A Study of a Scalable Distributed Stream Processing Infrastructure Using Ray and Apache Kafka

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 various sensors and the development of cloud computing technologies enable the accumulation and use of many live logs in ordinary homes. In addition, deep learning technologies have been widely used for image and speech recognition processing. However, a key issue for deep learning is heavy processing loads. To operate a service that utilizes sensor data, those data are transmitted from sensors in ordinary homes to a cloud and analyzed in the cloud. However, services that involve moving image analysis require large amounts of data to be transferred continuously and high computing power for the analysis; hence, it is difficult to process them in real time in the cloud using a conventional stream data processing framework. First, we perform preliminary experiments using Apache Spark [3] (hereinafter called Spark), which is a representative cluster computing platform that is designed to be fast and versatile, and Ray [4], which is a distributed execution framework. We investigate the characteristics of their distributed recognition processing and demonstrate that Ray enables scalable distributed processing. Next, We implement a prototype system of the proposed distributed stream processing infrastructure using Ray and Apache Kafka [1] (hereinafter called Kafka), which is a distributed messaging system, and demonstrate its performance.

II. Distributed processing efficiencies in Spark and Ray

In this paper, as preliminary experiments, to investigate the distributed processing efficiency of Spark and Ray, we construct clusters of Spark and Ray and aim at high efficiency of image identification processing using Chainer [5]. In the experiments, MNIST [6] is used as a data set. For each case of Spark and Ray, we measure the time that it takes for images to be evaluated by each worker after execution of the program and for the result to be returned to the master. We prepare 1000 MNIST data files, each of size 7.5 MB, for the experiments. Table 1 shows the performance of the computer used in the experiments. Nodes with the same performance are used for the master and all workers. In addition, each node is connected by a 1 Gbps network as shown in Fig. 1.

TABLE I
Performance of the computer that was used in the experiment

OS	Ubuntu 16.04LTS
CPU	Intel(R) Xeon(R) CPU W5590 @3.33 GHz 4 core×2 sockets(8 core)
GPU	NVIDIA GeForce GTX 980
Memory	48 Gbyte

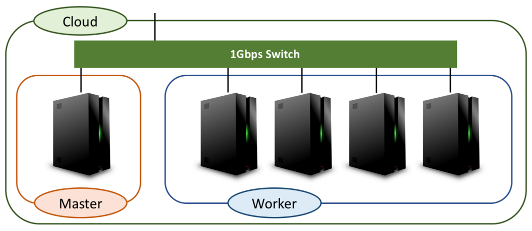

Fig. 1. Experimental environment.

A. The distributed processing efficiency in Spark

Fig. 2 shows master/worker processing in the case of Spark. The round rectangle drawn by a solid line represents the entire Spark cluster and the dotted rectangle represents the physical node. Among the physical nodes, a node that is used as a master is represented by a red dotted line and a node that is used as a worker is represented by a blue dotted line. Distributed processing in Spark is performed as follows: (1) Execute the Python program on the master. (2) Make Spark read the MNIST and create the RDD. (3) Pass the created RDD to the worker. (4) Identify MNIST using Chainer in workers. Each node is connected in Spark Standalone Mode.

The average value of 10 measurements, when the number of nodes is changed from 1 to 5 and the number of partitions is changed from 8 to 48 in 8 increments, are shown in Fig. 3. In Fig. 3, the horizontal axis is the number of nodes and the vertical axis indicates elapsed time; each color shows different experimental results when the number of partitions is set from 8 to 48. According to Fig. 3, when the number of nodes is 5, the processing time is shortened by approximately 40 seconds compared to when the number of nodes is 1. However, the results for 4 and 5 nodes were almost the same. In terms of the number of partitions, the results differed substantially between 8 and 16 and above; however, no significant difference was observed for 16 or more partitions.

To conduct a detailed behavior survey, the processing times for each task are measured and shown in Fig. 4. Measurement results are obtained when the number of nodes is 5 and the

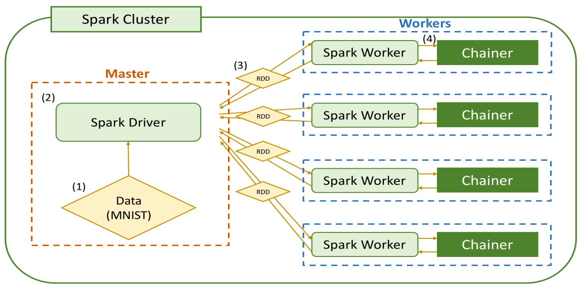

Fig. 2. Master/Worker processing using Spark and Chainer.

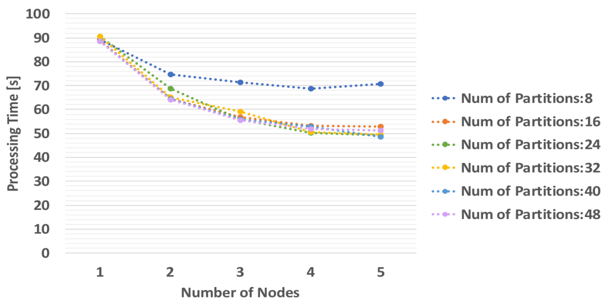

Fig. 3. Processing times of distributed processing by Spark.

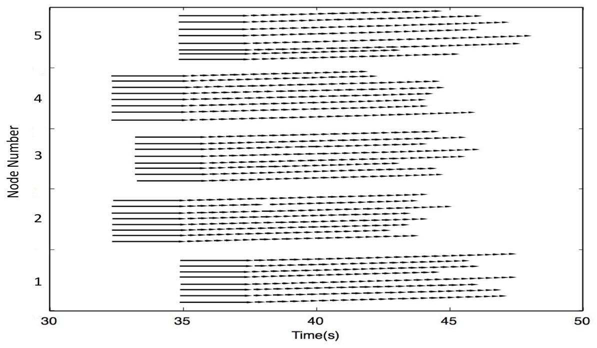

Fig. 4. Processing times of each task in the Spark environment.

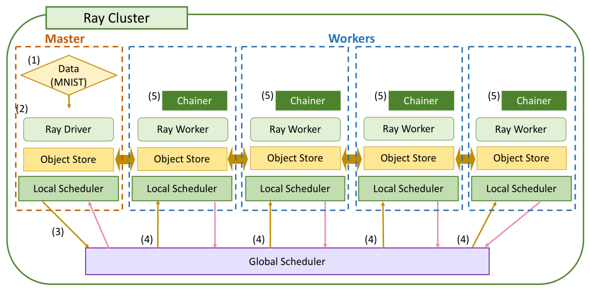

Fig. 5. Master / Worker processing using Ray and Chainer.

number of partitions is 40. In Fig. 4, the horizontal axis indicates elapsed time and the vertical axis is the node number. The start time and the end time are measured for each task and the processing times of each task are drawn one by one with arrows for each node. A group of arrows that point diagonally upward and to the right of the time axis indicate a single partition and it takes approximately 2.5 seconds to process the first task of the partition and approximately 0.3 seconds to process subsequent tasks. It takes longer to process the first task because it calls Chainer and the learned model. It is thought that task processing is faster for the second and subsequent calls because these calls are easier than the first task with Chainer library loading. In Fig. 4, processing is intimated for each node and 8 tasks are processed in parallel since the number of the machine cores is 8. However, the number of tasks in each partition does not become uniform; hence, the partitions in the positive direction of the time axis are scattered. The presence of such partitions suggests that the parallel processing efficiency degrades as the overall processing time increases.

B. The distributed processing efficiency in Ray

Fig. 5 shows master / worker processing in the case of Ray. The round rectangle drawn by a solid line represents the entire Ray cluster and the dotted rectangles represent the physical nodes. Among the physical nodes, a node that is used as a master is represented as a red dotted line and a node that is used as a worker is represented as a blue dotted line.

Distributed processing in Ray is performed as follows: (1) When a Python program is executed on the master, (2) the Ray driver in the master node places the data in the object store of its own node and contacts the local scheduler. (3) The local scheduler communicates to the global scheduler. (4) The global scheduler sends instructions to local schedulers of each node. (5) Ray workers evaluate MNIST using Chainer based on data that are copied via object storage and shared memory deployed over Object Stores in the Ray cluster nodes.

As with Spark, we measure the execution times of Ray on 1000 tasks as we change the number of nodes from 1 to 5. Since Ray does not have the concept of a partition as Spark does, we distribute data to workers 1 to 5 in a round-robin manner. We divide the processing time into three parts: launching a Ray worker, reading data, and evaluating via a remote function. The average values of 10 measurements is shown in Fig. 6. The horizontal axis is the number of nodes and the vertical axis indicates elapsed time. The graph is drawn of each of the 3 parts: launching a Ray worker, reading data, and identifying via a remote function. In Fig. 6, the processing time that is taken to read the data is nearly constant over the number of nodes; however, as the number of nodes increases, the time to launch a Ray worker is increasing. In contrast, the time that is required for the identification process via the remote function decreases as the number of nodes increases; it is faster than Spark.

We also measure the processing times for each task using Ray. The result for 5 nodes is shown in Fig. 7. The horizontal axis indicates elapsed time and the vertical axis is the node number. The set of arrows that point to the right obliquely upward are processed on the same node. In the Spark result in Fig. 4, the arrows are not continuously plotted for each

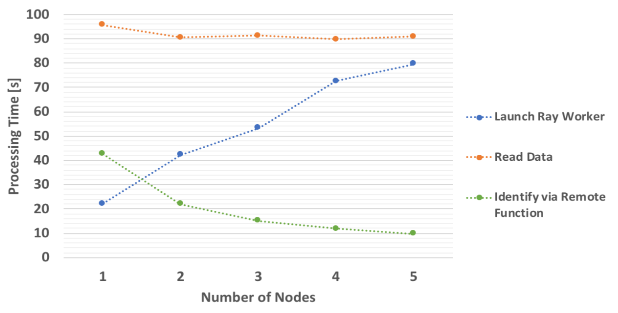

Fig. 6. Processing times of distributed processing by Ray.

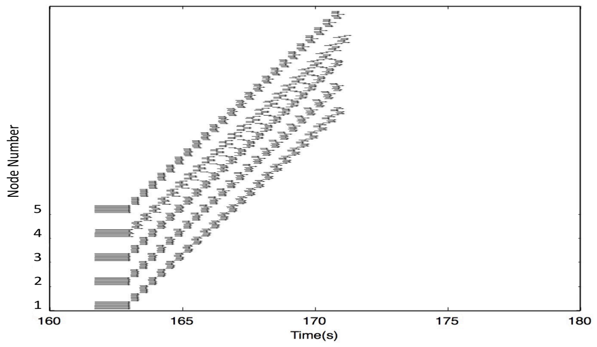

Fig. 7. Processing times of each task in the Ray environment.

core due to a partition-based scheduling; instead, tasks in Fig. 7 are being processed in parallel depending on the number of machine cores since tasks are assigned to each worker in a round-robin manner. In each node, it takes approximately 1.3 seconds to process the first task and approximately 0.35 seconds for subsequent tasks. In the case of Ray, task processing in all nodes are started at the same time; as a result, the image identification process is completed in approximately 10 seconds, which is faster than Spark.

III. Distributed Stream Processing Infrastructure Using Ray and Kafka

We perform distributed stream processing using Kafka and Ray. The experimental configuration is shown in Fig. 8. In this experiment, Kafka transmits image data provided by the ImageNet [8] to the Ray cluster and we investigate the performance of distributed recognition processing using Keras [7] and TensorFlow [2]. The round rectangles drawn by solid lines represent the Ray cluster and the Kafka cluster and the dotted rectangles represent the physical node. When Kafka is started and a Python program is executed, image data are transmitted from Producer to Broker. Upon receiving the data, the Broker sends the data to the Consumers deployed in the worker nodes in the Ray cluster and the Consumers identify the image. The number of Producers is set to 1, the number of Ray workers is varied among 1, 2, 4, 8, 12, and the batch size is varied among 1, 5, 10, 20, 40. The experimental results are shown in Fig. 9. The horizontal axis is the number of worker nodes and the vertical axis is the image processing throughput;

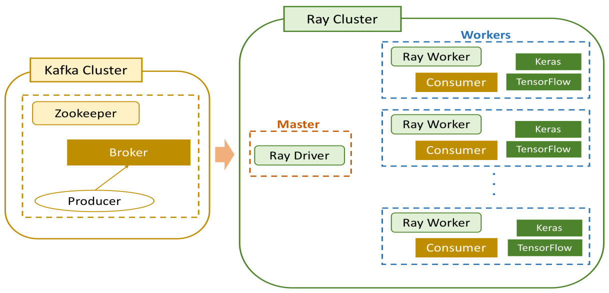

Fig. 8. Distributed stream processing composition.

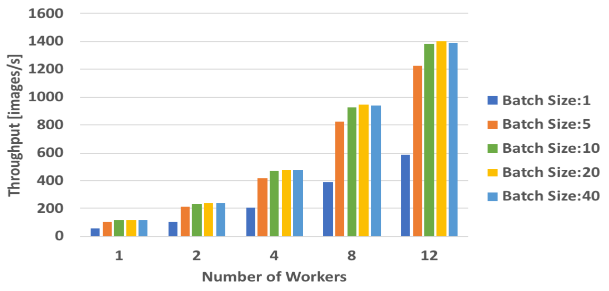

Fig. 9. Throughputs of batch-based distributed stream processing.

the graph is drawn for each batch size. Fig. 9 shows that the throughputs are improved by approximately a factor of two compared with batch sizes of 1 and 5, while the throughputs of batch size 10, 20 and 40 are comparable. In addition, the throughputs of all batch sizes are improved as the number of workers increases. Thus, the experimental results show that a scalable distributed stream processing infrastructure can be constructed using Kafka and Ray.

Acknowledgments

This paper is partially based on results that were obtained from a project that was 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 (FY2018).

References

[1] “Apache kafka,” https://kafka.apache.org/

[2] M. Abadi, et al., “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, http://download.tensorflow.org/paper/whitepaper2015.pdf. pp. 1-19. [Online]. Available: http://tensorflow.org/

[3] “Apache Spark,” https://spark.apache.org/.

[4] P. Moritz, et al., “Ray: A Distributed Framework for Emerging AI Applications,” 2017. http://ray.readthedocs.io/en/latest/index.html

[5] S. Tokui, et al., “Chainer: a next-generation open source framework for deep learning,” in In Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS), 2015, 6 pages.

[6] Y. Lecun, et al., “The MNIST Database of handwritten digits,” http://yann.lecun.com/exdb/mnist/.

[7] “Keras,” https://keras.io/

[8] J. Deng, et al., “ImageNet: A large-scale hierarchical image database.” IEEE Conference on Computer Vision and Pattern Recognition, 2009. http://www.image-net.org/