PAPER Special Section on Parallel and Distributed Computing and Networking

Distributed Video Decoding on Hadoop*

Illo YOON†, Saehanseul YI†, Chanyoung OH†, Hyeonjin JUNG†, *Nonmembers*, *and* Youngmin YI†a), *Member*

SUMMARY Video analytics is usually time-consuming as it not only requires video decoding as a first step but also usually applies complex computer vision and machine learning algorithms to the decoded frame. To achieve high efficiency in video analytics with ever increasing frame size, many researches have been conducted for distributed video processing using Hadoop. However, most approaches focused on processing multiple video files on multiple nodes. Such approaches require a number of video files to achieve any speedup, and could easily result in load imbalance when the size of video files is reasonably long since a video file itself is processed sequentially. In contrast, we propose a distributed video decoding method with an extended FFmpeg and VideoRecordReader, by which a single large video file can be processed in parallel across multiple nodes in Hadoop. The experimental results show that a case study of face detection and SURF system achieve 40.6 times and 29.1 times of speedups respectively on a four-node cluster with 12 mappers in each node, showing good scalability.
key words: *distributed video processing, Hadoop, extended FFmpeg*

1. Introduction

As we entered Big Data era, efficient processing of those data on a distributed computing environment such as Hadoop has become a must. Traditionally, most of the data were text, but recently video data is rapidly increasing with the wide spread of surveillance cameras and also with the advent of many smart devices capable of video recording. In addition, the frame resolution is constantly increasing: 1080p is already common also in commodity CCTVs, and many devices now support 4K.

On the other hand, as more and more powerful machine learning algorithms are developed, applying such algorithms to video processing has been widely attempted and video analytics applications can be easily found in many domains nowadays. For example, face detection and recognition is widely used not only in security but also in commercials and marketing purposes. Most machine learning algorithm is compute-intensive and takes much time even for a single frame. And, it requires to decode the video frames first to process them further in the execution pipeline.

Manuscript received January 6, 2018.
Manuscript revised May 22, 2018.
Manuscript publicized September 18, 2018.
†The authors are with the School of Electrical and Computer Engineering, University of Seoul, Korea
*This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. R0190-16-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development), and by the 2016 sabbatical year research grant of the University of Seoul.
a) E-mail: ymyi@uos.ac.kr (Corresponding author)
DOI: 10.1587/transinf.2018PAP0014

Thus, efficiently processing video data has become of keen interest. GPUs have gained much popularity with its huge computational power and the characteristics that suits well to video processing. However, a single GPU or even multiple GPUs in a single machine cannot excel the performance achievable through the distributed computing. Since Hadoop emerged, many attempts were made to use Hadoop for video processing, especially for transcoding. However, many works simply applied video decoding in Hadoop by using Java-based FFmpeg, and focused on input file-level parallelism, or job-level parallelism, where a video file itself is processed conventionally in each node: a job itself is not processed in parallel in those approaches. Such an approach which we will denote Multi-file approach is simple and can scale well only when there are many small video files of similar length. However, it could easily suffer from the load imbalance as the granularity for distributed processing is too coarse.

In contrast, the proposed approach in this paper divides a single input video file into multiple InputSplits. Thus, as many workers as the number of InputSplits can run concurrently, fully utilizing the resources in a cluster. In Multi-file approach such as [1], however, each input video file is mapped to a single InputSplit, and thus employs only as many workers as the number of video files. And an input video file itself is processed sequentially by a MapTask, and is not processed in a distributed manner.

In this paper, we propose an efficient distributed video decoding method by which a single large video file can be processed in parallel across multiple nodes in Hadoop, hence achieves very good scalability without any load imbalance. To enable this, the following contributions have been made:

1) FFmpeg, a widely used video decoder, was modified and extended carefully so that it can construct a decoding context and can decode video data in the buffer of key-value pair.
2) VideoRecordReader that can distinguish the variable sized Group-Of-Picture (GOP) boundary has been implemented.

The rest of the paper is organized as follows: the related work is reviewed in Sect. 2, and the backgrounds for Hadoop and FFmpeg is given in Sect. 3. The proposed method is explained in detail in Sect. 4 and the application used in the experiments are introduced in Sect. 5. Experimental results are discussed in Sect. 6, followed by a con-

Copyright © 2018 The Institute of Electronics, Information and Communication Engineers

clusion Sect. 7.

2. Related Work

2.1 Video Processing in Hadoop

Recently, there have been some efforts to process videos using Hadoop frameworks. Kim et al. proposed a distributed video transcoding system [2], which converts various input video formats into a MPEG-4 format using FFmpeg on Hadoop. Since FFmpeg is C/C++ based library while Hadoop is Java based framework, they used Xuggler [3], an open-source Java wrapper of FFmpeg library. They assume that a job transcodes multiple video files, in which scenario, a mapper only needs to read the given file and can decode it straightforwardly using Xuggler.

Tan et al. also proposed a distributed video processing framework on Hadoop [4]. They used JavaCV [5] which provides a Java wrapper of FFmpeg and OpenCV. Since JavaCV cannot read the video files in HDFS [6] directly, Fuse-DFS which can mount HDFS to local file system was used. In this approach of using JavaCV, it is necessary to transform the data type from JavaCV frames to Hadoop key-value pairs. They also assume that multiple video files are distributed and each file is decoded conventionally. Only as many map tasks as the number of files run for decoding, and the decoded frames are stored in HDFS. Then, reduce tasks read these frame data to process it for analytics. In this way, time-consuming analytics can be done in more distributed manner but large overhead of storing and retrieving each frame data to and from HDFS is unavoidable. In our approach, as many map tasks as the number of InputSplits run concurrently, thus the parallelism is already high and video analytics can be directly done in the map tasks immediately after the decoding, without storing and reading frames to/from HDFS.

Zhao et al. proposed a Hadoop Video Processing Intefrace (HVPI) [1], which also adopted Xuggler and employs Multi-file approach. Since each InputSplit is a separate video file, a mapper would process the given video file by decoding it sequentially using Xuggler. They compared different JNI [7] implementations to bridge Hadoop and OpenCV, and provides an efficient JNI implementation in HVPI.

Although the work proposed by Ryu et al. [8] is not clearly described, it is somewhat similar to our approach in that it assumes that a job decodes a single large video file and uses GOP-level parallelism within a video file, and also in that FFmpeg was extended. However, InputSplits have to end at GOP boundary so that TaskTrackers can read the GOP directly when an InputSplit is received. To do so, JobTracker has to partition the input video file into InputSplits at GOP boundaries, which is done sequentially before the MapReduce starts. As a larger video file is used, the splitting overhead by a JobTracker in [8] could increase. In contrast, our approach can find out the GOP boundaries in parallel by each YarnChild in a node, and hence is more scalable.

The approach in [9] is similar to ours in that InputSplits do not need to align at the GOP boundary. However, it extracts the sequence header information by executing a separate job, of which the launching overhead would not be negligible. Only the first InputSplit would be executed to extract the sequence header information and the other InputSplits should be skipped. For example, a 10GB video file would be divided into 157 InputSplits of 64MB size, out of which the 156 TaskTrackers or Mapper processes have to be launched in order to be simply skipped to the end of each InputSplit. This overhead in the first job processing could become a performance bottleneck unless the cluster has very large number of nodes.

Pereira et al. [10] proposed Split & Merge framework, a distributed video encoding system in the cloud. Although they presented video encoding framework, the main idea of splitting a large video into multiple chunks so that they can be processed in a distributed manner in the cluster is similar to ours. They also address identifying key-frames in the input is important if the input to be encoded already have some form of temporal compression, which is always the case in video decoding. However, they implemented their framework for the inputs without temporal compression, and only fixed interval splitting is presented, which cannot support GOP based distributed decoding. In contrast, our proposed approach aims at supporting GOP based chunking in Hadoop, discussing the design and implementation of distributed video processing in a detailed manner.

3. Backgrounds

3.1 Hadoop 2

Hadoop is a widely used distributed framework that supports MapReduce programming model, and the Hadoop 2 provides not only MapReduce but also many other frameworks such as SQL, Graph, etc [11]. The daemons and processes in Hadoop 2 MapReduce framework are shown in Fig. 1.

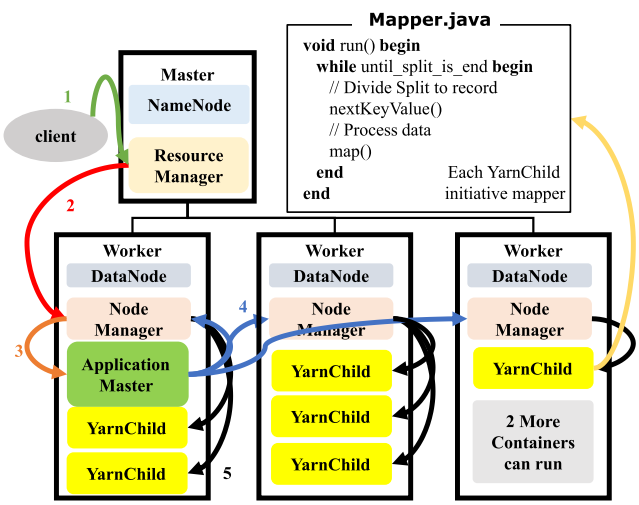

Fig. 1 Hadoop 2 daemons and processes

Fig. 2 Structure of AVC video format.

The two daemons, *ResourceManager* (RM) and *NodeManager* (NM), manage the resource in the cluster and in the node, respectively. While a RM resides in the master node, an NM resides in the each worker node. *ApplicationMaster* (AM) and *YarnChild* are processes that are executed per job. RM and AM altogether play the role of *JobTracker* in Hadoop 1, and a separate AM is invoked for different jobs. And *YarnChild* corresponds to a Child JVM in Hadoop 1, and is a process that *MapTask* or *ReduceTask* is running for the actual computations.

RM produces input chunks called *InputSplits*, each of which will be consumed by a MapTask. The number of InputSplit is identical to the number of MapTasks. A MapTask, or a *YarnChild*, iterates executing *nextKeyValue()* and *map()* until it consumes all the data in the assigned InputSplit. The *nextKevValue()* method returns one record for the map function.

Figure 1 illustrates the overview of Hadoop 2 architecture and the sequence for processing a job. When a client submits a job, RM allocates an AM, which in turn allocates *YarnChilds* in multiple nodes in the cluster with the help of NM in each node. Note that, as shown in figure, each node has the same number of Containers that can contain either AM or *YarnChild*.

3.2 FFmpeg and AVC Video Format

FFmpeg is a widely used open source C++ library for video encoding and decoding which supports various video formats as well as audio formats [12]. Since it is well optimized, it is widely used across many platforms from servers to embedded devices.

In the library, *ffmpeg* is the main set of APIs for encoding and decoding video and audio, while *ffprobe* is used to obtain information such as key-frames and packet size. The decoding process of *ffmpeg* mainly consists of three steps: reading the video header, finding the codec information, and decoding the actual frames. Once the video header is read and codec information is found enough to initialize the codec, decoding function is called repeatedly in the loop until all the necessary frames are decoded.

Figure 2 illustrates that an AVC video format (i.e., mp4 file), which is one of the most widely used video coding format, consists of multiple *atoms*: *ftyp* atom contains information about video file type, and *mdat* contains encoded video frame data which consists of a sequence of GOPs (Group Of Pictures). The *moov* atom contains information such as whether or not a frame is a key-frame (i.e., I-frame), the byte offset of the frame, and so forth.

Note that, although we used FFmpeg as our video decoder to apply the proposed technique, in principle, our approach can be applied to any video decoder based on GOP since GOP can be processed in parallel and a decoding context could be constructed in a similar way, which will be explained later.

4. The Proposed Hadoop Video Decoding Framework

4.1 Challenges

Several challenges arise when a video file is to be decoded on multiple nodes in a distributed system, which can be phrased mainly into two parts: First, the size of each GOP in a video file varies since it is encoded data, while Hadoop InputSplit is generated as a fix-sized chunk such as 128MB. Thus, it is required to tokenize GOPs in an InputSplit, although a generic InputSplit has no information about the boundaries of each GOP.

Second, the decoding context, which is a structure in FFmpeg library required to decode frames in a GOP, must be reconstructed in a distributed environment. Each InputSplit is a fragmentation of the original input video without any meta information needed to construct the decoding context. Thus, we should be able to reconstruct the decoding context in each MapTask.

To solve these problems, we propose a distributed video processing framework on Hadoop that extends the FFmpeg library and is comprised of the following three steps, of which the overall architecture is shown in Fig. 3

1) *Generation of Video auxiliaries*
2) *Partitioning an InputSplit into GOPs*
3) *Reconstructing a decoding context*

4.2 VideoRecordReader

A typical AVC format video file has a number of GOPs. Each GOP is made up of one I-frame (i.e., key-frame), which can be decoded by itself without referencing any other frame. It is followed by more than one P-frames which reference the preceded I-frame in order to be decoded. Since a GOP does not depend on other GOPs, it is a natural unit of parallel processing.

As mentioned in the previous section, *nextKeyValue()* defined in Hadoop RecordReader tokenizes a record from the InputSplit. When InputSplit consists of text data such as in WordCount example, it is easy to distinguish sentences, hence easy to tokenize the InputSplit. However, GOP is a set of encoded video frame data whose size differs frame by frame depending on the encoding efficiency, which in turn depends on the nature of the original frame data. Thus, the size information for each GOP is needed to correctly tokenize them from the InputSplit.

For this purpose, auxiliary file called *MetaFrameInfo* which contains size information for each GOP is generated by a modified *ffprobe*. The modified *ffprobe* can find out efficiently the packet information such as byte position, packet size, key-frame info, and the width and height of the video

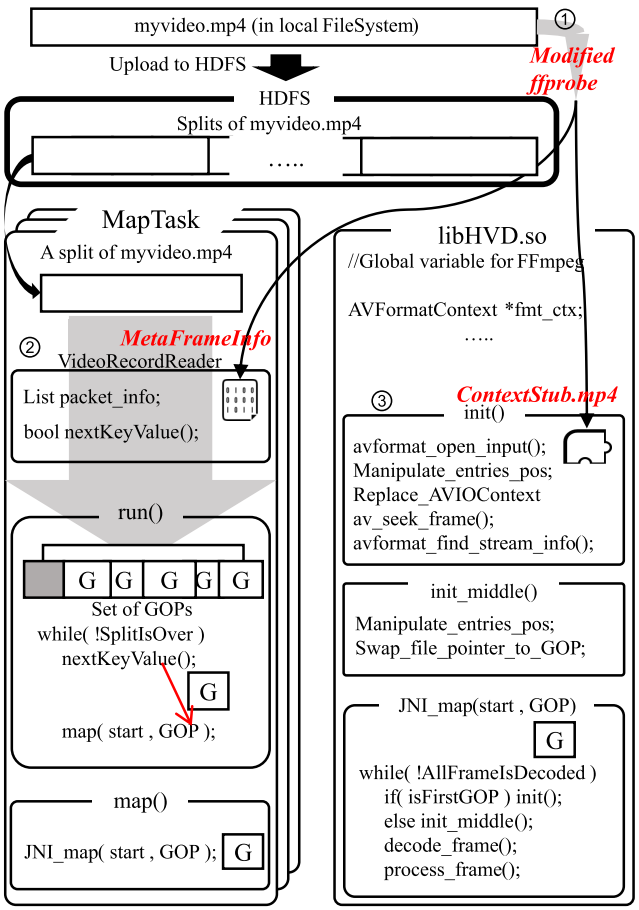

Fig. 3 Structure of Hadoop video decoding framework.

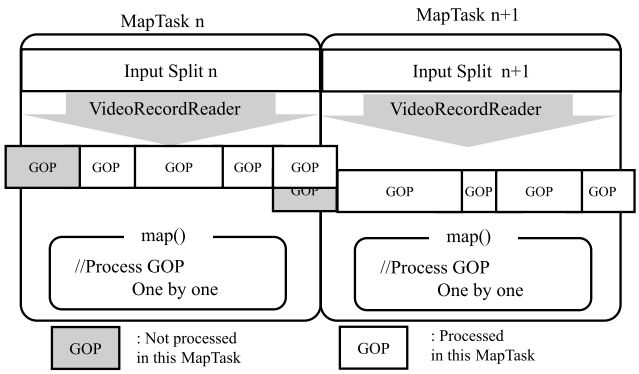

Fig. 4 The proposed VideoRecordReader can correctly partition each GOP of variable sizes, as well as InputSplit boundary handling.

frame. Among them, we store packet position and size in *MetaFrameInfo* file. Then, the generated file is scattered to the local file system of each node in the Hadoop cluster using *archives* option so that each MapTask can access this file later from the local disk of each node.

VideoRecordReader, to which we extended the RecordReader of Hadoop for GOP tokenizing, reads the information in the *MetaFrameInfo* file and keeps them in its data structure for later use in *nextKeyValue()* method. In our case, a record is a GOP with variable length, and now *nextKeyValue()* in VideoRecordReader can partition correctly each of GOPs in the InputSplit, referencing the size and the byte offset kept in its data structure. Also, it can partition correctly the GOP across the boundaries of the InputSplits. A GOP belongs to a MapTask depending on the start position: if it starts within an InputSplit of a MapTask, then the GOP belongs to the MapTask, even if the GOP ends in another InputSplit. Note that, even the part of a record resides in another InputSplit, Hadoop provides an API that returns such data. This is illustrated in Fig. 4.

4.3 Distributred Decoding by Extending FFmpeg APIs

To decode video frames, the decoder in *map()* function should construct the decoding context: i.e., the encoded type, the location of key-frames in a GOP, and the size and the location of each GOP. Moreover, the context construction should be done while accessing GOPs in the record delivered as a key-value pair.

The information that can construct the decoding context are kept not only in the header of the original encoded video file but also in the rear of the file, or even in the actual *mdat* data atom. To provide a decoding context to each MapTask in the distributed environment, we preprocess the video file in advance, and the required information is stored in a file named *ContextStub.mp4* which is scattered to each node in the cluster along with *MetaFrameInfo* file using Hadoop archives option. It contains only the required header and the fraction of data in the original mp4 file sufficient to construct a decoding context. Note that the size of *ContextStub.mp4* file is only about 10MB for a video clip of tens of GB and the processing time for generating the *ContextStub.mp4* is trivial as will be explained in the later section.

We directly modified and extended the FFmpeg library instead of using Xuggler since the proposed framework assumes JNI through which C libraries such as OpenCV are executed. Setting up the decoding context in the extended FFmpeg can be done in the following steps:

1) *avformat_open_input()* reads the atoms in the ContextStub.mp4 file and parses each atom. While parsing the atoms, the byte offsets of each frame including frame information such as key-frame (I-frame) info and the encoded frame size are retrieved and built as a table, which we call *entry table*.

To perform *avformat_open_input()* successfully, all the atoms in the video file except *mdat* atom are needed: it is provided in *ContextStub.mp4*, which is illustrated in Fig. 5. The video file always starts with *ftyp* atom, and most of the .mp4 file has *mdat* atom in the middle, followed by the index information in *moov* atom which is usually located at the end of the file as the index information can be decided only after the encoding is finished. Thus, the original video file needs to be cropped around the front and the rear of the *mdat* atom, and then merged again. In fact, the locations of *mdat* atom and *moov* atom can change depending on the files. Some .mp4 video files have *moov* atom in the middle,

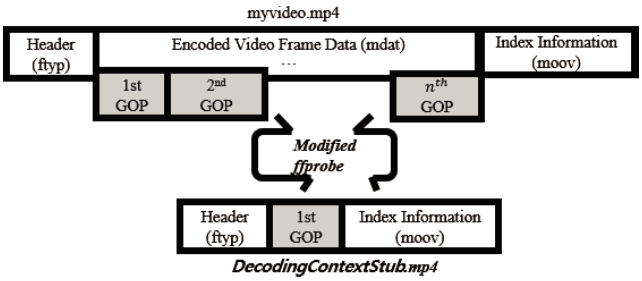

Fig. 5 ContextStub.mp4 generated by the modified ffprobe.

Frame-id [i]	Byte-offset entries[i].pos	Key-frame entries[i].flag
0	36	1
1	5603	0
2	9165	0
…		
240	4164261	1
241	4239430	0
…		

Byte-offset entries[i].pos	Key-frame entries[i].flag
36	1
5603	0
9165	0
…	
0	1
75169	0
…	

Fig. 6 Entry table.

while *mdat* atom is in the end of the file. We considered such variation in generating the auxiliaries using the modified ffprobe. It accepts any .mp4 format files and successfully generates ContextStub.mp4. After avformat_open_input() is called, the basic information in the decoding context is constructed.

2) The second step is to find out the frame id from the given key-value pair, referencing the *entry table* built in the previous step. As shown in Fig. 6, the table contains the byte offset of the packet and key-frame info. The *nextKeyValue()* method reads one GOP and passes it to *map()* function as a key-value pair, where the key is the byte offset of the GOP relative to the original sequential video file, and the value is the content of the GOP as in memory buffer. For example, if the key value is 4164261, then by referencing the *entry table*, the map function finds out that the id of the first frame in the given GOP is 240. Since the GOP no longer resides in the original sequential file but in the buffer, the byte offset of the first frame in the GOP is now truncated to 0, also adjusting the offset of the subsequent frames accordingly.

3) The decoding context is essentially constructed in *AVFormatContext* object, a data structure in FFmpeg library. In the original FFmpeg library, after *avformat_open_input()* completes, AVIOContext structure in the object is filled with the file pointer that locates the current frame in the original sequential video file. However, in our distributed framework, it simply locates the *ContextStub.mp4*, which does not contain any frame data except the first GOP. Thus, this structure needs to be modified so that it can point to the buffer where the given GOP is stored, not the *ContextStub.mp4*. In addition, the structure contains the functions that define how the video frame data can be read, written, and sought.

Reproduce the decoding context

1) call avformat_open_input() with *ContextStub.mp4*
2) Manipulate entry table
3) Replace AVIOContext to Custom AVIOContext
4) call av_seek_frame()
5) call avformat_find_stream_info()

Fig. 7 Sequence of filling decoding context.

Thus, these functions should be redefined so as to operate on the memory buffer, not in the file. To do that, we introduced *MemAVIOContext*, which replaces the *AVIOContext* structure in *AVFormatContext* object.

4) In addition to replacing *AVIOContext*, another member variable of *AVFormatContext* object that points to the current frame should be modified by *av_seek_frame()* function so that this variable can point the exact frame to be decoded.

5) *avformat_find_stream_info()* function finds the codec parameters and the stream information such as a codec type and a real frame rate, and initializes the codec object by actually reading the encoded frames. Thus, this function needs to read at least one GOP in *mdat* atom to retrieve the information.

Figure 7 summarizes the aforementioned steps to construct the decoding context of FFmpeg decoder in a distributed environment using *ContextStub.mp4*. Once the decoding context has been successfully constructed, *avcodec_decode_video2()* is finally called to decode a frame, and the subsequent tasks can process the decoded frame. Whenever *map()* is called again with a new GOP, only step 2) and 4) need to be done to adjust the byte offsets and the pointer to the current frame in the GOP. Other steps do not need to be repeated since the information have been already stored in the global variables. These are illustrated as *init()* and *init_middle()* in Fig. 3.

Although we explained our approach using specific FFmpeg APIs, the concept of reproducing the decoding context to enable distributed decoding, which is summarized in Fig. 7, can be applied to other GOP-based video decoders.

5. Applications

In this section, we briefly introduce the two video analytics applications that we used for the experiments: face detection application and SURF based object detection application. The processing is applied to the decoded frame using *process_frame()* as explained in Fig. 3.

5.1 Face Detection

Face detection is a typical video analytics and can be used in various applications. For example, it is often necessary to find a target subject in a large surveillance video sequence as soon as possible. Since it is usually very time-consuming to search through the entire video frames manually by human

beings, efficient processing of automatic face detection and recognition is necessary. In the experiments, we used the CPU implementation of Local Binary Pattern (LBP) based face detection presented in [13]. For the given frame, a pre-trained face classifier searches a face of any size using LBP features, varying the search window size.

5.2 SURF (Speed Up Robust Features)

SURF is a feature detection algorithm [14] that uses scale and rotation invariant features in the images. SURF is faster than SIFT by using Hessian detector, thus widely used in object detection and classification. Unlike face detection, no pre-trained classifier is required. Once target object is given, the SURF feature vectors are extracted. The same SURF feature extraction is applied to all the search windows in the given video frame, and if the distance calculated by the matching operation using both features from the target object and the one in the current search window is below threshold, the two objects are considered to be the same. We used SURF based object detection application in OpenCV 3.1.0 [15]. Since SURF involves a number of operations and is time-consuming, the GPU implementation in OpenCV was used.

6. Experimental Result

Experiments were performed in a Hadoop 2.7.1 cluster which consists of one master node and four worker nodes. As described in Table 1, each node has 12 CPU cores and 64GB memory, and are connected through Infiniband.

As input video files, up to 96 TV show video clips of 720p resolution were used. The duration of each video file is in the range of 1h 10m to 1h 30m, which corresponds to 1.2GB to 1.8GB.

6.1 Comparison of the Proposed with *multi-file* Approach

First, we compared our approach with *multi-file* approach where a video file is not split but the whole file is processed by a single MapTask: i.e., a job has only one MapTask and one InputSplit, thus multiple input files can only be processed by multiple jobs in the *multi-file* approach. Note that mapping the whole file into an InputSplit can be done by overriding *isSplitable* method in *FileInputFormat* class to return *false*, as addressed in [1].

We used 1, 24, 48, 72, 96 video files so as to compare execution time of our proposed approach with *multi-file* approach. The execution time of the workload is defined to be the difference between the start time of the first job and the end time of the last job completed.

As the cluster configuration, we set 24 containers per node for *multi-file* approach since each node has 12 CPU cores and a job needs one MapTask as well as one Application Master. With this configuration, up to 12 video files or jobs can run simultaneously in a node, fully utilizing the underlying cores in the machine. For the proposed approach, we set 14 containers so that a job consists of 13 MapTasks as well as one ApplicationMaster: with 12 MapTasks, when a MapTask is completed, the CPU core would be idle for short time until it receives a new MapTask. Having one more MapTask than the number of CPU cores can avoid this idle situation. And 128MB of the default HDFS block size was used.

Table 1 Specification of each node in the cluster.

CPU	12-core Xeon E5-2630
RAM	64GB
Network	Infiniband
GPU	NVIDIA K20c

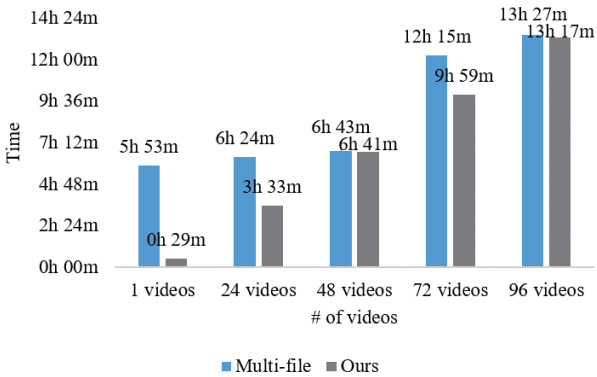

Fig. 8 Execution time of face detection application when processing various number of video clips using the proposed approach and the *multi-file* approach

Figure 8 shows the execution times of the face detection application with the proposed approach and *multi-file* approach. The execution time of the proposed approach when the number of input video files decreases, the execution time of our approach decreases linearly since the workload is fine grained and very well balanced across the nodes in the cluster with a number of GOPs even for a single job. However, the execution time of *multi-file* approach does not decrease linearly but decreases only at the multiple of 48. That is because the cluster has four nodes, each with 12 CPU cores. Thus, when the number of video files is 24 or even one, the execution time is more or less similar to the one with 48 files, meaning that the remaining 47 or 24 CPU cores are idle.

Note that the Infinitband network overhead in *multi-file* approach turned out to be negligible and as low as about 10 seconds for 1 GB file. This includes the replication transfer overhead: with the default replication number of three, each block is replicated and distributed across three nodes in the four-node cluster. Since the network overhead is negligible, it is confirmed that the inefficiency of *multi-file* approach mainly comes from the load imbalance mentioned before.

A more detailed analysis revealed that the load imbalance is caused not only by the coarse-grained workloads but also by the fact that Hadoop scheduler treats AM (*ApplicationMaster*) and *YarnChild* without distinction when allocating them to the nodes: they are simply the same Con-

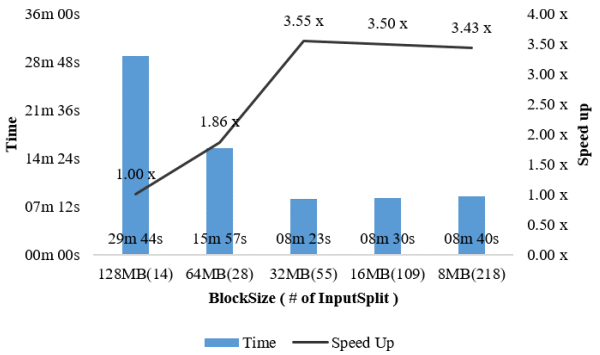

Fig. 9 Execution time of face detection with a 1.8 GB file when InputSplit size varies. The number in parenthesis is the total number of InputSplits.

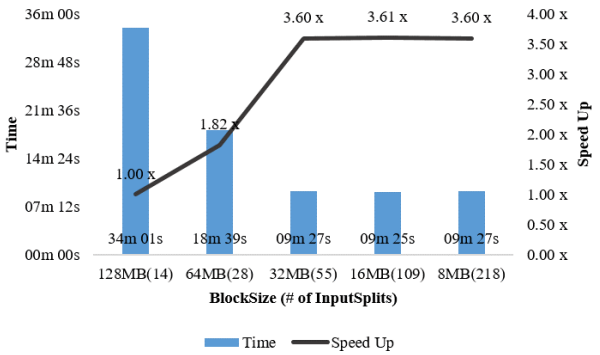

Fig. 10 Execution time of SURF with a 1.8 GB file when InputSplit size varies. The number in parenthesis is the total number of InputSplits.

tainer to Hadoop scheduler. Thus, it is possible that a node has more *YarnChilds* (i.e., MapTasks), while another has much less *YarnChilds* with more AMs instead. Since an AM is active only at the arrival of an InputSplit and is idle in the rest, this aggravates the load imbalance.

6.2 Effect of Changing InputSplit Size

Although it is obvious that the proposed GOP based distributed decoding approach is always more efficient than the *multi-file* approach, the proposed approach can also suffer from load imbalance when the number of InputSplits is not sufficient to feed all the Containers in the nodes. The number of InputSplits is determined by the size of an input video file and the InputSplit size.

Figures 9 and Fig. 10 show the execution time of the face detection and SURF applications, varying the size of InputSplits from 128MB of default size down to 8MB. Then, the number of InputSplits becomes from 14 to 218 for the 1.8 GB input file. Compared to the case with 128MB InputSplits, as the size of InputSplits gets smaller, the total execution time is decreased with the higher throughputs as more Containers in the cluster start to work. Beyond 32MB Split size, there is no further speedup since all the Containers in the cluster are working and the throughput does not increase: the number of Containers in the four-node cluster

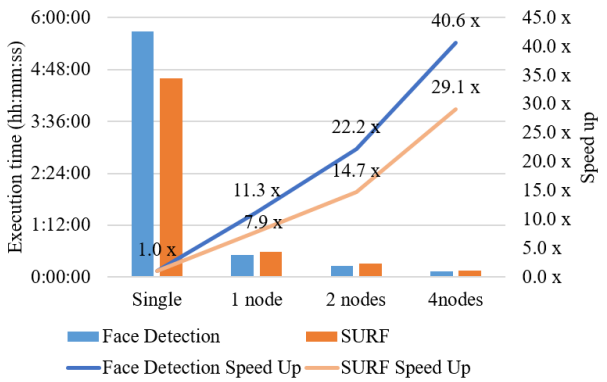

Fig. 11 Execution time as a number of nodes increases (1.8GB with 32MB InputSplits)

is 56 as we set 14 Containers per node. The slight decrease in the speedup if any, beyond 32MB Split size, is due to the process scheduling overhead since as many processes as the number of InputSplits are created and allocated.

As a result, both the face detection and SURF applications show 3.55 and 3.60 times of speedups with 32MB InputSplit size compared to the case with 128MB. It corresponds to 42 times faster processing compared to *multi-file* approach when there are one video file.

6.3 Scalability

As shown in Fig. 11, the proposed approach achieves very good scalability as the number nodes in the cluster increases. Compared to the single threaded execution without Hadoop, the speedup of each Hadoop application increases almost linearly as the number of nodes increases in the cluster. As 12 CPU cores in each node is fully utilized in the face detection application, it achieves about 11 times speedup with one node and 40.6 times speedup with four nodes. However, in the SURF application, a GPU is utilized as well as a CPU. Since the node in the cluster has only one GPU (k20c), it becomes the performance bottleneck. Although 12 MapTasks are executed per node, the speedups are about 8 times with one node and 29.1 times with four nodes.

A video analytics job for a large video file that would take several hours now takes only several minutes with our approach. This makes some applications such as interactive video search practical.

6.4 Preprocessing Overhead

There are two types of overheads in the proposed distributed decoding: generation of *ContextStub.mp4* and *MetaFrameInfo*, and copying these files to all nodes in the cluster using Hadoop archive option. With the careful extension of *ffprobe*, the generation time of the *ContextStub.mp4* and *MetaFrameInfo* takes only 0.7 second for about 2GB-long video file. Also, since the size of these files is only about 10MB, the broadcasting overhead is negligible.

6.5 Limitations

Currently, the proposed distributed video decoding framework works for .mp4 format, not .avi nor .mkv. However, these are the file container formats that contains h.264 encoded file. In fact, *ffmpeg* and *ffprobe* also support these formats. Thus, the proposed GOP based distributed decoding can be easily extended to support these formats.

The second limitation is that the preprocessing assumes the input video files to be on the local file system. If the encoding of raw data is written directly to HDFS, then downloading of the encoded file from HDFS to the local file system is necessary.

7. Conclusion

In this paper, we have presented a distributed video decoding framework, where even a single video file can be decoded in parallel in Hadoop. Compared to the conventional video decoding approaches on Hadoop where a job consists of a single InputSplit and multiple jobs are distributed in the cluster, the proposed approach is fine grained at GOP level so that it does not suffer from load imbalance when the number of input files is small but achieves good scalability in the distributed environment. To enable the GOP level distributed decoding, the widely used video codec library, *ffmpeg*, was modified and extended carefully, assuming JNI execution in Hadoop.

Specifically, VideoRecordReader has been implemented to tokenize variable-length GOPs correctly. Also, the decoding context is reconstructed in each MapTask so that the extended *ffmpeg* can decode the frames in the tokenized GOP in Hadoop. For this purpose, the original video file is preprocessed efficiently with the modified *ffprobe*, and auxiliary files are generated.

It was confirmed with two video analytics applications that the proposed approach achieves very good scalability even for a single video file, as long as there are enough InputSplits. Compared to the single threaded implementation, face detection application and SURF based object detection application achieved over 40 times and about 30 times speedups respectively, in four-node cluster.

References

[1] X. Zhao, H. Ma, H. Zhang, Y. Tang, and Y. Kou, "HVPI: Extending Hadoop to Support Video Analytic Applications," Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on, pp.789–796, IEEE, 2015.

[2] M. Kim, Y. Cui, S. Han, and H. Lee, "Towards efficient design and implementation of a hadoop-based distributed video transcoding system in cloud computing environment," International Journal of Multimedia and Ubiquitous Engineering, vol.8, no.2, pp.213–224, 2013.

[3] Xuggler, "Xuggler api," Avaliable: http://www.xuggle.com/xuggler/ [Accessed: 25 April 2015], 2012.

[4] H. Tan and L. Chen, "An approach for fast and parallel video processing on Apache Hadoop clusters," Multimedia and Expo (ICME), 2014 IEEE International Conference on, pp.1–6, IEEE, 2014.

[5] "Javacv project page on google code," Avaliable: http://code.google.com/p/ javacv [Accessed: 20 Aug. 2012].

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on, pp.1–10, IEEE, 2010.

[7] R. Gordon, Essential JNI: Java Native Interface, Prentice-Hall, Inc., 1998.

[8] C. Ryu, D. Lee, M. Jang, C. Kim, and E. Seo, "Extensible Video Processing Framework in Apache Hadoop," Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on, pp.305–310, IEEE, 2013.

[9] R. Radhakrishnan, "Using hadoop mapreduce for distributed video transcoding," Avaliable: https://content.pivotal.io/blog/using-hadoop-mapreduce-for-distributed-video-transcoding [Accessed: 22 May 2018], 2013.

[10] R. Pereira, M. Azambuja, K. Breitman, and M. Endler, "An Architecture for Distributed High Performance Video Processing in the Cloud," Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on, pp.482–489, IEEE, 2010.

[11] A. Hadoop, "Hadoop," Avaliable: http://hadoop.apache.org [Accessed: 27 Dec. 2017], 2009.

[12] F. Bellard, M. Niedermayer, et al., "Ffmpeg," Availabel from: http://ffmpeg.org, 2012.

[13] C. Oh, S. Yi, and Y. Yi, "Real-time face detection in Full HD images exploiting both embedded CPU and GPU," 2015 IEEE International Conference on Multimedia and Expo (ICME), pp.1–6, IEEE, 2015.

[14] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded Up Robust Features," European conference on computer vision, vol.3951, pp.404–417, Springer, 2006.

[15] OpenCV, "Opencv 3.1.0," Avaliable: http://opencv.org [Accessed: 27 December 2017], 2015.

Illo Yoon received the B.S. and M.S. degrees in Electrical and Computer engineering from the University of Seoul in 2015 and 2017, respectively. His research interest includes parallel software design and computer vision applications.

Saehanseul Yi received the B.S. and M.S. degrees in Electrical and Computer Engineering from the University of Seoul in 2013 and 2015, respectively. He is currently a Ph.D. student in the University of California, Irvine. His research interest includes parallel software design, heterogeneous computing, embedded GPU platforms, computer vision and high-performance distributed framework using manycore accelerators.

Chanyoung Oh received the B.S. degree in Electrical and Computer Engineering from the University of Seoul in 2015. He is currently a Ph.D. student in the University of Seoul. His research interest includes parallel software design, heterogeneous computing, embedded GPU platforms, computer vision and medical imaging.

Hyeonjin Jung received the B.S. degree in Electrical and Computer engineering from the University of Seoul in 2018. He is currently a M.S. degree student in University of Seoul. His research interest includes parallel software design, heterogeneous computing, and high-performance computing on a GPU cluster.

Youngmin Yi received the B.S. degree in Computer Engineering and Ph.D. degree in Electrical Engineering and Computer Science from Seoul National University in 2000 and 2007 respectively. He was a Postdoctoral Researcher at the University of California, Berkeley from 2007 and 2009, and a senior researcher at Samsung Advanced Institute of Technology from 2009 to 2010 before he joined the School of Electrical and Computer Engineering in the University of Seoul, where he is currently an Associate Professor. His research interest includes algorithm/architecture codesign for heterogeneous manycore platforms, GPU computing, and computer vision applications.