CS 230P Project Report

Distributed Multimedia Processing using Hadoop

Objective

The objective of this project is to build a distributed platform for media transcoding. Media transcoding is a very expensive operation in terms of computational power required, and has become more and more necessary as the number of media-enabled devices skyrocket. Each class of device has its own requirements in terms of video resolution, bitrate and codecs supported. Although there has been some effort to standardize in recent years, we still have the need to convert between different resolutions.

Fortunately, media transcoding happens to be easily scalable horizontally, since the work can be split up into smaller chunks and processed separately. This opens up an avenue for distributed processing platforms like Hadoop to be leveraged.

The goal of this project is to demonstrate this idea by implementing a basic video scaling program that leverages Hadoop and FFMPEG (media codec library) to perform video downscaling (converting video from a high resolution to a lower resolution).

Architecture

The media downscaling operation consists of 3 steps:

1. Splitting up source video into chunks of roughly equal size (IO bound).
2. Applying the transformation to each chunk independently (CPU bound).
3. Merging the result chunks back into one file in the correct order (IO bound).

Meanwhile, a typical Hadoop job looks something like this:

1. Input is read, and is split into key-value pairs (map operation)
2. These key-value pairs are shuffled so that the same keys are grouped together (shuffle).
3. Each group of key-value pairs (with the same key) is processed by a reduce task.
4. Further processing if any, is done, and the result is put back together.

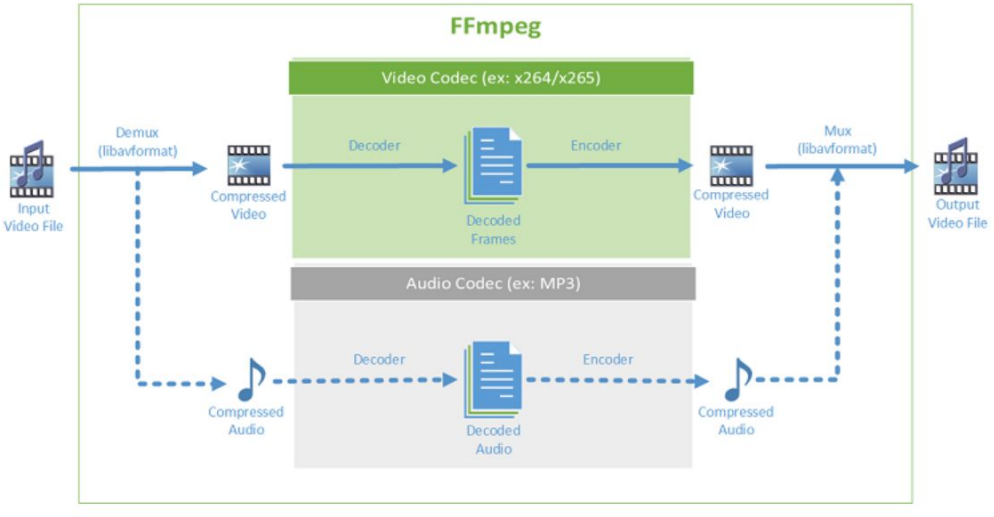
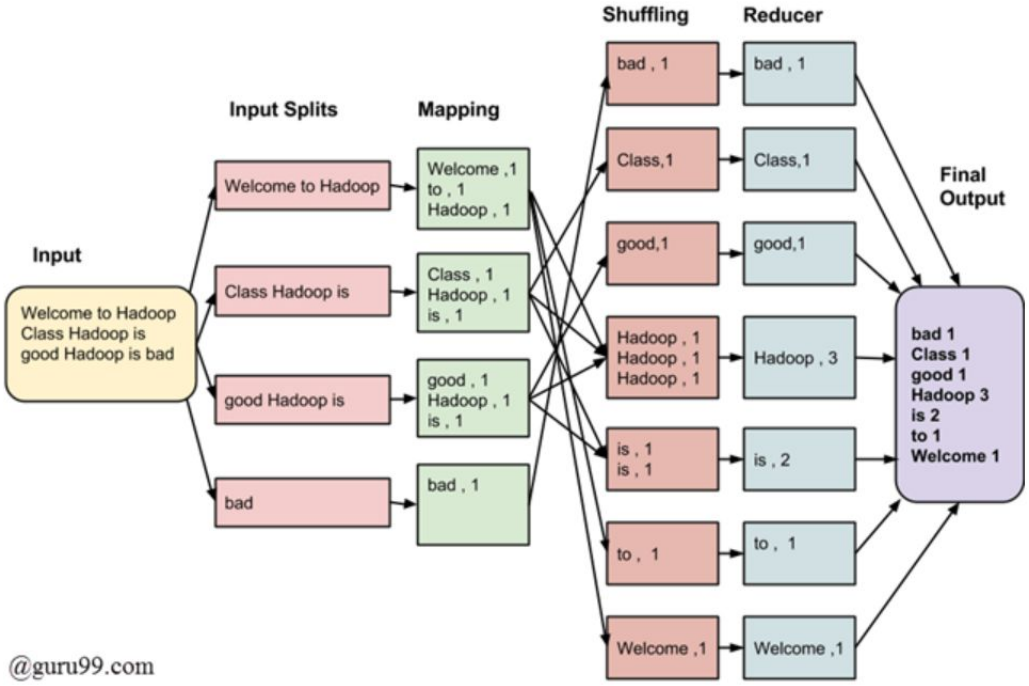


Figure 1: FFmpeg transcoding high-level flowchart.

High level overview of FFMPEG flow (source: intel.com)



Anatomy of a MapReduce operation that counts words in input text. (source: guru99.com)

For the purpose of this project, we can associate the reduce task with the actual video transformation operation. A Map task is strictly not needed, but can be leveraged to produce the chunks (which can otherwise be produced independently of Hadoop).

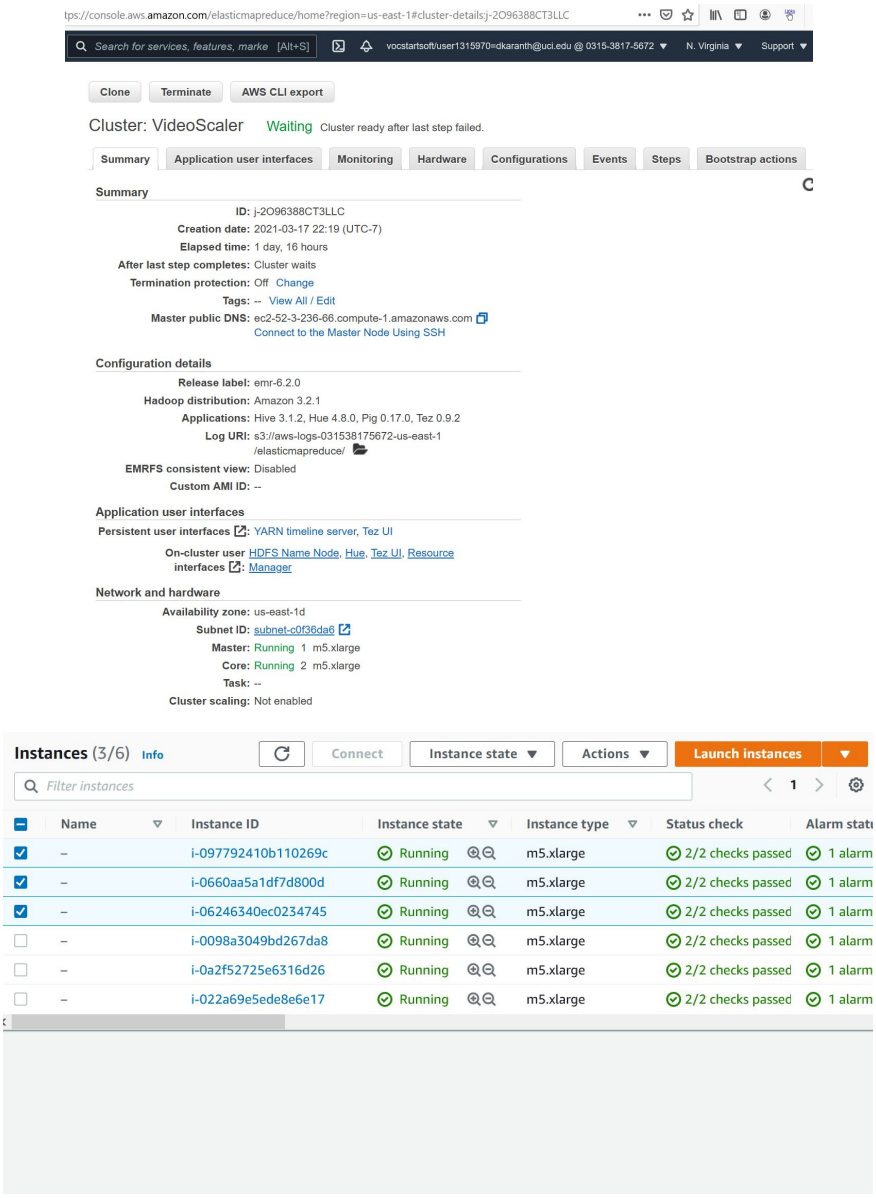
The combining operation is also best done outside Hadoop, since there is no major advantage in trying to do it in a distributed fashion.

Approach and Implementation

We have implemented the Mapper and Reducer as standalone Python programs, and are using Hadoop's Streaming API to inform the Hadoop platform of these executables. The Streaming API allows Hadoop to execute arbitrary programs as mappers and reducers. The programs themselves make use of FFMPEG library to query the media metadata, and to interact with HDFS storage.

All data (input, intermediary cache, output) is persisted in HDFS, and is cached locally as and when required.

The pipeline is run on AWS Elastic MapReduce (EMR) platform. However, we faced significant hurdles in making it work for our needs. FFMPEG was unavailable as an installable package in Amazon Linux instances, forcing us to build from source. There were also Python compatibility issues, with the Hadoop Command Runner refusing to execute Python3 programs. We had to fall back to the now-obsolete Python2. This also meant that we had to trigger the job manually, instead of using EMR web UI.



AWS UI: EMR cluster as well as the EC2 instances that make up the cluster.

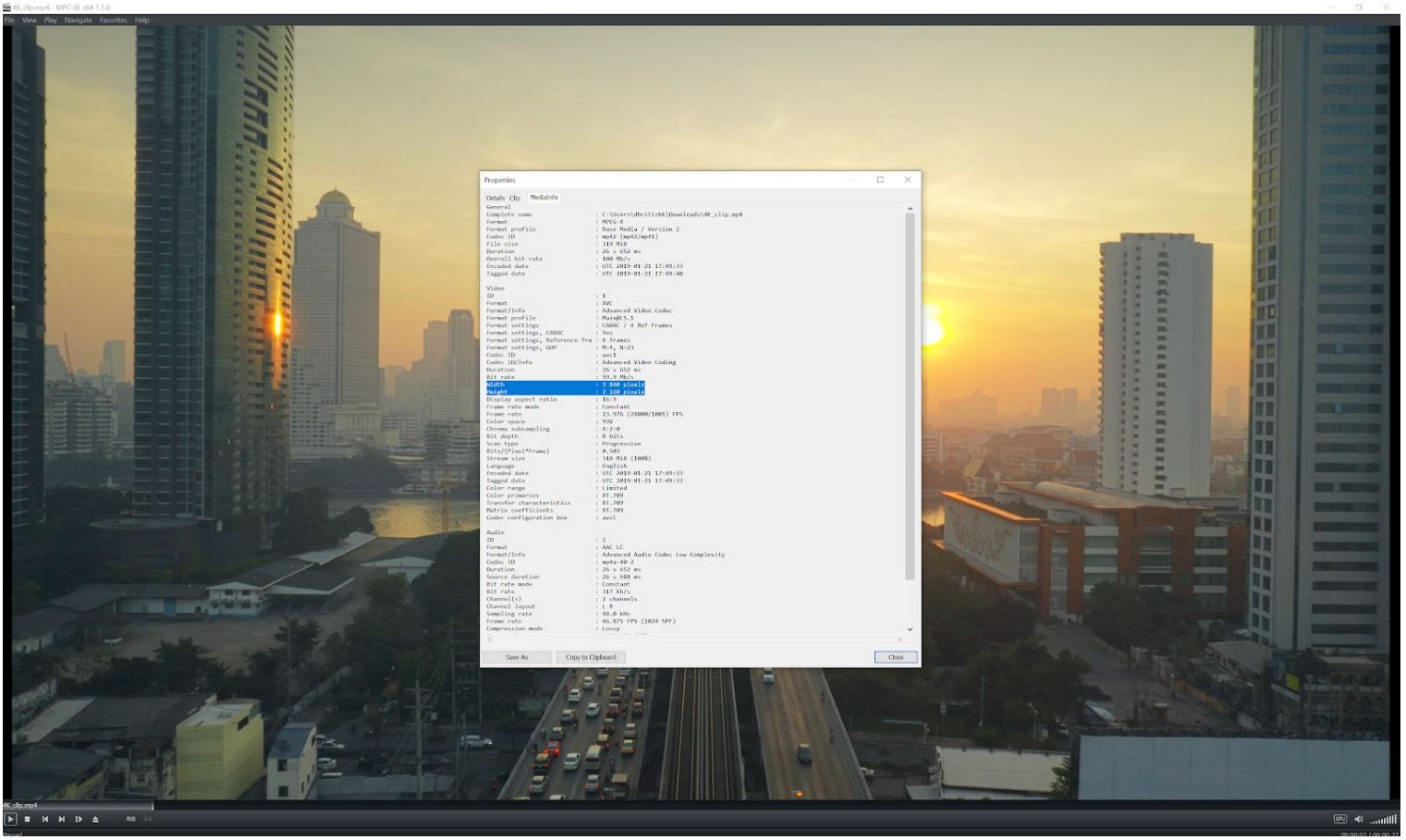
Results and Evaluation

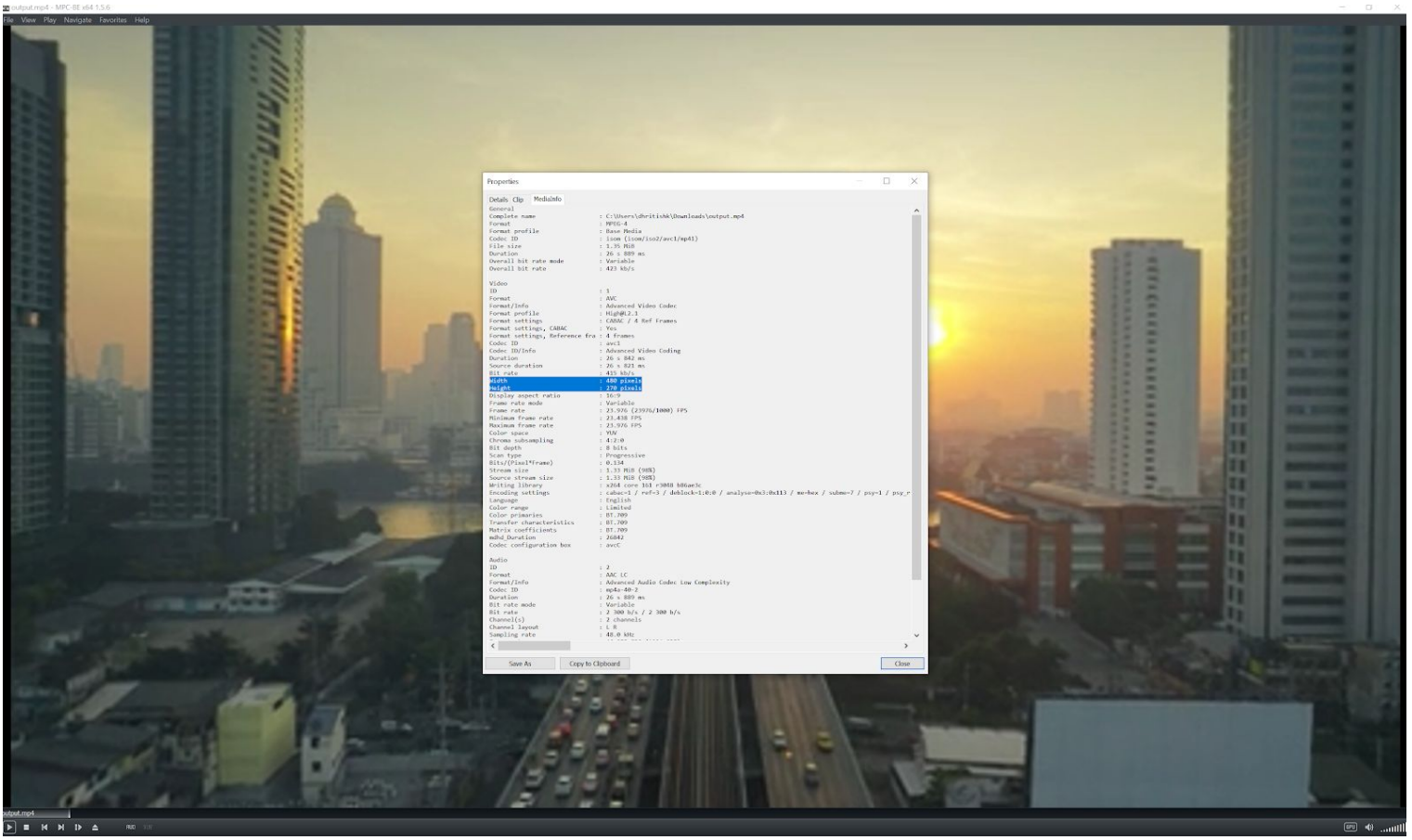
We performed evaluations on two media files: one a 20s clip and the other a 2 hour movie.

The clip was of 4K resolution, whereas the movie was of 1080p.

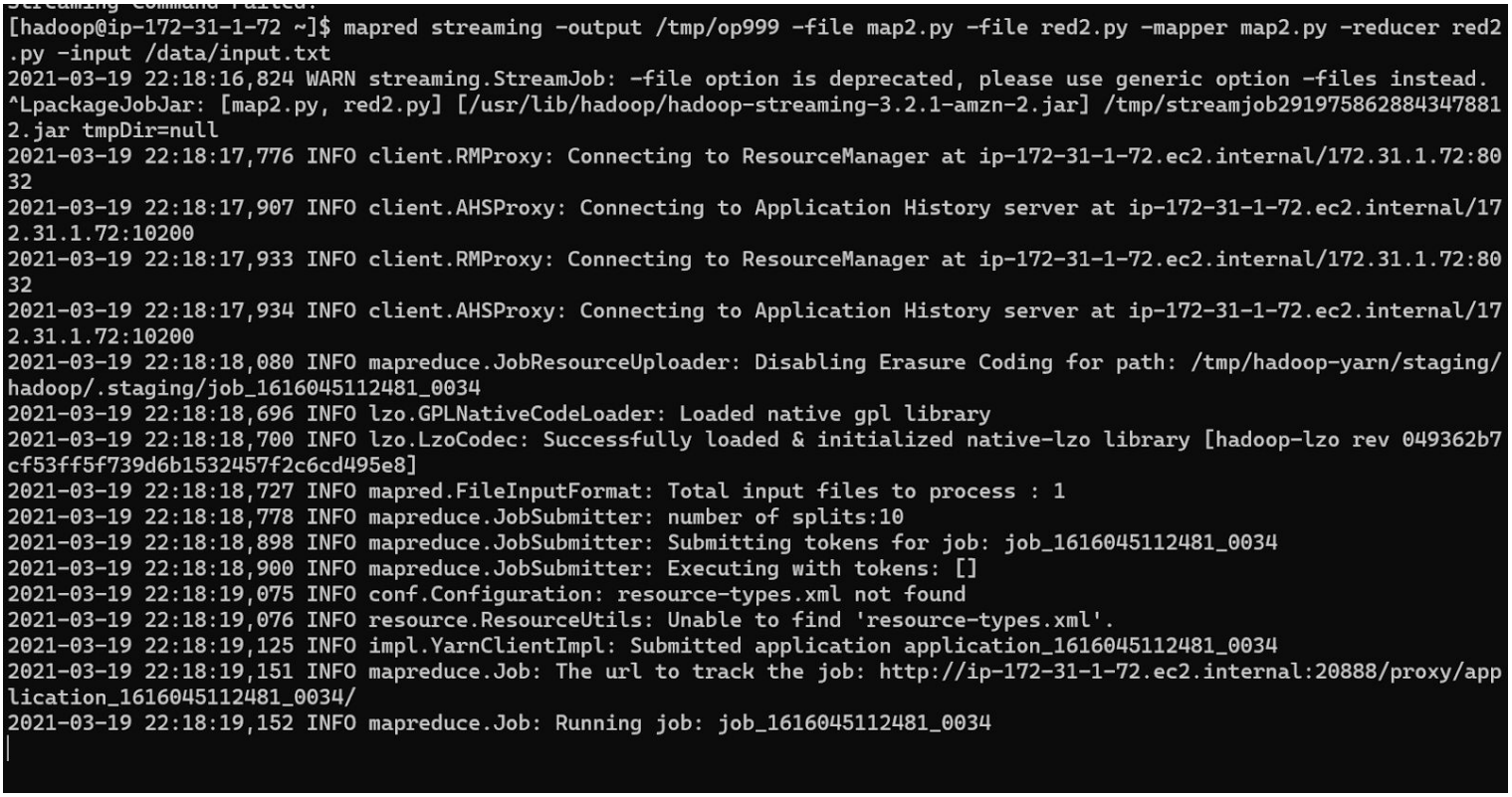
Media type	Duration for standalone mode	Duration for distributed mode
4K clip (30s)	15s	24s
1080p movie (2 hours)	56min	24min

As seen from the results, the distributed mode excels when the input file is very large. As the size decreases, the gain is negligible, and can even be slower than the standalone mode of operation, due to the overhead involved in coordinating inputs for each worker node. For large input, however, step #2 (processing chunks) becomes the most significant factor. Since this step is CPU bound, distributed mode is bound to have the advantage.





Source video (in 4K) and after downscaling to 480 x 270.



Execution of a job.

Future extensions

GPU-accelerated processing

Instead of using only CPU, we can leverage GPUs to speed up processing. There are plenty of GPU resources available today due to the proliferation of machine learning and gaming.

Use HDFS file streaming

Currently, the program splits input into chunks which are stored separately. Instead, we could implement file streaming where each reducer streams the necessary data from HDFS. This will alleviate the need to create chunks.

Implement in native Java

We faced quite a lot of issues when implementing this project in Python, due to poor compatibility between Hadoop and Python. Any future work should be done in Java, unless the compatibility issues are ironed out.

Conclusion

This project serves as a proof-of-concept, showing that video processing is easily scalable horizontally and can be further improved using off-the-shelf technologies.