Consideration of Parallel Data Processing over an Apache Spark Cluster

Kasumi Kato*, Atsuko Takefusa†, Hidemoto Nakada‡ and Masato Oguchi*
*Ochanomizu University †National Institute of Informatics ‡National Institute of Advanced Industrial Science and Technology (AIST)

I. INTRODUCTION

The Spread of cameras and sensors and cloud technologies enable us to obtain life logs at ordinary homes and transmit the captured data to a cloud for life log analysis. However, the amount of processing for video data analysis in a cloud drastically increases when a very large number of homes send data to the cloud.

In this research, we aim to improve the efficiency of distributed video data analysis processing by using the parallel deep learning framework Chainer [2] and the distribution processing platform Apache Spark [1] (Spark). In this paper, we construct a Spark cluster and investigate the performance of parallel data processing using Spark varying parameter settings.

II. RELATED TECHNOLOGY

A. Apache Spark

Spark is a fast and versatile cluster computing platform designed for large-scale data processing. Spark was developed by the University of California, Berkeley and was donated to the Apache Software Foundation in 2014. Spark can perform batch processing in a minimum unit, called micro-batch process. Because computation is performed on-memory in the Spark platform, the performance of data processing is improved avoiding expensive disk access. Processed data are stored in Resilient Distributed Dataset (RDD) defined by the Spark core and the RDDs are automatically distributed. The Spark project consists of multiple components tightly coupled together, such as Spark core, Spark SQL, MLlib [5]. The Spark core includes Spark's basic functions such as task scheduling, memory management, disaster recovery, and interaction with the storage system. By operating RDD, it is possible to easily perform a wide range of data processing including repetitive machine learning, streaming, complicated queries, batches. Spark can be combined with other big data tools such as Hadoop [4], and has excellent versatility.

B. Chainer

Chainer is a deep learning framework developed by Preferred Networks. It is provided as a Python library, and has three outstanding characteristics, "Flexible", "Intuitive" and "Powerful". Chainer enables users to install easily, write a code intuitively with a simple notation, and perform high-performance operation utilizing GPU with CUDA support. It is used in a wide range of fields including image processing, natural language processing, and robot control.

The most notable feature of Chainer is so-called 'define-by-run' nature, meaning it constructs computational graphs when it runs code at the first time. In contrast, most other deep learning frameworks take a different approach called 'define-and-run', where graph construction is performed in advance as a separate step. This feature is quite useful to debug complicated deep neural networks.

III. EXPERIMENTS

A. Outline of Experiment

We investigate the performance of parallel data processing using Spark varying parameter settings. We construct a Spark cluster for parallel machine learning processing using Chainer as shown in Fig. 1. The Spark cluster consists of a master node and worker nodes and the master node executes a Python program to read data and create RDDs of the data. We use MNIST[3], which is an image data set of handwritten digits of 28×28 pixels from 0 to 9, to be analyzed. Then the master node passes the RDDs to the worker nodes and the worker nodes evaluate them using Chainer. In the experiments, we use Spark v. 2.2.0 and Chainer v. 2.0.2. The Spark cluster consists of 1 master and 4 worker nodes connected in the Spark standalone mode and the details of each node are shown in the table III-A. The Spark cluster configuration is shown in the Fig. 2. Spark has the concept of partitions to split each RDD for processing and we can specify the number of partitions by using the method called partitionBy() in the program as follows:

\[
\text{partitionBy}(\text{numPartitions}, \text{partitioner} = \text{portable_hash})
\]
In this paper, we perform parallel data processing changing the following two parameters.

1) locality wait time
2) partitioner

1) locality wait time is the parameter that specifies a standby time before task allocation and 2) partitioner is the parameter that decide how to part the group of task. Spark implements a scheduling scheme called "delay scheduling" to achieve compatibility between Fairness and Locality [6]. If the job to be scheduled next is not able to be allocated on a node which stores processed data considering fairness, it makes the job wait and the other jobs start instead. In this paper, we call this standby time, "locality wait time", and specify it in the configuration file, we experiment by changing the specific gravity of Fairness and Locality. Since the partitioner is defined as portable_hash by default, we can adjust the partitioning of tasks by changing portable_hash.

B. Locality Wait Time

Locality wait time is set to 3 seconds by default. In this paper, locality wait time is set to 3 seconds and 0 seconds, which means a state not considering data locality. We divide 40 tasks into partitions and use 3 workers. Fig. 3 shows the behavior of task processing when the number of partitions is set to 9 and locality wait time is set to 3s. The horizontal axis shows the elapsed time, and the vertical axis represents nodes where tasks are processed. Each arrow in the figure indicates the elapsed time of each task processed in each node, and a group of arrows continuing in the upper right direction means a partition. We can see that task is biased in a certain node, and processing can not be performed efficiently. We assume that this phenomenon is caused by locality wait time. Fig. 4 shows the behavior of task processing when the number of partitions is set to 9 and locality wait time is set to 0. With this parameter adjustment, all nodes are used, but the total execution time was not improved much.

C. Partitioner

We divide 1,000 tasks into partitions and use 4 workers. Fig. 5 shows the behavior of task processing when the number of partitions is set to 32. We can see that the number of tasks in a partition is not uniform, so the total execution time becomes longer. Therefore, we modified portable_hash which decides how to make the number of data in a partition so that the partition sizes are adjusted to be equal. Portable_hash is an argument of partitionBy(). Fig. 6 shows the behavior of task processing after modification. The number of partitions is also set to 32. Fig. 6 shows that the number of tasks in each partition became almost uniform and tasks are processed in parallel. However, the number of partitions processed in each node decreases and partitions are executed consecutively in the time direction.

Fig. 7 shows the behavior of task processing when the number of partitions is set to 64. The size of each partition is finer than when number of partition is set to 32. In addition, variations can be seen in the number of included tasks for each partition. It is understood that the start time of the task process is delayed compared with the case of the number of partitions 32, and the execution time does not become short as the number of partitions is increased.

Fig. 8 shows the behavior of task processing after modification when the number of partition is set to 64. As with the number of partition 32, the number of tasks in each partition...
Fig. 5. The behavior of task processing with default settings (number of partitions is set to 32).

Fig. 6. The behavior of task processing with modified settings (number of partitions is set to 32).

Fig. 7. The behavior of task processing with default settings (number of partitions is set to 64).

Fig. 8. The behavior of task processing with modified settings (number of partitions is set to 64).

became almost uniform and tasks are processed in parallel. However, only three out of the four nodes have been used.

IV. CONCLUSIONS

In the experiments, we investigate the performance of parallel data processing using Spark varying parameter settings. The locality wait time is effective in that processing can be performed in parallel among multiple nodes, but the total execution time was not improved much between default setting and not considering data locality setting. Also, modified portable_hash can make the number of tasks in each partition almost uniform and tasks processed in parallel, but the number of partitions processed in each node decreases and partitions are executed consecutively. In the future, we will consider a method to control the number of allocated tasks on each worker in order to improve the processing performance.

ACKNOWLEDGMENT

This paper is partially based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO), JSPS KAKENHI Grant Number JP16K00177, and the open collaborative research at National Institute of Informatics (NII) Japan (FY2017).

REFERENCES