International Journal of Electronic Commerce Studies
Vol.3, No.2, pp.211-228, 2012
doi: 10.7903/ijecs.1092

PARALLEL IMAGE DATABASE PROCESSING WITH MAPREDUCE AND PERFORMANCE EVALUATION IN PSEUDO DISTRIBUTED MODE

Muneto Yamamoto
Kyushu University
Motooka 744, Nishi-Ku, Fukuoka-Shi, 819-0395, Japan
mnt.ymmt@gmail.com

Kunihiko Kaneko
Kyushu University
Motooka 744, Nishi-Ku, Fukuoka-Shi, 819-0395, Japan
kaneko@ait.kyushu-u.ac.jp

ABSTRACT

With recent improvements in camera performance and the spread of low-priced and lightweight video cameras, a large amount of video data is generated, and stored in database form. At the same time, there are limits on what can be done to improve the performance of single computers to make them able to process large-scale information, such as in video analysis. Therefore, an important research topic is how to perform parallel distributed processing of a video database by using the computational resource in a cloud environment. At present, the Apache Hadoop distribution for open-source cloud computing is available from MapReduce[1]. In the present study, we report our results on an evaluation of performance, which remains a problem for video processing in distributed environments, and on parallel experiments using MapReduce on Hadoop[2].

Keywords: Hadoop, MapReduce, Image Processing, Sequential Image Database

1. INTRODUCTION

With the spread of cloud computing and network techniques and equipment in recent years, a large amount of data collected in a variety of social situations have been accumulated, and so the need for analysis

techniques to take advantage of useful information that can be extracted from such data sets is increasing. This is also true for video data, in which images are sequential and the data includes the associated time and frame information of each frame. Today, because video cameras are set up to perform surveillance of moving objects such as pedestrians and vehicles, a large amount of video data is generated, and stored in database form. To improve these video database systems, which hold a large amount of video and related data, parallel processing using CPUs and disks at multiple sites is an important area of research.

Also, when considering operations such as search and other types of analysis of video images recorded by video camera and stored in a database, there are limits on what can be done to improve the performance of single computers to make them able to process large-scale information. Therefore, the advantages of parallel distributed processing of a video database by using the computational resources of a cloud computing environment should be considered. In addition, if computational resources can be secured easily and relatively inexpensively, then cloud computing is suitable for handling large video databases at low cost. Hadoop, as a *mechanism* for processing large numbers of databases by parallel and distributed computing has been recognized as promising. Nowadays, for reasons such as ease of programming, by using the function MapReduce on the Hadoop system, *open-source cloud-based systems* that can process data across multiple machines in a distributed environment have been studied for their application *to* various *database operations*. In fact, *Hadoop is in use all over the world*[3]. *Studies using Hadoop have been performed to treat one file as a text data file or multiple files as a single file unit, such as for the analysis of large volumes of DNA sequence data, converting the data of a large number of still images to PDF format, and carrying out feature selection/extraction in astronomy*[4]. *These examples demonstrate* the usefulness of this system, which is due to its having the ability to run multiple processes in parallel for load balancing and task management.

The rest of this paper is organized as follows. Section 2 provides the background knowledge related to this work, including an overview of the MapReduce programming model and why MapReduce is deployed in a distributed environment. In section 3, we discuss the architecture of our system and problems arising when processing video databases by using MapReduce on Hadoop in a distributed environment. Then, we give an overview on our experimental methodology and present the results of our experiments in section 4. Finally, we conclude the paper and propose future work in section 5.

2. HADOOP STREAMING FOR DATABSE PROCESSING

2.1 Parallel and Distributed Processing on Hadoop

As the structure of the system, Hadoop consists of two components, the Hadoop Distributed File System (HDFS) and MapReduce, performing distributed processing by single-master and multiple-slave servers. There are two elements of MapReduce, namely JobTracker and TaskTracker, and two elements of HDFS, namely DataNode and NameNode. In Figure 1, the configuration of these elements of MapReduce and HDFS on Hadoop are indicated. There is also a mechanism that checks the metadata for NameNode.

(A) JobTracker

JobTracker manages cluster resources and job scheduling to and monitoring on separate components.

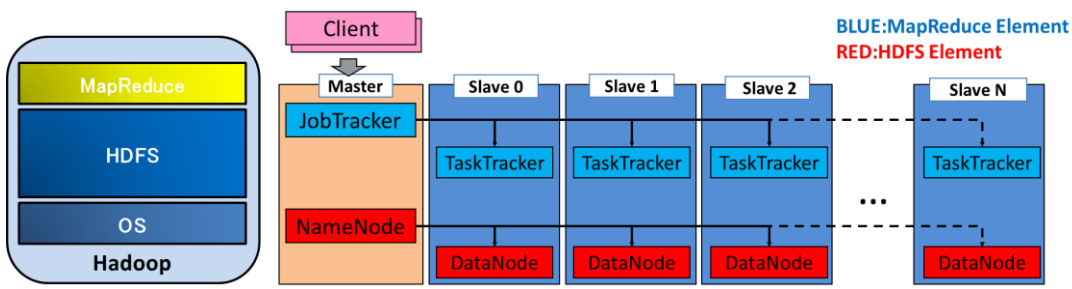

Figure 1. Structure with elements of MapReduce and HDFS

(B) TaskTracker

TaskTracker is a slave node daemon in the cluster that accepts tasks and returns the results after executing tasks received by JobTracker.

(C) NameNode

An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. NameNode executes file system name space operations, such as opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes.

(D) DataNode

The cluster also has a number of DataNodes, usually one per node in the cluster. DataNodes manage the storage that is attached to the nodes on which they run. DataNodes also perform block creation, deletion, and replication in response to direction from NameNode.

(E) SecondaryNameNode

SecondaryNameNode is a helper to the primary NameNode. Secondary is responsible for supporting periodic checkpoints of the HDFS metadata.

2.2 Hadoop Distributed File System (HDFS)

HDFS is designed to reliably store very large files across machines in a large cluster. It is inspired by the Google File System. HDFS is composed of NameNode and DataNode. HDFS stores each file as a sequence of blocks (currently 64 MB by default) with all blocks in a file the same size except for the last block. Blocks belonging to a file are replicated for fault tolerance. The block size and replication factor are configurable per file. Files in HDFS are write-once and can have only one writer at any given time.

2.3 MapReduce

MapReduce (implemented on Hadoop) is a framework for parallel distributed processing large volumes of data. In programming using MapReduce, it is possible to perform parallel distributed processing by writing programs involving the following three steps: Map, Shuffle, and Reduce. Figure 2 shows an example of the flow when Map and Reduce processes are performed. Because MapReduce automatically performs inter-process communication between Map and Reduce processes, and maintain load balancing of the processes.

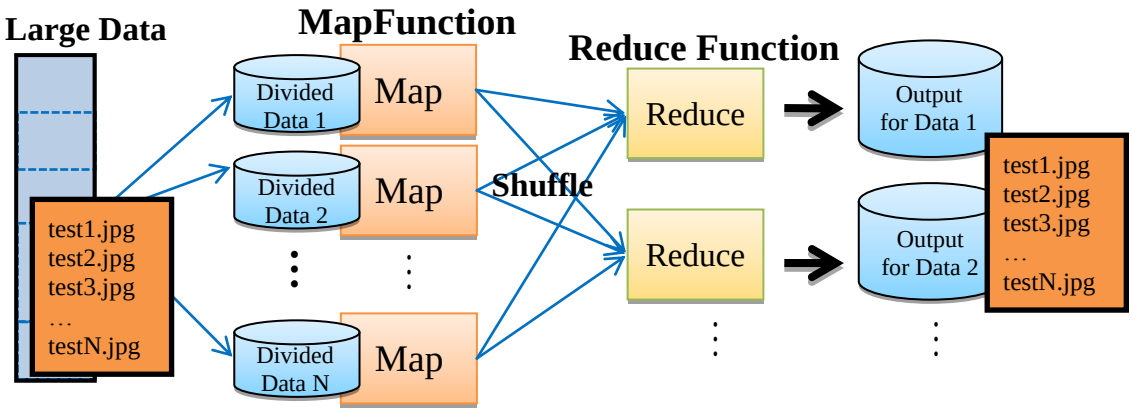

Figure 2. Processes performing the map and reduce phases

1. Map concept of data processing

The Map function takes a key-value pair <K, V> as the input and generates one or multiple pairs <K′, V′ > as the intermediate output.

2. Shuffle concept of data processing

After the Map phase produces the intermediate key-value pair or key-value pairs, they are efficiently and automatically grouped by key by the *Hadoop* system in preparation for the Reduce phase.

3. Reduce concept of data processing

The Reduce function takes as the input a <K′, LIST V′ > pair, where "LIST V′ " is a list of all V′ values that are associated with a given key K′. The Reduce function produces an additional key-value pair as the output.

By combining multiple Map and Reduce processes, we can accomplish *complex* tasks which *cannot* be done via a *single Map and Reduce* execution. Figures 3 and 4 respectively show the key-value data model and a Wordcount example of MapReduce.

map: <key, value> ⇒ list <key', value'>

shuffle: list <key', value'> ⇒ {<key'', list(value'')>}

reduce: {<key'', list(value'')>} ⇒ list(value''')

Figure 3. Key-value data model of MapReduce

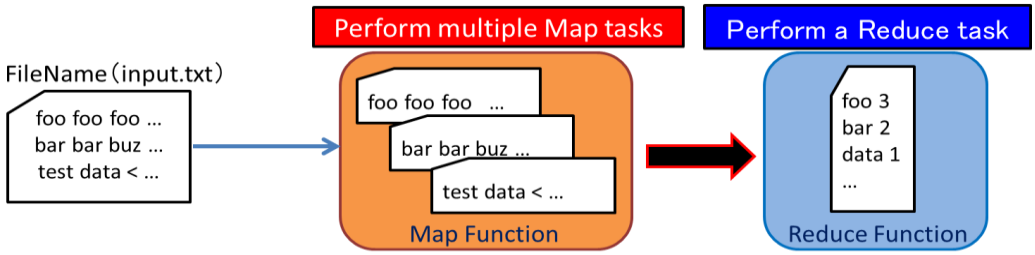

Figure 4. Wordcount example of MapReduce

2.4 Hadoop Streaming

Hadoop is an open-source implementation of the MapReduce platform and distributed file system. The Hadoop system is *written in Java.* Hadoop Streaming is a utility that comes with the Hadoop distribution and that can be used to invoke streaming programs that are not written in Java (such as Ruby, Perl, Python, PHP, R, or C++). Using this utility, we can execute database programs written in the Ruby programming language. The utility also allows the user to create and run *Map and Reduce* jobs with any executable programs or scripts as the mapper and the reducer. Figure 5 shows an example of an execution of a program when using Hadoop Streaming and a description of the Map and Reduce functions in the Ruby programming language. Key-value pairs can be specified to depend on the input–output formats.

./bin/hadoop jar contrib/hadoop-0.20.2-streaming.jar

-input myInputDirs [*HDFS Path*]

-output myOutputDir [*HDFS Path*]

-mapper map.rb [*map program file path*]

-reduce reduce.rb [*reduce program file path*]

-file filename [*local file system path*]

```
//map.rb
def map(key, value, output, reporter)
   # Mapper code
end
//reduce.rb
def reducer(key, value, output, reporter)
   # Reducer code
end
```

-cacheFile fileNameURI [*URI to the file that you have already uploaded to HDFS*]

-inputformat JavaClassName [*Input format should return key/value pairs of Text class*]

-outputformat JavaClassName [*output format should return key/value pairs of Text class*]

Figure 5. Usage example of the options for Hadoop Streaming using the Ruby programming language on Hadoop

3. VIDEO DATABASE PROCESSING ON HADOOP

3.1 MapReduce for Video Database Processing

The Map and Reduce functions of *MapReduce* are both defined with respect to data structured in key-value pairs. In short, we can perform distributed processing by creating key-value pairs in *MapReduce form. However, for* unstructured data such as video data, it can be assumed that it is more difficult to create key-value pairs and perform the processing than processing structured data.

In this experiment, in order to use the Ruby programming language, we *utilize* an extension package of Hadoop, namely Hadoop Streaming. With a view to performing parallel distributed processing on MapReduce forms, we need to create programs to be used as Map and Reduce functions in the Ruby programming language. Video database processing is performed by splitting the data in a video database and creating key-value pairs. For example, the frame number can be used as a key for a video frame. In the case of parallel processing of a video frame, the video frame is divided into multiple parts, and the part numbers can be the keys (identifiers) for these different parts. Sorting is carried out using the key number, and joining separated frames or separated parts is performed by the Reduce function. Figure 6 shows an example of processing flow using MapReduce. In this figure, each video frame is divided into four parts, and each part has a unique key number.

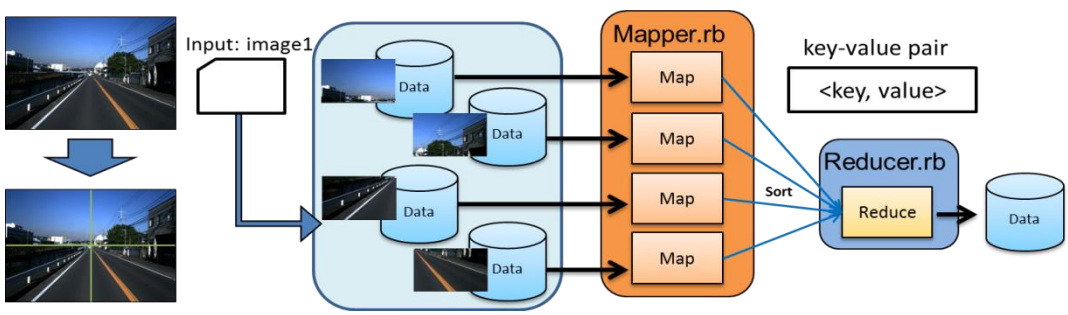

Figure 6. Image processing flow using MapReduce

3.2 MapReduce Processing of Video Database

We implemented the following video database processing steps using Hadoop Streaming. Multiple sequential video frames are input, and image processing is performed for each video frame. We implemented a Map function for image processing. The input of the Map function is a single video frame, and the Map function produces one video frame as output.

We process video frames in parallel with the slave servers, which use HDFS. The video database is stored in HDFS. Each Map process also outputs to HDFS. Figure 7 shows an example of video database processing flow using the Map function and HDFS.

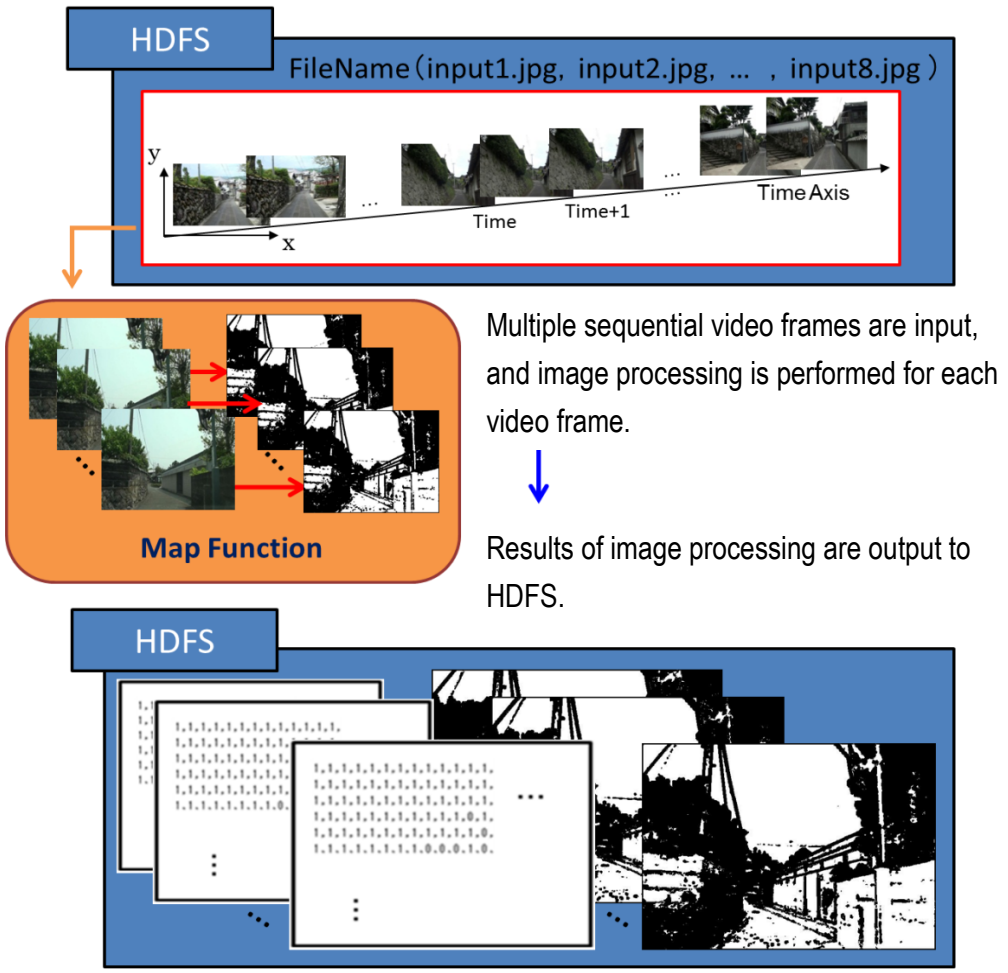

Figure 7. Sequential video frame processing flow using Map and HDFS

4. EXPERIMENT

In the experiment, we use a video database that contains a set of video data collected in the area of Nishi-ku in Fukuoka City and Usuki in Oita Prefecture.

The experimental environment is as follows:

- OS: Ubuntu 11.04, CPU: Intel core i5 2.8 GHz, Memory: 4 GB.

The software versions are as follows:

- Hadoop-0.20.2, Ruby 1.8.7, and Octave 3.4.2.

The image size is as follows:

- 640×480 pixels.

The experiment is described as follows. We first create a grayscale image of the original image in parallel by using MapReduce. Then, features of the grayscale image are extracted in parallel by using MapReduce. Next, we recall a sub-program written in the Octave language (a high-level interpreted language used for mathematical model fitting, signal processing, and image *processing*) from a Ruby program.

We configured the MapReduce system as a pseudo-distributed mode. Hadoop is composed of a master server which manages slave servers, which perform the actual image processing. Master and slave servers actually run on the same server for this configuration of the MapReduce system (pseudo-distributed mode). The number of copies of video data is set to 1. The parallel processing of the video database using Hadoop Streaming is distributed over all of the cores of a CPU of a single machine.

4.1 Grayscale Images Created by MapReduce

In this experiment, we create a grayscale image of the original image in parallel by using MapReduce. First of all, we perform the processing of dividing the image in the Map tasks. Next, the Reduce task accepts the processed images from the Map tasks, combines the divided image, and outputs the result image to HDFS.

We used grayscale creation as an example of image processing using MapReduce. In making the grayscale image from the original, we convert the RGB values to the NTSC color space as an yiqmap that contains the equivalent NTSC luminance (*Y*) and chrominance (*I* and *Q*) color components as columns, where Y, I, and Q are defined as follows:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (1)$$

Here, we specify the number of Map tasks as 4. Figure 8 shows the results of JobTracker when processing the video database, and shows that the number of Map tasks is 4 and the number of Reduce task is 1. Figure 9 shows the results given by a Hadoop Web interface program when

processing image files with MapReduce. Here, we specify the output file on HDFS when we perform the re-combination of the 4-divided image from *usuki-20110430.avi-000001_divide1.jpg* to *usuki-20110430.avi-000001_divide4.jpg*. Figure 10 shows the Task ID, the processing time from start to end, and the status of each TaskTracker on Hadoop in the experiment. Due to perform considerations, the maximum number of slave processes allowed to be executed is set to be 2, and then the tasks are performed in pairs, for example, *task_201201302155_0004_m_000000* and *task_201201302155_0004_m_000001*, followed by *task_201201302155_0004_m_000002* and *task_201201302155_0004_m_000003*. As a result, we confirmed a processing time of 4 sec and that two tasks can be performed at the same time in parallel for this experimental setup.

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	4	0	0	4	0	0 / 0
reduce	100.00%	1	0	0	1	0	0 / 0

Figure 8. Results of JobTracker when processing an image

Name	Type	Size
usuki-20110430.avi-0000001_divide1-2-3-4.jpg	file	42.04 KB
usuki-20110430.avi_0000001-hpoint.jpg	file	53.31 KB
usuki-20110430.avi_0000001-hpoint.txt	file	600 KB
usuki-20110430.avi_0000002-hpoint.jpg	file	53.64 KB
usuki-20110430.avi_0000002-hpoint.txt	file	600 KB
usuki-20110430.avi_0000003-hpoint.jpg	file	53.64 KB
usuki-20110430.avi_0000003-hpoint.txt	file	600 KB
usuki-20110430.avi_0000004-hpoint.jpg	file	52.17 KB
usuki-20110430.avi_0000004-hpoint.txt	file	600 KB
usuki-20110430.avi_0000005-hpoint.jpg	file	52.36 KB
usuki-20110430.avi_0000005-hpoint.txt	file	600 KB
usuki-20110430.avi_0000006-hpoint.jpg	file	53.2 KB
usuki-20110430.avi_0000006-hpoint.txt	file	600 KB
usuki-20110430.avi_0000007-hpoint.jpg	file	51.88 KB
usuki-20110430.avi_0000007-hpoint.txt	file	600 KB
usuki-20110430.avi_0000008-hpoint.jpg	file	53.13 KB
usuki-20110430.avi_0000008-hpoint.txt	file	600 KB

Figure 9. Results shown by a Hadoop Web interface program during grayscale video image creation using MapReduce

Task	Complete	Status	Start Time	Finish Time	Errors	Counters
task_201201302155_0004_m_000000	100.00%	Records R/W=1/1	30-1-2012 22:05:05	30-1-2012 22:05:07 (2sec)		10
task_201201302155_0004_m_000001	100.00%	Records R/W=1/1	30-1-2012 22:05:05	30-1-2012 22:05:07 (2sec)		10
task_201201302155_0004_m_000002	100.00%	Records R/W=1/1	30-1-2012 22:05:07	30-1-2012 22:05:09 (2sec)		10
task_201201302155_0004_m_000003	100.00%	Records R/W=1/1	30-1-2012 22:05:07	30-1-2012 22:05:09 (2sec)		10

Figure 10. Status of each TaskTracker on Hadoop when processing image files with MapReduce

4.2 Feature Extraction of Video Images by Using MapReduce

In this experiment, we create video images that depict extracted features in parallel by using MapReduce. First of all, we input multiple video frames and perform feature extraction processes using Map function in parallel. In the parallel processing, multiple slave servers use a video database stored in HDFS and output the processing results to HDFS.

For the process of extracting features of sequential video frames after multiple video frames are input to slave servers that execute a Map function, only the Map function is used to output a significant number of features for each image. First, when a pixel of a video frame contains a significant amount of features, such as corners and edges, a new pixel is generated in the output image, and the pixel value is set to be a feature value. Otherwise, a black pixel is generated in the output image. Let F and L be the original video frame and a smoothed image of F by a Gaussian distribution, respectively. We denote the individual pixels of L as $l(i, j)$. Then, in order to perform feature extraction, we use the Prewitt filter, which detects edges and lines in an image. In this process, the filter obtains the horizontal and vertical contours. The filter coefficient is assumed to be a 3×3 matrix, from which an auto-correlation *matrix is created.* The *image L is differentiated once in each of the horizontal and vertical directions, where the notation* l_i and l_j is used for the respective derivatives. In addition, A, B, and C are defined as follows:

$$l_i l_j = \left(\frac{\partial^2 l_k}{\partial i \partial j} \right), \; A = (l_i)^2 = \left(\frac{\partial l_k}{\partial i} \right)^2, B = (l_j)^2 = \left(\frac{\partial l_k}{\partial j} \right)^2, C = l_i l_j = \left(\frac{\partial^2 l_k}{\partial i \partial j} \right) \quad (2)$$

The auto-correlation matrix, denoted by M, is given in terms of A, B, and C as follows:

$$M = \begin{bmatrix} A & C \\ C & B \end{bmatrix} \quad (3)$$

Then, the feature points of the Harris corner detection are extracted using the value $c(i, j)$ of each pixel[5, 6]:

$$c(i, j) = \det(M) - k(\mathrm{tr}(M))^2 \quad (4)$$

Det and tr represent the determinant function and the sum of the diagonal elements, respectively. Here, k is an adjustable parameter and is generally taken to be within the range 0.04 to 0.06. In our experiment, k is taken to be 0.04. In addition, the eigenvalues λ_1 and λ_2 are obtained as follows:

$$\det(M) = \lambda_1\lambda_2 = AB - C^2, \quad \mathrm{tr}(M) = \lambda_1 + \lambda_2 = A + B \quad (5)$$

Threshold T is used to decide which $c(i, j)$ are used in the output image. A pixel value in the output image is set to be 0 if $c(i, j) < T$. The absolute value of c(i, j) as calculated from the two eigenvalues λ_1 and λ_2 is denoted by $R(i, j)$. In addition, the maximum and minimum values max($R(i, j)$) and min($R(i, j)$) are normalized to 0 and 255, respectively, in the output image. The pixel values in the output image have a 256-level range from 0 to 255. Using this method, we create a mask image with a threshold using a feature extraction image. Figure 11 shows examples of the original image (left), feature extraction image (center), and mask image (right). The feature extraction image is the result of processing the original image using the Harris corner detection, and the mask image is the result of processing the feature extraction image taking advantage of the threshold T.

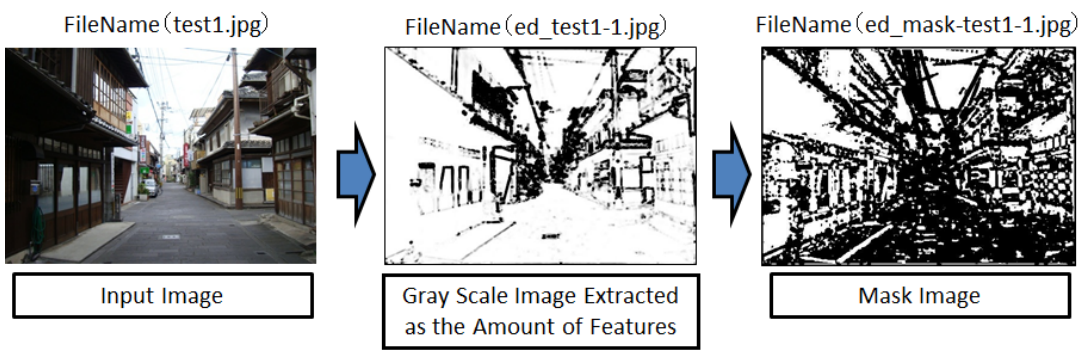

Figure 11. Examples of an input image, feature extraction image, and output image. The output image is used as the mask image

For the Map processing, we input a sequence of eight images and perform the process. As a result, we output eight corresponding image files. The output image files are used for masking the input image. Here, we specify the output file on HDFS when we perform image processing of eight images, for example, as the eight images *usuki-20110430.avi-000001jpg* through *usuki-20110430.avi-000008.jpg*. Figure 12 shows the MapReduce status in halfway through the execution; at this point, six of the eight tasks have been performed and 25% of the processing has been completed. Figure 13 shows Task ID, processing time from start to end, and status of each TaskTracker on Hadoop. Due to perform considerations, the maximum number of slave processes executed is set to be 2, two tasks are performed, such as *task_201111161359_0007_m_000000* and *task_201111161359_0007_m_000001*, and then the eight tasks are performed in pairs in parallel, for example, *task_201111161359_0007_m_000000* and *task_201111161359_0007_m_000001*, followed by *201111161359_0007_m_000002* and *task_201111161359_0007_m_000003*. As a result, we achieved a processing time of 2 min 27 sec. Figure 14 shows a MapReduce status report used to confirm the successful termination of MapReduce tasks. Here, we specify that the number of Map tasks is 8 and the number of Reduce tasks is 0.

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	25.00%	8	6	1	1	0	0 / 0
reduce	0.00%	0	0	0	0	0	0 / 0

Figure 12. MapReduce status in the middle of execution

Task	Complete	Status	Start Time	Finish Time	Errors	Counters
task_201111161359_0007_m_000000	100.00%	Records R/W=1/14	29-11-2011 05:48:16	29-11-2011 05:48:53 (37sec)		8
task_201111161359_0007_m_000001	100.00%	Records R/W=1/14	29-11-2011 05:48:16	29-11-2011 05:48:53 (36sec)		8
task_201111161359_0007_m_000002	100.00%	Records R/W=1/14	29-11-2011 05:48:52	29-11-2011 05:49:29 (37sec)		8
task_201111161359_0007_m_000003	100.00%	Records R/W=1/14	29-11-2011 05:48:52	29-11-2011 05:49:30 (37sec)		8
task_201111161359_0007_m_000004	100.00%	Records R/W=1/14	29-11-2011 05:49:28	29-11-2011 05:50:05 (37sec)		8
task_201111161359_0007_m_000005	100.00%	Records R/W=1/14	29-11-2011 05:49:29	29-11-2011 05:50:07 (37sec)		8
task_201111161359_0007_m_000006	100.00%	Records R/W=1/14	29-11-2011 05:50:04	29-11-2011 05:50:41 (37sec)		8
task_201111161359_0007_m_000007	100.00%	Records R/W=1/14	29-11-2011 05:50:05	29-11-2011 05:50:43 (37sec)		8

Figure 13. Status of each TaskTracker on Hadoop

Finished in: 2mins, 27sec
Job Cleanup: Successful

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	8	0	0	8	0	0 / 0
reduce	100.00%	0	0	0	0	0	0 / 0

Figure 14. MapReduce status after termination of execution

The total number of Map tasks is set to be 8, and the maximum number of slave processes executed is set to be either 2, 4, 6, or 8 in our experimental test. The creation time of the mask image was 2 minutes and 27 seconds when we performed the slave process in parallel using 2 processor-cores. Without MapReduce, processing time of the same task averages 2 minutes and 6 seconds, implying a favorable difference of about 21 seconds. We assume that this difference is due to the processing time for the input–output of video database on HDFS, the access time of external programs written in the Octave and Ruby programming languages, and the latency time of performing the Map processing. Figures 15 and 16 show respectively the total processing time and the processing time of each TaskTracker when a sequence of eight images is processed by using MapReduce in *pseudo-distributed* mode. Figure 17 shows an example of a part of the program code for feature extraction of video images using MapReduce.

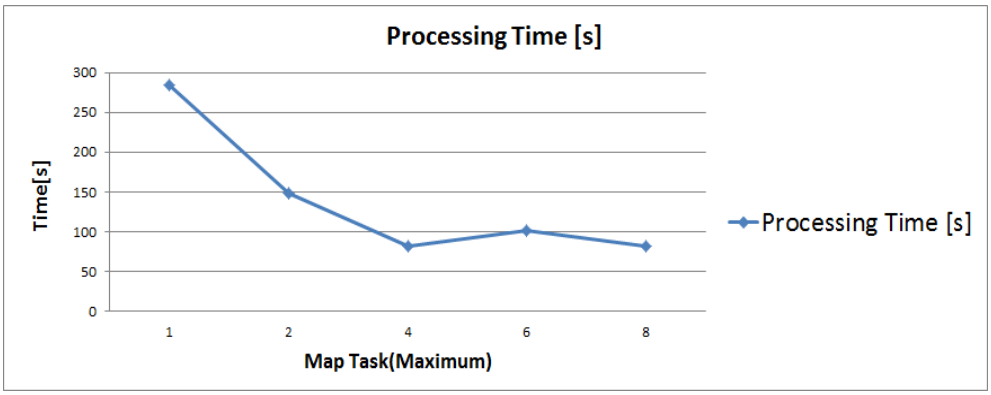

Figure15. Versus the maximum number of slave processes

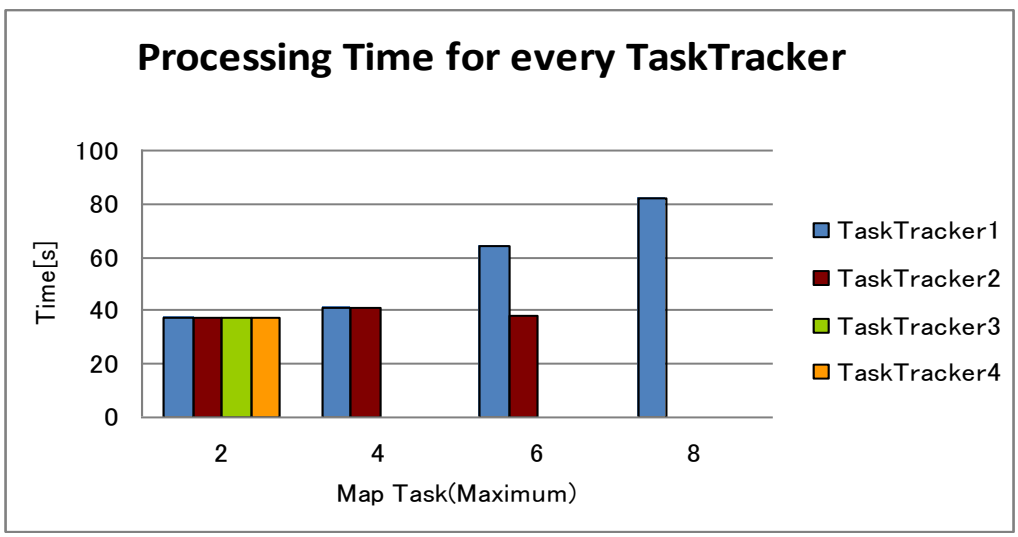

Figure16. Processing time for each TaskTracker for different numbers of Map tasks performed simultaneously

5. SUMMARY AND FUTURE WORK

In this paper, we describe using a video database of video collected with a video camera. In an experiment, we processed sequences of video frames with MapReduce to create grayscale images and extracted some features of the video images. In the process of creating the grayscale images, each video frame was divided into multiple parts. In the extraction, frame numbers were used as the key numbers to extract some features of the video images. In future, it is necessary for us to build a distributed environment that combines multiple machines, conduct large-scale experiments involving sequential video images, evaluate the

performance speed of the implementation with MapReduce, and verify the efficient key-value pairs.

```
#Map
…
  key1 = "#{k}"
  value1 = "#{v}"
  f_name = File.basename(value1,".jpg")
  ef_name = f_name + "_divide" + key1 + ".jpg"
…
  key2 = "#{k}"
  value2 = ef_name + "#{v}"
  puts "#{key2}¥t#{value2}"
end

#Reduce
image = Hash.new {|h, k| h[k] = 0 }
…
  imgList["1-2"] = ef_name1-2
  imgList["3-4"] = ef_name3-4
  A = image["1"]; B = image["2"]; C = image["3"]; D = image["4"]
…
def image_append(image1, image2, image3, image4)
str = "octave -q Reduce_img-devide.m "
                          + image1 + " " + image2 + " " + image3 + " " + image4
p system(str)
end
…
image_append(A, B, C, D)
…
```

Figure17. Example program for feature extraction of video images using MapReduce

6. ACKNOWLEDGMENT

This work was supported by Grant-in-Aid for Scientific Research (C) 50274494.

7. REFERENCES

[1] D. Jeffrey, and G. Sanjay, MapReduce: Simplified data processing on large clusters. *Paper presented at OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, USA, December 6-8, 2004.

[2] D. Sudipto, S. Yannis, S.B. Kevin, G. Rainer, J.H. Peter, and M. John, Ricardo: Integrating R and hadoop. *Paper presented at the 2010 international conference on Management of data*, Indianapolis, USA, June 6-11, 2010.

[3] S. Chris, L. Liu, A. Sean, and L. Jason, HIPI: A hadoop image processing interface for image-based map reduce tasks, B.S. Thesis. *University of Virginia, Department of Computer Science,* 2011.

[4] W. Keith, C. Andrew, K. Simon, G. Jeff, B. Magdalena, H. Bill, K. YongChul, and B. Yingyi, Astronomical image processing with hadoop. *Paper presented at Astronomical Data Analysis Software and Systems XX*, Boston, USA, November 7-11, 2010.

[5] H. Chris, and S. Mike, A combined corner and edge detector. *Paper presented at the 4th Alvey Vision Conference, Manchester, En*gland, August 31 - September 2, 1988.

[6] K. Yasushi, and K. Kenichi, Detection of feature points for computer vision. *The Journal of the Institute of Electronics, Information, and Communication Engineers,* 87(12), p1043-1048, 2004.