords: Docker, Collective Communication, Distributed Deep Leaning, Multi-GPU

This research was supported by Next-Generation Information Computing Development Program (2015M3C4A7065646) and Basic Research Program (2020R1H1A2011685, 2020R1F1A1072696) through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT, GRRC program of Gyeong- gi province (No. GRRC-KAU-2020-B01, "Study on the Video and Space Convergence Platform for 360VR Services"), ITRC (Information Technology Research Center) support program (IITP-2021-2018-0-01423) and "HPC Support" Project supported by the Ministry of Science and ICT and NIPA.

http://doi.org/10.3837/tiis.2021.03.006 **ISSN : 1976-7277**

1. Introduction

Large learning models and a great number of training datasets are required to considerably improve the accuracy of deep learning based systems [1-4]. Inevitably, this significantly increases the required learning time. To overcome this problem, distributed deep learning using multiple GPUs has been proposed [5]. This has been verified practically as well. For instance, Facebook achieved 76% accuracy while training the ImegeNet dataset with the Resnet-50 model over one hour using 256 Tesla P100s [6]. Additionally, Sony used the 2176 Tesla V100s in their research which achieved 75% accuracy while training the ImageNet dataset with the Resnet 50 model over 224 seconds [7]. Based on these two studies, we can confirm that similar levels of accuracy can be achieved over much shorter training times by using distributed processing for deep learning with multiple GPUs.

Every year, NVIDIA announces a new GPU with better computing power, and the price of the GPUs keep increasing. Therefore, hardware acquisition costs quickly become prohibitive. To solve this problem, many laboratories conduct deep learning in a multi-GPU environment using cloud services such as Amazon Web Services and Google Cloud Platform. These cloud services provide a multi-GPU environment to users via a Docker container environment rather than a virtual machine environment [8, 9]. Therefore, extensive analysis of the communication performances of distributed deep learning architectures in Docker containers is necessary to achieve efficient distributed deep learning in multi-GPU environments akin to those provided by cloud services.

In this paper, we present experimental analysis of various collective communication libraries in a Docker container environment, which is a typical environment in the training process of a distributed deep learning system. We employed MPI (Massage Passing Interface), which is a representative parallel programming library [10], and NCCL (NVIDIA Collective Communication Library) [11], which is NVIDIA's multi-GPU collective communication library, to analyze the collective communication performances of multi-GPU based distributed deep learning systems, especially focusing on multi-GPU resource allocation in the Docker container environment. We compared OpenMPI, with MPICH, both of which are representative open source MPI libraries [12, 13]. Further, we analyzed the performance of CUDA-aware OpenMPI.

The contributions of this paper are two-fold. To begin with, this is the first experimental result that demonstrates the difference in performance between a single GPU Docker container environment and a multi-GPU one. Second, an extensive analysis of collective communication performances under typical distributed deep learning training scenarios is also presented.

Our experiments focus on the collective communication performances of MPI, NCCL, and CUDA-aware MPI in a distributed deep learning environment as described below.

- We consider two typical distributed deep learning architectures in our experiments – the parameter server architecture and the All-reduce architecture. We analyze each communication library's performance in each distributed deep learning architecture. In the parameter server archtecture, CUDA-aware OpenMPI performed the best. However, NCCL was the best performer in the All-redeuce archtecture.
- We consider two types of GPU resource allocation policies — allocating a single GPU to a Docker container and allocating multiple GPUs to a Docker container. In all experimental senarios, allocating multiple GPUs to a Docker container yielded better performances than allocating a single GPU to a Docker container because of the commucation overhead between separate Docker containers.
- To evaluate GPU scalability, we experimented with increasing the number of GPUs

used. In the parameter server architecture, CUDA-aware MPI exhibited the best scalability. On the other hand, NCCL proved to be the most scalable in the All-reduce architecture.

In the parameter server architecture, we show that using CUDA-aware OpenMPI with multi-GPU per Docker container environment reduces communication latency by up to 75%. When we use NCCL with multi-GPU per Docker container environment in All-reduce architecture, communication latency is reduced by up to 93% compared to using other libraries.

This paper presents the most efficient communication libraries for each distributed deep learning architecture in the Docker container environment via a comparative analysis of experimental results. Section 2 presents a detailed discussion about the communication libraries studied in this paper and about distributed deep learning in general. The expermental environment is described in Section 3. In Sections 4 and 5, we analyze the experimental results for each distributed deep learning archtecture. The experimental results are summarized in section 6. Section 7 describes the works that are related to our research. Finally, we conclude our paper in Section 8.

2. Background

2.1 Collective Communication Libraries

MPI is a standard communication library for parallel programs. MPI assigns jobs based on processes, and each process corresponds to a set called 'the communicator'. Processes can exchange messages with other processes corresponding to the same communicator. Processes in a communicator are identified via a special number assigned to the process called the 'rank'. There are two kinds of communication methods in MPI — point-to-point communication and collective communication [14]. *MPI_Send* and *MPI_Recv* are typical point-to-point communication subroutines in MPI. Collective communication is any communication in which all processes corresponding to the communicator participate together. Collective communication is implemented based on point-to-point communication. *MPI_Bcast*, *MPI_Gather*, *MPI_Allgather*, *MPI_Reduce*, and *MPI_Allreduce* are all representative collective communication subroutines. OpenMPI and MPICH are representative open source MPI libraries. The experiments presented in this paper test *MPI_Bcast*, *MPI_Gather,* and *MPI_Allreduce*. **Fig. 1** depicts the working procedure of collective communication subroutines.

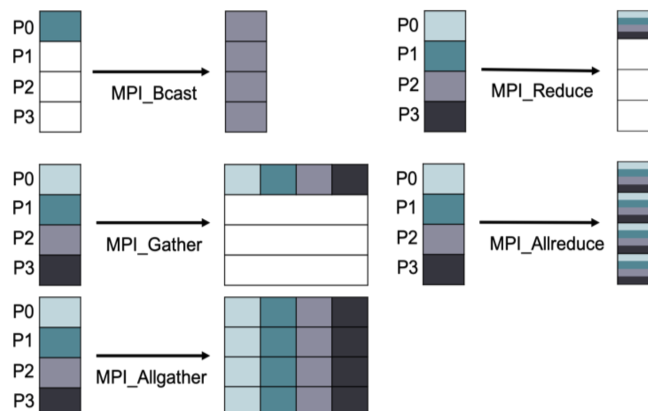


Fig. 1. MPI collective communication

NCCL is a multi-GPU based collective communication library developed by NVIDIA. NCCL provides subroutines such as *ncclAllgather*, *ncclReduce*, *ncclBroadcast,* and *ncclAllGather*. It constructs a ring-style group to provide maximum bandwidth. The first version of NCCL only supported communication among GPUs corresponding to a single node. On the contrary, the second version of NCCL not only supports multi-GPU communication at a node but also supports communication between GPUs at separate nodes [11].

2.2 Distributed Deep Learning

The increase in training time in deep learning can be solved by distributed deep learning using the multi-GPU system. Training methods of distribued deep learning can be of two kinds — data parallelization and model parallelization [5, 15]. Data parallelization is a scheme in which multiple GPUs divide one large dataset and all GPUs train using the same model. In data parallelization, each GPU undergoes training on separate datasets, so that the gradients computed through back propagation are different across the GPUs. Therefore, the gradients computed on each GPU are reflected in the whole learning process after aggregation. Parameter server architecture and All-reduce architecture are typical architectures used to aggregate gradients. **Fig. 2** and **Fig. 3** depict the operation of each architecture.

Processes involved in the parameter server architecture can be classified into two groups — the parameter server and workers. The parameter server receives the gradients computed by the workers, updates the parameters of the learning model, maintains the latest parameters, and provides the parameters to the workers whenever any worker requests them. Workers compute the gradients based on input data and model parameters based on the latest parameters received from the parameter server. The parameter server can operate both in synchronouns and asynchronous methods. In the synchronous method, parameters are updated once all workers finish computing gradients. Thus, each worker is able to begin its task after receiving the latest parameters from the parameter server once the parameters are updated. As synchronization is performed at each iteration in this method, the accuracy of the whole learning process is quite high. However, the throughput of a whole cluster depends on the slowest worker because all workers are synchronized at each iteration. This is a disctinct disadvantage. In the asynchronous method, parameters are updated whenever the parameter server receives gradients computed by any worker. Thus, each worker is free to begin its task immediately, without waiting for the other workers to receive the latest parameters from the parameter server. The throughput rate of the whole cluster is consequently higher than that in the synchronous method. However, accuracy is compromized because each worker computes gradients using different parameters. Tensorflow [16] currently supports the parameter server architecture for distributed deep learning.

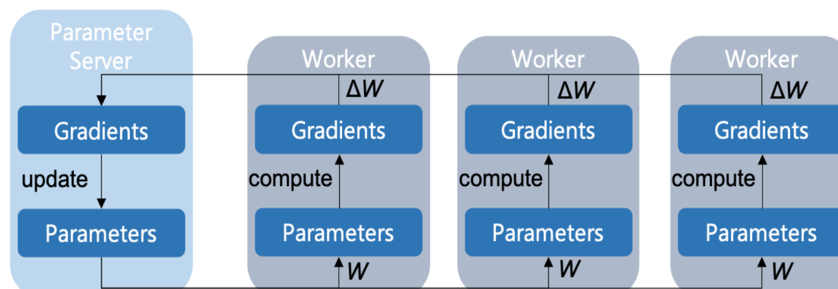


Fig. 2. Parameter server architecture

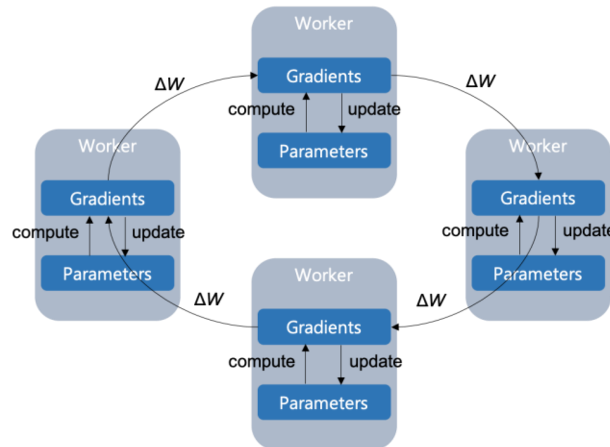


Fig. 3. All-reduce architecture

In the All-reduce architecture, instead of having a parameter server to maintain the latest parameters, each worker communicates with the others via P-to-P communication to maintain the synchronization of parameters. This method has the advantage of reducing communication overhead compared with methods employing a parameter server because of the lack of any communication between the parameter server and the workers. However, its inflexibility is a disadvantage because it can only operate synchronously. An example of a representative framework that uses the All-reduce architecture is Horovod [17].

3. Related Works

A few studies exist in the literature that evaluate HPC(High-Performance Computing) application performances in a Docker environment. Xavier, Miguel G. et al. compared the performances of container-based virtualization tools (e.g., LXC, Docker, Singularity, bare metal) using HPL-benchmarks and MPI based applications [18]. Additionally, Saha, Pankaj et al. compared the performances of MPI applications in bare metal and Docker environments using various HPC benchmarks [19]. This paper shows that the performance overhead of MPI applications in the Docker environment are negligible. However, in the aforementioned study, the authors do not consider MPI applications that use GPUs. Thus, their paper suffers from limitation that the impact of the Docker environment on MPI applications using GPUs is not explored.

There is study that evaluate the performance of GPU applications in a Docker environment. Kømären, Teemu et al. evaluated GPU performances in host, QEMU, and Docker environments in cloud gaming systems [20]. The study concludes that GPU performance for cloud gaming systems with Docker is much better than that of QEMU, and is identical with that of the host. However,this study suffers from the limitation that it does not analyze the high communication overhead that may be incurred by using multi-GPU environments.

Futher, some studies analyze the performances of deep learning applications using GPUs in Docker environments. Siddaramana, Vintha G., and Anto Ajay Raj John analyzed the performance of deep learning applications in native and Docker environments [21]. Their performances were compared by running DeepBench and Fathom workloads in a host environment and in the container environment. Performance comparisons showed that the time taken to run the kernel was the same on the host and on the container. In the study published by Xu, Pengfei, Shaohuai Shi, and Xiaowen Chu, the performance of deep learning

frameworks such as Caffe, CNTK,MXNet, and Tensorflow, all of which are being widely used recently, are evaluated in the Docker container environment [22]. They compared the performances of I / O, CPU, and GPU in the Docker container environment with their performances in a host environment. The conclusion of their study was that the performance of the deep learning frameworks in a Docker container environment was identical to that in a host environment.

Their research shows that there is no performance degradation on systems with single GPUs. However, the study suffers from the limitation that the degradation of performanc for deep learning frameworks due to possible communication overheads has not been explored. In this paper, we propose a method that can minimize the communication overhead that arises in multi-GPU based deep learning distributed processing. Based on the results presented in this paper, it can be concluded that efficient multi-GPU based deep learning distributed processing in a Docker container environment is viable.

Zhang, Jie, Xiaoyi Lu, and Dhabaleswar K. Panda evaluate the performance of MPI in Singularity-based container [23]. Their research shows that there is no degradation of MPI communication performance for both Intel Xeon and Intel Knights Landing in a Singularity-based container environment. Unlike this research, we use Docker container. Also, we evaluate the performance of communication between GPUs.

A study published by Z. Li, M. Kihl, Q. Lu and J. A. Andersson Compared the performance overhead between hypervisor and container based virtualization [24]. According to this study, in most cases, the performance overhead of the container is less than the performance overhead of the virtual machine. This result means that virtualization using a container is more efficient than using a virtual machine. Thus, we evaluate the performance of the communication library in Docker container environment rather than a hypervisor.

4. Experimental Environment

In order to analyze the communication performance of multi-GPU based distributed deep learning training in the Docker container environment, experiments were conducted on the parameter server architecture and the All-reduce architecture. For each architecture, the experiments were conducted in two distinct modes. One mode proceeded by assigning a single GPU to one Docker container, and the other proceeded by assigning multiple GPUs to one Docker container. In the case of MPI, there are various types of implementation. Most MPI libraries are implemented based on OpenMPI or MPICH. Also, NCCL is developed by NVIDIA as a collective communication library for communication between GPUs. NCCL is

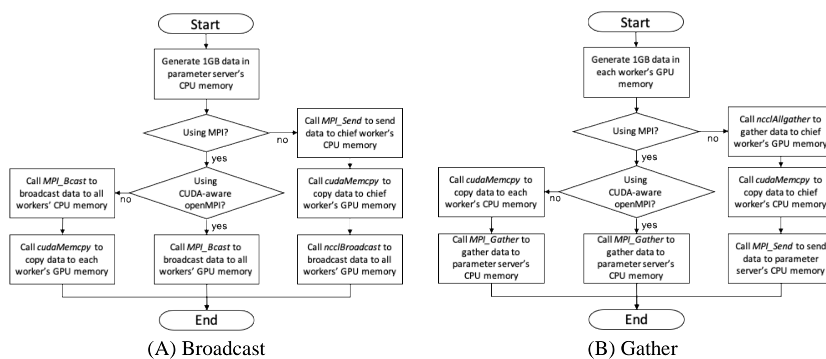
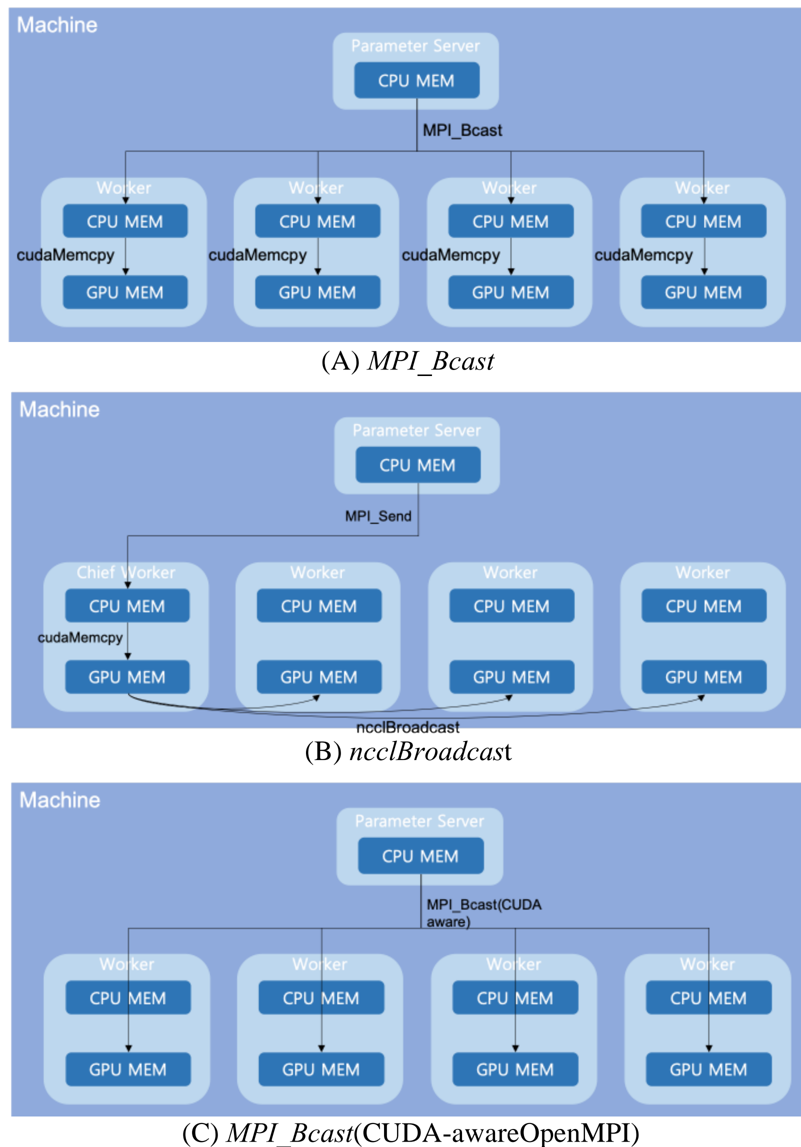


Fig. 4. Flowchart of parameter server architecture

the most efficient library to communicate between NVIDIA's GPUs. Thus, the experiments used MPICH, OpenMPI, and NCCL as communication libraries. The recent version of OpenMPI supports the CUDA-aware MPI function. Therefore, we used CUDA-aware OpenMPI in our experiments. Intel Xeon CPU E5-2650 CPUs and four NVIDIA GTX Titan XP were the GPUs used in the experiments. The software used were NVIDIA driver 410.79, CUDA 10.0, OpenMPI 4.0.0, MPICH 3.3, NCCL 2.4, and Docker 18.09.5. In each experiment, we exchanged a randomly generated 32-bit floating point array of size 1 GB to evaluate the communication performance. Since the experiments measure the communication latency, the results of the experiments do not change depending on the type of the dataset. All experiments were repeated 1000 times.



(A) *MPI_Bcast*

(B) *ncclBroadcas*t

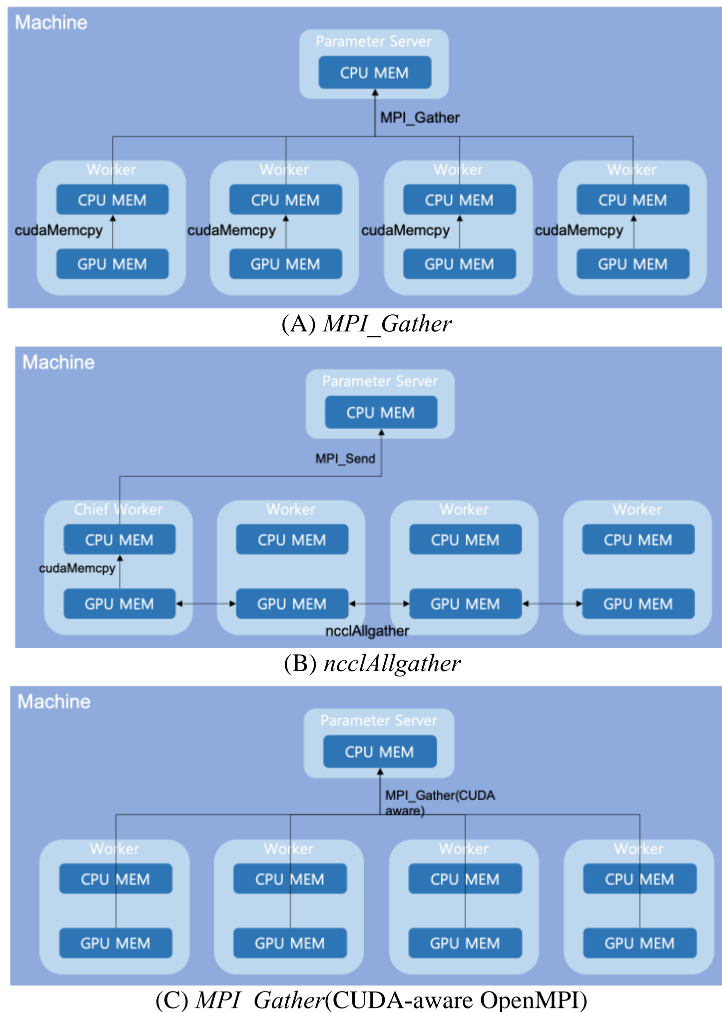
(C) *MPI_Bcast*(CUDA-awareOpenMPI)

Fig. 5. Broadcasting data in the parameter server archtecture

4.1 Parameter Server Architecture

In the parameter server method, communication latency was measured twice — once when a parameter server broadcasts the parameters to the workers and once when the gradients computed by the workers are aggregated into the parameter server. Additionally, each experiment was repeated using two separate allocation policies — allocating one GPU to a Docker container and allocating multiple GPUs to a Docker container. Fig. 4 is a flowchart of experiment in parameter server architecture.

In parameter server architecture, the parameter server needs to broadcast model parameters to all workers [25]. So, we use *MPI_Bcast* of MPI, *ncclBroadcast* of NCCL, and *MPI_Bcast* of CUDA-aware OpenMPI, all of which are one-to-all broadcast subroutines. We measured and compared the durations required to copy relevant data from the CPU memory of the parameter server to the GPU memories of all workers. In the experiment using *MPI_Bcast*, the parameter server first broadcasts all data in its CPU memory to the workers' CPU memories. Following that, each worker copies the data from its CPU memory to its GPU memory via *cudaMemcpy*. When *ncclBroadcast* is used for broadcasting data, the data in the CPU memory of the parameter server is transmitted to chief worker's CPU memory using MPI_Send.



(A) *MPI_Gather*

(B) *ncclAllgather*

(C) *MPI_Gather*(CUDA-aware OpenMPI)

Fig. 6 Gathering data in the parameter server archtecture

The chief worker copies the data from its CPU memory to its GPU memory via *cudaMemcopy*. Finally, the chief worker broadcasts the data from its GPU memory to the remaining workers' GPU memories via *ncclBroadcast*. In the case of transmission via CUDA-aware OpenMPI, the parameter server broadcasts all the data in its CPU memory directly to the GPU memories of the workers via *MPI_Bcast*. **Fig. 5** depicts the experimental scenarios of broadcasting data via each subroutine.

During the aggregation of gradients, MPI and CUDA-aware OpenMPI use the *MPI_Gather* subroutine, but NCCL uses the *ncclAllgather* subroutine because there is no *ncclGather* subroutine. Using *MPI_Gather*, each worker calls *cudaMemcpy* to copy the data from its GPU memory to its CPU memory. The parameter server gathers data from each worker's CPU memory to its CPU memory via *MPI_Gather* as well. Workers gather data from their GPU memories to the chief worker's GPU memory using *ncclAllgather*. The chief worker copies the gathered data from its GPU memory to its CPU memory via *cudaMemcpy*, and then transmits it to the CPU memory of the parameter server via *MPI_Send*. In the case of CUDA-aware OpenMPI, it is possible to directly copy all the data in the workers' GPU memories to the CPU memory of the parameter server by simply calling *MPI_Gather*. **Fig. 6** depicts the experiment scenarios of gathering data via each subroutine.

4.2 All-reduce Architecture

In the All-reduce architecture, we measured the duration required to execute a reduce sum operation on the data in the GPU memory of each worker and to store the result in the GPU memory of each worker for various subroutines. We compared *MPI_Allreduce* of MPI, *ncclAllreduce* of NCCL, and *MPI_Allreduce* of CUDA-aware OpenMPI. **Fig. 7** is a flowchart of experiments in All-reduce architecture. As in the previous case, each experiment is conducted in two modes — once by assigning one GPU to each Docker container and once by assigning multiple GPUs to each Docker container. When *MPI_Allreduce* is used, each worker copies the data in its GPU memory to its C PU memory via *cudaMemcpy*. The reduce sum operation is then executed on the copied data using *MPI_Allreduce*. Each worker copies the resultant data saved in CPU memory of each worker to their respective GPU memories via *cudaMemcpy*. On the other hand, when *ncclAllreduce* is used, it is possible to directly execute

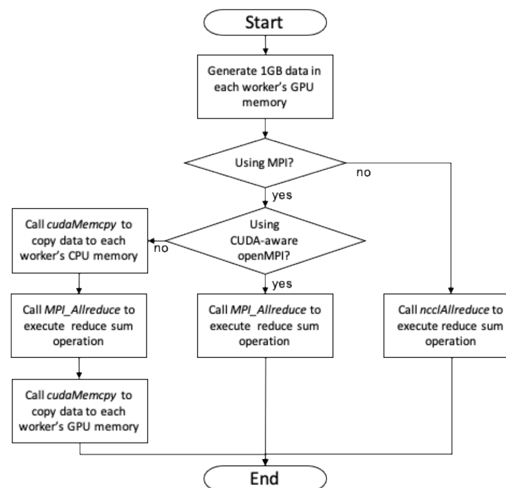
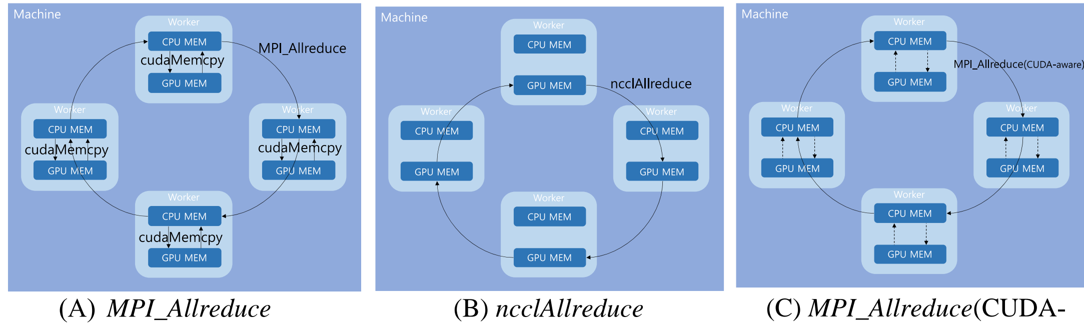


Fig. 7. Flowchart of All-reduce architeture

the reduce sum operation on the data present in the workers' GPU memories and to save the resultant data in GPU memory by simply calling *ncclAllreduce*. The final candidate — CUDA-aware OpenMPI — does not require replication of data between CPU and GPU memories via *cudaMemcpy*. Instead, it is possible to execute the reduce sum operation on the data stored in the GPU memory of each worker by simply calling *MPI_Allreduce*. **Fig. 8** depicts the experimental scenarios arising in the All-reduce architecture.



(A) *MPI_Allreduce* (B) *ncclAllreduce* (C) *MPI_Allreduce*(CUDA-

Fig. 8. All-reduce data in All-reduce architecture

5. Experimental Results for the Parameter Server Architecture

5.1 Broadcast of Data from the Parameter Server

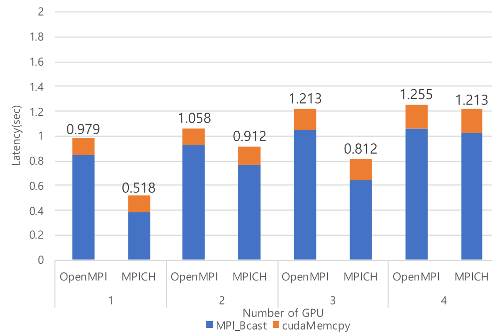
5.1. 1 Single GPU per Docker Container

Fig. 9(A) depicts the communication latency with the increase in the number of GPUs, during broadcast using *MPI_Bcast,* when a single GPU is assigned to each Docker container. The total communication latency when a single GPU was used was 0.518 s for MPICH. For OpenMPI, the total communication latency was 0.979 s, which is about 1.9 times that of MPICH. Thus, OpenMPI performed worse than MPICH in this case. However, when four GPUs were used, OpenMPI took 1.255 s to broadcast the data while MPICH took 1.213 s. Therefore, increasing the number of GPUs reduces the difference in communication latencies caused by the choice of specific MPI libraries. The communication latency recorded for *ncclBroadcast* is presented in **Fig. 9(B)**. When the parameter server broadcasts data to the workers via *ncclBroadcast*, the variance in communication latency caused by the difference in the MPI library is negligible except when three GPUs are used. **Fig. 9(C)** records the communication latencies when data is broadcasted via *MPI_Bcast* of CUDA-aware OpenMPI using varying number of GPUs. When two GPUs were used, the time taken to broadcast the parameters was 1.668 s, which is about 1.5 times the time taken to broadcast the same data using one GPU. Using four GPUs increased the communication latency to 1.747 s, which is 1.04 times the time taken by two GPUs. Therefore, CUDA-aware OpenMPI exhibited the best scalability. According to the data presented in **Fig. 9**, *MPI_Bcast* is the optimal choice to reduce communication latency if only one GPU is active. Further, it can be seen that, if four GPUs are active, the variance in communication latency caused by the different choices of MPI libraries is negligible.

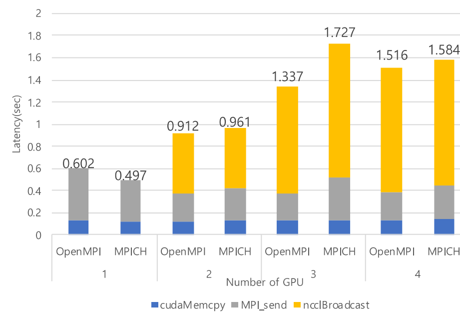
5.1.2 Multi-GPU per Docker Container

Fig. 10 presents a graph depicting communication latencies in data broadcasting as the number

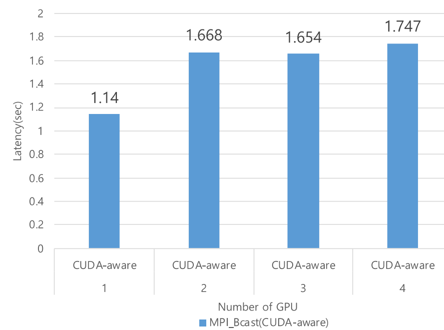
of GPUs is increased by allocating multiple GPUs to each Docker container. **Fig. 10(A)** records the communication latencies when *MPI_Bcast* is used as the broadcasting subroutine. When three GPUs are used, the communication latency for OpenMPI is about 1.08 times that for MPICH. However, when four GPUs are used, MPICH took about 1.3 times that for OpenMPI. Therefore, OpenMPI is a better choice when four or more GPUs are assigned to each Docker container. **Fig. 10(B)** presents the communication latencies when data is broadcast via *ncclBroadcast*, which increased by about 1.7 times as the number of GPUs used was increased from one to four. However, when *MPI_Bcast* was used, communication latency increased by about 1.9 times as the number of GPUs used was increased from one to four. Thus, the scalability of NCCL is better than that of MPI. **Fig. 10(C)** shows a graph of the results of the experiments conducte d using *MPI_Bcast* of CUDA-aware OpenMPI. In this case, communication latency increased by roughly 1.2 times as the number of GPUs used was increased from one to four and it exhibited the smallest increase among the three candidates. Moreover, in the case of four GPUs, the communication latency was the smallest at 0.436 s.



(A) Using *MPI_Bcast*

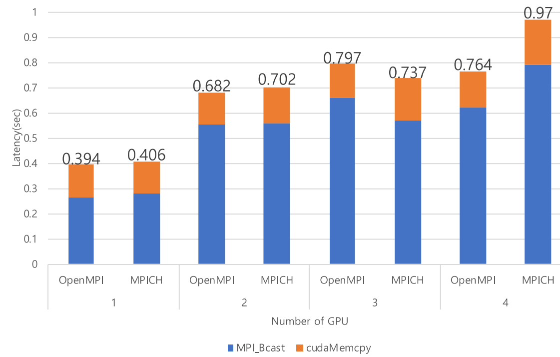


(B) Using *ncclBroadcast*

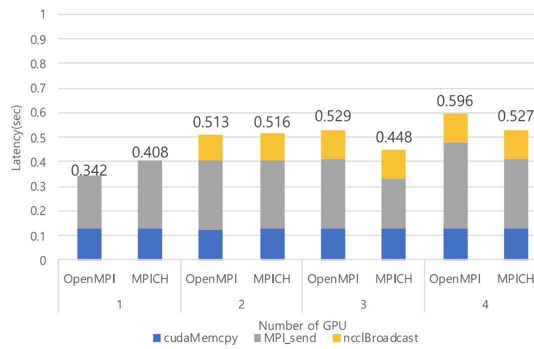


(C) Using *MPI_Bcast* of CUDA-aware OpenMPI

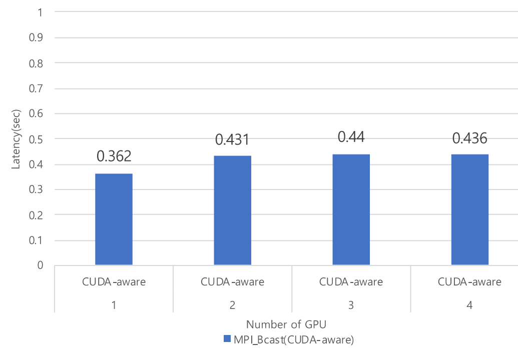
Fig. 9. Data broadcating latencies corresponding to increasing number of GPUs in single GPU per Docker container environment



(A) Using *MPI_Bcast*



(B) Using *ncclBroadcast*



(C) Using *MPI_Bcast* of CUDA-aware OpenMPI

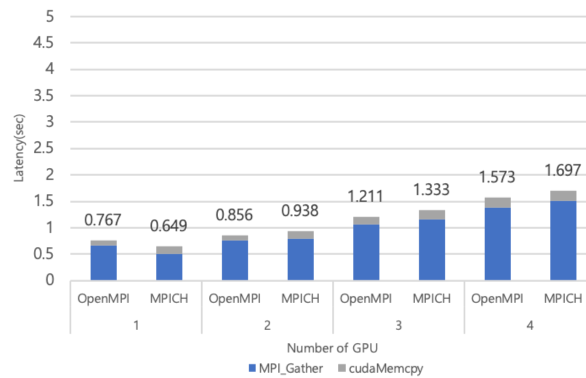
Fig. 10. Data broadcating latencies corresponding to increasing number of GPUs in multiple GPUs per Docker container environment

5.2 Aggregation of Data in the Parameter Server

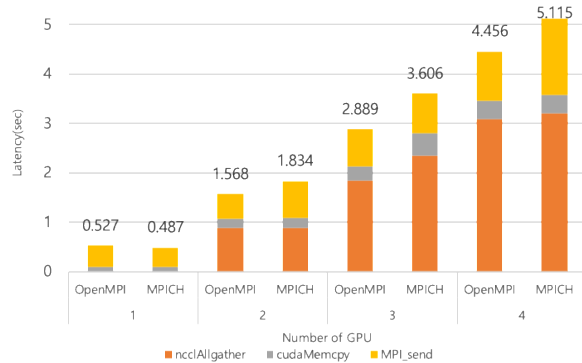
5.2.1 Single GPU per Docker Container

Fig. 11 depicts the communication latencies for data aggregation on a parameter server corresponding to varying number of GPUs, when each Docker container is assigned a single GPU. As shown in **Fig. 11(A)**, MPICH exhibits a latency that is about 1.1 times that of OpenMPI in cases in which more than two GPUs are used to call *MPI_Gather*. Further, as the number of GPUs was increased from one to four, the communication latency increased by more than a factor of two. **Fig. 11(B)** illustrates the communication latencies when data is aggregated using *ncclAllgather*. As in the case recorded in **Fig. 11(A)**, the communication latency while using MPICH was about 1.1 times that while using OpenMPI. Further, with four

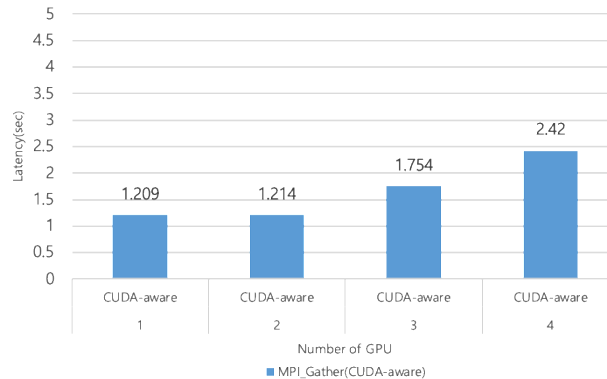
GPUs compared to using one GPU, the communication latency exhibited a significant increase by a factor of about 10 when four GPUs were used instead of one. Fig. 11(C) is a communication latency graph for the case in which *MPI_Gather* of CUDA-aware OpenMPI was used. In this case, the communication latency almost doubled when the number of GPUs was increased from one to four. This is a small increase in latency compared to the 10-fold increase in the case of *ncclAllreduce*. Fig. 11 indicates that using general *MPI_Gather* achieves the smallest communication latency for data aggregation when a single GPU is assigned to each Docker container. It was also confirmed that using general *MPI_Gather* shows the smallest increase in communication latency as the number of GPUs used is increased. OpenMPI performs better than MPICH in this scenario.



(A) Using *MPI_Gather*



(B) Using *ncclAllgather*

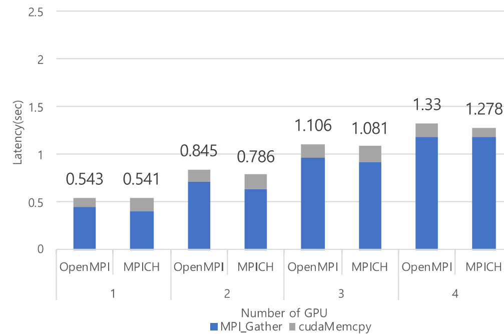


(C) Using *MPI_Gather* of CUDA-aware OpenMPI

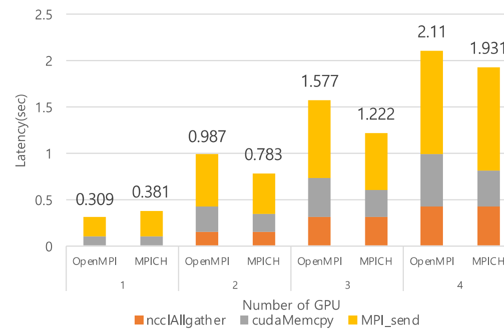
Fig. 11. Data gathering latencies corresponding to increasing number of GPUs in single GPU per Docker container environment

5.2.2 Multi-GPU per Docker Container

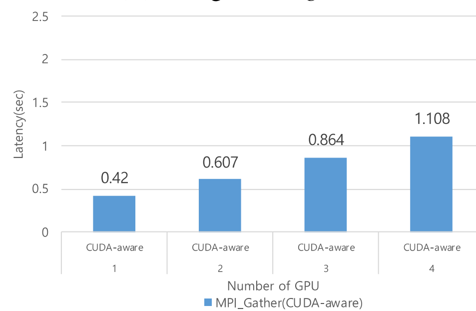
Fig. 12 depicts the communication latencies corresponding to increasing number of GPUs during aggregation of data on a parameter server in a multi-GPU per Docker container environment. Fig. 12(A) shows the graph of communication latencies when *MPI_Gather* was used. As is evident from the figure, there is almost no variation in communication latencies due to different choices of MPI libraries. Fig. 12(B) is the graph of communication latencies when *ncclAllgather* was used for data aggregation. Overall, the communication latency for MPICH was about 20% smaller than that for OpenMPI. The graph presented in Fig. 12(C) shows the same data for the case of *MPI_Gather* of CUDA-aware OpenMPI. With four active GPUs, the total communication latency in this case was 1.108 s, smaller than that for both MPI and NCCL. This case also exhibited the smallest communication latency during data aggregation with multiple GPUs in the Multi-GPU per Docker container environment, as is evident from Fig. 12. Further, *ncclAllgather* requires the longest duration for data aggregation because the size of the data being used in communication is increased in its case.



(A) Using *MPI_Gather*



(B) Using *ncclAllgather*



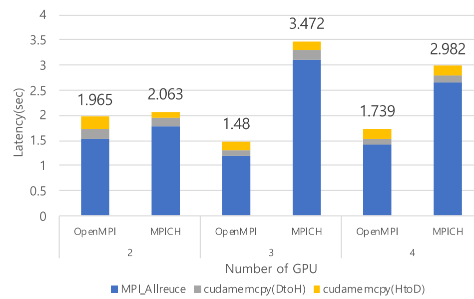
(C) Using *MPI_Gather* of CUDA-aware OpenMPI

Fig. 12. Data gathering latencies corresponding to increasing number of GPUs in multiple GPUs per Docker container environment

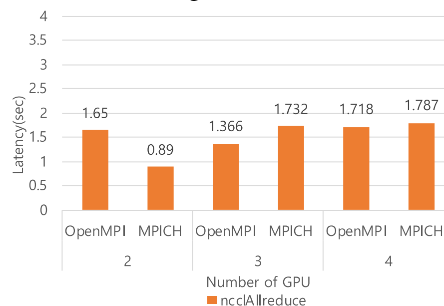
6. Experimental Results for the All-reduce Architecture

6.1 Single GPU per Docker Container

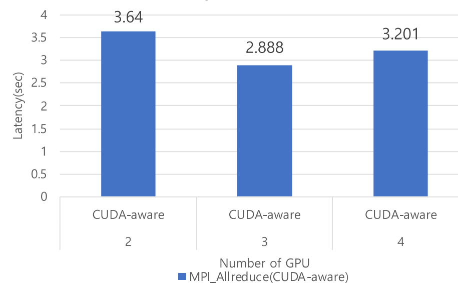
Fig. 13 depicts communication latencies of varying subroutines in the All-reduce architecture corresponding to increasing number of GPUs in a single GPU per Docker container environment. **Fig. 13(A)** presents the communication latencies when *MPI_Allreduce* is used. In this case, OpenMPI shows communication latency which is up to 60% smaller than for MPICH. **Fig. 13(B)** is the communication latency graph when *ncclAllreduce* is used. In this case, the communication latencies for different libraries is almost identical as long as four GPUs are employed. With three active GPUs, MPICH required a duration that was about 1.3 times that for OpenMPI. However, with two active GPUs, OpenMPI took about 1.9 times longer than MPICH. **Fig. 13** indicates that, overall, in a single GPU per Docker container environment, OpenMPI tends to require a longer duration when the two GPUs are active than when three GPUs are active. Therefore, if OpenMPI is to be used, it is advisable to use three or more GPUs. Further, *ncclAllreduce* exhibits better overall performance than *MPI_Allreduce*.



(A) Using *MPI_Allreudce*



(B) Using *ncclAllreduce*

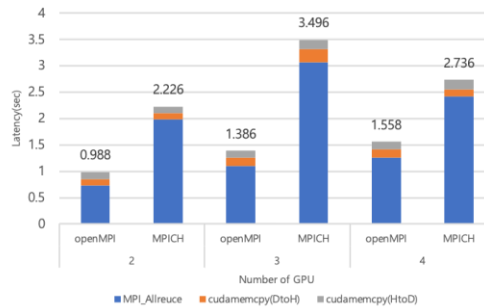


(C) Using *MPI_Allreuduce*CUDA-aware OpenMPI

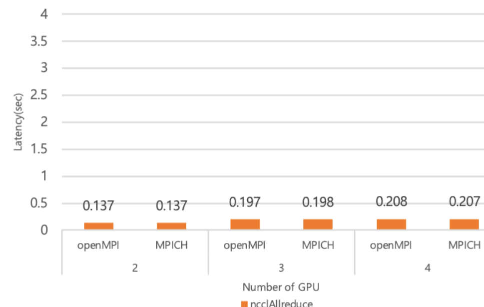
Fig. 13. Data Allreduce latencies corresponding to increasing number of GPUs in single GPU per Docker container environment

6.2 Multi-GPU per Docker Container

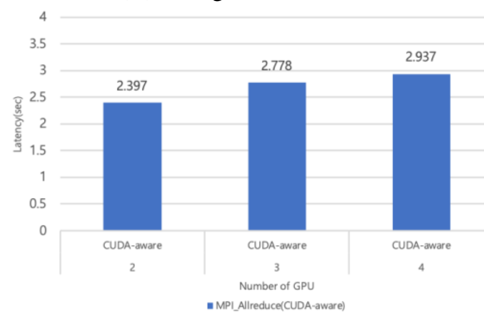
Fig. 14 depicts the communication latencies for various subroutines in the All-reduce architecture corresponding to varying number of GPUs used in a multi-GPU per Docker container environment. **Fig. 14(A)** shows that communication latency is reduced by up to 60% by using *MPI_Allreduce* of OpenMPI instead of using *MPI_Allreduce* of MPICH. **Fig. 14(B)** is a graph of latencies in the case in which All-reduce is executed via *ncclAllreduce*. As is evident from the figure, as the number of GPUs increases from two to four, the communication latency increases by a factor of about 1.5. *NcclBroadcast* exhibits the smallest communication latency of 0.208 s using four GPUs. When CUDA-aware OpenMPI was used as the communication library, the communication latency was about 1.9 times that for OpenMPI. Therefore, *ncclAllreduce* is the most optimal method to reduce communication latency on the All-reduce architecture when multiple GPUs can be assigned to each Docker container. If NCCL is not available, OpenMPI exhibits smaller communication latency compared to MPICH or CUDA-aware OpenMPI and is thus a better choice.



(A) Using *MPI_Allreuduce*



(B) Using *ncclAllreduce*



(C) Using *MPI_Allreudce*CUDA-aware OpenMPI

Fig. 14. Data Allreduce latencies corresponding to increasing number of GPUs in multiple GPUs per Docker container environment

7. Summary of Experimental Results

Fig. 15, **Fig. 16** and **Fig. 17** summarize the results of the experiments conducted in this paper. We choose OpenMPI as MPI library to summarize our experiments with four active GPUs.

Fig. 15 depicts that the durations required by the parameter server to broadcast data to the workers in the parameter server architecture. In the environment in which multiple GPUs are assigned per Docker container, the best performance was achieved with *MPI_Bcast* of CUDA-aware OpenMPI, yielding a duration of 0.436 s. The poorest performance was obatined with *MPI_Bcast* of OpenMPI. However, in the single GPU per Docker container environment, *MPI_Bcast* of OpenMPI exhibited the best performance with a duration of 1.255 s. Therefore, if multiple GPUs can be assigned to each Docker container, *MPI_Bcast* of CUDA-aware OpenMPI is the optimal choice. If only one GPU can be allocated to each Docker container, *MPI_Bcast* of OpenMPI is the best choice.

Fig. 16 presents the durations required by the workers to execute data aggregation to the parameter server in the parameter server architecture. In the environment with multi-GPU per Docker container, CUDA-aware OpenMPI performed the best with a duration of 1.108 s. However, in the environment with a single GPU per Docker container, *MPI_Bcast* of OpenMPI performed the best at 1.573 s. Therefore, as in the previous case, allocating multiple GPUs to each Docker container and using *MPI_Bcast* of CUDA-aware OpenMPI provides the best method to reduce communication overhead. If only one GPU can be allocated to each Docker container, MPI_Bcast of OpenMPI is the best at reducing communication latency. In the parameter server architecture, the CUDA-aware OpenMPI is the optimal choice to execute data transfer in the multi-GPU per Docker container environment. However, if multiple GPUs cannot be allocated to each Docker container, OpenMPI is the best option to transfer data while reducing communication overhead.

Fig. 17 records the durations required by various subroutines to execute the reduce sum operation in the All-reduce architecture. In particular, when allocating multi-GPU to a Docker container, using *ncclAllreduce* shows the lowest communication latency with 0.208 seconds. In all environments of the All-reduce architecture, *ncclAllreduce* is the best performer. Therefore, in the context of multi-GPU deep learning in the All-reduce architecture, using *ncclAllreduce* of NCCL minimizes communication latency.

Table 1 and **Table 2** show the best and worst results of each experiment. In parameter server architecture, CUDA-aware OpenMPI shows the best performance for both broadcasting and gathering. When allocating multi-GPU to Docker container, data broadcasting latency via *MPI_Bcast* of CUDA-aware OpenMPI is 0.436 seconds. However, when allocating single GPU to Docker container, data broadcasting latency is increased by about four times to 1.747 seconds. In single GPU per Docker container environment, Data gathering via *ncclAllgather* takes 4.456 seconds. On the other hand, data gathering via *MPI_Gather* of CUDA-aware OpenMPI takes 1.108 seconds in multi-GPU per container environment. In case of All-reduce architecture, using NCCL as a communication library shows the best performance. *ncclAllreduce* takes 0.208 seconds to execute All-reduce in Multi-GPU per Docker container environment. While, *MPI_Allreduce* of CUDA-aware OpenMPI takes 3.201 seconds in single GPU per Docker container, which is about 15 times that of *ncclAllreduce*.

Because of the overhead caused by communication between Docker containers, assigning a single GPU to a Docker container exhibits lower performance. Communication overhead between Docker containers is smaller when OpenMPI is used rather than NCCL or CUDA-aware OpenMPI in the parameter server architecture. However, in the All-reduce architecture, communication overhead between Docker containers is minimized by using NCCL.

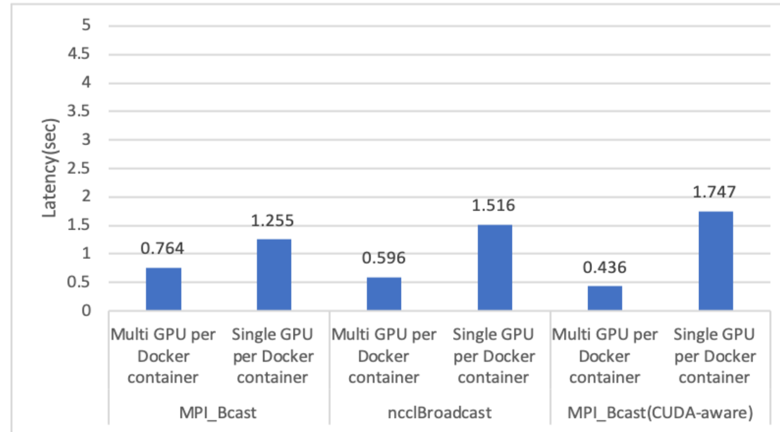


Fig. 15. Data broadcast latencies in the parameter server architecture

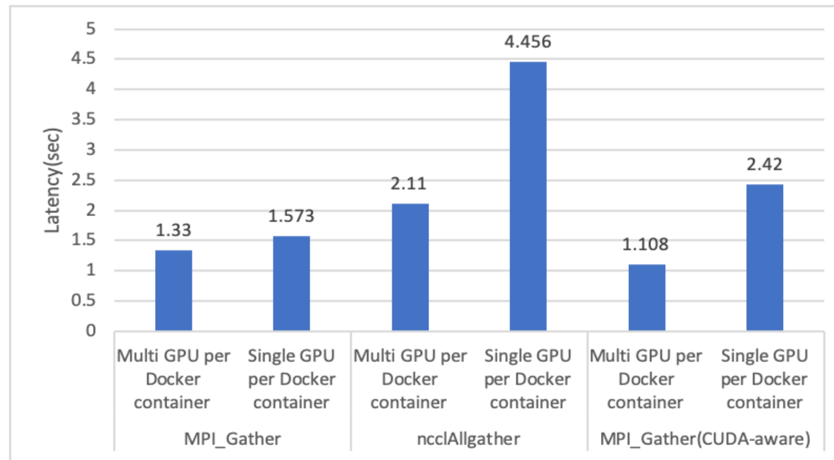


Fig. 16. Data gathering latencies in the parameter server architecture

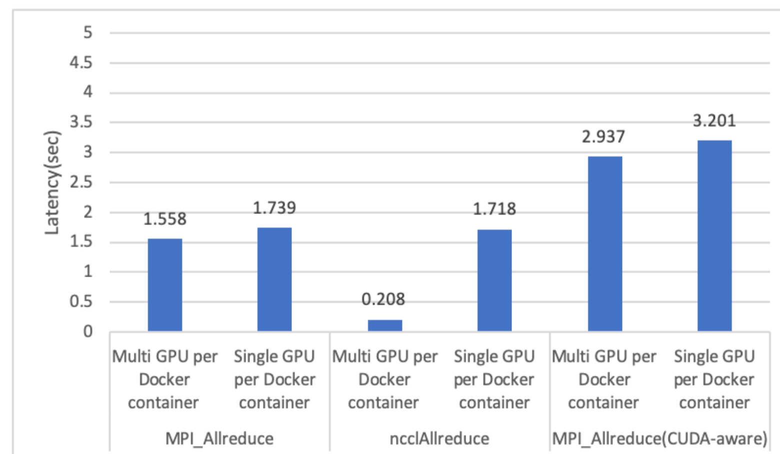


Fig. 17. Data Allreduce latencies in the All-reduce architecture

Table 1. Best results of each experiment

Experiment	GPU allocation type	Subroutine	Latency(sec)
Broadcasting	Multi-GPU per container	*MPI_Bcast*(CUDA-aware)	0.436
Gathering	Multi-GPU per container	*MPI_Gather*(CUDA-aware)	1.108
All-reduce	Multi-GPU per container	*ncclAllreduce*	0.208

Table 2. Worst results of each experiment

Experiment	GPU allocation type	Subroutine	Latency(sec)
Broadcasting	Single GPU per container	*MPI_Bcast*(CUDA-aware)	1.747
Gathering	Single GPU per container	*ncclAllgather*	4.456
All-reduce	Single GPU per container	*MPI_Allreduce*(CUDA-aware)	3.201

8. Conclusion

In this paper, we analyze the performances of collective communication techniques used in the parameter server architecture and the All-reduce architecture via extensive experiments to ascertain the viability of multi-GPU based deep learning in Docker container environment. The experiments used the libraries MPICH, OpenMPI, NCCL, and CUDA-aware OpenMPI.

In a Docker container environment, owing to the communication overhead between the docker containers, allocating multiple GPUs to each Docker container yielded better performances than allocating a single GPU to each Docker container. Therefore, allocating multiple GPUs to each Docker container can reduce performance degradation.In the parameter server architecture, CUDA-aware OpenMPI was ascertained to be the most optimal choice to reduce communication latency when multiple GPUs are allocated to each Docker container. However, if multiple GPUs cannot be allocated to each Docker container, using OpenMPI instead is most optimal. Further, in the All-reduce architecture, using NCCL is the best method to reduce communication latency regardless of GPU resource allocation policies. Further experiments on performances in multi-node environments will be performed in the future.

References

[1] J. Deng, W. Dong, R. Socher, L. Jia, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," *in Proc. of IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248-255, 2009. Article (CrossRef Link)

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. of IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1-9, 2015. Article (CrossRef Link)

[3] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proc. of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 1, pp. 4780-4789, 2019. Article (CrossRef Link)

[4] Y. Huang, Y. Cheng, A. Bapna, O. First, D. Chen, M. Chen, H. Lee, K. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in Neural Information Processing Systems*, vol. 32, 2019. Article (CrossRef Link)

[5] M. Zinkevich, M. Weimer, L. Li, and A. Smola, "Parallelized stochastic gradient descent," *Advances in Neural Information Processing Systems*, 2010. Article (CrossRef Link)

[6] P. Goyal, P. Dollar, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017. Article (CrossRef Link)

[7] H. Mikami, Hiroaki, P. Uchupala, Y. Tanaka, and Y. Kageyama, "Massively distributed SGD: ImageNet/ResNet-50 training in a flash," *arXiv preprint arXiv:1811.05233*, 2018. Article (CrossRef Link)
[8] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81-84, Sep. 2014. Article (CrossRef Link)
[9] Overview of amazon web services, *Amazon Whitepapers*, 2020. Article (CrossRef Link)
[10] J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "An introduction to the MPI standard," *Communications of the ACM 18*, 1995. Article (CrossRef Link)
[11] J. Sylvain, "Nccl 2.0," GTC, 2017. Article (CrossRef Link)
[12] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proc. of European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, vol. 3241, pp. 97-104, 2004. Article (CrossRef Link)
[13] W. Gropp, E. Lusk, N. Doss, and A. Skjullum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789-828, 1996. Article (CrossRef Link)
[14] B. Barker, "Message passing interface (MPI)," in *Proc. of Workshop: High Performance Computing on Stampede*, vol. 262, 2015. Article (CrossRef Link)
[15] J. Dean, G. S. Corradeo, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, and P. Tucker, "Large scale distributed deep networks," in *Proc. of the 25th International Conference on Neural Information Processing Systems*, pp. 1223-1231, 2012. Article (CrossRef Link)
[16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," in *Proc. of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*, pp. 265-283, 2016. Article (CrossRef Link)
[17] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018. Article (CrossRef Link)
[18] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Proc. of the 21st Euromicro International Conference on Parallel*, pp. 233-240, 2013. Article (CrossRef Link)
[19] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, "Evaluation of docker containers for scientific workloads in the cloud," in *Proc. of International Conference on Advanced Research Computing*, pp. 1-8, 2018. Article (CrossRef Link)
[20] T. Kämäräinen, Y. Shan, M. Siekkinen, and A. Ylajaaski, "Virtual machines vs. containers in cloud gaming systems," in *Proc. of International Workshop on Network and Systems Support for Games (NetGames)*, pp. 1-6, 2015. Article (CrossRef Link)
[21] V. G. Siddaramanna and A. A. R. John, "Effect of Performance on Containerized Deep Learning Applications," *Presented at WinTechCon-2018, organized by IEEE CAS Bangalore Chapter, IEEE Bangalore Section, and IEEE WiE Council*, pp. 1-6, 2018.
[22] P. Xu, S. Shi, and X. Chu, "Performance evaluation of deep learning tools in Docker containers," in *Proc. of the 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, pp. 395-403, 2017. Article (CrossRef Link)
[23] J. Zhang, X. Lu, and D. K. Panda, "Is Singularity-based Container Technology Ready for Running MPI Applications on HPC Clouds?," in *Proc. of the 10th International Conference on Utility and Cloud Computing*, pp. 151-160, 2017. Article (CrossRef Link)
[24] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, "Performance Overhead Comparison between Hypervisor and Container Based Virtualization," in *Proc. of IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pp. 955-962, 2017. Article (CrossRef Link)

[25] G. Heigold, E. McDermott, V. Vanhoucke, A. Senior, and M. Bacchiani, "Asynchronous stochastic optimization for sequence training of deep neural networks," in *Proc. of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5587-5591, 2014. Article (CrossRef Link)

Hyeonseong Choi is a M.S. candidate in Electronics and Information Engineering at Korea Aerospace University (KAU). He received 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.

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.

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.

Yoonhee Kim is the professor of Computer Science Department at Sookmyung Women's University. She received her B.S in Computer Science from Sookmyung Women's University in 1991, her M.S and Ph.D. in Computer and Information Science from Syracuse University in 1996 and 2001, respectively. She was a Research Staff Member at the Electronics and Telecommunication Research Institute during 1991 and 1994. Before joining the faculty of Sookmyung Women's University in 2001, she was the faculty of Computer Engineering dept. at Rochester Institute of Technology in NY, USA. Her research interests span many aspects of runtime support and management in distributed computing systems. She is a member of IEEE and ACM, and she has served on variety of program committees, advisory boards, and editorial boards.