MapReduce Audio and Video Transcoding

Student Name: *Tom Stordy-Allison*

Supervisor Name: *David Budgen*

Submitted as part of the degree of ***Computer Science*** *to the Board of Examiners in the Department of Engineering and Computing Sciences, Durham University.*

Background – *MapReduce* is a paradigm for distributing large amounts of data across a cluster of machines and then analysing the data in parallel in a two-stage process taken from the world of functional programming. *MapReduce* allows for processing to be easily scaled across multiple machines.

Aims – To investigate the feasibility of using the *MapReduce* paradigm to process video and transcode it into different formats and codecs. Our aim was to achieve a performance improvement over a sequential single machine transcoder, and investigate this improvement as the number of machines used in a cluster is increased.

Method – The transcoding problem was formulated for *MapReduce* using *Apache Hadoop*, and then it was used in various scenarios to test its feasibility in clusters of up to 20 machines. The performance of our solution as the number of nodes in the cluster is increased was analysed, and we examined how the performance was affected by the size of the input. We also analysed the output of our transcoder in terms of visual quality and compression.

Results – Our implementation of video transcoding on *MapReduce* shows a performance improvement when large files are used for input, for cluster sizes up to 20. We found input file size to have a considerable effect on our system, larger input reducing overhead and increasing scalability.

Conclusions – Transcoding on *MapReduce* allows for large video files to be processed on clusters of machines, achieving a performance improvement with respect to the number of machines used. The output is of similar quality and of similar file size to that of a sequential encoder.

Keywords – Cloud, Distributed, EC2, MapReduce, Parallel, IaaS, Transcoding, Hadoop.

I. INTRODUCTION

A. *Project Background and Motivation*

Video transcoding is the process of converting video (and usually its associated audio) from one encoding format to another (Fhurt, 2008 : 951). Transcoding can also involve keeping the encoding format identical while adjusting the quality of the video so that size of the output is smaller. As video resolution increases, so does the amount of time it takes to transcode. Transmitting video via the web often requires various encodings of different types and quality to accommodate various devices and their bandwidth availability. This makes the conversion even more time consuming, as it has to be performed multiple times for each output type. The media industry, as well as many video-sharing websites, has to perform this type of transcode frequently, processing many terabytes of video per day[1]. As the popularity of web video continues to increase, the expectation from consumers is that video content should be available almost instantly on whichever device or connection they choose to use. Improving the performance of this transcode operation allows for the delay between content acquisition and the availability of the content on the various devices to be reduced.

MapReduce (Dean and Ghemawat, 2004) is a parallel data processing paradigm popularised by Google for 'web scale'[2] processing of information using a cluster of machines.

[1] http://www.bbc.co.uk/blogs/bbcinternet/2008/03/bbc_iplayer_on_iphone_behind_t.html ("peak data rate of over a gigabit per second")
[2] http://jacobian.org/writing/web-scale/ [last accessed: 19/04/12]

A *MapReduce* program uses a variant of the *Map* and *Reduce* functions from functional programming, to define a method of processing input that can scale across machines. The data is processed in segments requiring no global communication. This makes it ideal for parallel data processing problems, but also allows for other problems to be expressed in a way that takes advantage of the simple parallelism model it offers.

Video transcoding is getting slower as resolution increases, and more widespread as more devices begin to receive web video. *MapReduce* is a candidate for improving its performance. To use *MapReduce* we need to segment the video and audio efficiently for processing without damaging its structure and consequently preventing it from being decoded. Making good use of the *Map* and *Reduce* functions, so that the problem scales across machines without large overhead, requires careful consideration of the architecture of the system, and is difficult to test thoroughly.

It should also be noted, however, that *MapReduce* is still immature; the platforms for its implementation are only beginning to see their APIs and performance stabilise. *MapReduce* is usually used for data-driven scenarios, but as it is becoming more popular, and available in the cloud as part of various IaaS (Infrastructure as a Service) offerings, using it for computationally intensive tasks is becoming more attractive. Assessing to what extent highly computational problems can be used on a *MapReduce* cluster is of interest and the focus of this project is to use video transcoding to investigate this.

B. *MapReduce*

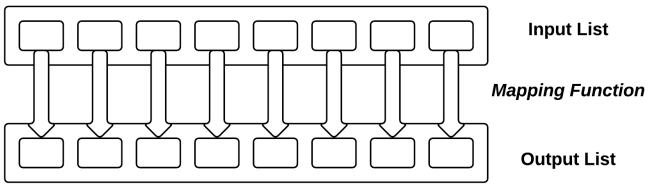

Figure 1a. Map

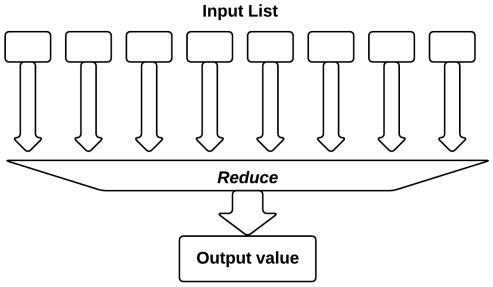

Figure 1b. Reduce

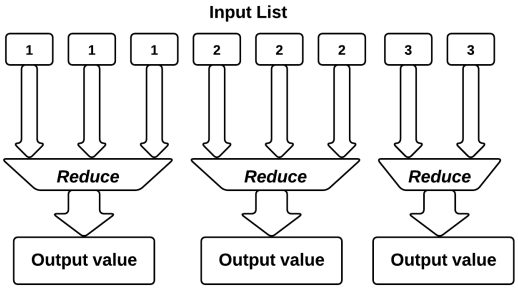

Figure 1c. Multiple reducers

As *MapReduce* forms the basis of our implementation, we will explain its background in some detail. *MapReduce* programs are designed to process large amounts of data in parallel (White, 2009:15), requiring that the data be in such a format that it can be split into and processed in independent chunks. These chunks are then distributed between many machines, and, as the processing operation for each chunk is independent of the others, their processing can scale easily without communication overhead between nodes.

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 1a). Data elements are provided from a list to a function called the *Mapper*, which transforms each element

individually to an output data element. This allows for the data elements to be processed in parallel over many machines, and the input list to be partitioned over many machines. 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 input values; namely the output from the map phase (Figure 1b).

In *MapReduce*, no data value stands alone without a key; each data element is part of a key-value pair. The environment is 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 (or none). 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 of *map* output/*reduce* input can be customised, and the scheme used usually relies on the input value keys. Each *reducer* only ever receives 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 1c). This is important for the application of the *reducer* in our implementation.

C. *Video Transcoding*

Video and audio, when stored digitally for distribution, is nearly always encoded by 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 computational power required for playback. Often the quality of the video is impaired to keep the file size down and/or make the encoding playable. Transcoding involves changing some of the 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 high a quality for playback on end-user devices.

D. *Project Deliverables*

The research question for this project is therefore, 'Can the *MapReduce* paradigm be used for video transcoding in an effective manner, so that it scales well across machines, has comparable output quality to current sequential encoders, and improves performance overall?'

We propose a method for segmenting video and audio correctly and efficiently, so that it can be processed in a *MapReduce* program, and a complimentary method for correctly putting the output data back together into a playable video file.

We evaluated various aspects of the implementation:

- We tested the performance of the *MapReduce* program as the number of parallel tasks was increased, for 8 different files sizes ranging from 4MB to 10GB. Smaller file sizes were unable to scale as well, as the segmentation relies on input file size.
- We compared the output quality of our parallel implementation with that of a reference sequential encoder, highlighting the effect that segmenting the video has on the ability of the encoder to maintain visual quality.
- We compared the output file size, indicating where the segmentation had been detrimental to the compression.
- We also performed a small user study verifying that the output from our solution is comparable to that of a reference sequential encoder, and that the visual quality had not been perceptibly affected.

II. RELATED WORK

In this section, we survey previous work in the areas of *MapReduce* and Distributed Video Transcoding.

A. *MapReduce*

Many different operations have been implemented using *MapReduce* to improve their performance. In Chen and Schlosser (2008) processing is achieved through a *MapReduce* implementation that estimates geographic information ('where is this scene?'), given a specific image, by leveraging a data set of around 6 million GPS-tagged images and using scene matching to find the most similar image through a reduce-less program. One of the first large scale demonstrations of the power of *MapReduce* in a real world scenario was a New York Times archive conversion, where the TIFF images of the public domain scanned articles from 1851 to 1922 were converted to PDF format (New York Times, November 1st, 2007). Using the *MapReduce* model, 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.

Fewer examples are available for video processing, one of them being HP Labs' *VideoToon* implementation[3]. They implement a service to "cartoonize" videos using a series of scripts that break the video up into its sub-streams for the *map* phase and then put them back together in the *reduce* phase after processing. In the specific context of video transcoding, Pereira et al (2010) propose a 'Split and Merge' generalisation of the *MapReduce* model, and have compared the use of this against several sequential encoders to show the performance improvements. They make a strong case for using this type of model for video encoding in the 'cloud', but only show how their 'Split and Merge' architecture performs with video that does not have temporal compression – commonly found in most videos to achieve a reasonable reduction in file size. Garcia et al (2010) also use *MapReduce* to convert video from a DVD, in real time, using a user requested quality setting, for live streaming to their device. Their implementation, however, requires that the video be initially converted into a segmented format (namely MPEG transport stream segments) that is then processed independently, and they target a specific type of output (HTTP Live Streaming) that also leaves the output in segments.

There are many different implementations of the *MapReduce* paradigm – *Hadoop* is the open-source favourite, with the best support and feature set. "[Apache] Hadoop Map/Reduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner"[4]. *Hadoop* implements the *MapReduce* model, and supports several languages to directly interface with its API (namely Java, among others). *Hadoop* is available to run wherever Linux does, and several pre-built distributions exist for its use (Yahoo Hadoop[5] and Cloudera[6]). On-going research exists looking to improve areas of *Hadoop*, for example on distribution and load balancing (Zaharia et al, 2008). This makes the use of the *MapReduce* model in *Hadoop* an excellent way of taking advantage of these improvements as they occur.

B. *Distributed/Parallel Video Processing*

Parallel video processing has been studied extensively over the past 10 years. Shen and Delp (1995) identify the two primary strategies: spatial and temporal parallelism. Spatial parallelism partitions a single frame of the video into pieces, allowing for the various stages of the encoding of each frame to be distributed among processors. Temporal parallelism

[3] http://www.hpl.hp.com/open_innovation/cloud_collaboration/cloud_demo_transcript.html [last accessed: 19/06/11]
[4] http://hadoop.apache.org/common/docs/current/mapred_tutorial.html#Overview [last accessed: 18/04/12]
[5] http://developer.yahoo.com/hadoop/distribution/ [last accessed: 18/06/11]
[6] http://www.cloudera.com/ [last accessed: 18/06/11]

partitions the video into groups of frames that can then be processed by a sequential encoder on the various processors (or nodes in a cluster). Spatial parallelism allows for high quality coding that can be identical to the output of a sequential encoder, but suffers in terms of scalability. On the other hand, temporal parallelism scales very effectively but can produce sub-par quality results if the segments of video are too small. Combinations of the two are also possible, breaking up the video temporally and then using spatial parallelism on the sequence of frames (Rodriguez et al, 2006).

We focus primarily on temporal parallelism for use with *MapReduce*. Any form of spatial parallelism requires a large amount of communication between threads of execution, at high speed, making it suited for shared memory and/or multi-processor environments rather than a network of distributed machines. Temporal parallelism gives us independent segments of work that can then be distributed easily with a small communication overhead.

Pereira et al (2010) highlight that one of the key problem areas when splitting the input is temporal compression. Many of the codecs that we want to transcode to and from use temporal compression to remove redundant data. When using temporal compression, rather than simply compressing each frame (essentially image compression, usually based on the JPEG format) the codec uses references to previous and future frames to make up the image. This takes advantage of the small comparative differences between frames due to the nature of their use (e.g. the camera not moving). When required, or at a set interval, the encoder places a 'key-frame' in the stream; a frame that does not reference any other and holds all of the required information to make up the image (Richardson, 2002). Other frames can then reference this key-frame to get the extra information they need. The exact format of the temporal compression needs to be understood by any distributed video encoding system, so that it does not split the video at a point that would leave frames unable to reference data to make up the images. As noted in the previous section, Pereira et al (2010) do not address the temporal compression problem, rather they show how video that is compressed frame-by-frame can be broken up and distributed for processing among machines. We address this key issue in our implementation, providing a method of breaking up each of the video and audio streams in the file into chunks, without affecting the temporal compression structure.

Schmidt and Rella (2012) further investigate how the size and structure of this split can affect the workload distribution in a cluster. They propose an algorithm that attempts to select a number of frames that will take a similar amount of processing time for each segment. Implementing this was deemed outside of the scope of this work, and we chose to simply break the segments up by using a target chunk size in bytes, taking advantage of our segments being of a size that is a multiple of the underlying distributed file system, improving data locality (See – III. B. Demultiplexing and Chunking).

IV. SOLUTION

Our solution for transcoding video and audio using *MapReduce* is presented in this section. To begin, an overview of the design of the solution is given and how this fits conceptually with the *MapReduce* paradigm. We then describe the process of splitting the input data for parallel processing, and detail several issues that need to be addressed to make this process robust. Finally, we present the architecture for running a *MapReduce* cluster in 'The Cloud' using Amazon Web Services, and describe its workflow.

A. *High-level MapReduce Process*

Figure 4 describes our overall video transcoding process conceptually 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 pre- and post-processing operations.

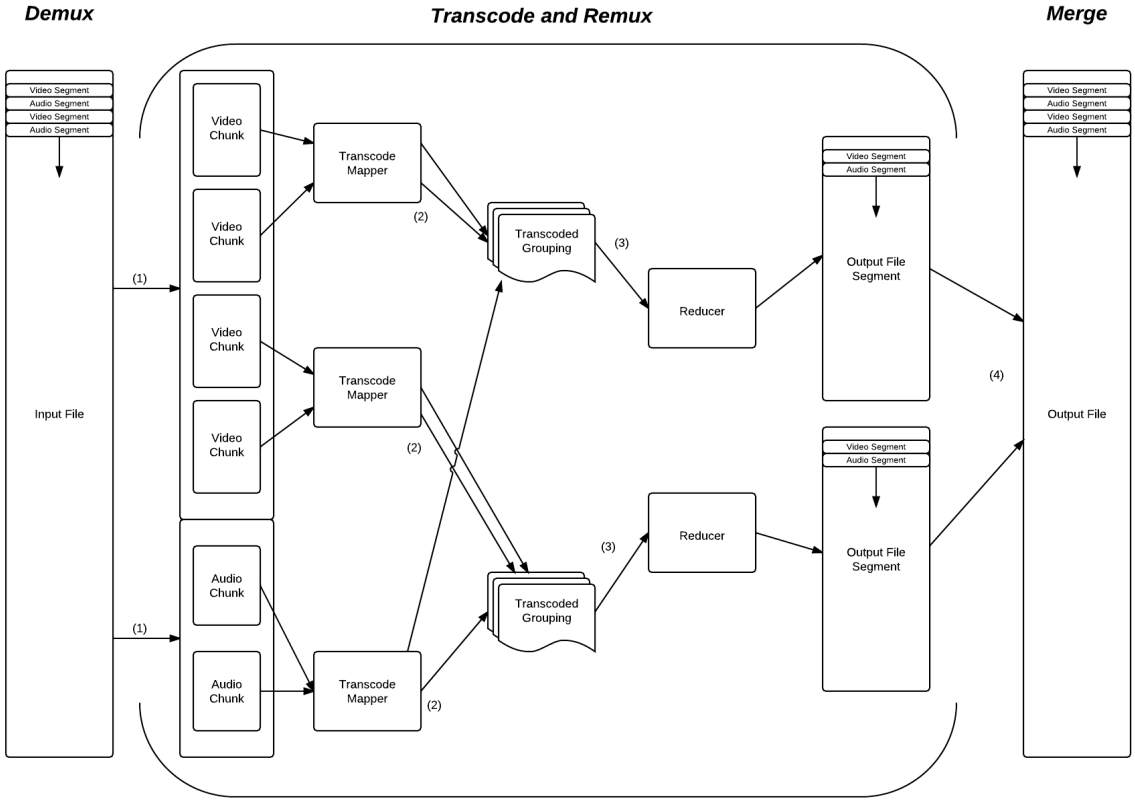

Figure 4. Conceptual overview diagram of our *MapReduce* program for transcoding.

1. **Demux:** The input file is broken down into chunks of data, per stream, ready for the *MapReduce* operation. This process is called the *Demux* phase.
2. **Transcode:** 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.
3. **Remux:** These output chunks are then grouped and partitioned according to the number of available *Reducers*. The grouping is based on the key that was associated with the output from the mapper. This key is the timestamp of the chunk for a given stream. This means that the grouped input to the *reducer* can then be used to perform the reverse of the *Demux* phase, interleaving each of the streams for a particular timestamp into a given output container. Because several of these *reducers* are run, the output is in segments, equal to the number of *reducers*.
4. **Merge:** 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.

B. *Demultiplexing and Chunking*

1) Atomic Processing Units

At the core of any distributed processing is the method by which the input is separated into Atomic Processing Units (APUs). We call this process the *Demux* (short for De-multiplexing), and it is the first phase of the process of transcoding a video file. This process separates input file into its constituent streams of audio and video, and into APUs 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 the APU so that it has all of the information it needs when a *Mapper* is processing it. Once the APUs are defined and sent to a *Mapper* there is no way for them to get any additional information.

A 'chunk' target size is defined that is used to decide when an attempt to split the input data should occur. This target size needs to provide a balance between output visual quality and scalability. Having small processing units can improve scalability across a cluster in some circumstances, but reduces the amount of context the encoder has to maximise compression and visual quality. 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 which we call key-frames. In a video stream, these represent points in the video where an 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 the structure of the file. Most input formats call these key-frames 'Intra-frames', but their nomenclature varies. 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' target size that was defined, it is emptied to create a new chunk. When using video we empty the buffer up until (but not including) the most recent key-frame, ensuring that the GOP structure (the 'Group Of Pictures' that lie between two key frames) is kept together. When using audio, we simply empty the buffer entirely, as it can be broken up at any point.

A header is then added to the 'chunk' data, that stores information about the stream that the 'chunk' came from, and what the output format should be when this 'chunk' is transcoded. Figure 5 shows how a chunk of data (the value) is represented along side its key.

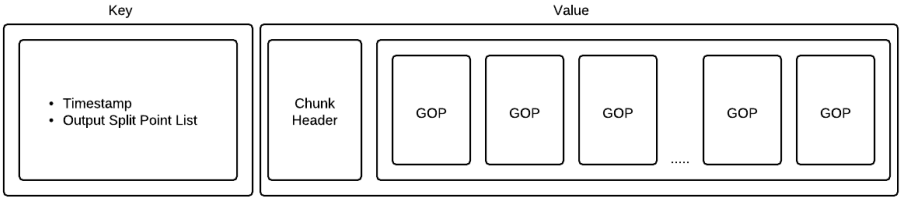

Figure 5. Layout of the chunk key-value data for a video stream.

The data is output using a Hadoop (the platform we are using for MapReduce) 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 (the timestamp), and where it needs to be split further during the encode process. When grouping these chunks before the *Reduce*, we the timestamp as our key.

2) Split Points

When each of the chunks is processed in the *Map* phase, it represents a portion of time for a particular stream, be that video or audio. The length of time may not be equal between chunks, as the data size of the chunk does not correspond to the length of time it represents. This can be seen in Figure 6 in the input chunks to the *Map* function. In the example in Figure

6, for simplicity, the video uses twice as much data to encode the same amount of time as audio. Because we break up the chunks by data size, we have twice as many video chunks as audio.

To successfully *Remux* (merge the audio and video streams back together again), we need to have chunks of equal time to pass to the *Reduce* function. To do this, we define 'split points' in each of the chunks input to the *Map* function, that describe where the output should be split. When we partition each of these output chunks from the *Map* phase we group them by their key and can be sure the length of time of each of the chunks is the same. Figure 6 shows the flow of data for this process.

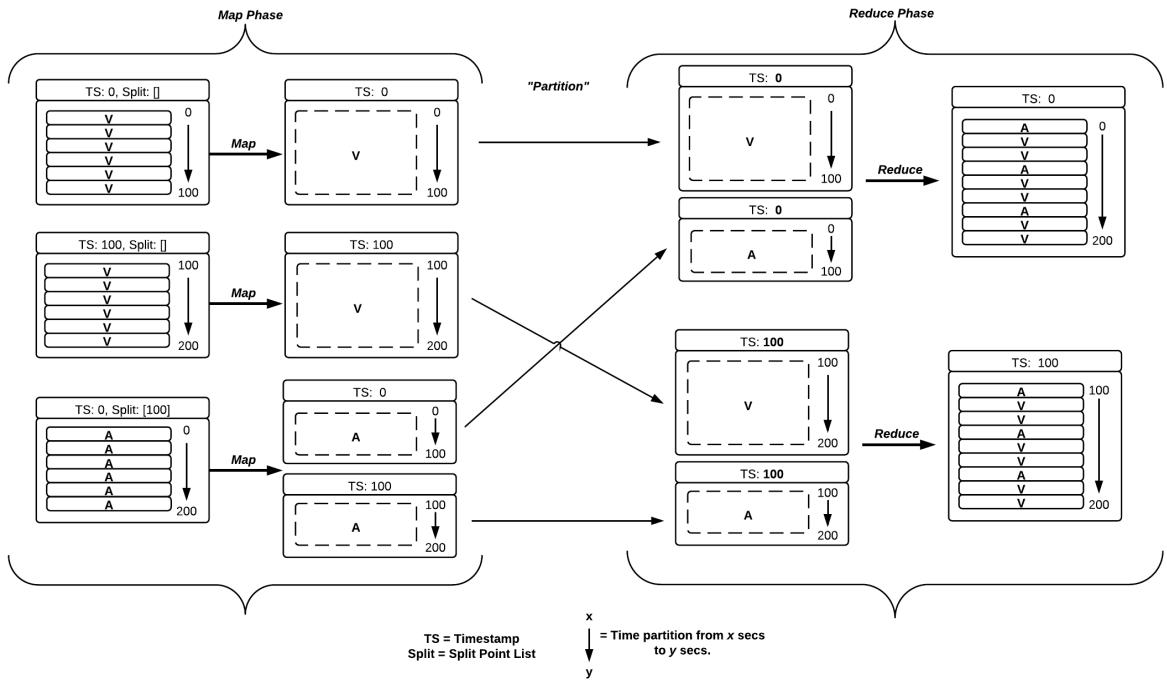

Figure 6. Data flow through MapReduce showing time partitioning

Calculating where these split points need to occur 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.

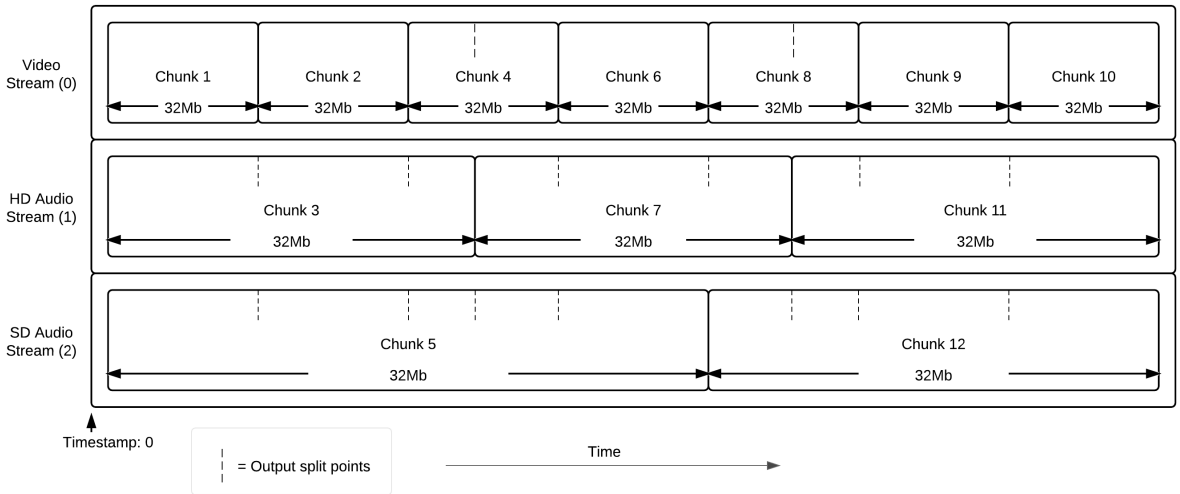

Figure 7. Chunks of data inside of streams with respect to time

Figure 7 shows an example of the different streams that might be found in a given video file, with respect to time. Each chunk has approximately the same data size, but the amount of time it represents differs greatly depending on the stream. In our example, the video stream has the highest number of chunks, as more data is required to store it than audio.

As we generate each of the chunks of data by reading the packets out of the file, we look back through all of the previous chunks we have output and calculate the split points. Each of the end timestamps of previous chunks are marked in the key of the chunk we are currently processing (See Figure 5). This order is shown in Figure 7, where the split points are always from previous chunks. This provides the *Mapper* with all split points, and the *Reducer* with groupings that are complete and are ready to be interleaved. This process relies on the premise that the timestamp for every stream is always increasing as we progress through the file, irrespective of the actual stream, and that the timestamps from different streams never overlap.

3) *Stream Timestamp Monotonicity*

Whilst it is true that a correctly interleaved file, with highly accurate timestamps, will have non-decreasing timestamps across all of its streams; it is not required that this is the case for the file to be playable. When playback occurs, each of the streams is buffered and played back separately whilst being re-synchronized occasionally by the player. This allows for the frame order to be different to the timestamps and file play perfectly.

If we assume that the file has non-decreasing timestamps across all of its streams, when we mark each of our chunks with its split points, we run the risk of missing a split point that we were not aware of that simply occurs later in the file. This is as we are assuming that we have seen all of the data up until the most recently seen timestamp. To counter this issue, we keep a fixed number of chunks in a buffer before output, and also keep a variable number of chunks in a queue to improve read performance.

By doing this, we can check as we write each chunk to the queue, that the end timestamp of this new chunk is not missing as a split point from any of the yet-to-be-written chunks that are still in memory, and, if they are, change them accordingly. We found that a fixed buffer of 4 chunks fixed all of the monotonicity issues for our test files, and that a maximum queue size of 5 resulted in a read performance increase, as well as a longer output buffer for changes, especially when the data was being streamed over the network.

C. *System Architecture and 'The Cloud'*

1) *Hadoop and HDFS*

Apache *Hadoop*[7] was chosen as our implementation of *MapReduce*. As described in the original *MapReduce* paper, it also contains an implementation of a distributed file system, called HDFS (*Hadoop* Distributed File System). This is key to understanding how each of the *map* and *reduce* tasks run, and how they allocated resources.

When the data is initially copied onto the cluster, a block size is specified, much like on a traditional file system. However, this block size is much larger, usually around 32-128 MiB, and it is used to determine when to split the input and commit the block. Each block is randomly committed to different nodes in HDFS, and optionally replicated to other nodes for redundancy. The result is that the input file is spread almost evenly across all of the machines in the cluster.

When the data is processed on the cluster using *MapReduce* each of the APUs in a given block is processed as part of a *Map Task*. Each *Map Task* may run the *Map* function several times, once for each APU that it finds inside of the block. A fixed number of *Map Tasks* run on one machine at any given time, each taking up a *Map Task Slot* on the machine while they run. A *Map Task Slot* represents a multi-dimensional resource slice, consisting of CPU, memory and hard disk on a given machine in the cluster, and a fixed number are allocated prior to execution.

[7] http://hadoop.apache.org/ [last accessed: 17/04/12]

This method of breaking up the input is the fundamental means by which *MapReduce* achieves its parallelism automatically. In the *Demux* phase we aim to split our input video/audio into 'chunks' that are themselves a single APU and are as close as possible to the size of the blocks that HDFS is using to store and distribute the data. This makes each *Map Task* in our scenario nearly always a single invocation of the *Map* function, a single 'chunk' and single APU. This reduces the overhead in each *Map Task*, and also makes the 'chunks' as large as possible, helping improve the visual quality of the output.

2) Amazon Cloud Architecture

In order to run a *Hadoop* cluster for testing with access to the computational resources required for video transcoding, we designed and implemented a system for operating the cluster in the 'Cloud' using Amazon Web Services (AWS). Amazon Elastic *MapReduce* (EMR) is a service that coordinates the setup and configuration of a Hadoop cluster of a given size, allocating the correct number of computers and charging for the cluster on a pay-as-you-go basis. Data is stored using Amazon's Simple Storage Service (S3), where transfer to and from the cluster, keeping inside of the AWS data centre, is free-of-charge and data storage costs are charged per gigabyte-month.

A 'Transcode Job' is defined as the process where a cluster transcodes one or more input files. We define a 'Transcode Job Definition file' that is stored on S3 and describes the settings for each of the transcodes that take place as part of a job.

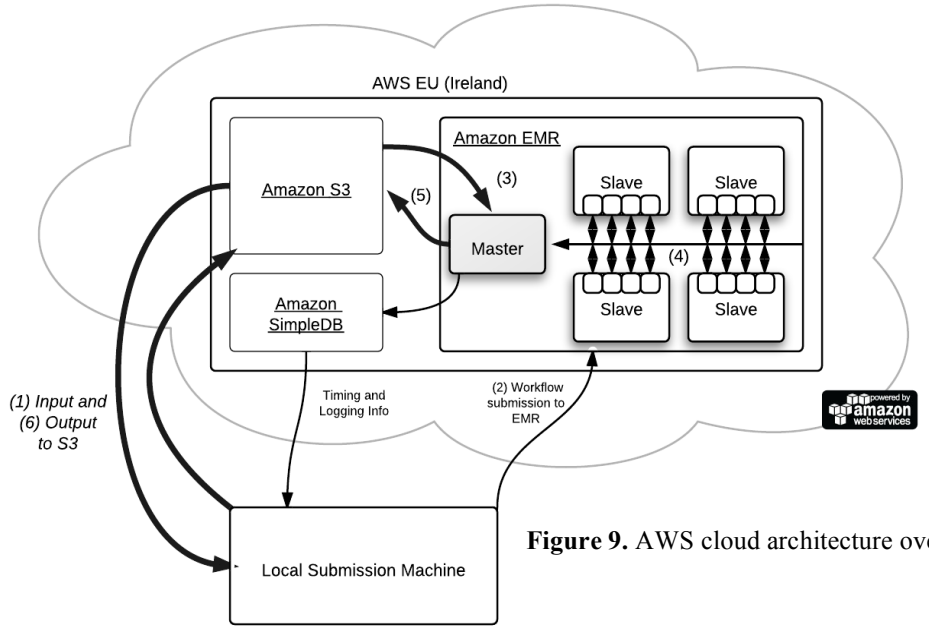

Figure 9. AWS cloud architecture overview

The workflow for a 'Transcode Job' using AWS is as follows and can be traced through Figure 9:

1. The file(s) to be transcoded are uploaded to S3 from the submission machine using one or more HTTP connections.
2. The workflow is submitted to EMR, specifying the processing parameters for the transcode job through the use of a transcode job definition file that is stored and uploaded to S3.
3. Each transcode operation is then processed from the definition file. The master node then copies the file from S3, and the *Demux* process splits the file into chunks, distributing it randomly amongst each of the slave nodes.
4. A *Hadoop* job is then submitted to the cluster's 'JobTracker', the process in *Hadoop* that coordinates the allocation of tasks. Each slave begins processing the *Map Tasks*

that it gets assigned. The 'JobTracker' assigns the *Map Tasks* to the slaves that are storing the data locally first, and the random distribution of blocks in HDFS means that nearly all slaves are able to process local data. Once all of the *Map Tasks* are complete, the *Reduce* phase runs, transferring the intermediate data from the appropriate nodes in the cluster and storing the output in the HDFS.

5. The *Merge* operation then begins on the Master node, taking each of the segments of output from the *Reduce* phase from the distributed file system, and merging them into a final file that it then copies onto S3 for persistent storage.
6. The final output is then copied from S3 back onto the machine that submitted it (or left on S3 and served to clients on the web directly). We then repeat from step 3 until all the files in the job definition file are processed.

Throughout the execution of a 'Transcode Job' the Master node on the cluster and each of the *Map Tasks* that are running on the slave nodes report their progress back to a key-value database system provided by Amazon called SimpleDB. We record the progress and the log output of each *Map* execution so that we can give an overall progress of the job as it executes, and inspect the logs if needed. We also record the time that each of the phases begins for each file being processed, and we use this information to analyse the performance of the cluster overall. All of this information is read directly out of SimpleDB by the machine that submits the job, and can be read incrementally as it is added.

Figure 9 also shows the *Map Task Slots* running on each slave (denoted as small squares inside of the slaves), a total of 4 per machine. Each of these slots runs a 'TaskTracker' and it is this 'TaskTracker' that communicates with either the underlying HDFS instances on the local machine, or that of a remote machine to get the data for input to the *Map Task* it is currently running. It should also be noted that *Hadoop* manages the retry of *Map Tasks* that fail, and also the retry of any nodes that fail, providing a limited amount of redundancy to independent failures in the cluster.

3) Map Task Resource Allocation

Each of the slave machines running in the cluster are Amazon's High-CPU Extra Large instances, with 7Gb of memory and 8x 2.33Ghz cores (more specifically two quad core Xeon E5410 processors). This makes them ideal for video transcoding, and when combined in a cluster, a formidable amount of computational power (~1.459 teraFLOPS, for a 19 node cluster). Whilst some of the memory in the physical machine is lost to the virtualisation system Amazon use to implement the Elastic Compute system (a modified version of Xen[8]), very little CPU power is lost, as the majority of the instructions operate directly on the underlying hardware.

This would appear to make it ideal to have *Hadoop* allocate 8 *Map Task Slots*, and 8 *Reduce Task Slots* to each machine, essentially forcing the CPU affinity of each task to one of the given processing cores. However, after some initial benchmarks, it became clear that for high-resolution files there was not enough memory available for 8 *Map*/*Reduce* Tasks, HDFS, the TaskTracker system and the Linux kernel on each machine. *Map Tasks* failed frequently and the Linux kernel was forced to kill high memory usage tasks, making the cluster highly unstable. In response to this issue, 4 *Map Task Slots* were allocated to each machine and the video encoder was set to enable multi-threading. After further benchmarking with different numbers of threads, 4 low priority threads per encoder were allocated, over subscribing the number of cores (32 threads vs. 8 cores). The side-effect of this is that in return for some overhead from the process scheduler in the kernel, we are able to use all of the processing power of the machine, even when one or more of the *Map Tasks* running on the machine can only use a single processing core (such as audio transcoding). It also brought down the overall

[8] http://www.xen.org/ [last accessed: 12/04/12]

execution time of *Map Tasks* for large resolution files, which helped to improve the performance of the cluster overall (this effect is explained further in our evaluation section).

D. *Solution Development*

1) *Implementation Tools and Languages*

Apache *Hadoop* accepts *Map* and *Reduce* functions in the form of Java classes. However, Java does not have the support built-in or an external library with functionality to encode video with the performance characteristics that are required to be able to outperform sequential transcoders that use assembly optimisations extensively. FFmpeg[9] 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. We used the Java Native Interface (JNI) to implement the required interoperability layer so that FFmpeg could be used for our transcoder and to break up the video into chunks and merge them into a playable file. Serialisation of the input data into chunks for the *Map* phase and intermediate chunks for the *Reduce* phase uses a C serialisation library called TPL[10]. We define TPL structures that mimic the internal C structs of the FFmpeg libraries to store the video/audio data and stream information.

2) *Building and Testing*

The project was developed using a mixture of C++ and Java using Xcode and Eclipse respectively, primarily on Mac OS. The native libraries were complied for Mac OS X and Linux 2.6, linking dynamically against the FFmpeg components. The Linux libraries were complied on-demand as the cluster was launched using a series of shell scripts. git[11] was used for version control throughout the project, committing frequently and keeping a log of changes as development progressed. The *Map* and *Reduce* functions were tested independently using a mock Hadoop testing framework as part of JUnit. The transcoding components were tested with small console applications that called their respective functions without the use of Java. Each phase of the overall transcode process was then tested separately to ensure its output was correct, and then finally the overall system was brought together and tested in the cloud using Hadoop. The verification and validation of this output, and the performance of these tests form the basis of our results and evaluation.

Development of the project took place incrementally, passing through several phases after which each could be tested:

1. Initial sample prototypes in C++ interfacing with FFmpeg
2. JNI boundary 'boiler-plate' code
3. Demux system
4. Map function
5. Reduce function
6. Map and Reduce job definition
7. Transcode Job submission API for EMR.

[9] http://www.ffmpeg.org/general.html#SEC6 [last accessed: 21/01/12]
[10] http://tpl.sourceforge.net/ [last accessed: 17/04/12]
[11] http://git-scm.com/ [last accessed: 17/04/12]

VI. RESULTS

We performed several experiments focusing on three key areas; parallel scalability, output quality, and compression. For each of these areas we used a common set of test input files chosen specifically to provide us with a range of input formats, quality settings and sizes. Table 1 lists these input files and their attributes in detail. The output in each of the tests is H.264 video, and AAC audio, in a Matroska[12] container. Every audio and video stream is processed and present in the output; only the data streams are skipped (e.g. chapter markings and subtitles). We present their resolution in pixels/sec to simplify the comparison of actual video content stored per second, as the test cases have differing frame rates.

TABLE 1. TEST INPUT FILES.

#	File Size[13]	Video pixels/sec	Audio samples/sec	Length (sec)	Approx. bit rate	Type (Video/Audio)
1	4.6MB	2.4M pixels/sec x 4[14]	64Khz	34	1090 kbit/s	WMV/WMA
2	183MB	5.76M pixels/sec	96Khz	1308	1121 kbit/s	MPEG4/MP3
3	237MB	3.84M pixels/sec	96Khz	2745	691 kbit/s	H.264/AAC
4	364MB	7.92M pixels/sec	96Khz	2607	1119 kbit/s	MPEG4/MP3
5	798MB	8.16M pixels/sec	96Khz x 6 streams[15]	2653	2405 kbit/s	H.264/AAC+AC3
6	1.54GB	15.6M pixels/sec ('HD')	88.2Khz	2470	4988 kbit/s	H.264/AAC
7	6.32GB	16.8M pixels/sec ('HD')	96Khz and 288Khz[16]	8284	6103 kbit/s	H.264/AAC+AC3
8	10.52GB	49.9M pixels/sec ('HD')	288Khz	9701	8670 kbit/s	H.264/HEAAC

Each of the input files, when broken up for parallel processing in the *Demux* phase, creates a different number of *Map Tasks.* Table 2 presents the number of *Map Tasks* per file, and the target block size that we chose to help increase their scalability. The block size settings in Table 2 are used throughout all of the experiments. This block size dictates the approximate size of each 'chunk' that the *Demux* phase produces for processing, and also the size of each block in HDFS. Each block is distributed across the cluster of machines. We chose these block sizes based on a performance test on a subset of our cluster size range, using 2, 4 and 6 machines, varying the block size between 4MiB and 32MiB.

TABLE 2. CHOSEN TEST INPUT FILE BLOCK SIZES.

#	File Size	Block Size	Number of Map Tasks
1	4.6MB	4 MiB	1
2	183MB	8 MiB	21
3	237MB	8 MiB	28
4	364MB	8 MiB	43
5	798MB	8 MiB	95
6	1.54GB	16 MiB	91
7	6.32GB	16 MiB	376
8	10.52GB	24 MiB	417

A. *Scalability Results*

Our most extensive test was of the performance of the *MapReduce* solution across a varying number of machines. This was to ensure that our solution could scale properly, and to analyse the effect adding machines to the cluster has on the overall performance increase.

[12] http://matroska.org/ [last accessed: 26/04/12]
[13] All file size units are presented using SI decimal prefixes, e.g. MB = 1,000,000 bytes (not 1,048,576 bytes).
[14] Test file 1 has multiple video streams, effectively increasing its length without adding extra audio.
[15] Test file 5 has more audio than the other files to test performance when many audio streams are present.
[16] Test file 7 contains a 6-channel (5.1 Dolby Digital) stream at a sample rate of 48Khz per channel.

Figure 10 presents the performance increase that is gained over using a single machine for each of our test files, using cluster sizes ranging from 1 to 19 (and so with 4 *Map Task Slots* per machine, 4 to 76 *Map Task Slots*). We processed each of test files in three rounds to produce average performance values for the varying cluster sizes. To reduce the overall cost of the experiment, only odd numbered cluster sizes were tested in each round. We denote 'cluster size' to be the number of slave machines in the cluster. The 'speed up' is with reference to the average time it took to process the file on a machine with a single slave, shown on the graphs below in Figure 10. This does not include the time for *Demux* or *Merge*, concentrating on the scalability of the distributed phases of our solution.

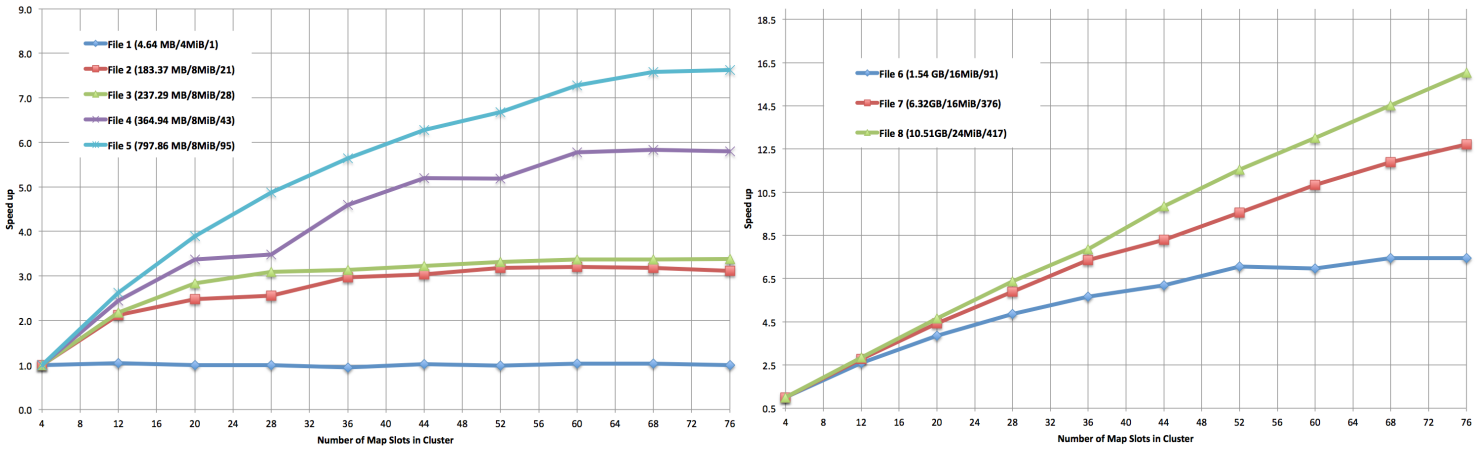

Figure 10. Performance increase over a single machine as the number of *Map Task Slots* in the cluster increases. N.B. *The y-axis scale differs between the graphs.*

It is clear from Figure 10 that the scalability of our solution varied depending on the input size and type. In the three larger test cases presented in the graph on the right, File 8 was able to achieve a performance increase of 16.4x that of a single node in the cluster, when using 19 machines. In the graph on the left, it can be seen that the smaller files were limited by their number of *Map Tasks,* achieving smaller performance increases.

B. *Visual Quality (PSNR)*

To analyse the visual quality of the output from the transcoder, we chose to use a common objective visual quality measurement called PSNR (Peak Signal to Noise Ratio, a logarithmic representation of Mean Squared Error). PSNR compares the output from a transcoder to a reference file to give a measurement of the quality lost during the transcode. As we are only interested in the difference between the output from our *MapReduce* transcode implementation, and that of our reference sequential transcoder[17], we focus on the difference between the two PSNR values. Taking PSNR measurements compares every output frame with every original frame, producing a large amount of data per test file. We present the percentage of the output that produces frames of lower quality, the same quality and better quality, giving a succinct indicator of difference between the sequential and parallel transcoders. We also include the difference between the two average PSNR values for the reference and *MapReduce* output, giving us an indicator of whether the overall quality has been significantly affected. The standard deviation for the PSNR differences is also presented to make it clear if the differences vary wildly or are close to the average.

We took measurements using two common types of transcode, Constant Rate Factor (CRF) and Average Bitrate (AB) to ensure that we assessed the quality for both output. CRF aims to maintain a given visual quality throughout, whereas AB aims to target a particular output bitrate (and so file size) for the file overall. AB is important for files that are streamed via bandwidth-constrained connections, where a large change in bitrate for a scene that

[17] The transcoder used was the FFmpeg command line transcoder, found at: http://ffmpeg.org/about.html [last accessed: 16/04/12]

requires more data to maintain its quality (as happens in CRF) would prevent the video from playing back smoothly. CRF is mainly used for files that are on local secondary storage, such as downloads from online video stores.

In Table 3 positive differences imply an improvement in visual quality over the reference file and negative differences imply degradation.

TABLE 3. PSNR DIFFERENCE VALUES FOR CRF AND AVERAGE BITRATE ENCODINGS

#	Input Size	CRF PSNR Difference Counts			AB PSNR Difference Counts			Average & Standard Deviation CRF PSNR Difference		Average & Standard Deviation AB PSNR Difference	
		Lower	Same	Higher	Lower	Same	Higher	Average	Std. Dev.	Average	Std. Dev.
1	4.6MB		*N/A*[18]			*N/A*[18]		*N/A*[18]		*N/A*[18]	
2	183MB	15%	69%	15%	52%	1%	47%	0.0011 dB	0.0850 dB	-0.0651 dB	0.8877 dB
3	237MB	42%	18%	40%	45%	0%	54%	0.8878 dB	0.2718 dB	-0.0006 dB	0.9693 dB
4	364MB	29%	45%	26%	43%	0%	57%	-0.0972 dB	1.0717 dB	0.0345 dB	1.5434 dB
5	798MB	27%	46%	26%	44%	0%	56%	0.0008 dB	0.2504 dB	0.1986 dB	1.3777 dB
6	1.54GB	40%	15%	44%	47%	0%	52%	0.0307 dB	0.8156 dB	1.1796 dB	4.5603 dB
7	6.32GB	33%	35%	32%	41%	0%	59%	-0.0198 dB	1.3605 dB	0.7466 dB	3.3237 dB
8	10.52GB	34%	38%	28%	44%	0%	56%	-0.0185 dB	0.2992 dB	0.0385 dB	1.9680 dB
							Mean Values:	0.1121 dB	0.5935 dB	0.3046 dB	2.0900 dB

When the encoder is using CRF, it is clear that the average quality produced is not affected greatly by our method of 'chunking' the input. The average PSNR difference in all but one of the files is less than 0.05 dB. In some examples (especially File 2) the average quality is greater than that of the sequential transcoder for CRF. It should be noted however, that the differences in PSNR for CRF occurred largely around the chunk boundaries. In all of the CRF tests a sizable percentage of the frames had identical quality to that of the reference transcoder. The standard deviation of the PSNR differences for CRF was low, indicating that most of the differences were close to the mean and a low number of frames had a greatly improved or degraded quality.

When encoding using AB, it is also clear that the average quality produced is not affected greatly by our 'chunking' of the input. However, unlike CRF, nearly every frame was different to that of the reference transcoder, resulting greater standard deviation and overall visual quality degradation. Figure 11 shows an example of a sample of 10,000 frames from Test file 4 when encoding using AB and CRF. It shows that using CRF produces better visual quality than AB, resulting in many frames that have no difference to the reference transcoder.

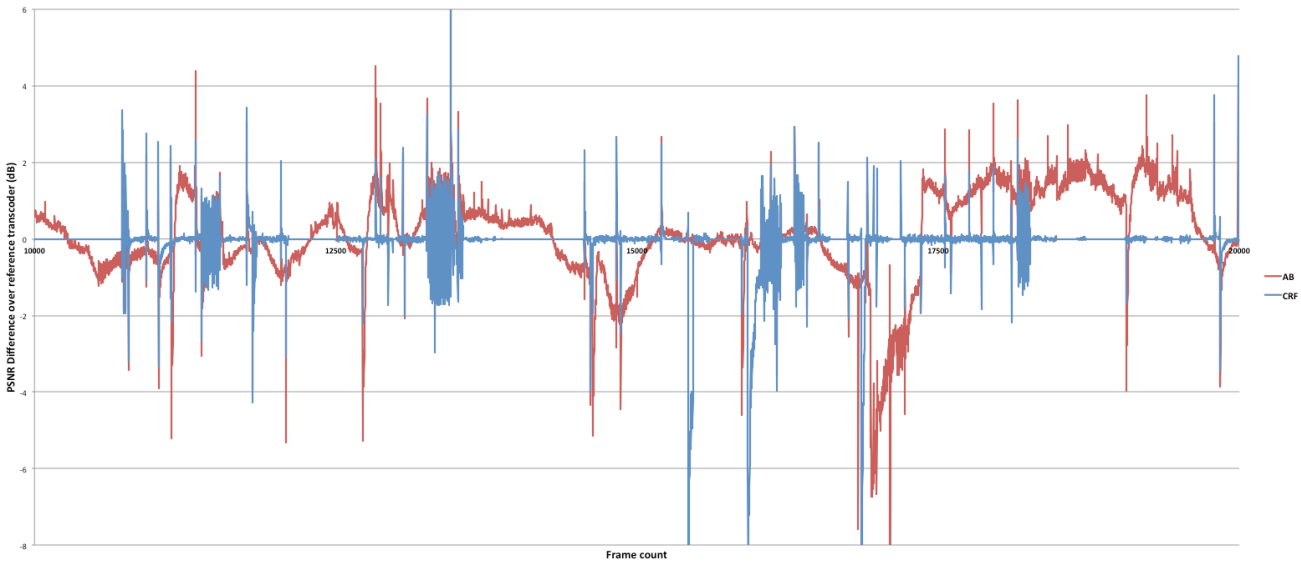

Figure 11. PSNR difference over reference transcoder for a 10,000 frame sample in Test file 4.

[18] As Test file 1 contains multiple video streams, it is excluded from this test. Our *MapReduce* implementation does not support multiple output target average bitrates for different video streams, and so it would be unfair to include this test file.

C. *User Study Results*

To ensure that the quality difference we found during a visual quality analysis was not perceptible to the viewer we conducted a user study. The candidates compared 2 sets of 2 videos; each set comparing the output from our solution with the output from a reference sequential transcoder. The first set was a video encoded using CRF, and the second set was a video encoded using AB. For each comparison we asked participants which video, if any, they preferred with respect to colour, visual quality, audio quality, audio/video synchronisation, and overall preference. The results from the trial produced 2 scores for each pair of video clips, one for each video clip in the set. Each score corresponded to a preference the user gave to a given property of the clip, adding 1 to the score for each, with a maximum of 5. Each candidate was presented with the clips in a random order for each pair, and the order of the video sets was randomised. This ensured that the order the videos were presented in had no unexpected effects on the results. Each random order from the 8 possible permutations was performed twice, totalling 16 candidates.

By taking an average of these scores across all of the candidates, we are able to obtain enough data to perform a two-tailed T-test. The T-test shows us that there is no significant difference between the mean scores for either video clip, for either of the test videos trialled. We can therefore be confident that the output from our solution is not perceptibly different from the output of the reference sequential transcoder. Table 4 presents the mean and standard deviation scores for each of the clips, and the T-value for the two-tailed T-tests.

TABLE 4. USER STUDY RESULTS

	Video 1 (CRF) MapReduce	**Video 1 (CRF) Reference**	**Video 2 (AB) MapReduce**	**Video 2 (AB) Reference**
Mean:	0.69	0.56	0.88	1.31
Standard Deviation:	1.14	0.96	1.15	1.45
T-value:	0.34		-0.95	

D. *Output File Size Comparisons*

To evaluate the effect that breaking up the transcode operation has on the compression of the video, we compared the output files sizes of our *MapReduce* transcoder with that of the reference transcoder for both CRF and AB encoding, using the same output data as the visual quality tests.

In table 5, positive differences imply an increase in file size over the reference, conversely negative values imply a decrease.

TABLE 5. OUTPUT FILE SIZES FOR VARYING TRANSCODE SETTINGS

#	**Input Size**	**CRF File Size % Difference**	**Average Bitrate % File Size Difference**
1	4.6MB	-1.8%	N/A
2	183MB	0.2%	0.9%
3	237MB	-6.2%	0.7%
4	364MB	-2.5%	1.1%
5	798MB	-11.4%	0.5%
6	1.54GB	-0.8%	-1.6%
7	6.32GB	5.8%	-0.1%
8	10.52GB	0.0%	-0.6%

As we can see from the Table 5, the AB file compression was not affected by our input segmentation by any more than 1.6%. CRF, however, was affected more substantially, with a maximum of 11.4%.

VII. EVALUATION

Our original research question for this project was, 'Can the *MapReduce* paradigm be used for video transcoding in an effective manner, so that it scales well across machines, has comparable output quality to current sequential encoders, and improves performance overall?'

We will now evaluate each aspect of this question, and comment on the overall suitability of *MapReduce* for video transcoding based on our performance results and evaluation of the output.

A. *Solution Strengths and Limitations*

1) *Can the solution scale over many machines effectively?*

Making use of the cluster effectively is an important part of any distributed system, especially one that uses pay-as-go IaaS service, such as ours. Ensuring that when further machines are added to the cluster that their extra computational power is taken advantage of, is important to our solution being useful in real world scenarios, which often have various elastic loads.

Our scalability tests (Figure 10) clearly showed that files smaller than the block size do not gain any performance increases (e.g. test file 1). This is because with only one block to process, only one *Map Task* is created and so the capacity of the cluster is always under-utilised. Test files 2, 3 and 4 clearly stop gaining any large performance increase after the number of *Map Task Slots* in the cluster exceeds the number of *Map Tasks* they have to process (21, 28 and 43 respectively). Files 5 and 6 scaled well initially and always had enough *Map Tasks* to fully utilise the cluster, although their rate of increase in performance decreased, as in the other test files. In 5 and 6, there was no increase in performance between 17 and 19 machines, highlighting the extremes of their scalability. Our two largest test files scale best overall. 7 and 8 achieve constant performance increases through all of the tests, and look to be able to scale further as cluster size is increased, beyond our maximum of 19.

None of our test files showed an increase i5 and 6 n performance equal to the number of machines in the cluster. This is because of various overheads and inefficiencies in *Hadoop* and the *MapReduce* process. The believe the two main contributory factors are:

Task Overhead: Every *Map Task* that runs incurs an overhead in start-up and shutdown time, and numerous small tasks cause this overhead to take up more of the execution time per *Map Task*.

"Straggler Tasks": The last 'wave' of *Map Tasks* to run contains many tasks of different execution times, but the total time it dictated by the slowest *Map Task* in the wave instead of utilising the entire cluster efficiently. This execution time difference could be reduced through better choice of the size of APUs in each block, at the sacrifice of some data not being local to every machine. Making the *Map Tasks* smaller at the sacrifice of visual quality could also reduce the disparity, but increase start-up/shutdown overhead.

It appears that the best use case for scalability for our implementation is for large files (e.g. 7 and 8), which split well, and have medium-to-high pixels/sec to process during each *Map Task*.

Although there is a best-case scenario for transcoding video using *MapReduce*, our scalability results succinctly show that the system does work well for most non-trivial tasks, where a performance improvement would be desirable. In our tests, we did not allow for multiple jobs to be submitted to the cluster at once and we did not allow for any sort of dynamic resizing of the cluster. This was to ensure the results were comparable and fair. If the implementation was to be used in a real world scenario, the cluster utilisation could be kept high by running multiple transcode jobs at a time and adding more machines as needed. This makes the *MapReduce* model much more flexible in a cluster scenario than simply allocating a single input file to each machine in a cluster.

2) Does the 'chunking' degrade the visual quality of the encoding?

It is clear from the PSNR data in Table 3 and Figure 11 that the output quality from our solution differs from the reference in both the CRF and AB encoding types. We compared an example CRF setting, and an example AB setting to give us an idea of how much effect the chunking of the input has on quality.

The AB algorithm relies on information about previously encoded data to make optimisations to quality without affecting the average output bitrate. As it only has a record of the frames it has previously encoded in the current chunk, and not the whole file, it cannot make these optimisations as well as in the reference transcoder. This results in AB's output frame quality varying much more wildly than that of CRF, as Figure 11 shows. CRF relies on the ability to 'look ahead' into the input data before making a decision about frame quality, but it seems to be less sensitive to being parallelised.

Whilst the output quality did differ, the user study confirmed that it was difficult for a human to perceive the difference overall. Our study aimed to see if a human could detect the difference between our encoded videos and their associated reference video, as we know there is a difference from the PSNR analysis. The focus of the study was on whether participants could perceive sizable differences in a normal viewing situation, rather than examining the video carefully to notice minor differences. To this extent, participants were presented each of the clips only once and were unable to compare the clips 'side-by-side'. The results showed participants were unable to notice any significant differences between the videos. Many of the candidates commented in their questionnaire on how difficult it was to actually notice any difference at all. It should be noted that in Video 2 of the user study, when using AB, the average score for the reference version was higher than the *MapReduce* version. This was a correct observation by the participants, but was not statistically significant enough to change our overall analysis.

Overall, our *MapReduce* implementation performs adequately with respect to visual quality. It is clear that the output is different from the reference transcoder, especially when using AB, but the amounts by which they differ do not appear to be significant in our test cases.

3) Is the file compression affected?

The difference in file size is least apparent in the Average Bitrate (AB) case, where the encoder is targeting a specific bitrate and file size. The differences for CRF encoded video vary much more than AB. In most cases, the file size output by the *MapReduce* transcoder is smaller, evidence of the CRF algorithm being affected by the chunking. CRF makes use of a large input buffer, something that chunking prevents from occurring. It uses this buffer to assess which frames can be encoded with less data and have the least perceptible change in quality (rather than actual statistical quality difference, such as PSNR). It is clear from Table 5 that CRF does not produce the same file compression when it is been transcoded using our *MapReduce* solution.

Overall, however, the file size differences are not large. In the AB case, where the output size is of most importance to the transcode, the differences are negligible. In the CRF case, where quality is the focus of the transcode, the difference is most substantial, but does not change the output file size by more than 12%.

4) Is there a performance increase overall?

In most of the test files we were able to achieve a performance increase over a reference sequential encoder when using a cluster of machines. In the best case, test file 8, the performance increase, if we include the time to *Demux* and *Merge* the file, was 10.2x that of the reference transcoder when using 19 identical machines (and one low cost master node).

This took the overall transcode time from 15,267 secs on a single machine with the reference transcoder, to 1492 secs on the cluster.

When comparing our results against our reference sequential transcoder, it is clear that the overall performance increase is reduced when include our sequential *Demux* and *Merge* phases. If the input file for processing was to be output in several different types and quality levels, the *Demux* process could be performed once, and then reused. The cost of the *Demux* operation is largely constant irrespective of cluster size, and so its percentage cost worsens as execution time is reduced. This makes the use of our solution most attractive when the input is processed several times for different settings, and the *Demux* cost can be spread across each of the executions, especially when the input is large and well suited toward scalability.

Making the *Demux* and *Merge* phases *Hadoop* jobs, that run in parallel over many machines would be possible in some cases. The *Demux* phase could be made parallel through the use of the distributed file system (S3) that it is stored on, and in some cases, where the output type supports segmentation, the *Merge* could be parallelised as well. However, this was deemed outside of the scope of our work, due to its complexity and restriction of our output type.

B. *Project Approach*

1) *MapReduce*

The choice of *MapReduce* as a paradigm for distributed transcoding allowed for rapid development of the initial solution, but finding a balance for block size was difficult. It was desirable for the block size to be as large as possible to reduce the visual quality degradation but small to increase scalability.

Also, tuning the settings of the cluster to maximise performance showed *Hadoop* and the *MapReduce* paradigm to be quite inflexible. The settings for *Map Task Slots* and general machine utilisation are static, and chosen when the cluster is initially setup. Using blocks of data as the only method to break up input for parallelisation limits the level of parallelism that can be achieved, especially when this has to be fixed and cannot be modified easily after *Demux*. The current version of *Hadoop* does not yet offer enough dynamic resource allocation to fully utilise the cluster it runs on for processing tasks, such as ours, that have different requirements per job.

Hadoop 2.0[19] (which is currently in beta) aims to address some of these dynamic requirements by decoupling resource management from the applications that run on the cluster. It allows for the cluster to run modified versions of *MapReduce*, or even MPI (Message Passing Interface) paradigm applications, whilst allowing for the resource management to customized and controlled separately. *Hadoop* 2.0 should be able to address our dynamic configuration and resource allocations issues, however it is currently unavailable for use in Amazon Elastic MapReduce and so outside the scope of this project.

2) *Stream Decomposition*

The decision to break the content up into its constituent streams of audio and video had ramifications on the design of the *Demux* phase, resulting in the complexity described in III. B, Demultiplexing and Chunking. However, it allowed us to improve the simplicity of the *Transcoder* and *Mapper* and allowed for us to use spatial parallelism over several threads on each machine. This use of spatial parallelism was made efficient by only having to consider one stream at a time and resulted in us being able to process high-resolution *Map Tasks* without running out of memory on our nodes.

Splitting up the streams did add more complexity to the *MapReduce* design than was anticipated. It arguably added more complexity in the *Demux*, *Reduce* and *Merge* than it

[19] http://hadoop.apache.org/common/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html [last accessed: 28/04/12]

gained in simplicity of the *Transcoder*, but we feel the performance and memory advantages it gave us justified the design decision.

VIII. CONCLUSION

We have successfully verified the feasibility of *MapReduce* as a solution for video transcoding and proposed a method for decomposing the streams of input efficiently. Whilst we found that the scalability of the solution is limited with smaller files, large test cases perform well. The visual quality of our output was degraded when using both the AB and CRF encoding strategies, but our user study confirmed that this was not noticeable when perceived in normal viewing conditions. The video compression achieved by our solution did not differ significantly from that of the reference transcoder.

We believe that future work should be concentrated on implementing a 2-pass encoding procedure that would improve the quality and compression of our output, at the detriment of some performance. 2-pass encoding decodes the entire video initially, calculating statistics about each of the frames and the video as a whole, which are then used to optimise the chosen encoding strategy. Whilst this does add extra computation to the beginning of our transcode operation, it would provide the encoders with all the information they require to perform optimally. The complexity of this procedure limits the flexibility of the transcoder and increases the complexity of the transcode overall. Now that we have proven *MapReduce* as a platform for transcoding and presented a solution for processing video correctly, we feel that 2-pass would be an ideal extension to further develop the solution into a feasible production transcoder, for use in elastic load scenarios in 'The Cloud'.

Further investigation into how *Hadoop* 2.0 can be integrated into our solution to solve the resource allocation issues, and to allow us utilise the cluster efficiently in the last 'wave' of tasks, is of interest. Combining this work with 2-pass transcoding, would allow custom task scheduling to break up the input into different sizes for each wave, ensuring that we process the video as efficiently as possible and still maintain visual quality throughout.

To summarise, we believe *MapReduce*, whilst having originally had a data-processing oriented background, provides a good platform for distributed video transcoding. *MapReduce* can clearly be used for computational loads, as our results and evaluation have shown, and improvements to *Hadoop* look to make this more viable in the future.

REFERENCES

Chen and Schlosser. Map-Reduce Meets Wider Varieties of Applications. *Intel Labs Pittsburgh Tech Report, IRP-TR-08-05, May, 2008*

Dean, J. and Ghemawat, S, 2004. MapReduce: Simplified data processing on large clusters. *Communications of the ACM,* 51(1), pp 107-113.

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

Garcia, A. and Kalva, H. and Furht, B. A study of transcoding on cloud environments for video content delivery, *Proceedings of the 2010 ACM multimedia workshop on Mobile cloud media computing, October 29-29, 2010, Firenze, Italy*, pp13-18.

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: 21/06/2011]

Pereira, Azambuja, Breitman and Endler, An Architecture for Distributed High Performance Video Processing in the Cloud, *Proceedings of Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference, July 2010, Miami, Florida, USA,* pp 482-289.

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

Rodriguez, A. Gonzalez A., Malumbres M.P., Hierarchical parallelization of an H.264/AVC video encoder, *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06), September 2006*, pp. 363–368.

Schmidt R. and Rella M, An approach for processing large and non-uniform media objects on mapreduce-based clusters, *Digital Libraries: For Cultural Heritage, Knowledge Dissemination, and Future Creation, 2011*, pp. 172-181.

Sharma B., Prabhakar R., Lim S., Kandemir M. and Das C., MROrchestrator: A Fine-Grained Resource Orchestration Framework for Hadoop MapReduce, *Department of Computer Science and Engineering, The Pennsylvania State University - Technical Report CSE-12-001, January 2012,* unpublished.

Shen and Delp, A parallel implementation of an MPEG encoder: Faster than real-time!, in: Proceedings of the SPIE Conference on Digital Video Compression: Algorithms and Technologies, San Jose, California, February 1995, pp. 407–418.

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

Zaharia, M. and Konwinski, A. and Joseph, A.D. and Katz, R. and Stoica, I. Improving MapReduce performance in heterogeneous environments, *Proceedings of the 8th USENIX conference on Operating systems design and implementation, San Diego, CA, USA 2008.* pp 29-42.