



Distributed deep learning platform for pedestrian detection on IT convergence environment

Seong-Soo Han¹ · Yoon-Ki Kim² · You-Boo Jeon³ · JinSoo Park³ ·
Doo-Soon Park³ · DuHyun Hwang² · Chang-Sung Jeong²

Published online: 15 February 2020
© The Author(s) 2020

Abstract

IT technology and traditional industries have been combined recently, resulting in IT convergence technology in various fields. Through convergence with the automobile, pedestrian detection technology, in particular, is used in the autonomous navigation control service of autonomous vehicles and also applied in various fields such as intelligent CCTV and robot recognition technology. For pedestrian detection, hierarchical classification and feature vector were used in early stage, and deep learning is under active progress. However, since deep learning for pedestrian detection is time-consuming for processing a large volume of image data, it requires a lot of computing resources, and hence building such a system is very expensive. Therefore, in this paper we shall present a distributed deep learning platform which can easily build a cluster, and execute deep learning process in the distributed cloud environment, while achieving performance improvement in various ways. Our platform provides a convenient interface for easily and efficiently executing the deep learning process in a distributed environment by providing a multilayered system architecture. Our system builds and utilizes computing power in easy and efficient way by leveraging container technique, so-called OS-level virtualization, rather than traditional hypervisor-based virtualization. In our system, we improve the whole performance by exploiting both of data and parameter parallelisms at once and reduce the synchronization overhead by exploiting asynchronous communication for parameter updates. Also, we propose an efficient resource allocation scheme for parameter servers and slaves which can improve the performance from the experiment.

Keywords IT convergence · Pedestrian detection · Faster R-CNN · Deep learning · Distribution processing · Parallel processing · Virtual machine

✉ Chang-Sung Jeong
csjeong@korea.ac.kr

Extended author information available on the last page of the article

1 Introduction

IT convergence technologies have been recently emerging from various fields, combining IT technology and traditional industries. In IT convergence technology, pedestrian detection is very essential and crucial in particular, since it has been utilized in many different fields in relation with human life such as intelligent CCTV, robot recognition, security, autonomous navigation, crime investigation and entertainment [1]. Pedestrian detection involves finding and recognizing a person in an image or video, and his exact location. This information can be utilized for various uses, providing services needed by the pedestrian or as a method of communication [2]. Such pedestrian detection has been converged with the automobile field and currently used in the intelligent services for autonomous navigation of vehicles.

Pedestrian detection technology in the early years consisted of feature extraction and learning process [3–9] which have a relatively high false detection rate compared to the deep learning model. Convolutional neural network (CNN) [10–12] through deep learning is used widely in recent years, which involves learning pedestrian data and detecting objects through classification. This method has a lower false detection rate compared to the method from early stages, but making the detector is strenuous, and the learning time takes very long [13]. Therefore, in this paper, we adopt faster region with convolutional neural network (Faster R-CNN) [14] technique with excellent detection performance, which uses region proposal network (RPN) for faster detection. However, since deep learning for pedestrian detection is time-consuming for processing a large volume of image data, it requires a large number of high-performance computers or several high-performance graphics processing units (GPUs). Unfortunately, it takes money and time to build such an environment; thus, it must put a lot of pressure on researchers who are in financial difficulty. Cloud for deep learning processing is one of solutions to solve this problem. By providing a deep learning cloud service from an enterprise company which is building a large data center, deep learning researchers can conduct research without any financial or time burdens.

In this paper we shall present a distributed deep learning platform which can easily build a cluster, and execute deep learning process fast in the distributed cloud environment while achieving performance improvements. Our platform provides a convenient interface for easily and efficiently executing the deep learning process in a distributed environment by providing a multilayered system architecture. Our system utilizes computing power in easy and efficient way by providing interface for managing distributed resources and by leveraging container technique, so-called OS-level virtualization, rather than traditional hypervisor-based virtualization. Hypervisor-based machine is operated in the same hierarchy as the emulator and guest OS, making it rather inferior and utilization of resources difficult [15]. On the other hand, the container is an OS-level virtualization method which provides a virtual environment, isolated from the physical system including the CPU, memory and distributed network. By providing deep learning clouds through container-based virtualization, physical resources can be

used as much as possible without overhead. We achieve performance improvement in various ways. First, we improve the whole performance by exploiting both of data and parameter parallelisms at once. For data parallelism, input data are distributed among slave nodes, each updating the gradient, while for parameter parallelism parameter data among several parameter nodes, respectively. Next, we reduce the synchronization overhead by exploiting asynchronous communication for parameter updates. Most of distributed learning frameworks were developed using a high-performance distributed processing framework called Apache Spark [16], which was based on memory technology. It generally operates in MapReduce and batch processing in synchronous approach for parameter exchanges. Our system improves performance by adopting the asynchronous method for parameter exchange, as the synchronous method takes more time compared to the asynchronous one. Also, we propose an efficient resource allocation scheme for parameter servers and slaves which can improve the performance from the experiment. When parameter servers and slaves are placed in VMs of different physical nodes, remote communication occurs frequently. Therefore, we can reduce the network overhead by placing the parameter server and slave in each VM in one physical node.

The outline of our paper is as follows: In Sect. 2, we explain about related works. In Sect. 3, we describe about system architecture and flow of our platform, and in Sect. 4, we explain about deep learning process. In Sect. 5, we describe about experimental result, and in Sect. 6, we give a conclusion.

2 Related work

This chapter explores deep learning models for pedestrian detection and programming library which are widely used recently and then describe about distributed parallel processing technique for deep learning and distributed environments.

2.1 Deep learning models for pedestrian detection

Convolutional neural networks (CNN) have been widely used recently, due to its excellent performance in the object recognition field [17]. CNN is used in the process of classifying whether or not the extracted image is a pedestrian. This method involves inputting an image of a certain size and outputting whether or not it is a pedestrian in the final output stage through CNN. Feature extraction and classification process are all included in one structure. On the other hand, region with convolutional neural network (R-CNN) [18] uses image segmentation for selective search in order to extract all the object prediction regions and inputs the extracted regions into the learned CNN to classify the results. However, the problem with R-CNN is that training is complicated, causing slow test speeds. Fast R-CNN was developed to solve this problem. Like the existing R-CNN, Fast R-CNN also predicts the object region from the image, but unlike the R-CNN, it can deduce the entire object prediction regions with one CNN calculation for one image. There is no damage to

the original image, since the crop and resize operations from the existing technique have been omitted, and processing is completed using the ROI pooling layer in the CNN computed feature. However, region proposal selective search is still conducted outside of the neural network in Fast R-CNN, and region proposal selective search becomes the bottleneck of the entire performance. To solve this problem of Fast R-CNN, Faster R-CNN was proposed, combining RPN to Fast R-CNN. Instead of the slow external selective search, Faster R-CNN uses the fast internal RPN. It also has the advantage of sharing the detection network and convolutional feature map by applying the back-propagation algorithm of RPN. In this paper, Faster R-CNN model is adopted for pedestrian detection due to its fast object detection and excellent accuracy.

2.2 Deeplearning4j framework

Our deep learning programming is based on Deeplearning4j [19] which is open-source library based on java and JVM. It supports various deep learning algorithms such as the restricted Boltzmann machine, deep belief net, deep autoencoder, stacked denoising autoencoder and recursive neural tensor network, word2vec, doc2vec and Glove. These algorithms can be executed in parallel using Hadoop and Spark. Deeplearning4j is mainly written in Java and includes Scala application programming interface (API). Numerical computation uses ND4J.

Deeplearning4j's open-source library is compatible with both central processing units (CPUs) and GPUs. Deeplearning4j's learning process is structured to take advantage of distributed computing. In other words, it is possible to learn with repetitive MapReduce model based on Hadoop and Spark. In addition, it supports CUDA kernel, which enables computation on GPU and distributed learning using multiple GPUs. With Deeplearning4j framework, any combination of constrained Boltzmann machines, convolution neural networks, autoencoders and recurrent neural networks can be used.

2.3 Distributed parallel deep learning

In this section, we shall describe various techniques for distributed parallel deep learning including parallelism, parameter sharing and communication.

2.3.1 Parallelism

Distributed parallel training is used in deep learning for accelerating training, which can be classified into two ways: data parallelism and model parallelism. Figure 1a outlines the data parallelism method, where the input data set for learning is divided among multiple computers for training. In this method, the entire model is loaded into each distributed computer, and the input data are divided into several parts each of which is distributed among computers for training. Then, the weights computed by each computer are merged to update the whole parameters [20].

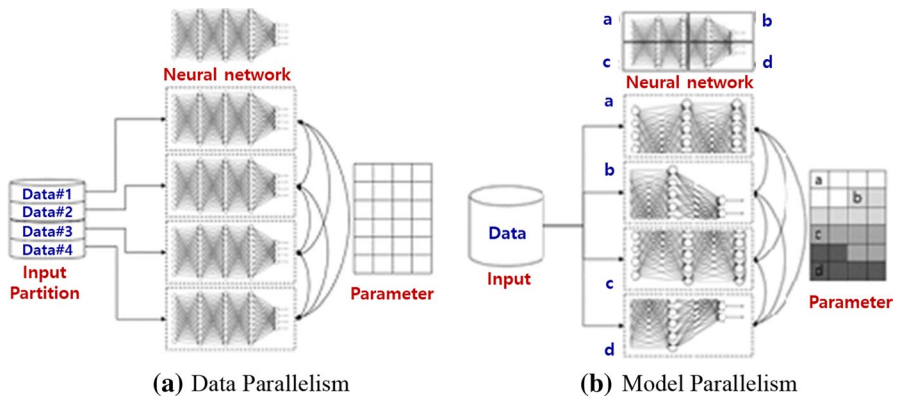


Fig. 1 Parallelism

Figure 1b outlines the model parallelism method, where multiple computers divide the deep learning model for training. Each computer performs training for the entire input data and exchanges propagation data which have been partially calculated with other distributed computers.

2.3.2 Parameter sharing

There are two ways to share parameters: full mesh topology-based sharing and star topology-based sharing technique [21, 22]. Figure 2a outlines full mesh technique, where each computer delivers the parameter to all the other computers directly. This approach has the advantage of not requiring separate shared storage management, but its expandability is reduced, since the number of communication increases by

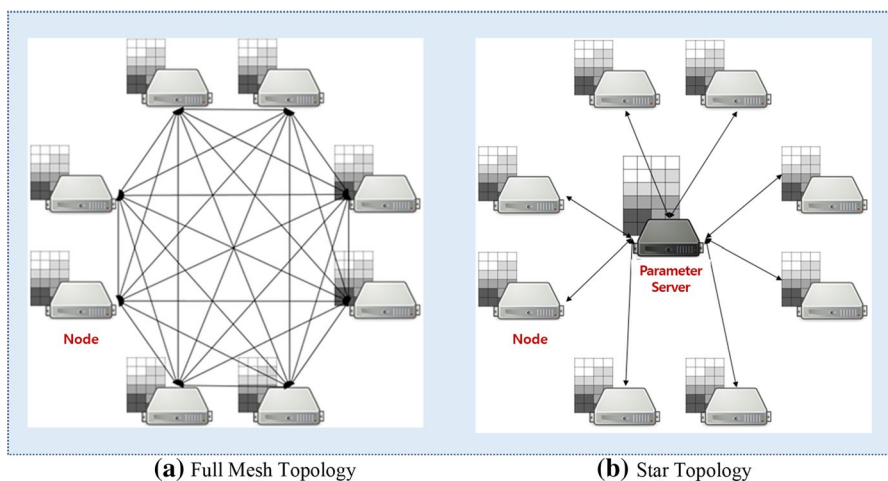


Fig. 2 Parameter sharing

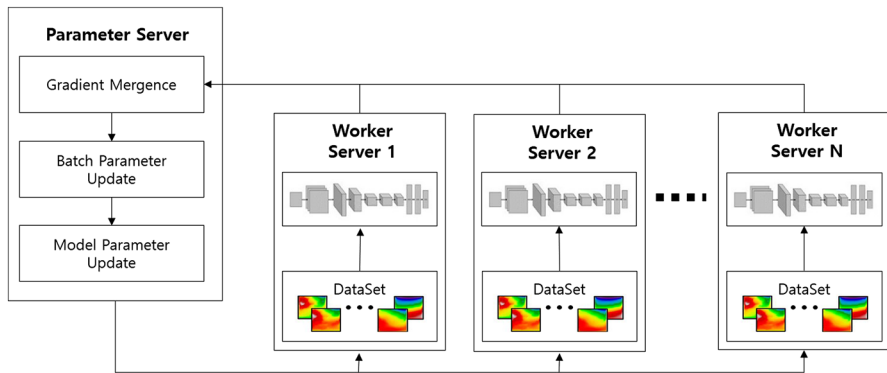


Fig. 3 Synchronous communication

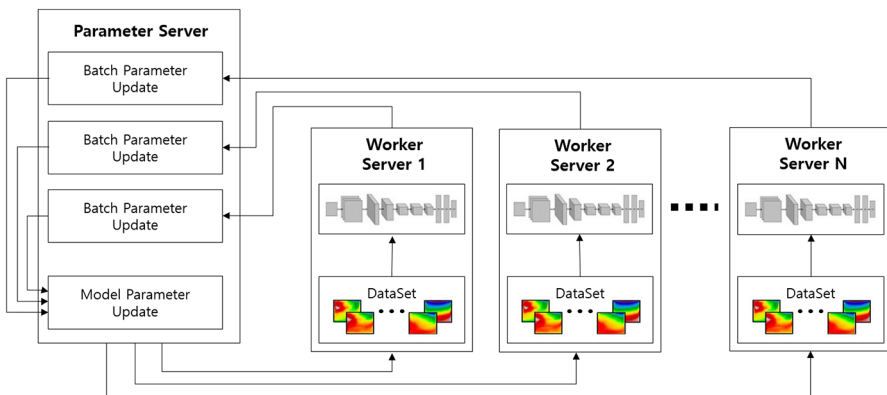


Fig. 4 Asynchronous communication

N^2 as the number of distributed computers increases. On the other hand, Fig. 2b outlines star approach, which requires the concurrency control and synchronization of the shared parameter storage, but has the advantage of better expandability, since communication does not increase when the number of computers increases.

2.3.3 Synchronous and asynchronous communication

In case of star topology-based parameter sharing technique, the parameter synchronization between the computers is necessary, since the distributed computers need to update their respective parameters in the central computer. In case of a synchronous update, the entire training performance is adjusted to the performance of the last computer which sends its parameter to the central computer, since the execution time differs for each distributed processing computer, depending on their performance and

the current workload. This is shown in Fig. 3. To solve this problem, an asynchronous update approach can be used as shown in Fig. 4. This involves executing the training without synchronizing the parameters received from each computer. Most of the distributed frameworks are based on the synchronous technique and provide additional asynchronous technique to partially improve the training speed [23]. Asynchronous processing is more complicated than synchronous processing and has the disadvantage of deadlock.

2.4 Distributed environment

2.4.1 Akka

In our system, Akka [24] plays an important role in driving system and communication between nodes, which is a free and open-source toolkit, and runtime simplifying the construction of concurrent and distributed applications on the JVM. It supports multiple programming models for concurrency, but it emphasizes actor-based concurrency, with inspiration drawn from Erlang. The actor model implemented by Akka is based on the mathematical model proposed by Carl Hewitt in 1973. Now that concurrent programming is becoming increasingly difficult to write in multithreading environments and the number of CPU cores used by a single computer is increasing rapidly, Akka provides an intuitive and convenient programming model for writing concurrent code. The actor model has several features. An object's methods cannot be called directly, and a message can be only delivered. It is basically asynchronous and non-blocking. It also works concurrently. These Akka features are very powerful. With Akka, it is possible to eliminate all or at least the sequential parts and blocking calls which exist throughout the program. In addition, Akka has the advantage of automatically guaranteeing scale-out.

2.4.2 HDFS

Hadoop Distributed File System (HDFS) [25] is a block-structured file system based on Google File System (GFS). Large files of over tens of terabytes can be stored in the distributed server, and storage can also be configured using the low-end server, which has advantages over existing large file systems (NAS, DAS, SAN, etc.). Network File System (NFS) is a general distributed file system that was developed for many users to access data in a network environment, but there is a restriction that only one logic volume stored in a single device can be accessed remotely [26]. HDFS was designed to overcome the limitations of NFS. It can store terabytes or petabytes by distributing data among multiple computers, supporting files much larger than NFS. Since the reliability of data storage in HDFS is high, individual computers can use the data even if they cause an error.

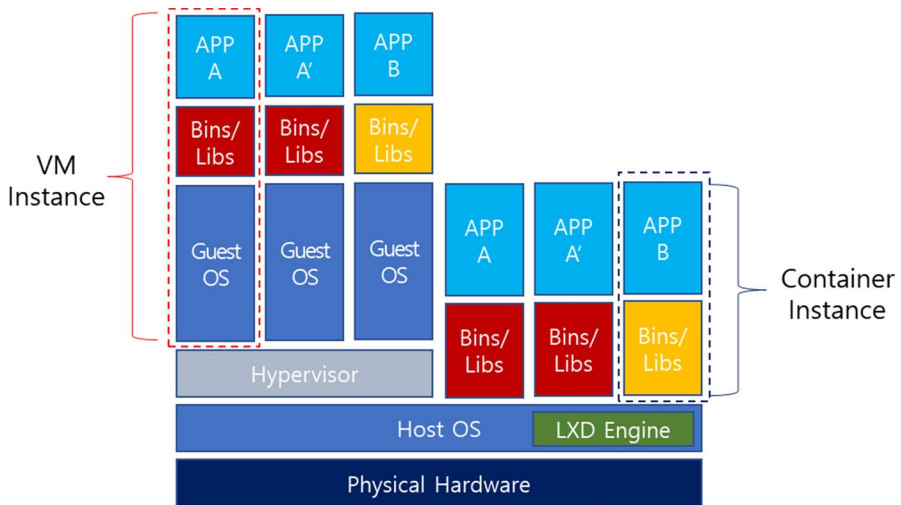


Fig. 5 Hypervisor and container virtualization

2.4.3 Container

Container, also known as OS virtualization, is the process of performing one operating system in another operating system. As shown in Fig. 5, the container shares the kernel of the original OS with other containers and has its own runtime, library, etc., to execute application programs. Unlike the existing hypervisor-based virtualization, the container can make efficient use of resources as it does not require guest OS and emulator.

Linux container (LXC) [27] is an OS-level virtualization technique for running multiple isolated Linux systems (containers) from a single control host. LXD [28] is a pure container hypervisor that performs Linux OS and VM-style operations at a remarkable speed and density. New Linux hypervisors that were developed to replace kernel-based virtualization, such as KVM container technology like Docker, have security issues. As a result, in spite of the speed and density that enables efficient resource utilization, containers have been used as a traditional hypervisor, without maximizing resources. However, LXD, a new pure container-centric hypervisor, uses the same technology as the Docker container to provide the same level of security as traditional hypervisors like KVM.

3 Distributed deep learning platform

In this chapter, we shall describe about system architecture for distributed deep learning system.

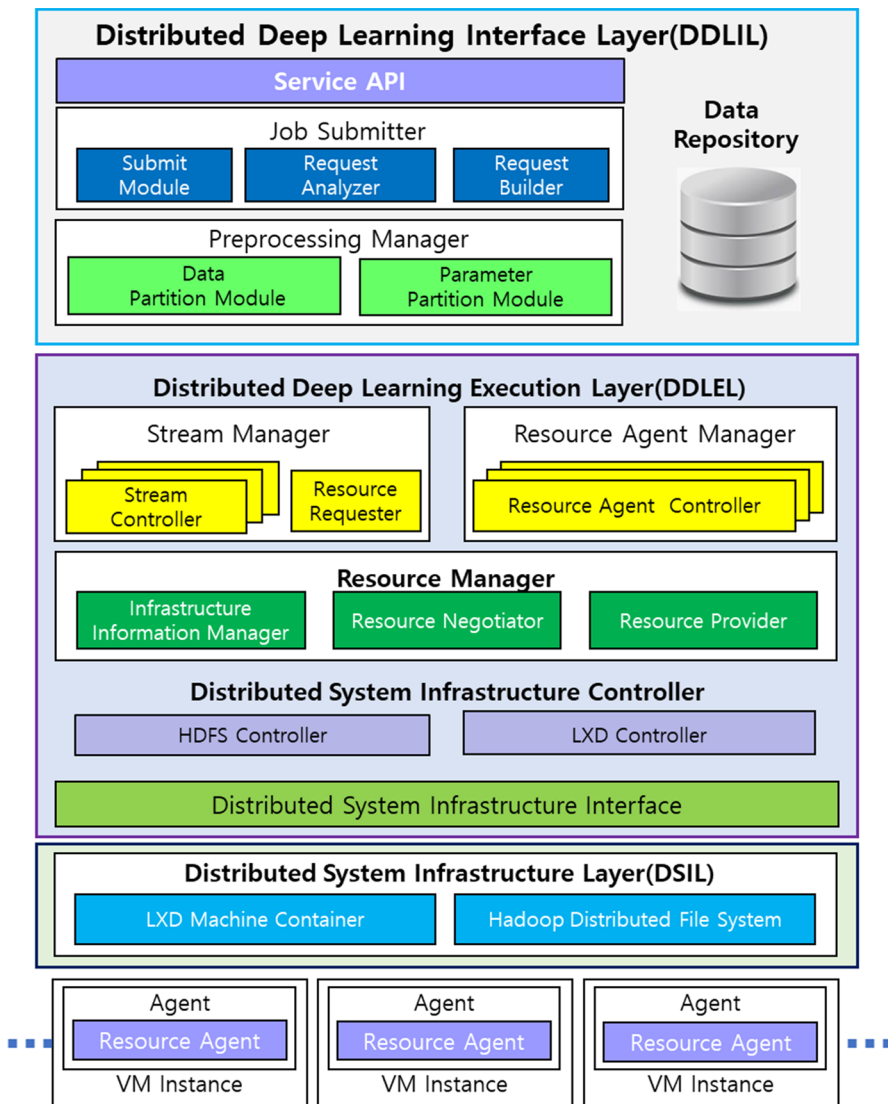


Fig. 6 System architecture

3.1 System architecture

Our distributed deep learning system consists of three layers: Distributed Deep Learning Interface Layer (DDLIL), Distributed Deep Learning Execution Layer (DDLEL) and Distributed System Infrastructure Layer (DSIL) as shown in Fig. 6. In DDLIL, user develops deep learning application programs and submits them to DDLEL, which in turn manages, allocates system resources in the distributed environment and then runs the programs. DSIL exploits HDFS for storing distributed

files and deploys virtual programming environment and deep learning model on container-based cloud infrastructure. Each layer shall be explained in detail in the following sections.

3.1.1 Distributed Deep Learning Interface Layer (DDLIL)

DDLIL is responsible for receiving the request of the user for running application, building a job and delivering it to the system. It consists of job submitter and parallel model manager as shown in Fig. 7. Job submitter consists of submit module, request analyzer and request builder. Submit module receives requests from users for application program execution, and request analyzer analyzes the user request such as parallelism, build file path, main class path, the number of parameter servers and slaves, etc., so that request builder builds parallel application program by interacting with parallel manager which generates a proper parallel model with data and parameter partition modules. Data partition module is responsible for the data parallelism of distributed deep learning. Each application allocates data according to the number of slave nodes and the size of total input data for distributed deep learning. Parameter partition module is responsible for the parameter parallelism of distributed deep learning. It appropriately allocates parameters of the model according to the number of parameter servers. The generated model is created as a job with various information, which is transmitted to DDLECP for deployment and running in the system.

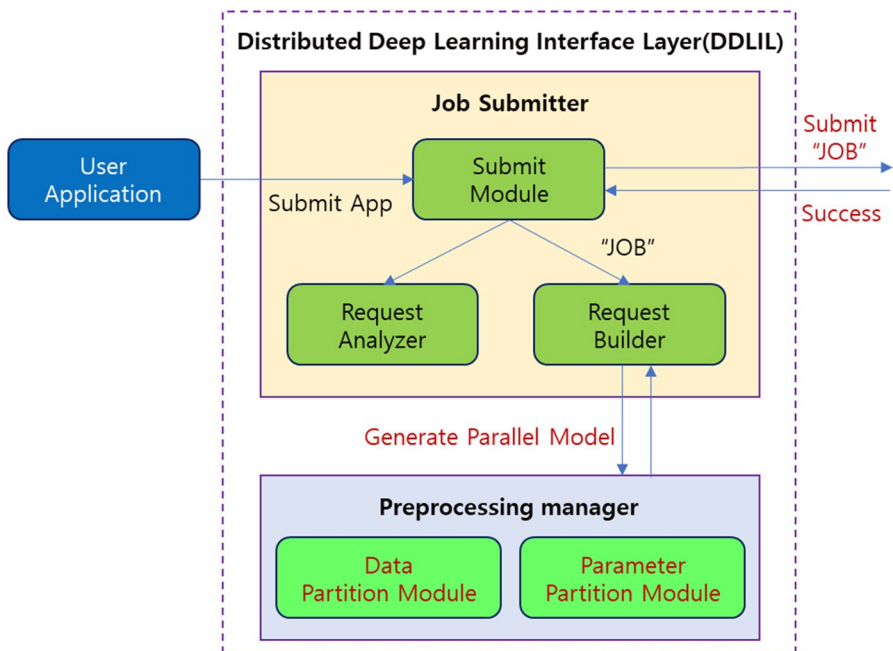


Fig. 7 Components of DDLIL

3.1.2 Distributed Deep Learning Execution Layer (DDLEL)

DDLEL manages the clustered resources, distributes the user application to the selected nodes and executes applications. The first is to manage cluster and infrastructure state. The second is to manage the resources to execute the application. The third is to deploy the job to the allocated resources. The forth is to run the application. Figure 8 outlines the control flow and configuration components.

When DDLEL receives a job from user request from DDLIL, stream manager generates a stream controller to handle the job. Stream manager manages the lifecycle of the stream controller. Stream manager creates stream controllers according to the number of jobs submitted from the DDLIL. Each stream controller corresponds to one job requested by the user and managers several resource agent controllers. Stream manager is maintained until the stream controller is finished. Stream controller requests resources to resource manager through resource requester. To communicate with the allocated resources, the stream controller generates resource agent controllers as many as the number of nodes allocated through the resource agent manager. The requested job is deployed to each agent through resource agent controller. Agent actually executes the deployed job. After receiving the job, resource agent generates a task for execution.

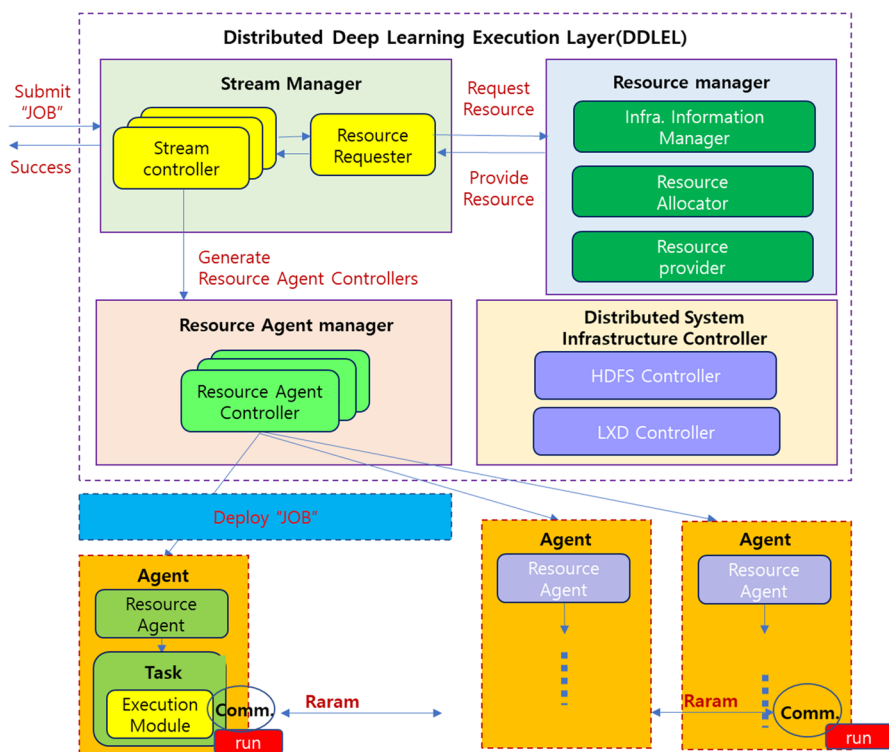


Fig. 8 Components of DDLEL

Resource agent directly communicates with the resource agent controller. Once the task is allocated to resource agent, the task is created to execute the user application program. Also, it communicates with infrastructure information manager of the resource manager periodically and sends the condition of the node. Task consists of the execution module and communication module for the user application. It manages the lifecycle of the user application program and is maintained until the application program is completed. Execution module runs the actual application program. The user application program is processed by loading the build file's class in the job, using the class loader. Communication module is used in the communication between nodes and occurs through the distributed processing of the user application program. It is used in the exchange of parameters between the parameter server and the slave during distributed deep learning processing.

Resource manager provides as much resources as user application needs. When the use of resources is completed, the corresponding resources are collected. If there are not enough resources for the user's request, the process is terminated and a warning is issued. Infrastructure information manager manages the status of resources. If one of the nodes in the cluster is killed, remove it from the resource pool. Resource negotiator is responsible for determining the appropriate resources according to the status of each resource when the stream controller requests resources through the resource requester. When the resource negotiator determines the requested resource, resource provider deploys the allocated resources through DSIL.

Stream manager also manages a buffer which consists of several topics each storing real-time images coming from source such as CCTV as shown in Fig. 9. Real-time images in each topic are distributed onto multiple partitions residing in the single or distributed broker nodes based on the file system using Kafka. Each of multiple partitions in the topic is processed by a single node so that multiple partitions can be processed simultaneously for deep learning as a preprocessing step.

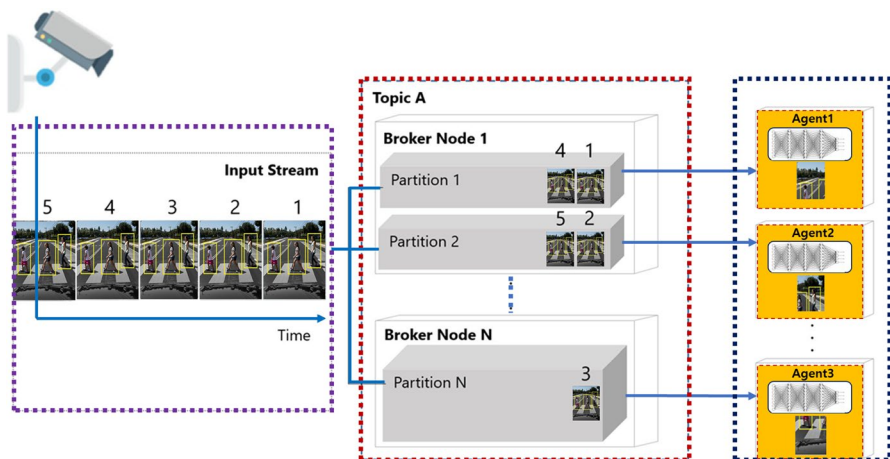


Fig. 9 Stream processing for real-time image

3.1.3 Distributed System Infrastructure Layer (DSIL)

DSIL provides container-based cloud and HDFS for distributed file storage as shown in Fig. 10. Distributed System Infrastructure refers to the infrastructure on which our system operates. Distributed System Infrastructure consists of LXD controller and HDFS controller. LXD controller is used to create a container-based virtual machine on the physical node and set the usage amount of each resource. HDFS controller is used to create Hadoop Distributed File System (HDFS) for storing the inputs of the deep learning process. Since HDFS stores files in block units on the distributing nodes, it can reduce the network load caused by requesting the files from multiple nodes.

3.2 System control flow

In this section, we shall describe the control flow as well as a hierarchy of our system as shown in Fig. 11. The control flow is largely divided into two parts: DDLIL and DDLEL. An application written by the user is delivered to DDLIL responsible for building and submitting it to our system. The application delivered to the DDLIL is created as a job containing a variety of information for running applications on our system. One example of the information is about the parallelism of the model. The job is delivered to DDLEL, which in turn requests resources from resource manager to run the application. It also deploys the job to the agent of the allocated distributed resource. Each agent receives a job and generates a task to execute the application and creates a communication module in the distributed deep learning application. Then, the application is executed as a task by agent.

Our system has a software hierarchy diagram as shown in Fig. 12. Physical nodes are distributed at the bottom to explain at the lowest level, which provides real resources for running the deep learning application program. Next is LXD machine container for providing resource virtualization service and then HDFS layer for loading input files of the application program. It provides input data to the

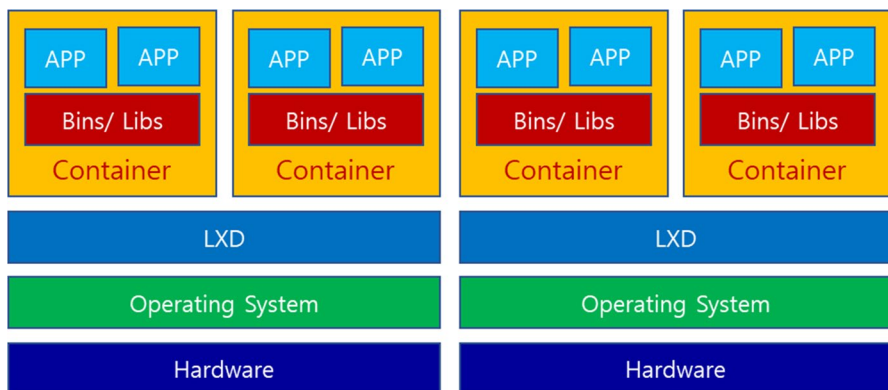


Fig. 10 LXD container

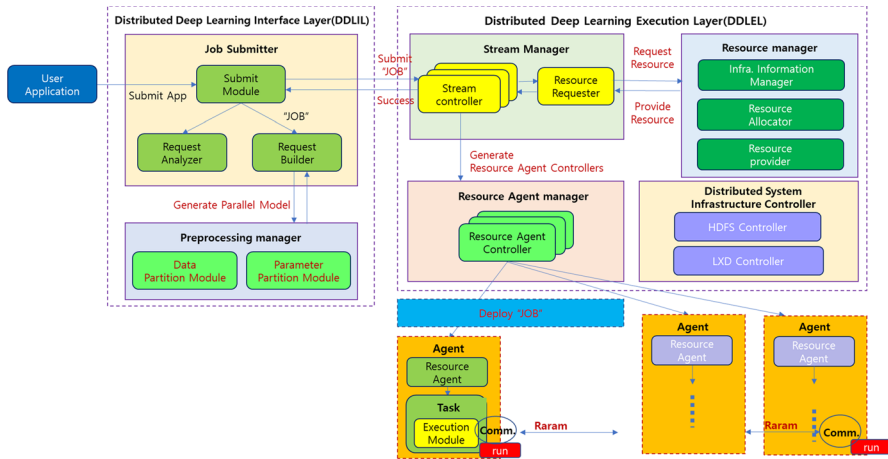


Fig. 11 Control flow of system

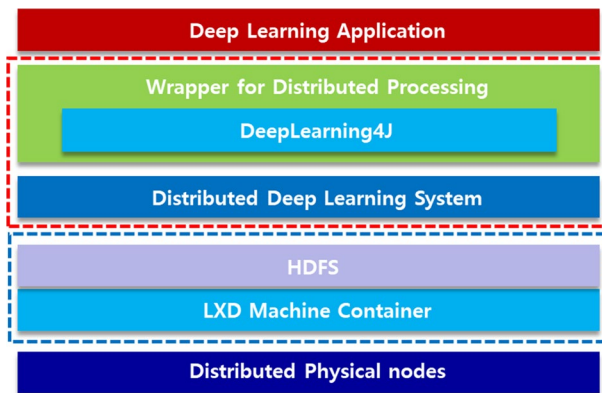


Fig. 12 System software diagram

distributed nodes for data parallelism. As mentioned previously, resources can be used efficiently if OS-level virtualization is used. Next is distributed deep learning system software for supporting distributed and cloud environment. Our proposed deep learning system uses Deeplearning4j and provides a wrapper for deep learning application.

4 Distributed deep learning processing

Distributed deep learning processing is executed as follows: First, input data are distributed among slaves for distributed deep learning. All parameters are also distributed among parameter servers. Each slave receives parameters from

parameter servers and then finds the gradient by processing the part of input data with specified batch size, based on the imported parameters. The calculated gradients are sent back to the parameter servers, each updating its corresponding parameters. This train process is repeated until sufficient accuracy is obtained.

4.1 Data and parameter parallelism

Our system provides two kinds of parallelism in the distributed environment: data parallelism and parameter parallelism. For the former, the whole model is loaded for each slave, and the input data is distributed among slaves for training as shown in Fig. 13. Each time the training process in each slave is repeated, the modified gradients are exchanged with the parameter servers. Since deep learning training processes very large input data, data parallelism can be an efficient method to reduce the overall training time through parallel processing.

The size of overall model parameters is very large when large-scale deep learning processing is performed as shown in Fig. 14. Therefore, it may incur a network bottleneck during parameter exchange in distributed environment. In order to solve this problem, parameter parallelism is used by distributing the entire parameters across several parameter servers, thus reducing the communication overhead arising in the centralized parameter server.

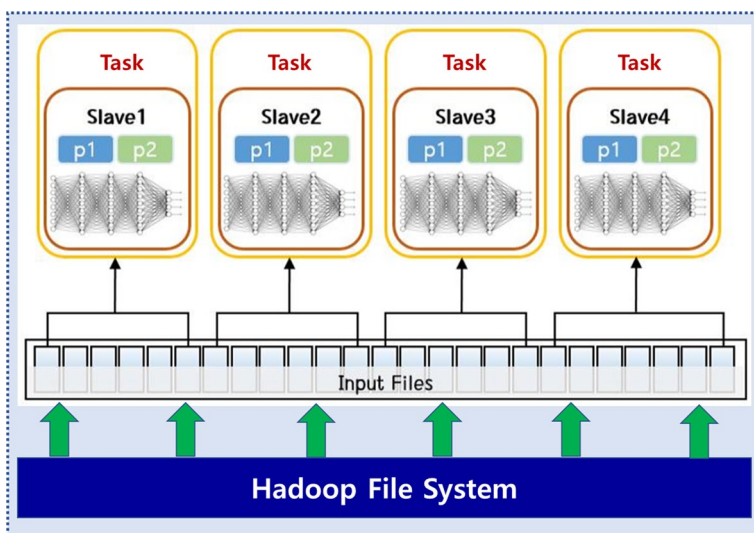


Fig. 13 Data partition

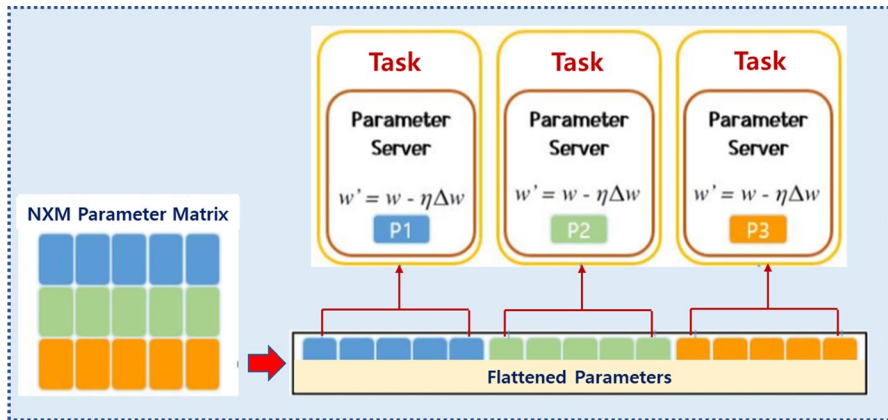


Fig. 14 Parameter partition

4.2 Parameter exchange method

There are two methods for parameter exchange: asynchronous and synchronous. Figure 15 shows parameter exchange method for both methods. Our system adopts the asynchronous method for the parameter exchange method.

Synchronous method, when performing updates on the parameter servers, is performed synchronously on all nodes. It has advantage for accuracy and implementation cost. However, synchronous method causes all the other slaves to wait, while the parameter server performs each update. Therefore, the overall performance is adjusted to that of the worst performing node, resulting in performance degradation. In other words, each slave node which has already performed the update should

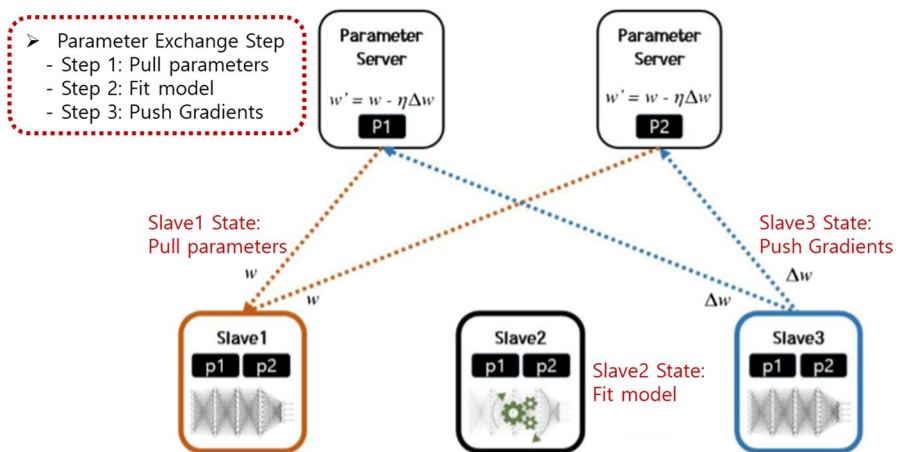


Fig. 15 Parameter exchange

wait until the latest update of the gradient is performed. On the other hand, in the asynchronous mode, each slave communicates asynchronously with the parameter servers to update parameter. For this reason, asynchronous method can be quickly trained without sacrificing accuracy as compared to synchronous method. Asynchronous method performs updates more concurrently.

Generally, many distributed frameworks are based on synchronous methods. In particular, Deeplearning4j, the foundation of our deep learning process, implements distributed processing based on the Apache Spark, which is a synchronous model based on MapReduce as a large-scale distributed processing framework. Therefore, distributed deep learning processing based on Apache Spark exchanges parameters in a synchronous manner. As a result, performance is somewhat lower than that of the asynchronous method. However, although our system is based on Deeplearning4j, we shall provide a way for exchanging the parameters asynchronously without deploying Apache Spark, improving the overall performance.

5 Performance evaluation

For the experimental evaluation, Akka and Deeplearning4j libraries are used in our system for distributed deep learning processing. Akka is open-source toolkit for simplifying the construction of concurrent and distributed programming. It enables us to build our distributed system including clustering, deployment of job, resource management and so on. Our deep learning processing is based on Deeplearning4j, an open-source deep learning library written in Java. It supports a variety of deep learning models, including limited Boltzmann machines, CNN, recurrent neural network and so on. We provide DistMultiLayerNetwork class to run Deeplearning4j in distributed mode on our system, which is a class wrapping MultiLayerNetwork class of Deeplearning4j. The application program written by the user is submitted to the system by executing the main submit class, which analyzes the request, creates the task and sends it to the submit module. It generates Akka Actor System before sending the task to submit module, since submit module is Akka's actor,

5.1 System environment

Our system environment for performance evaluation consists of 8 container systems written by LXD based on the OS virtualization technology from 4 physical nodes each with 8 CPU's (Intel (R) Xeon (R) CPU E5606 @ 2.13 GHz) and 24 GB DDR3 main memory. Each container system is allocated 4 CPU's with 4 cores and 10 GB of RAM. Performance is evaluated under various experiments. Figure 16 shows our experimental environment.

We experiment KITTI data set for pedestrian detection on Faster R-CNN deep learning model. We measure performance by changing the number of parameter servers and slaves as well as the placement of parameter servers and slaves. For performance evaluation, we are more concerned with comparing synchronous and asynchronous modes for various configurations in terms of the overall time taken for

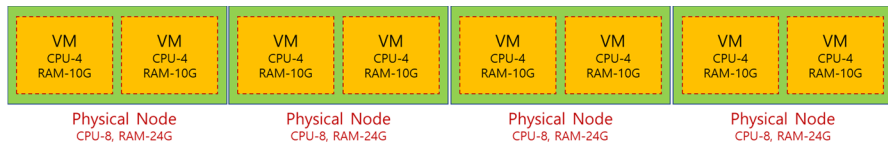


Fig. 16 Experiment environment

the whole training process, since it is hard to differentiate the communication time from computation time in asynchronous mode. We try to find the optimal placement from the viewpoint of the whole training process performance by showing that asynchronous mode is better than synchronous one due to the overlapped duration between computation and communication rather than from communication/computation cost.

Our data set for the pedestrian detection experiment on Faster R-CNN is a set of images from KITTI data set. It consists of 7481 training sets and 7518 test sets, each with 1242×375 image size. Figure 17 shows examples of our KITTI data sets.

5.2 Experimental result

First, we compare performance when assigning various placements to VMs. Table 1 shows four experimental configurations with the fixed number of slaves and parameter servers: four slaves in one VM, two slaves in two VMs in one node and two nodes, respectively, one slave in one VM in four nodes. Figure 18 shows performance for each configuration for both of synchronous and asynchronous cases. Experimental result shows that if multiple slaves are placed in a single

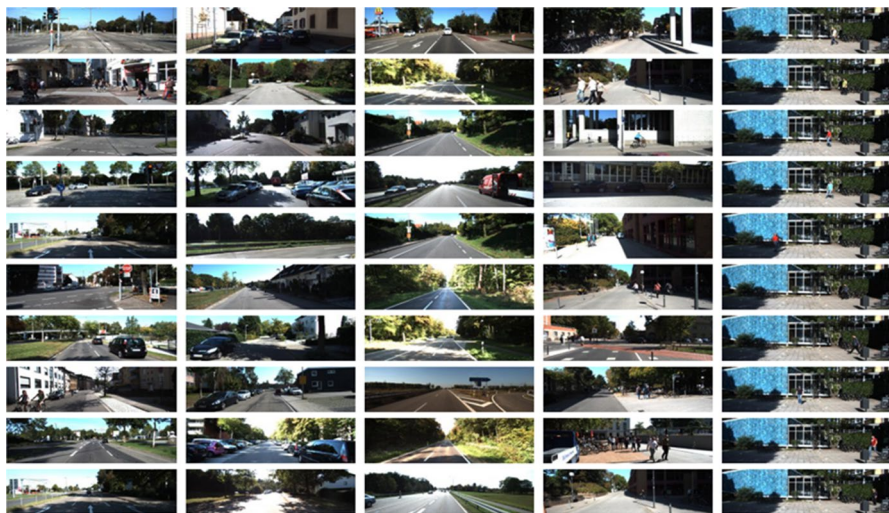
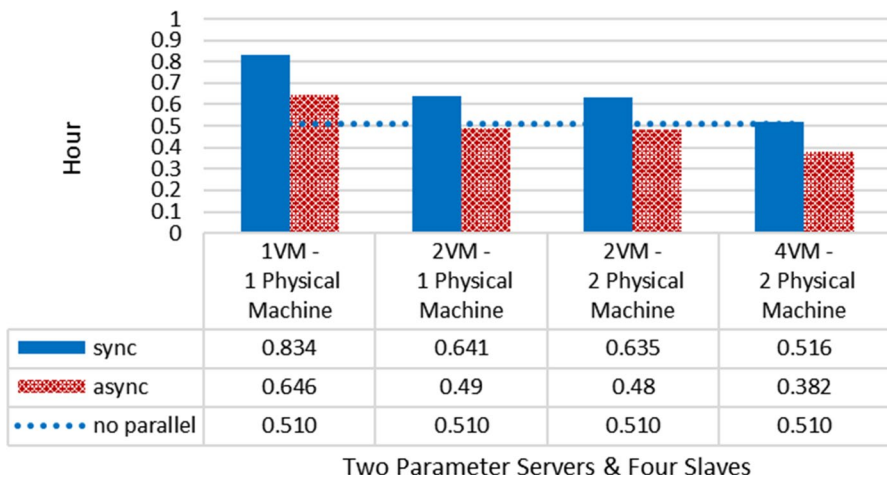


Fig. 17 KITTI data sets

Table 1 Placement configurations of parameter servers and slaves

	Placement							
Config.1	Slave CPU-4 RAM-10G	Slave CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	Param Server CPU-4 RAM-10G	Param Server CPU-4 RAM-10G
Config.2	Slave CPU-4 RAM-10G	Slave CPU-4 RAM-10G	Slave CPU-4 RAM-10G	Slave CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G
Config.3	Slave CPU-4 RAM-10G	Slave CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	Param Server CPU-4 RAM-10G	Param Server CPU-4 RAM-10G
Config.4	Slave CPU-4 RAM-10G	Slave CPU-4 RAM-10G	Slave CPU-4 RAM-10G	Slave CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G

**Fig. 18** Performance for each placement configuration

VM, the performance is not improved, since CPUs of single VM are shared. In other words, assigning one role of parameter server or slave to one VM shows the best performance.

Next, we measure the performance by varying the number of parameter servers and slaves. First, we compare the performance by increasing the number of slaves each in a single VM with respect to different number of parameter servers as shown in Figs. 19, 20 and 21. Both of synchronous and asynchronous cases show that performance increases and then decreases as the number of slaves increases, since frequent networks occur as the slave increases. In particular, the asynchronous case shows better performance than the synchronous one. The best performance case is for one parameter server and five slaves in asynchronous mode. In this case, it is 1.98 times faster than when it is not parallelized and 1.66 times faster than the synchronization case. In addition, it is 1.31 times faster than the best performance when using multiple parameter servers, 4 parameter servers and 4 slaves.

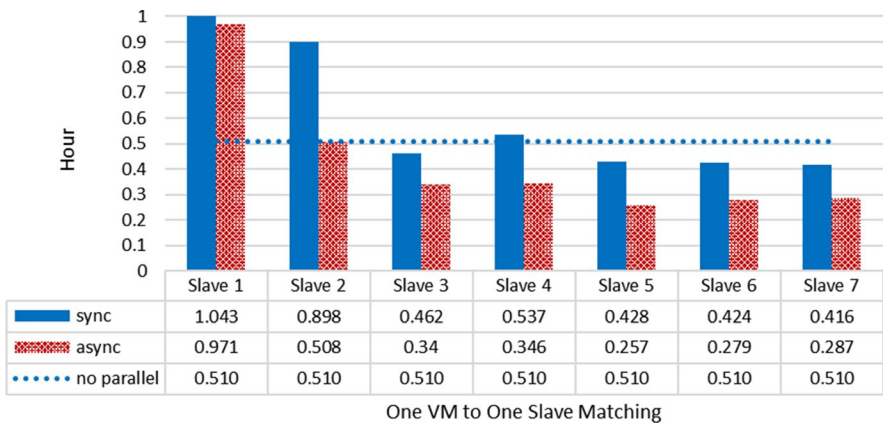


Fig. 19 Performance with respect to different number of slave nodes on 1 parameter server

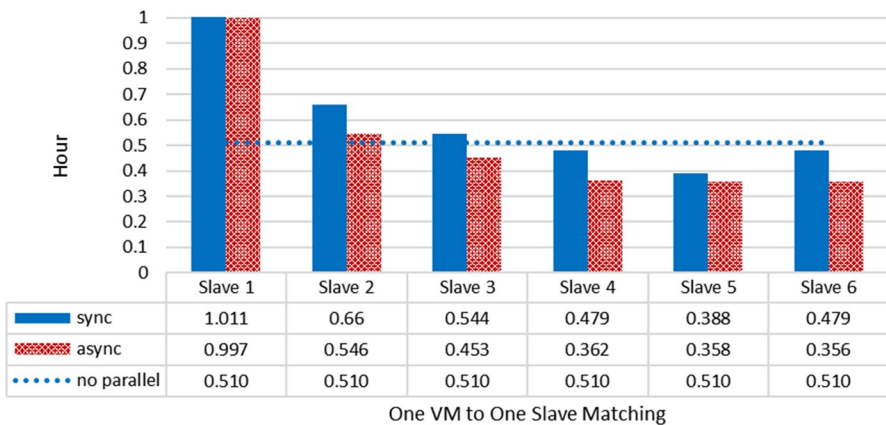


Fig. 20 Performance with respect to different number of slave nodes on 2 parameter servers

Next, we compare the performance by increasing the number of parameter servers with respect to different number of slaves as shown in Figs. 22, 23 and 24. This experimental result shows that the performance does not always increase when the number of parameter servers increases. Rather, the overall performance decreases, since the size of the parameters is rather small. The best performance can be achieved by selecting the number of parameter servers according to the number of parameters and data distributed to each node and the network speed.

5.3 Optimal placement of slave and parameter server

In this section, we suggest a placement strategy of parameter servers and slaves to get optimal performance of distributed deep running. It attempts to reduce the

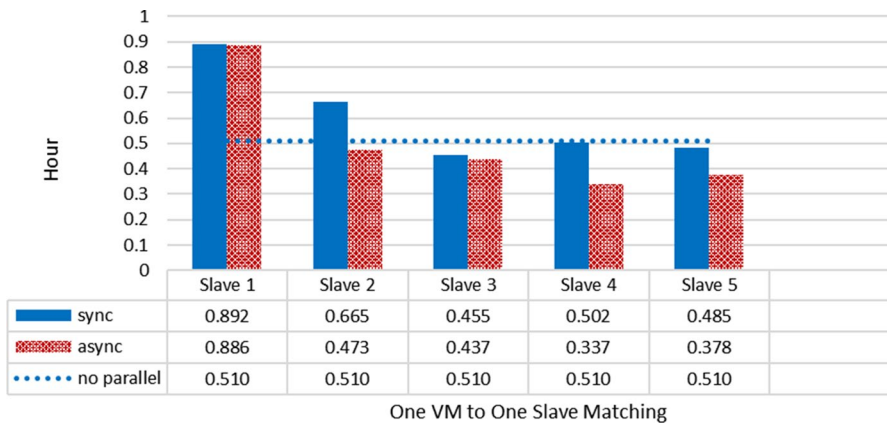


Fig. 21 Performance with respect to different number of slave nodes on 3 parameter servers

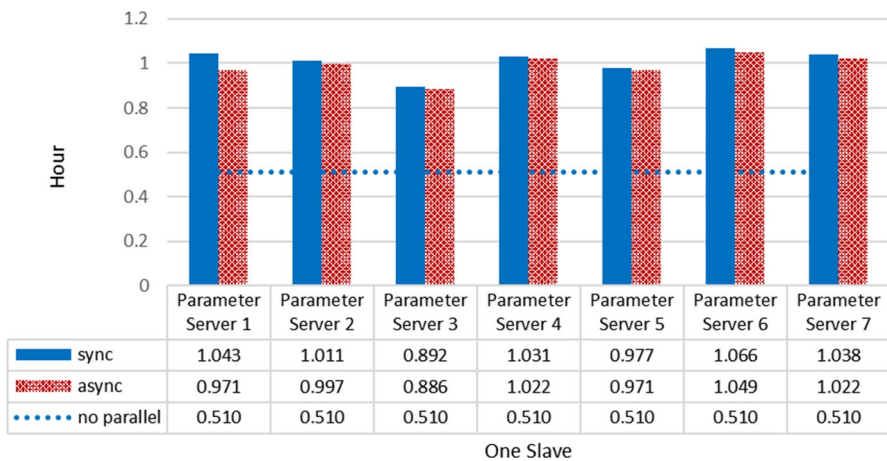


Fig. 22 Performance with respect to parameter server on 1 slave

number of remote communications that occur in parameter exchanges by placing a parameter server and a slave on each VM in a physical node. Table 2 shows the placement of these strategies. Figure 25 shows that this placement is performed better than the randomly placed method. In optimal results, the asynchronous communication method is 1.55 times faster than that without parallelism and 1.27 times faster than the synchronous communication method with 4 parameter servers and 4 slaves.

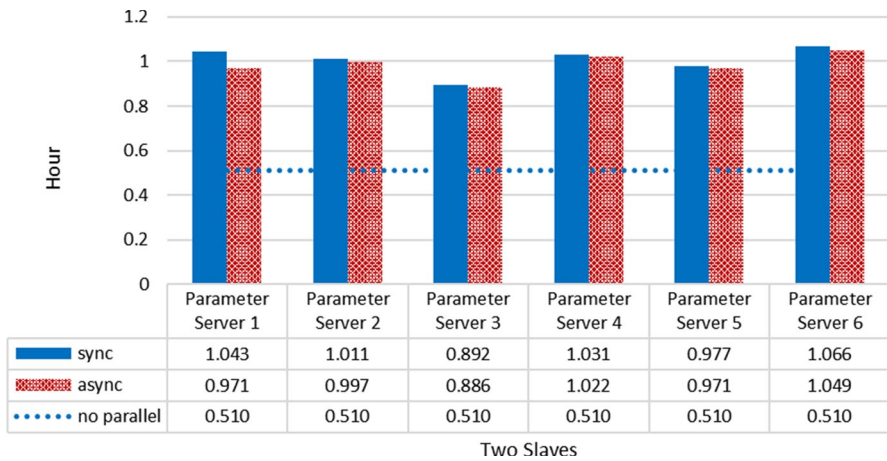


Fig. 23 Performance with respect to parameter server on 2 slaves

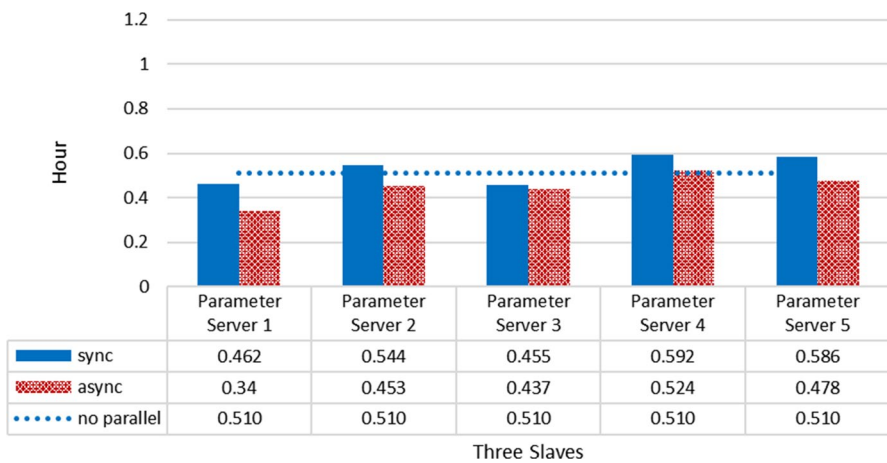


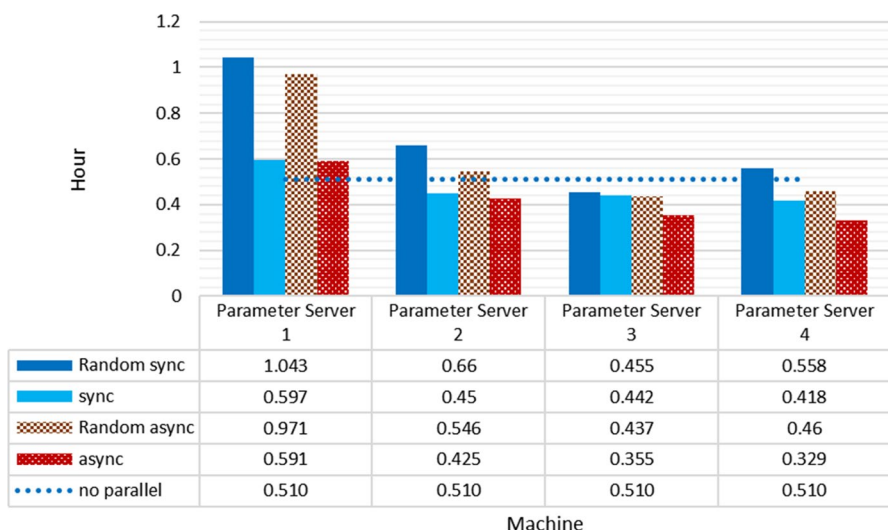
Fig. 24 Performance with respect to parameter server on 3 slaves

6 Conclusion

In this paper we have presented the distributed deep learning platform which can easily build a cluster, and execute deep learning process fast in the distributed cloud environment while achieving performance improvement. Our platform provides a convenient interface for easily and efficiently executing the deep learning process in a distributed environment by providing a system architecture which consists of three layers: DDLIL, DDLEL and DSIL. In DDLIL, user develops deep learning application programs and submits them to DDLEL, which in turn allocates system resources in the distributed environment and then runs the programs. DSIL provides

Table 2 Placements of parameter servers and slaves

	Placement							
Config.1	Slave M CPU-4 RAM-10G	Param Server M CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G
Config.2	Slave M CPU-4 RAM-10G	Param Server M CPU-4 RAM-10G	Slave M CPU-4 RAM-10G	Param Server M CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G
Config.3	Slave M CPU-4 RAM-10G	Param Server M CPU-4 RAM-10G	Slave M CPU-4 RAM-10G	Param Server M CPU-4 RAM-10G	Slave M CPU-4 RAM-10G	Param Server M CPU-4 RAM-10G	VM CPU-4 RAM-10G	VM CPU-4 RAM-10G
Config.4	Slave M CPU-4 RAM-10G	Param Server M CPU-4 RAM-10G	Slave M CPU-4 RAM-10G	Param Server M CPU-4 RAM-10G	Slave M CPU-4 RAM-10G	Param Server M CPU-4 RAM-10G	Slave M CPU-4 RAM-10G	Param Server M CPU-4 RAM-10G

**Fig. 25** Performance with respect to optimal placement

HDFS for storing distributed files and deploys virtual programming environment and deep learning model on container-based cloud infrastructure. Our system can easily build clustering and run distributed processing in a distributed environment using one simple command.

The various features of our platform can efficiently handle deep learning processing efficiently in a distributed environment. Our system utilizes computing power in easy and efficient way by leveraging container technique, so-called OS-level virtualization, rather than traditional hypervisor-based virtualization. We achieve performance improvement in various ways. First, we have improved the whole performance by exploiting both of data and parameter parallelisms at once. Next, we have reduced the synchronization overhead by exploiting asynchronous communication for parameter updates. Also, we have proposed an efficient resource allocation

scheme for parameter servers and slaves which can improve the performance from the experiment. When parameter servers and slaves are placed in VMs of different physical nodes, remote communication occurs frequently. We have suggested a way to improve performance by reducing the frequency of remote communication by placing parameter servers and slaves on the same physical node. Experimental results have shown that our scheme for distributed deep learning provides higher performance on processing time by adopting asynchronous communication method. In both experiments, asynchronous communication method has shown better performance than synchronous communication method. In addition, the strategy of placing parameter servers and slaves on each VM of a single physical node is able to boost performance.

As a future work, we are going to further develop distributed parallel platform for fast deep learning in other computation-intensive applications.

Acknowledgements This study was supported by 2019 Research Grant from Kangwon National University and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2017R1D1A1B03035461) and the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2018-2014-1-00720) supervised by the IITP (Institute for Information and communications Technology Promotion) and the Soonchunhyang University Research Fund. And this research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2019-2014-1-00720) supervised by the IITP (Institute for Information and communications Technology Planning and Evaluation).

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.


References

1. Lee SH, Chu SK, Kwon KB, Cho NI (2017) Pedestrian detection and re-identification for intelligent CCTV systems. *J Korean Inst Commun Sci* 34(7):40–47
2. <http://www.yonhapnews.co.kr/bulletin/2017/01/16/0200000000AKR20170116036600014.html>
3. Song SH, Hyeon HB, Lee H (2017) A pedestrian detection method using deep neural network. *J KIISE* 44(1):44–50
4. Dalal N, Triggs B (2014) Histograms of oriented gradients for human detection. In: *CVPR*, pp 886–893
5. Zhang S, Bauckhage C, Cremers AB (2014) Informed haar-like features improve pedestrian detection. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*
6. Ahonen T, Hadid A, Pietikainen M (2006) Face description with local binary patterns: application to face recognition. *IEEE Trans Pattern Anal Mach Intell* 28(12):2037–2041
7. Lowe DG (2004) Distinctive image features from scale-invariant keypoints. *Int J Comput Vis* 60(2):91–110
8. Joachims T (2006) Training linear SVMs in linear time. In: *Proceedings of International Conference on Knowledge Discovery and Data Mining (KDD)*, Philadelphia, Pennsylvania, pp 217–226

9. Gerónimo D, Sappa AD, López A, Ponsa D (2006) Pedestrian detection using Adaboost learning of features and vehicle pitch estimation. In: *PROceediNGS of the International Conference on Visualization, Imaging and Image Processing (IASTED)*, Palma de Mallorca, Spain, pp 400–405
10. Girshick R, Donahue J, Darrell T, Malik J (2014) Rich feature hierarchies for accurate object detection and semantic segmentation. In: *CVPR*, pp 580–587
11. He K, Zhang X, Ren S, Sun J (2015) Spatial pyramid pooling in deep convolutional networks for visual recognition. In: *ECCV*, pp 346–361
12. Sermanet P, Eigen D, Zhang X, Mathieu M, Fergus R, LeCun Y (2013) Overfeat: integrated recognition, localization and detection using convolutional networks. [arXiv:1312.6229](https://arxiv.org/abs/1312.6229)
13. Kim J, Lee JK, Lee KM (2016) Accurate image super-resolution using very deep convolutional networks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*
14. Ren S, He K, Girshick R, Sun J (2015) Faster R-CNN: towards real-time object detection with region proposal networks. In: *NIPS*, pp 91–99
15. Hwang D (2010) Cloud platform based on container for distributed deep learning. Korea University, Seoul
16. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) Spark: cluster computing with working sets. *HotCloud* 10:10
17. Koo K-M, Cha E-Y (2017) Image recognition performance enhancements using image normalization. *Hum Centric Comput Inf Sci* 7(33):1–11
18. He K, Zhang X, Ren S, Sun J (2014) Spatial pyramid pooling in deep convolutional networks for visual recognition. In: *ECCV*, pp 346–361
19. ND4j. <http://nd4j.org/>
20. Mosa A, Paton NW (2016) Optimizing virtual machine placement for energy and SLA in clouds using utility functions. *J Cloud Comput Adv Syst Appl* 5:17
21. Ozaki Y, Yano M, Onishi M (2017) Effective hyperparameter optimization using Nelder–Mead method in deep learning. *IPSJ Trans Comput Vis Appl* 9:20
22. Kim J-J (2017) Hadoop based wavelet histogram for big data in cloud. *J Inf Process Syst* 13(4):668–676
23. Akka. <http://akka.io/>
24. Shvachko K, Kuang H, Radia S, Chansler R (2010) The hadoop distributed file system. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp 1–10
25. Moon YJ, Yu HC, Gil J-M, Lim JB (2017) A slave ants based ant colony optimization algorithm for task scheduling in cloud computing environments. *Hum Centric Comput Inf Sci* 7(1):28
26. LXC. <https://linuxcontainers.org/lxc/>
27. LXD. <https://linuxcontainers.org/lxd/>
28. Deeplearning4j. <https://deeplearning4j.org/>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Seong-Soo Han¹  · Yoon-Ki Kim² · You-Boo Jeon³ · JinSoo Park³ · Doo-Soon Park³ · DuHyun Hwang² · Chang-Sung Jeong²

Seong-Soo Han
postsky0@korea.ac.kr

Yoon-Ki Kim
vardin@korea.ac.kr

You-Boo Jeon
jeonyb@sch.ac.kr

JinSoo Park
vtjinsoo@gmail.com

Doo-Soon Park
parkds@sch.ac.kr

DuHyun Hwang
doohh88@korea.ac.kr

- ¹ Visual Information Processing, Korea University, Seoul, Republic of Korea
- ² Department of Electrical Engineering, Korea University, Seoul, Republic of Korea
- ³ Department of Computer Software Engineering, Soonchunhyang University, Asan-si, Chungcheongnam-do 31538, Republic of Korea