A Lightweight Distributed Media Processing System for UHD Service

Xusheng Zhang[1], Li Song[1,2], Rong Xie[1,2], Yanan Zhao[1]
[1] Institute of Image Communication and Network Engineering, Shanghai Jiao Tong University
[2] Cooperative Medianet Innovation Center
Shanghai, China
{zhangxusheng, song_li, xierong, ya_nanzhao}@sjtu.edu.cn

***Abstract*—With the rapid development of media services, it's an urgent demand that videos uploaded to the Internet need to be processed with high efficiency and high quality. We set up a flexible lightweight distributed system for media processing to address this problem. The architecture of our system consists of three layers, which are connected by three flows. This system shows high performance after deployed for a public video processing service for UGC in a private cloud environment. Moreover, the functional extension and efficiency optimization are discussed in this paper. Overall, the deployed public service shows remarkable performance in both processing efficiency and network load.**

Keywords—video processing; cloud media service; UHD

I. INTRODUCTION

In recent years we have witnessed an evolution in multimedia industry, from increasingly powerful underlying infrastructures, to ubiquitous HD/UHD/HDR consumer electronics, then massive volumes of user-generated (UGC), provider-generated (TV series, movies, games, etc.) and device-generated (auto captured contents from e.g. monitors) contents. Those new changes take us into a new visual era, but also pose many new challenges to the multimedia industry. For instance, UHD and HDR videos are getting increasingly popular, while the high resolution, high dynamic range, and high frame rate result in huge volumes of data to be processed, stored, and transmitted, which is orders of magnitude than before, and traditional processing method and tools could not handle this anymore. On the other hand, content providers may have even more critical requirements on to-market time, e.g. burst news in high definition format, hot TV series or movies be transcoded for VoD services. They favor the shortest time from content generation to ready-for-market. Because the huge volumes of data, traditional processing patterns which processes a job sequentially usually takes hours or days to finish just one video clip, which is far from satisfactory.

Researchers and industries have both made efforts in alleviating this burden. In all those tries, cloud computing, or distributed processing, is one of the most promising one. Cloud computing, or distributed processing, tackles the problem by splitting a big task into many sub-tasks and parallel processing them, and merges the sub-results into the final output at the end. Compared with the original sequential fashion, this could achieve orders of magnitude speedup. More importantly, the solution usually has very good horizontal scalability, which means as long as we have enough computing resources, we could handle any amount of volumes as input. In terminology, this is called IaaS, a typical product is Amazon AWS.

To fully utilize the advantages of IaaS, we have to design models which fit well to the parallel pattern. Designing and implementing a model on top IaaS are far more complex than sequential ones, since we have to take many new problems into consideration, e.g. as we have multiple machines simultaneously processing the same big job, we have to handle problems such as communication between machines, network bandwidth, control flow design, status monitoring, synchronizing between tasks or node, etc. But good news is that, since the processing model is so similar, there are already some general purpose models implemented, usually call big data processing platforms. This layer is called PaaS. The typical one is Apache Hadoop [1].

On top of PaaS, researchers and developers could implement their core processing logic, and provider specific services to users and customers. This is called SaaS.

The real ROI of today's multimedia industry is to provide best services (SaaS) to end users in the first time, but a missing piece is that there are no such general purpose and highly efficient multimedia processing platforms (PaaS) on top of the existing and mature enough underlying infrastructures (IaaS).

Let's explain why the existing big data PaaSes like Hadoop are not suitable for multimedia processing. Existing platforms, such as Hadoop, is designed for text processing, while multimedia content is binary. For example, the hello-world program of Hadoop is "Word Counting", which reads inputs from input file, and outputs the occurrences of each word. Compared with the original input (maybe GB, TB or even PB in size), the output is almost neglectable in size (usually KB, MB). For multimedia processing, things are totally different, the input is binary data (usually raw or compressed video files, in GB, TB, PB order), while the output also is binary data, and still in huge volume (for compressing, results may in be GB, MB order according to different compression parameters; for transcoding or pre/post processing, the results are not guaranteed to be smaller than input). Researchers have made some efforts to turn Hadoop into a framework for video

processing framework. However, it's hard to have a further optimization because of the limit of its architectural characteristics.

Under the circumstances, we proposed a custom system for huge volume video processing, which address the following challenges:

A. Lightweight

Nowadays, industries establish their multimedia processing platforms with the help of some popular heavyweight frameworks such as Hadoop, which contains distributed storage solution HDFS and distributed processing pattern MapReduce. Heavyweight tools promise the robust and full functioning of commercial system, but it's hard to be deployed in private multi-machine environments. To rid of the tightly-coupling of heavyweight frameworks, it's a better choice to construct system with lightweight frameworks, which can be easily deployed and freely replaced according to different individual needs. However, It's extremely challenging to set up a robust system, which consists of lightweight frameworks.

B. Scalability

The main challenge of a distributed system is how to cope with the explosive growth of under processed data: it must process fast, and must be flexible to extensions. Firstly, it's a common situation that the number of users may increase rapidly over a short period of time. In this condition, the system have to make full use of hardware resources to achieve a given performance. Secondly, scaling new video processing algorithms in system cluster can be extremely complex for developers. It's better for a mature system to decrease the costs of administration and maintenance.

C. Heterogeneity

While most algorithms are platform independent, their realizations are usually not. For instance, some algorithms may utilize underlying libraries only implemented on windows; some others may utilize GPU for acceleration; while some others are Linux dependent. Therefore, it's a better choice to build up several type of clusters deployed with different software and hardware environments.

D. Flexibility

Existing distributed video processing systems are usually handling one or several specified media processing tasks such as transcoding, denoise and super resolution. However, it's a common circumstance that above processing methods need to be combined to achieve better video enhancement. Thus, it's important to meet user's different requirements in a systematic way of processing.

In order to overcome these challenges, we design and set up a light-weight distributed system using open source frameworks, and do some optimization work for video processing. In the following parts, we first introduce the related work in distributed media processing system research. In section III, we introduce the design of three layers and three flows of our system. In section IV, we introduce a typical UHD service application of our system and propose the optimism patterns including Slice-Merge and data locality. In section V, we show function comparison between our system and other classical systems, and do some experiments to analyze the system performance. Finally, we conclude our work in section VI.

II. Related Work

Recently, many researchers set up their own cloud-based media processing services based on open source frameworks for distributed computation. Pereira R. et al. proposed a high-performance architecture dealing with large-scale video processing [2]. They realize a Split&Merge approach and deploy it in the public cloud AWS (Amazon Web Services). M. Kim et al. presented a Hadoop-based multimedia transcoding system upon the private media service platform SMCCSE [3][4]. They utilize MapReduce and HDFS to video transcoding, and improve the transcoding efficiency remarkably. While most researchers do many significant cloud video processing works based on Hadoop, others taking advantage of other distributed frameworks also make a great progress. Yang H. et al. proposed a real-time video stream processing system using Spark [5] in [6] and do optimization for reducing network traffic and latency. Zhang W. et al. proposed a cloud framework in [7] for real-time video processing based on Storm [8]. And in this framework, they designed a load-aware algorithm for CPU-GPU collaboration.

All above researches reasonably take advantage of open-source frameworks. However, these frameworks are not light enough to be easily deployed. In [9], Agrawal H. build CloudCV running on Amazon Web Services using lightweight tools, such as Celery [10] and Redis [11]. The CloudCV provides popular computer vision algorithms and popular datasets for users including computer vision researchers, scientists and non-scientists. However, it is mainly used for image processing and doesn't optimize enough for large scale video processing. Besides, if users want a complex processing for media content, they must submit tasks many times. Therefore, we build up a Celery-based distributed video processing system supporting Windows-Linux collaboration, CPU–GPU collaboration, sliced video processing and chain processing. The purpose of this distributed video processing system is re-targeting the offline video processing software, program or tools to distributed environment.

III. System Architecture

In this part, we introduce the design and implementation of our distributed media processing system. The overall architecture is shown in Fig. 1. The system is divided into three low coupling layers from bottom to top: data storage layer, data processing layer, and data presentation layer. Between three layers, we design three flows for information exchange: control flow, data flow, and state flow.

The system consists of several physical components, including web server nodes, controller nodes, worker nodes, and data storage nodes. Web server nodes provide the UI for user interaction. Controller nodes and worker nodes are similar in hardware resources, but for different utilization. Data storage nodes manage the persistent storage.

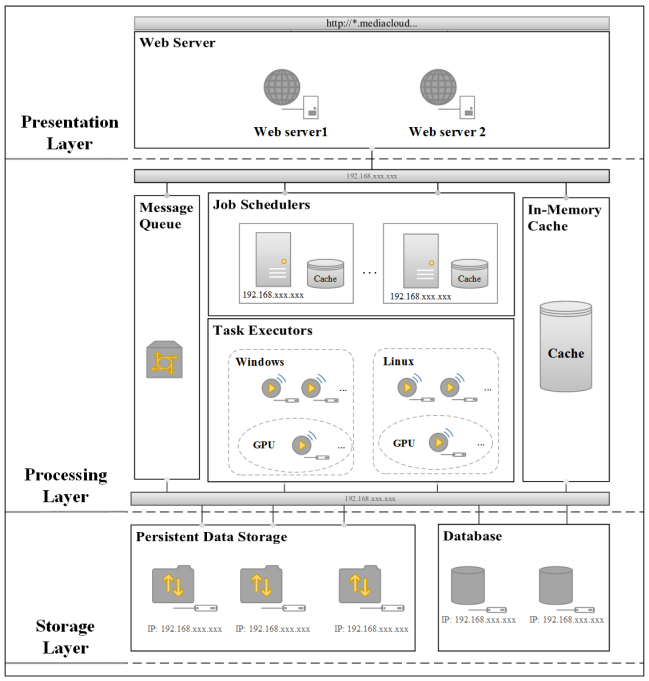

Fig. 1. High level architecture of the proposed lightweight video processing system

A. System Layers

1) Storage Layer

The storage layer is designed for permanent data of system. One of the characteristics of media processing is the high order of magnitude of video files, which poses great challenge on storage system and bandwidth. Besides, concurrent read/write will also slow down the I/O performance. Since our target is a light-weight multimedia processing system, we do not utilize any heavy storage solutions here. Eventually, we have a central storage design, in which the storage nodes provide FTP services to upper layers. This is enough in a private cloud or cluster environment, in terms of I/O performance, data integrity, and service usability. Besides the unprocessed multimedia data, the storage layer also records permanent information through database. The data storage and database can be connected as standalone module, which is convenient for distributed development.

2) Processing Layer

The core of the system is the video processing layer. It consists of several components: message queue workers, in-memory state servers, job schedulers, and task executors, which contains different types of filters. A job means a user submitted media processing request, e.g. transcode, a given video into another format. A job scheduler is a service on controller node, which is responsible for job handlings, e.g. receiving jobs, assigning jobs to proper worker nodes, terminate jobs, etc. Task executors focusing on executing specific tasks.

The scheduling module is based on message queue, which is designed in producer consumer pattern. One job scheduler node is deployed by job scheduler programs and a cache server. Users' unprocessed jobs are distributed to job scheduler cluster asynchronously according to load-balance principle. After jobs received, job scheduler, which could handle several jobs concurrently, obtains corresponding video content from data storage server, does preprocessing work, and transforms job to one or several tasks pushing to the specified message queue for task executors to pop. In our system, the main content of backend is multi types of task executor groups. One task executor contains task executing program, video processing algorithms and their dependencies. Task executors may differ from each other in terms of operating systems, whether or not has on-board GPU, and so on.

As a flexible system, we provide some basic filters following "one-in-one-out" pattern as basic units, each of which could fulfill one specific task, such as transcoding, denoising, super resolution, etc. Users could achieve complex jobs by combining this filters as a processing chain – just like Unix pipe. We implement this by job schedulers' monitoring. Although task distribution is an asynchronous procedure, the job scheduler periodically queries the state of task executing until complete. When the task executor processed task complete, the task executing parameters are updated for next processing filter. The update manner will be discussed in 3.2.

3) Presentation Layer

The presentation layer implements a WEB UI as the service entry. To support asynchronous video processing, which is necessary because of the large volumes of high quality video, there are user login/Registration pages, video management pages, and video processing submit/log pages. The system also exposes RESTful API to the outside, which benefits batch processing greatly, and support users to define more flexible and complex processing chains.

B. Flows

1) Control Flow

Control flow is in JSON format, which contains job description information, such as URIs of input/output files, processing filters, and filter parameters. The initial JSON, a sample of which can be seen in Fig. 2, is defined by users through web pages or RESTful API. The key "filters" record the first filter in the processing chain by a key-value couple, "Denoise" and its parameters in the sample. The job scheduler recognizes the key of first filter and submits the corresponding task to message queue. While one spare task executor popped task and processing it, the job scheduler starts a polling thread until the task processed complete. In the control JSON, there is a parameter named "next" – just like the linked list, recording the next filter and its parameter of the processing chain. When the task processed complete, the task executor will do JSON update work, including set the value of "filter" to the origin value of "next", and update the input file path. The updated JSON of Fig. 2 can be seen in Fig. 3. Therefore, when job scheduler gets the result of first task, it's convenient to recognize the key of second filter "Transcode" and re-submit the updated JSON to specified queue. In this circulation pattern, the processing chain can be as long as users' demand. The circulation will be stopped until the next is empty. Then the task executor will read the parameter of "output", and upload the final video file to permanent data storage.

```
job_control_json = {
    task_id: 'task_id',
    input: {filename: 'filename', file_id: 'file_id', file_path: 'ftp://ip:
            port@path/xx.mp4'},
    filters: {
      Denoise: {smooth_level: '1',
                    next: {Transcode: {bitrate: '4000',
                                video_codec: 'libx265',
                                    next: {},
                                output: {filename: 'filename1',
                    file_path: 'ftp://ip: port@path/xx_out1.mp4'},
                                    }},
                                }}
}
```

Fig. 2. The initial JSON instance of control flow.

```
job_control_json = {
    task_id: 'task_id',
    input: {filename: 'filename', file_id: 'file_id', file_path: 'ftp://ip:
            port@path/xx.mp4'},
    filters: {
      Transcode: {bitrate: '4000',
                        video_codec: 'libx265',
                            next: {},
                        output: {filename: 'filename1',
                file_path: 'ftp://ip: port@path/xx_out1.mp4'},
                            }},
}
```

Fig. 3. The updated JSON instance of Fig. 2 when processing.

2) State Flow

State flow is for monitoring purposes, which tracks the state of physical nodes, as well as the state of each running jobs. It's a better choice to take advantage of in-memory cache server for rapid state read/write. The temporary states are send to web server for user interaction, while some critical state such as "data download complete" and "data processed complete" will be record in permanent database at the same time. In our system, the monitoring module relies on Redis for messaging passing, and uses MySQL as database.

3) Data Flow

The main content of data flow is video slices, the volume of which is obviously much lager than control flow and state flow. In our system, the videos are stored in permanent data storage FTP after uploaded by user. When one video being processed, there is a cache video generated by every temporary filter. The cache videos have to been uploaded to cache servers for download by the next filter because of the distribution of filters. Therefore, we give two solutions for video data exchange, data storage server FTP, and in-memory cache server Redis. The Redis gives a quicker speed of read/write, while FTP gives larger storage space. Users could set the cache type to any one of above, if the volume of video haven't exceeded the maximum of Redis.

IV. Application In UHD Service

This lightweight distributed system has been deployed for a public UHD video processing service for UGC [12] in a private cloud environment. Fig. 4 shows the web interface of this UGC service. In this part, we describe the implement of video processing service in detail, and analyze the optimization of the system framework.

Fig. 4. The web page interface of the public UHD video processing service for UGC.

A. Application

The service using our framework aims to migrate UHD video processing procedures to media cloud service, which supports asynchronous video processing.

In the backend of this system, we implement the web server by Django [13], a lightweight web framework. Celery, a task queue using RabbitMQ as message broker, is the exchange methods between web server, job schedulers and task executors. Every job scheduler or task executor is taken as a Celery worker node. All worker nodes using the same broker form a consumer cluster. Celery supports different worker nodes to execute different tasks by the Queue. For one worker node, Queues is the list of message queues from which the worker will consume tasks. When system started, several types of Queues are initialized. it's worth noting that a special Queue is Job Scheduling Queue, which receives job message from web server. In this system, one job scheduler consumes 10 jobs at the same time. Other Queues are received various of task messages from job schedulers and wait for popping from task executors.

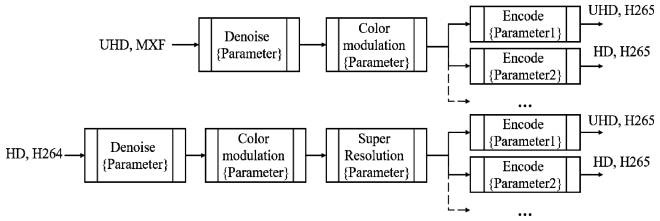

Fig. 5. Sample of 4k videos in MXF format encoding and HD videos enhanced transcoding.

There are two kinds of typical jobs in the project, UHD video (MXF format) coding and HD video enhanced transcoding. Fig. 5 shows the processing procedures. These two processing chains utilize many useful filters including denoising filter, color modulation filter and super resolution filter, etc. Filters' cores are vision algorithms based on both Windows and Linux environments. Since some algorithms,

TABLE I. COMPARISON OF SYSTEM FEATURES

Feature	P. R. et al. [4]	SMCCSE [5][6]	Yang H. et al. [8]	Zhang W. et al. [9]	CloudCV [11]	Our System
Scalability	✓	✓	✓	✓	✓	✓
Transcoding	✓	✓	×	✓	✓	✓
Video Enhancement	×	×	✓	✓	✓	✓
Slice&Merge	✓	×	×	×	×	✓
Data Locality	×	×	✓	×	×	✓
Windows-Linux Collaboration	×	×	×	×	✓	✓
CPU-GPU Collaboration	×	×	×	✓	✓	✓
Chain Processing	×	×	×	×	×	✓

such as super resolution algorithm SRCNN, can be accelerated by GPU [14], we also set up some GPU worker node groups.

As can be seen in Fig. 5, the enhanced videos usually need to be transcoded into multiple videos with different resolution, bitrate or framerate. Hence, we designed a one-in-multi-out pattern instead of one-in-one-out. To implement the design, we update JSON in control flow discussed in 3.2 by changing the type of parameter "next" to list.

B. Optimization

Besides above functional extensions, further efficiency improvements have been achieved in two aspects.

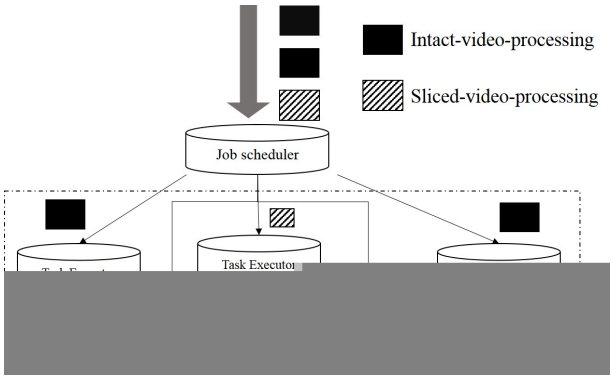

Fig. 6. Two kinds of job processing mode when taking Slice-Merge Pattern.

1) Slice-Merge Pattern

Video processing obviously can be a very time-consuming procedure when the video is high quality in resolution, framerate, and bitrate. Meanwhile, it's a waste of resources if not make full use of it during system idle periods. To speed up the video processing, we add a Slice-Merge module in job scheduler to improve the ability of distribution. Users can submit a Sliced-processing-job by specified interface. The difference between Intact-processing-job and Sliced-processing-job is shown in Fig. 6. When job scheduler received an Intact-processing-job, the job scheduler only parses the user's command and distributes the video processing tasks. However, after received a Sliced-processing-job, job scheduler will preprocess video. We add Slice module in job scheduler, which splits videos into many segments evenly or content adaptively. The job scheduler distributes every video slice to corresponding task executing cluster for parallel computing. At the meantime, the job scheduler monitors all states of processing tasks to maintain the processing chain. After all processing tasks completed, the video slices will be downloaded by job scheduler and merged by deployed Merge module. This Slice-Merge pattern improve the system resource utilization effectively when resources spare in idle periods.

2) Data Locality Pattern

In our designed processing chain, because of the lack of cache servers in task executing nodes, the task executors store the cache files produced by tasks for their next filters to remote cache servers, which are typically deployed in the corresponding job scheduling worker node. However, since most of task executors could handle multiple types of similar or related tasks, it's a recurrent phenomenon that the video data are retargeted to the same task worker nodes, which increase the consumption of time and network I/O. To address this problem, we add data locality module to decrease the waste of network resources. When switching to this mode, the Task executor will judge if "next" tasks can be processed in this node and decide whether continue to process next task or not. If the "next" tasks can be processed in the same node, the task executor will call the next filter functions directly, and the new filter function acquires video data from local node. This module accelerates the processing when the intermediate data is large in the chain. To ensure not exceeding the resource utilization limitation of computation nodes, it's a better choice to execute the "next" tasks serially, which is slower than executing parallel in multi worker nodes.

V. FUNCTION AND PERFORMANCE ANALYZE

A. Function Comparison

In this part, we make a feature comparison between our lightweight distributed system and other representative video processing system discussed in part 2.

The overall comparison can be seen in Table 1. The first problem to address of every system is scalability. It's convenient to utilize open-source framework to implement the distributed system. Heavyweight frameworks support explosive

growth of needs well, while we accomplish it by robust message queue servers.

Most of list systems supports video transcoding or encoding, which is discussed mostly about how to speed different types of videos in different environments. Video enhancement also is a popular function, which is supported in [8] [9] [11]. These two aspects are all implemented with the help of the existing video processing algorithms in our system.

Most of researchers only distribute large volumes of videos to cluster for parallel processing. However, the high-quality videos are too large to be processed in high speed. System in [4] and our system both develop Slice-Merge module to distribute large video to several worker nodes, which accelerate the video processing speed remarkably. [8] and our system both realize data locality pattern, which also is a great way to speed up the video processing, since it avoids unnecessary data exchange.

Our system realizes Windows-Linux collaboration and CPU-GPU collaboration through lightweight distributed tools based on Python, a cross-platform program language. These characters ensure that most of video processing tools can be integrated into our system. The unique character of our system is chain processing pattern, based on which the basic units – filter could be combined flexibly for complex video processing.

B. Performance Analyze

For system performance analyze, we set up a benchmark workflow in offline environment for experiments. The detailed information about running platform is shown in Table 2.

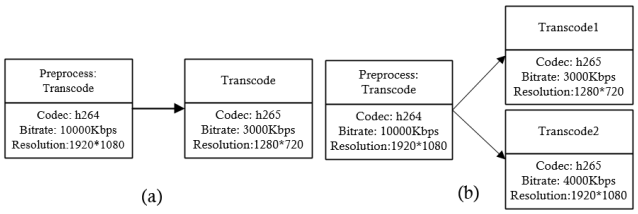

Fig. 7. Two types of processing chains in our test environment.

TABLE II. TEST ENVIRONMENT

Nodes	CPU	Memory	Number
Job scheduler	Intel Xeon E5 2.6GHz Cores: 24	64GB	1
Task executor	Intel Xeon E5 2.4GHz Cores: 16	32GB	5

We first test a sample processing chain (shown in Fig. 7) with two typical transcoders, which represent the preprocessing with high quality filter and the transcoding for television broadcasting or VOD. We take several test videos in 1 minute and get the average time consuming results. Fig. 8 illustrates the trends of processing times measured in seconds with the growth of job number in two environments: our test distributed system and standalone system. while the processing time lineally go up in standalone environment, it's remain the same in our system when using the Intact-video-processing system. We test the sliced-video-processing pattern by slicing video into pieces in equal length, one of which is limited in 10 seconds. Comparing with intact-video-processing, it can be noticed that sliced-video-processing considerably decrease the processing time when the job number is far more than task executor number. It's reasonable that when there are enough computational resources, parallel sliced video processing pattern saves processing time significantly, since it makes full use of system resources.

However, when the job number is close to task executor number, the executing time will increase slowly, because the sliced tasks processing requests exceed the ability of task executor cluster, while some tasks in message queue must wait for executing until other sub-tasks processed complete. It's worth noticing that when the job number equal to worker node number, the sliced-video-processing jobs execute longer time than intact-video-time-processing. The limit of task executor numbers results that every node must process tasks equals one intact job at least in average. Therefore, the exceeding time is scheduling time for distributing multiple sliced video clips.

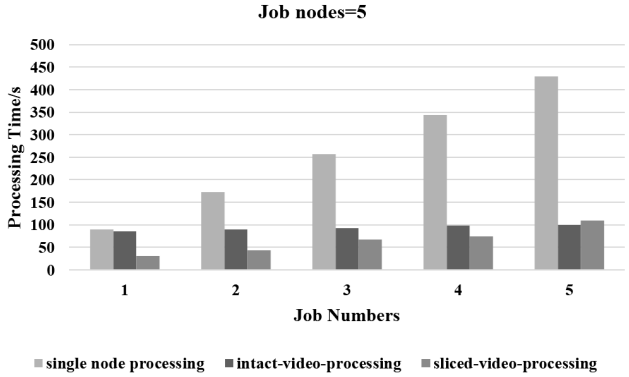

Fig. 8. Processing time of jobs in our test distributed system and standalone system.

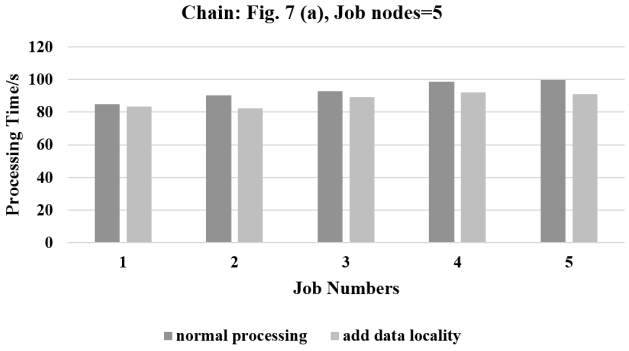

Fig. 9. Processing time (Processing chain: Fig. 7(a)) comparison between the job processing patterns whether utilize data locality or not.

The data locality pattern is designed for saving the task scheduling time, data transmission time, and network I/O. We first test this pattern using the processing chain shown in Fig 7 (a) and test video clips mentioned above. Fig. 9 shows the processing time comparison between the two job processing whether utilize data locality patterns. Data locality improves

processing efficiency by 5.93% in average in this experiment. The saving time can be considered as task scheduling time, clips upload and download time, whose proportions usually are much lower than video processing time. On the other hand, as introduced in part 4.2, when the processing chain includes "one-in-multi-out" design, the filters could be executed in parallel, which leads to low efficiency since the executing nodes only support serial execution. Fig. 10 shows the processing time when submitting the processing chain shown in Fig. 7(b). When the job number far less than task executing node number, the data locality distinctly increases the processing time comparing to normal processing. The reason is that Fig. 7(b) includes two transcoding filters in second step, but the two filters can't be executed at the same time because of the computational resource limit of one worker node. Under this circumstance, although the data locality decreases the network load and eliminates meaningless data transmission, the processing efficiency decrease much. Fig. 10 also illustrates that when not using data locality pattern, the executing time rapidly increase along with the jobs going up, which is caused by the quantity limit of worker nodes.

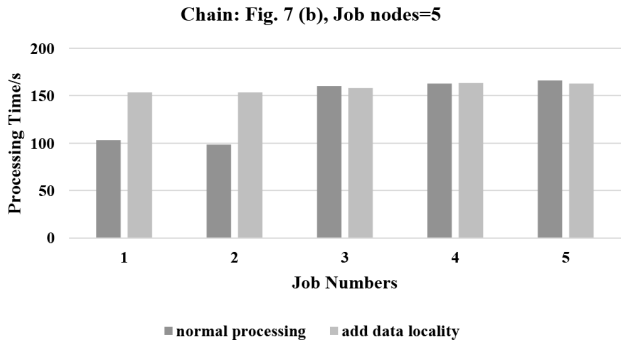

Fig. 10. Processing time (Processing chain: Fig. 7(b)) comparison between the job processing patterns whether utilize data locality or not.

The experiments show that the Slice-Merge module significantly improve the video processing efficiency in the system with adequate computing resources. Data locality pattern, mainly designed to decrease network load, also promote processing efficiency in a degree. However, when the control JSON's "next" parameter in processing chain has more than one filters, data locality pattern is not recommended as it will reduce the processing efficiency.

VI. Conclusion

In this paper, we introduce the design of our lightweight media processing system. The permanent data including video clips are stored in data storage layer. The data processing layer formed by job schedulers, task executors, message queue brokers and cache servers, migrates media processing algorithms to distributed computing cluster. And the representation layer provides the Web UI and RESTful API for user interaction. We also define three flows for message exchange between three layers.

In our system, we task basic processing units implementing video processing algorithms as filters, which can be combined flexibly to processing chains. This pattern improves the ability of extensive application remarkably.

This system has been deployed for a UHD video processing service for UGC. To accelerate the video processing, we designed Slice-Merge and data locality module for the system. These two module significantly improve the media processing efficiency.

Acknowledgment

This work was supported by NSFC (61671296, 61521062 and U1611461), the 111 Project (B07022 and Sheitc No.150633) and the Shanghai Key Laboratory of Digital Media Processing and Transmissions.

References

[1] Apache Hadoop - http://hadoop.apache.org.

[2] Pereira R, Azambuja M, Breitman K, et al. An Architecture for Distributed High Performance Video Processing in the Cloud[C]// IEEE International Conference on Cloud Computing, Cloud 2010, Miami, Fl, Usa, 5-10 July. 2010:482-489.

[3] Kim M, Han S, Cui Y, et al. A Hadoop-based Multimedia Transcoding System for Processing Social Media in the PaaS Platform of SMCCSE[J]. Ksii Transactions on Internet & Information Systems, 2012, 6(11):2827-2848.

[4] Kim M, Cui Y, Han S, et al. Towards efficient design and implementation of a Hadoop-based distributed video transcoding system in cloud computing environment[J]. International Journal of Multimedia & Ubiquitous Engineering, 2013, 8.

[5] Apache Spark - http://spark.apache.org.

[6] Yang H, Guo J, Liang C, et al. An Optimization of the Delay Scheduling Algorithm for Real-Time Video Stream Processing[M]// Frontier Computing. Springer Singapore, 2016.

[7] Zhang W, Duan P, Gong W, et al. A Load-Aware Pluggable Cloud Framework for Real-time Video Processing[J]. IEEE Transactions on Industrial Informatics, 2016:1-1.

[8] Apache Storm - http://storm.apache.org.

[9] Agrawal H, Mathialagan C S, Goyal Y, et al. CloudCV: Large-Scale Distributed Computer Vision as a Cloud Service[M]// Mobile Cloud Visual Media Computing. Springer International Publishing, 2015.

[10] Celery - http://www.celeryproject.org.

[11] Redis - https://redis.io.

[12] 4k evideocloud - http://4k.evideocloud.com.

[13] Django - https://www.djangoproject.com.

[14] Zhao Z, Song L, Xie R, et al. GPU accelerated high-quality video/image super-resolution[C]// IEEE International Symposium on Broadband Multimedia Systems and Broadcasting. 2016.