# MapReduce based Audio and Video Transcoding

Student Name: Tom Stordy-Allison

## Background

Many companies have begun offering SaaS (Software as a Service) in the 'cloud' for transcoding video into various formats and codecs, but they mostly only use one machine per transcoding job, limiting the performance of an already computationally intensive and lengthy operation.

MapReduce is a paradigm popularised by Google in the early 2000's for distributing large amounts of data across many machines and the analysing the data across all of the machines in a two stage process taken from the world of functional programming; Map and then Reduce. This method of processing allows for problems to be formulated such that they can scale across machines easily.

## Aims

To investigate the feasibility of using the MapReduce paradigm to process video and transcode it into different formats and codecs, whilst achieving almost linear performance increases as the number of machines rises.

## Method

The transcoding problem will be formulated using the MapReduce primitive methods, and then used in various scenarios to test its feasibility for use in the real world. The performance of the system as the number of nodes in the cluster is increased will be analysed, along with various options for distributing the data, and how the cluster performance is affected when multiple jobs are ran simultaneously. The cost of running the solution vs. performance as the number of nodes rise will be studied. A small user study will also be undertaken to ensure that subjective quality of the output does not differ from that of a reference single machine transcoder.

## Proposed Solution

The project will use the Apache Hadoop platform; a Java based implementation of the MapReduce work from Google, to implement a transcoding solution. It will use the popular open source FFmpeg project and its associated libraries to perform the actual conversion, creating a set of MapReduce libraries for Hadoop that will formulate the problem correctly and split the audio and video data such that it can be distributed. A Java API will be written for job submission from local clients. The system will be tested on the Amazon Elastic MapReduce platform, with the data stored on Amazon S3.

***Keywords*** – Hadoop, MapReduce, Transcoding, Cloud, EC2, SaaS, Distributed, Parallel.

# I. Introduction

The conversion of video into different encodings is a computationally intensive operation. As video resolution continues to increase, it is becoming harder and harder to convert raw video into various encodings for display in (near) real-time. Transmitting this video via the web often requires various encodings of different types and quality, making this conversion even more difficult. This project will investigate the use of 'MapReduce' (Dean *et al, 2004)*, a parallel data processing paradigm popularised by Google several years ago, for the use of video conversion. It will investigate the use of this paradigm when data IO is not the necessarily the bottleneck, exploring the use of MapReduce in compute intensive scenarios.

## *Background: Video Transcoding*

Video and audio when stored digitally for distribution is nearly always encoded using some sort of 'lossy' compression. The choice of encoding type, and the settings used when the encoding is generated, dictate three attributes of the video: the output file size, the viewing quality and the performance required for playback. Often to either keep the file size down and/or make the encoding easily playable, the quality of the video is impaired.

Video (and audio) transcoding is the conversion of one of these encoding types of video to another encoding (Fhurt, 2008 : 951). This might involve changing some characteristics of the video or audio (e.g. the frame rate, or frame dimensions), or the format of the encoding, or both. It also encompasses the process of encoding of the video into a 'lossy' format, from its original lossless capture format.

In its raw form, video generally has a higher bitrate than what is possible for transmission to end users, and is also often of too higher quality for playback on end-user devices. As the distribution of video via the web increases in popularity, the need to have video available in various formats, at various quality levels and file sizes for transmission is becoming more and more important. However, the conversion of the original high quality video into these various formats is invariably bound by the available computational power, and there are many situations where the video needs to be ready as soon as possible for distribution to end-users.

## *Background: Containers and compression*

Video files are made up of a number of streams of audio and video, each of which has a specific compression format. These streams of different formats are interleaved with each other so that they can be read simultaneously for playback. The container format defines how the interleaving of the different streams takes place, and stores metadata describing the streams that are contained inside of it.

This container allows for the data to read out of the file in small chunks that are often called packets. Each packet belongs to a particular stream, and usually maps to a single frame of video, or a fixed number of audio samples. The packet also denotes whether it is a *key-frame* or not. If it is a key-frame then it is able to stand alone in the decoder without any other frames. The implementation details of this frame dependence are specific to the chosen codec, but the key-frame flag is useful for understanding the layout of the file without having to spend time decoding it.

Most video compression formats are a mixture of intraframe compression, and interframe compression (Richardson, 2002: 28). Intraframe compression is the compression of each frame as a standalone image, and is often very similar to that of image compression and formats like JPEG. Interframe compression uses one or more earlier or later frames from the stream to compress the current frame. The most simple example being where the current frame is compared with the previous frames and if areas of the frame are found to be the same only a reference to the area and frame is stored (Richardson, 2003:30).

These interdependent frames are grouped into chunks, called groups of pictures (GOP). Figure 1 shows an example of how these groups of pictures can be structured.

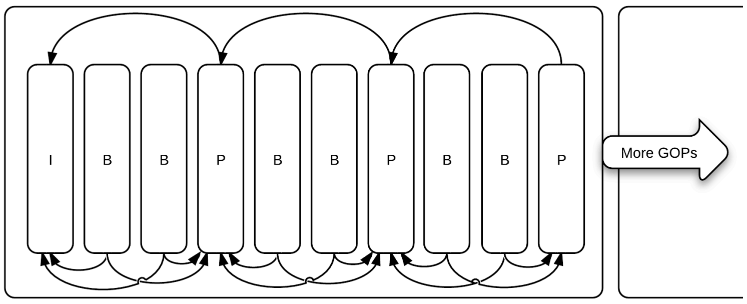

**Figure 1**. Typical GOP structure, with various types of frames referencing each other.

In a GOP the first frame is always an interframe, or an I-frame. The frames that follow this I-frame are usually intraframes, of one of two types, a B-frame or P-frame. P-frames are predictively coded frames; they base themselves on a previous frame in the stream. B-frames a bi-directionally predictive frames, they base themselves on frames both earlier and later in the stream.

The I-frame at the beginning of the GOP is called a key-frame. It can be decoded independently of other frames (such as for when seeking through a video) and also marks the point where a stream can be safely split (e.g. for editing the video). One variation of this rule exists called ‘Open GOP’, where frames are allowed to reference frames in the preceding GOP, as well as those of the current GOP to further improve compression. In this case the key-frame doesn’t always mark a safe place to split the stream.

## *Background: MapReduce*

MapReduce programs are designed to process large amounts of data in parallel (White, 2009:15), thus requiring that the data be split up in some fashion so that it can be processed in independent chunks. These chunks are then distributed between many machines, and as they are independent of each other they can scale easily without communication overhead.

Conceptually, MapReduce programs transform lists of input data elements into output data elements, using two different list processing idioms from functional programming: *map* and *reduce*.

The first phase of MapReduce is called *mapping* (Figure 2). Data elements are provided from a list, one at a time, to a function called the Mapper, which transforms each element individually to an output data element. This allows for the data elements to processed in parallel over many machines.

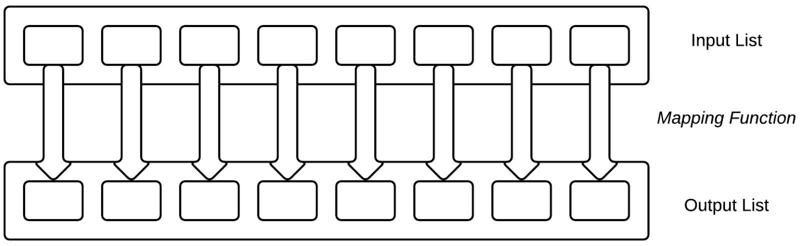

**Figure 2.** The map phase.

The second phase of MapReduce is called the *reducer*. This allows data elements to be aggregated together to produce an output from a set of values. It takes the output from the map phase as input and produces an output value (Figure 3).

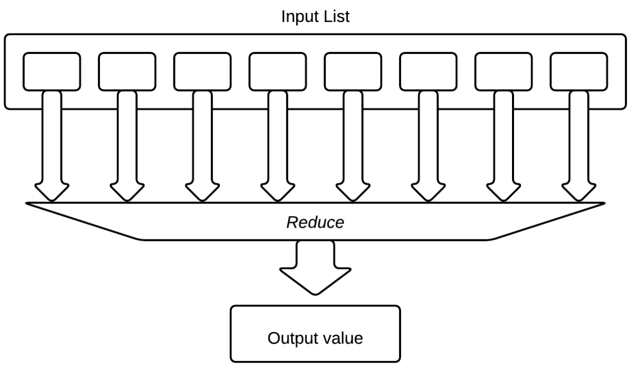

**Figure 3**. The reduce phase.

In MapReduce, no data value stands alone without a key. Each data element is part of a key-value pair. The environment is slightly less strict than that of a functional programming setting – allowing for the mappers and reducers to produce more than one key-value pair per input value. A reducer may also output multiple values and there is support for multiple reducers aggregating data based on its key.

Supporting multiple reducers requires that the input (the output from the map phase) be partitioned, but increases efficiency overall by allowing each reducer to run on separate machine. This partitioning can be customised, and the scheme used usually relies on the input value keys. Multiple reducers only ever receive all of the values for a specific key, allowing them to process the whole key, and its multiple data elements in one independent operation (Figure 4). This is important for our application of the reducer later in this document.

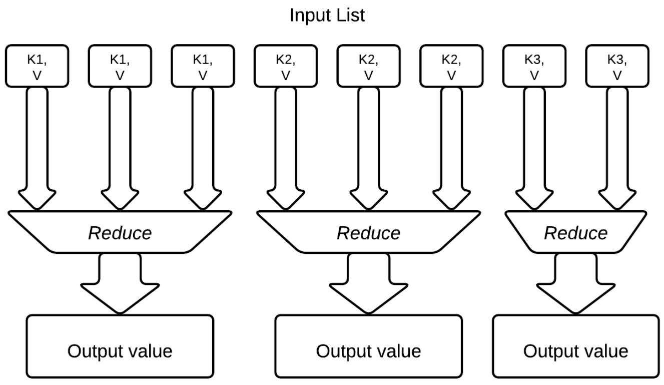

**Figure 4.** Multiple reducers.

MapReduce is primarily used for large data processing applications. But the model has the capacity to be used for various other applications too. One of the first public demonstrations of the power of MapReduce in a real world scenario, was the New York Times archive conversion in 2007, where TIFF images of public domain scanned articles from 1851 to 1922 were converted to PDF format (New York Times, 2007). 11 million articles were converted and ‘glued’ together in less than 24 hours of processing using around 100 nodes of Amazon Web Services’ EC2 instances using the MapReduce model.

Several different platforms exist that support the development of MapReduce programs, Apache Hadoop[1] being the open source favourite, with development contributions from Yahoo and Facebook. Apache Hadoop is also used by Amazon Elastic MapReduce[2], a cloud IaaS (infrastructure as a service) product, which allows clusters of machines to be rented at a fixed price per hour. This project will use Hadoop as the platform on which the video transcoding application will be developed, and Amazon Elastic MapReduce for testing.

*Project Purpose*

This project is a feasibility study to ascertain if MapReduce is a good candidate for video transcoding, and more widely if MapReduce can be used for applications that are not strictly large data processing, but more general computational applications. It will investigate how well the application performance scales as the number of nodes is increased. It will also investigate any decrease in compression efficiency caused by the implementation, and any degradation in viewing quality.

*Deliverables*

This project aims to produce a working MapReduce based transcoder that is able to take a variety of different input files and transcode them into either MPEG4 or H.264 based video. It will use the MapReduce paradigm to break up the problem and distribute the computation such that it is quicker than using a sequential (single machine) transcoder. It will only be required to perform single pass encoding (that is no first pass of the file will occur to gather statistics).

[1] http://hadoop.apache.org/ [last accessed: 19/1/12]
[2] http://aws.amazon.com/elasticmapreduce/ [last accessed: 19/1/12]

## II. Design

### *A. Functional Requirements*

The functional requirements for the project are detailed in table 1.

**TABLE 1.** FUNCTIONAL REQUIREMENTS.

| | **Description** | **Priority** |
|---|---|---|
| FR1 | Transcodes video and audio files to different codecs/containers using MapReduce for the computation | High |
| FR2 | Allows for output video frame size to be specified and rescaled | Medium |
| FR3 | Allows for output video quality settings to be specified | Medium |
| FR4 | Allows for output video and audio average bitrate to be specified | Medium |
| FR5 | Allows for submission of transcode job from a local client (non-cloud) using Java | Low |
| FR6 | Allows for several different transcode profiles to be specified for the same input file, and produces all of the outputs simultaneously | Low |

### *Non-functional Requirements*

The non-functional requirements for the project are detailed in table 2.

**TABLE 2.** NON-FUNCTIONAL REQUIREMENTS.

| | **Description** | **Priority** |
|---|---|---|
| NFR1 | The time taken to transcode a given file must be quicker than that of a single machine encoder. | High |
| NFR2 | The *Map* and *Reduce* functions are to be written for Hadoop 0.20.205. | High |
| NFR3 | The native binaries (used through JNI) should be complied for support on Linux (kernel 2.6) and Mac OS X. | High |
| NFR4 | The file compression for a transcode for a given file must be no more than 10% higher than that of a single machine encoder. | Medium |
| NFR5 | There should be no discernable quality difference against an encoding from a single machine encoder when encoding at comparable settings A user study will be performed to confirm. | Medium |
| NFR6 | The system should adhere to the MapReduce paradigm correctly, using the *Reduce* function effectively (it should not be missing a reducer, or only compatible with a single reducer). | Medium |
| NFR7 | The system should support use in Amazon Elastic MapReduce (EMR). | Low |

*B. High Level Overview*

*1) MapReduce Job*

Figure 5 describes the overall video transcoding process in the context of MapReduce. The *Transcode and Remux* section is a MapReduce program, and the *Demux* and *Merge* sections are single machine tasks that are executed as a pre and post processing operation.

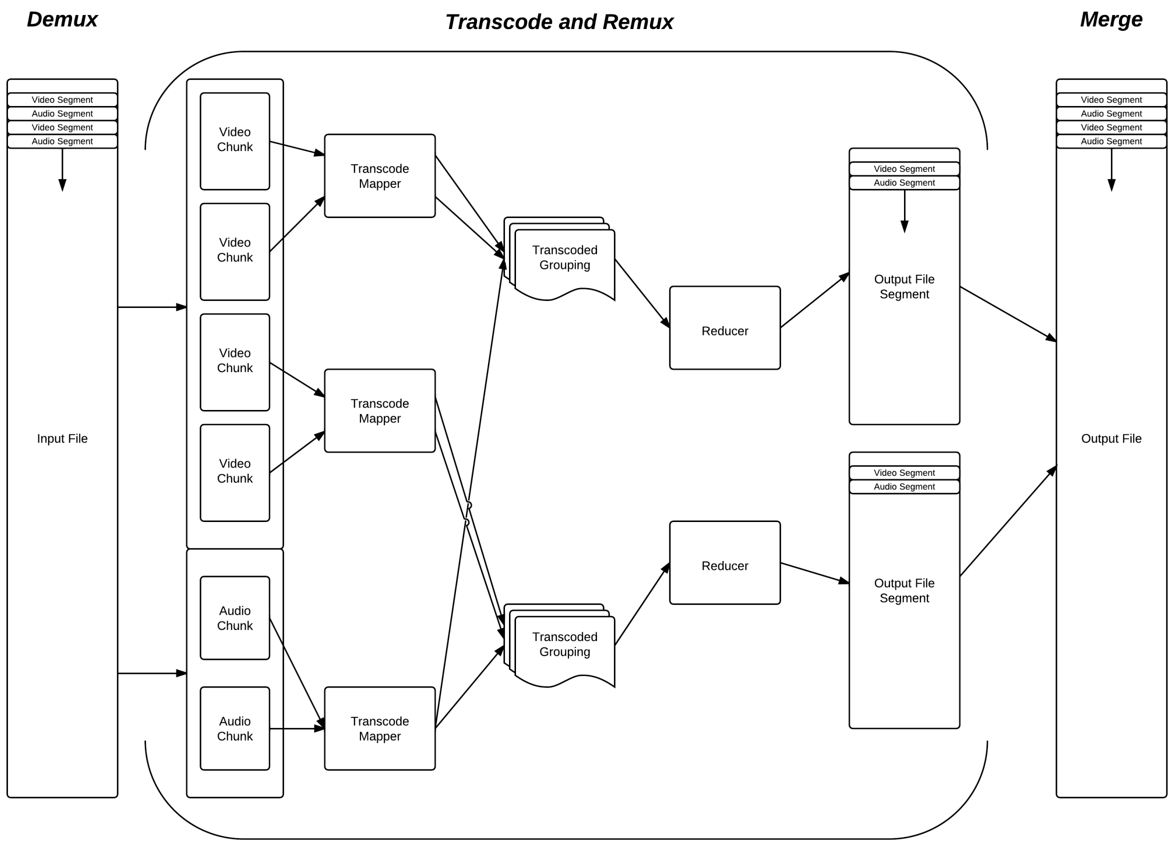

**Figure 5.** Conceptual overview diagram of the MapReduce program for transcoding.

Initially, the input file is broken down into chunks of data, per stream, ready for the MapReduce operation. This process is called the *Demux* phase. Each of these chunks is then assigned an available *Mapper*, which is where the actual conversion takes place, converting the chunk of data into the new format as desired. These output chunks are then grouped, and partitioned according to the number of available *Reducers* (usually slightly less than the number of *Mappers*). The grouping is based on the key that was associated with the output from the mapper, this key determining where in the output file the chunk should be. This means that the grouped input to the reducer can then be used to perform the reverse of the *Demux* phase, interleaving the streams again into a given output container. Because several of these reducers are run, the output is in fact in segments, equal to the number of reducers. The final phase of the overall transcode is to copy the output to its final destination, merging these segments as the copy operation takes place.

Parallelisation of this process becomes trivial, with each of the *Mappers* and each of the *Reducers* running on separate machines. The Hadoop platform coordinates the only piece of global communication when the output chunks are grouped and partitioned by their key between the Map and Reduce phases.

Each of the phases of the transcoding process will now be explained in further detail.

*a) Demux*

The first stage of processing a transcode job, and the most important in our case, is to ‘demux’ (meaning de-multiplex) the input file into its constituent streams, and into chunks of data that can be processed separately by the mappers. This phase of the process also has to determine several pieces of information in advance that are stored with each chunk so that it has all of the information it needs when a mapper is processing it. Once the chunks of data are defined and sent to a mapper there is no way for them to get any additional information.

Firstly, a chunk size is defined that is used to decide when an attempt to split the data should occur. This size is specified in bytes, so as to give each mapper a similar amount of work (be it audio or video). It is preferable to keep the work for each mapper similar, because if any of the mappers take a longer amount of time than another, the performance of the system will be reduced. The input is only ever split on specific boundaries called key-frames. In a video stream, these represent points in the video where and entire frame is defined that can stand alone, and does not depend on any other frames, and so we can split at this point without breaking in the structure of the file. In an audio stream we can split at any point, as the packets that we read out of the container are self contained.

As new packets of data are read from the input file, they are stored in temporary buffers for each stream. When the size of the data in the buffer is greater than the chunk size that was defined, it is emptied up until the last key-frame, ensuring that the GOP structures (the group of pictures that lie between two key frames) are kept together. A full GOP from the end of the previous chunk is also added to the start of each video chunk, as video that is using open GOP (that is, reference frames can refer to frames in the current GOP *and* the one preceding it) might need this data to decode correctly.

Finally a header is added to the chunk data that stores information about the stream that the chunk of data came from, and what the output format should be when this chunk is transcoded. Figure 6 shows an example of how a chunk of data (the value) is represented along side its key.

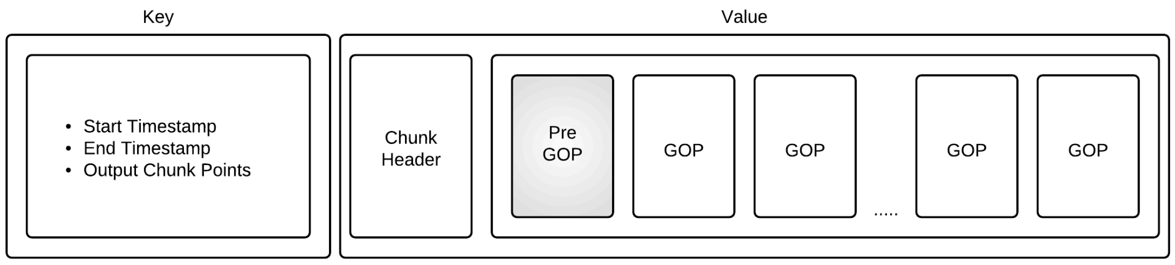

**Figure 6.** Layout of the chunk key-value data for a video stream.

The data is output using a Hadoop storage format called a *SequenceFile*. This allows us to store the data in key-value pairs that describe where this chunk was in the original file, how to output this chunk, and where it needs to be split during the encode process in the *Mapper*. Storing the output in this format also makes the data trivial to read in using Hadoop, as the key-value structure matches that of the *Mapper* input.

Calculating the points where extra splits in the mapper need to occur and why we need them requires more understanding of how the different chunks of data in the stream interrelate with respect to time. Figure 7 will be used to explain this relationship.

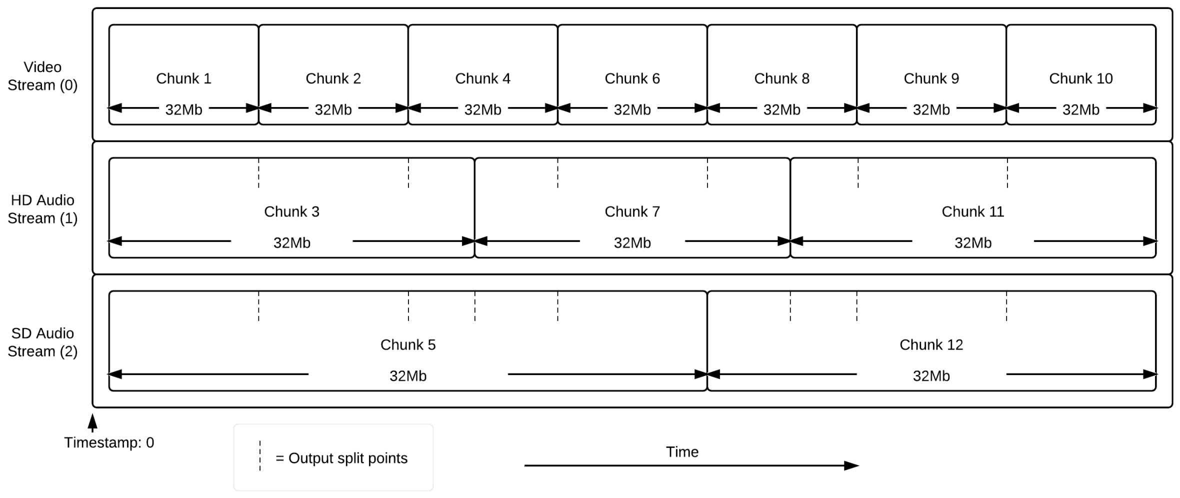

**Figure 7.** Chunks inside of streams as time elapses.

Figure 7 shows an example of the different streams that might be found in a given video file. Each chunk of data has approximately the same size, but the amount of time that the stream takes up in the video differs greatly. The video stream has the highest number of chunks, as more data is required to store it than audio. Once each of these chunks are processed in the Mapper, their output will not necessarily group together so that the chunks end and start at the same position. However, so that the reducer can run and output a segment of the video file correctly (see Figure 5 – the overview diagram), all of the output must be present in the transcode grouping. Instead, specific points in the chunk are specified where the output is to be split in the Mapper. This results in many smaller chunks in the output from the Map phase than were inputted, that when grouped by their starting timestamp, all of them with the same key end at exactly the same time. This allows for any number of reducers and the partition that is created will still be valid.

Note the order in which the chunks are generated in Figure 7; the split points only ever originating from previously outputted chunks (these points are shown by a dotted line). All the *previous* end points of chunks that were have outputted that lie between the start and end of the current chunk's timestamp are recorded in the key that is stored with the chunk data (see Figure 6). This provides the Mapper with all of the points to split on correctly, and the Reducer with groupings that are complete and are ready to be interleaved.

*b) Transcode, Sort and Reduce.*

This phase of the process is the actual MapReduce job that will be submitted to Hadoop. The MapReduce job will read in the file that was generated in the *Demux* phase that contains the chunks and their related data. These key-value structures will be allocated to *Mappers*, and the processing of the data will begin as soon as they receive their first chunk.

The Mapper will interact with the low level transcoding API, giving it all of the split points for the output from the chunk, the input format and desired output format, and the data itself. When the processing of the chunk is complete, the mapper will return several key-value objects, with the key as the timestamp that the data begins at, and the value as the data.

Each of these chunks will be guaranteed to have the same length of time when played back as described in the previous section.

As each chunk of data completes, Hadoop will read the key, and assign it a Reducer and begin copying the data to the machine if needed (it tries to keep the data on the same machine). This assignment will be dependent on the timestamp, the partitions being simply a division of the maximum timestamp by the number available reducers (see Figure 4). Hadoop will also sort the input data by timestamp, so the Reducer will always receive a complete set of chunks of data, one for chunk each stream, in order by timestamp.

The Reducer will then write out this data to the chosen container format, interleaving the streams and producing a segment of the final output. If there is only one reducer, then this is the final output. However, the system will support multiple reducers so that this process is as efficient as possible.

#### *c) Merge and Copy.*

The segments of output from the reducers are then read in one by one and merged into a final single file. As this merge take place, the file is copied to its chosen output destination (in the case of Amazon Elastic MapReduce, this will be Amazon S3, their storage system). The transcoding process will now be complete.

### *C. Low Level Overview*

#### *a) JNI Boundary*

Apache Hadoop (the chosen MapReduce platform) accepts Map and Reduce functions in the form of Java classes. However, Java doesn't have the support built-in or an external library with good enough functionality to encode video with the performance characteristics that are required to be able to outperform sequential transcoders. FFmpeg[3] is an open-source project; written in the C programming language, that has good performance, support for the decoding and encoding of numerous compression formats, and the multiplexing of many container formats. Its underlying functionality is made available through several well-defined libraries. For this reason, the system will use Java Native Interface (JNI) to implement the required interoperability layer so that FFmpeg can be used and the format support and abstraction layer offered by it can be used in this project.

A JNI interface will be written with a subset of the FFmpeg functionality, such that it is simple for the Java code to interact with. The library can be split up into 4 main parts: file introspection, de-multiplexing, transcoding and multiplexing.

The file introspection API will be used to get container format/codec information about the file into Java, so it can be compared with the requested output. This is an initial pre-process stage that is used to calculate the actual execution plan to get our desired output. This might mean that we don't have to re-encode the video as it is already in the correct format, or that we will keep one of the audio streams intact and simply add to it with a new one.

[3] http://www.ffmpeg.org/general.html#SEC6 [last accessed: 21/01/12]

The de-multiplexing API will be used to take the actual raw file, and split it up into its raw streams. It will also return points that the stream can be split up safely. It will achieve this by using the key-frame flag that is available as part of the FFmpeg API.

The transcoding API will take raw stream data from each Mapper and convert it into a new stream that uses a different codec. It will require that it be told the original and desired stream types, as all header information will have been lost by this point. It will support video and audio through the same API. It will accept a Java *InputStream* object, and a Java *OutputStream* object to obtain the raw data, and return the new byte stream. Some performance optimisation will be attempted here in JNI to make the data pass across the boundary with the lowest overhead possible, ensuring that as little data as possible is copied and trying to only maintain references to the original.

The multiplexing API will be used to put raw streams into the desired container format. It will need to know the target container, along with all of the stream types and their order, and then accept raw data from each in that order. The reducer will deal with ensuring this order is correct and that for each segment of data (that is a collection of streams that all start with the same timestamp and are of the same length) it has put all of the streams into the multiplexer before completing the segment. This in a similar way to the transcoding API will use an *InputStream* and *OutputStream* to keep IO abstraction as simple as possible in Java.

### *D. Development Plan*

The development will take place in several phases; loosely following in the data flow through the system:

1. Initial sample prototypes in C++
2. JNI boundary 'boiler-plate' code
3. Demux system
4. Map and Reduce system
5. Job submission API

The project will be developed using a mixture of C++ and Java using Xcode and Eclipse respectively, primarily on Mac OS X. The native libraries will be complied for Mac OS X and Linux 2.6, statically including the FFmpeg components so that they can be distributed easily on to remote machines. git will be used for version control throughout the project, although no major code integration will take place as there will only be a single developer. Unit testing on the Map and Reduce functions will be performed using JUnit, the tests themselves being trivial to setup owing to the stateless manner of the functions. The C++ sections of the system will tested using UnitTest++; namely the demuxer, transcoder and muxer.

### *E. Analysis*

Once the implementation is complete, the project will be analysed as follows:

- **Performance Tests**
  - Time taken as number of nodes increases on various different size files, compared to on single machine sequential FFmpeg run.
- **File Size/Compression Comparison Test**
  - Compared to input, and a sequential FFmpeg run.
- **Objective Quality Tests**
  - PSNR (peak signal to noise ratio) compared to input, and a sequential FFmpeg run.
  - SSIM (structural similarity index) compared to input, and a sequential FFmpeg run.
- **Subjective User Study Quality Tests**
  - Can users tell the difference between the distributed output, and one generated by FFmpeg sequentially
- **Cloud Encoder Comparison**[4]
  - Test run of my solution against 3 other market leading cloud transcoding solutions.
  - Involves several different file types, and tests throughout a given week on a cloud platform (in this case Amazon Elastic MapReduce).
  - Comparison against existing study.

Each of these tests will be run with varying file sizes, file types and numbers of processing nodes. It is expected that for small (single digit) cluster sizes that the performance increase should be close to linear. The file compression is not expected to be as efficient as a sequential transcode, as extra key frames will be introduced into the streams adding intraframes where there needn't be any. Quality is not expected to be visibly different by the naked eye, and the objective quality tests are not expected to show any major differences.

Varying the size of the chunks will be investigated, as well as analysis of the overall MapReduce performance (mapper time per chunk standard deviation, buffer spill rates onto hard disk). Use of higher specification machines will also be explored to see if it costs less to add more machines, or increase performance on the cluster generally.

## REFERENCES

Dean, Jeffrey & Ghemawat, Sanjay, 2004. *MapReduce: Simplified Data Processing on Large Clusters* Available at: http://labs.google.com/papers/mapreduce.html. [last accessed: 19/1/12]

B. Furht, 2008, *Encyclopaedia of Multimedia*, 2nd ed. Springer. ISBN: 0387747249

New York Times, November 1st, 2007. *Self-service, Prorated Super Computing Fun!*, [online] Available at: http://open.blogs.nytimes.com/2007/11/01/self-service- prorated-super-computing-fun/ [last accessed: 19/1/12]

Iain E. G. Richardson, 2002. *Video codec design: developing image and video compression systems,* Wiley and Sons. ISBN: 978-0-471-48553-7

Iain E. G. Richardson, 2003, *H.264 and MPEG-4 video compression: video coding for next-generation multimedia*, Wiley and Sons. ISBN: 978-0-470-84837-1

T. White, 2009. *Hadoop: The Definitive Guide*, O'Reilly Media, Inc., ISBN: 978-0-596-52197-4

[4] http://blog.cloudharmony.com/2011/10/encoding-performance-comparing-zencoder.html [last accessed: 19/1/12]