**MapReduce Audio and Video Transcoding**

Student Name: *Tom Stordy-Allison*

Supervisor Name: *David Budgen*

# Introduction

## Project Background and Motivation

Video transcoding is the process of converting video from one encoding format to another. The media industry has to do this frequently, processing many terabytes of video per day[[1]](#footnote-1) as the popularity of web video increases and different devices require different encodings. The project aim was to speed up the processing by employing a concept from functional programming (*MapReduce*) to spread the work across several parallel computers. We ask ‘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?’.

## MapReduce

**Figure 1b.** Reduce

*MapReduce* programs are designed to process large amounts of data in parallel (White, 2009:15), requiring that the data be organised so 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.

**Figure 1a**. Map

*MapReduce* is a parallel data processing paradigm popularised by Google for ‘web scale’[[2]](#footnote-2) processing of information using a cluster of machines (Dean and Ghemawat, 2004). Conceptually, programs that use *MapReduce* 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*. The *reducer* aggregates the processed data elements to produce an output from a set of input values; namely the output from the map phase (Figure 1b).

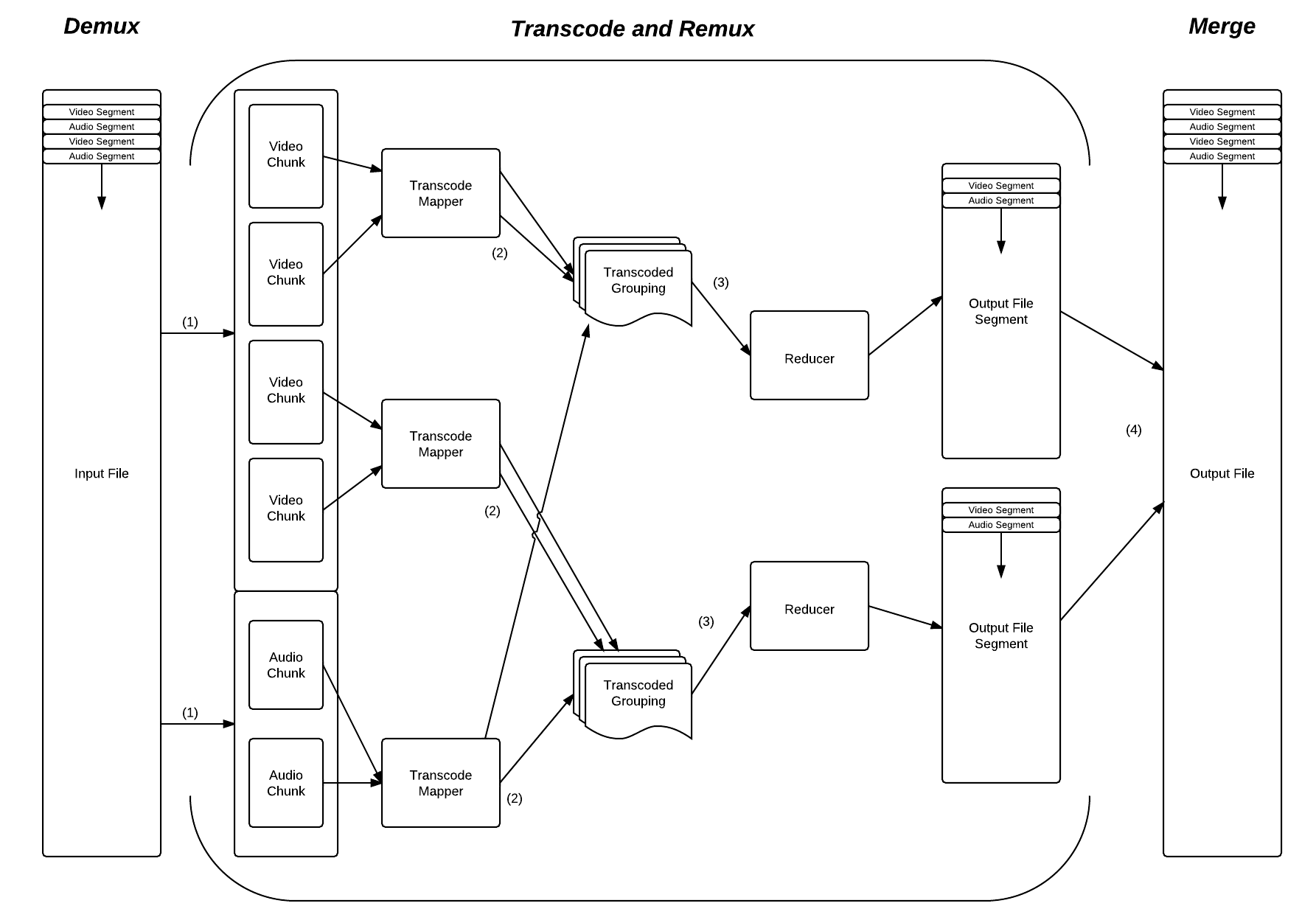
# Solution

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. Finally, we present the architecture for running a *MapReduce* cluster in ‘The Cloud’ using Amazon Web Services.

## High-level MapReduce Process

Figure 2 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.

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.



**Figure 2.** Conceptual overview diagram of our *MapReduce* program for transcoding.

1. **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.
2. **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*.
3. **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.

## Demultiplexing and Chunking

At the core of any distributed processing is the method by which the input is separated into Atomic Processing Units (APUs). We call this first phase of the process the *Demux* (short for De-multiplexing). This process separates the input file into its constituent streams of audio and video, and into APUs that can be processed separately by the *mappers*. Once the APUs are defined and sent to a *Mapper* there is no way for them to get any additional information, so all necessary information is collected and stored at creation.

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.

The data is output in key-value pairs that describe where this ‘chunk’ was in the original file (the timestamp). When grouping these chunks before the *Reduce*, we use the timestamp as our key.

## System Architecture and ‘The Cloud’

### Hadoop and HDFS

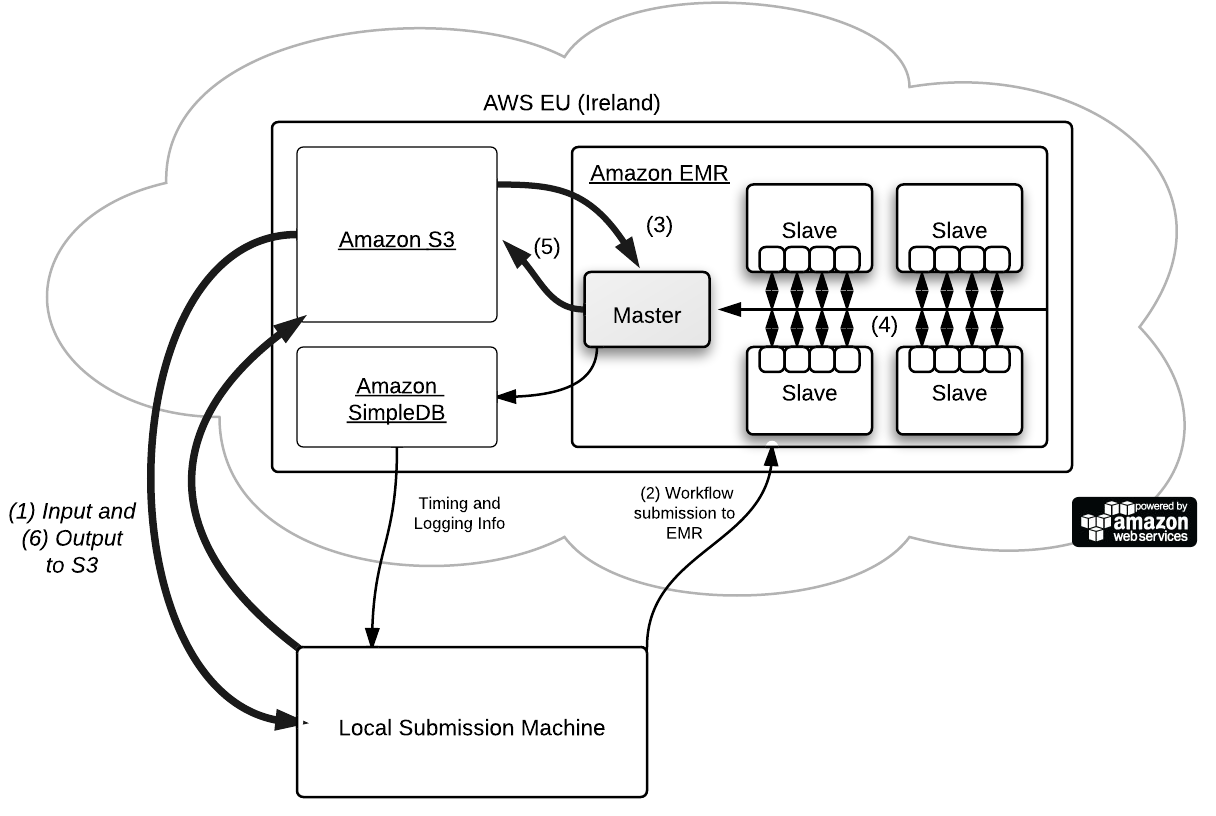
Apache *Hadoop*[[3]](#footnote-3) was chosen for our *MapReduce* implementation. As described in the original *MapReduce* paper, it also contains an implementation of a distributed file system, called HDFS (*Hadoop* Distributed File System). HDFS ensures that data is spread almost evenly across all of the machines in the cluster, broken up into blocks of a fixed size.

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.

### 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).

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 3.** AWS cloud architecture overview

The workflow for a ‘Transcode Job’ using AWS is as follows and can be traced through Figure 3:

1. The file(s) to be transcoded are uploaded to S3.
2. The workflow is submitted to EMR with the transcode job definition file.
3. Each transcode operation is then processed from the definition file. The master node 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, and the *Map* phase begins. Once all of the *Map Tasks* are complete the *Reduce* phase runs, 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, and merging them into a final file that it 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.

### Map Task Resource Allocation

Each of the slave machines running in the cluster has 7Gb of memory and eight 2.33Ghz cores. This makes them ideal for video transcoding, and when combined in a cluster, a formidable amount of computational power. Whilst some of the memory in the physical machine is lost to the virtualisation system Amazon use to implement the Elastic Compute Cloud system (a modified version of Xen[[4]](#footnote-4)), very little CPU power is lost, as the majority of the instructions operate directly on the underlying hardware.

# Results and Evaluation

We performed several experiments using a range of test input files focusing on three key areas; parallel scalability, output quality, and compression (see the ‘Results Data Appendix’). We will now evaluate each aspect of our research question, and comment on the overall suitability of *MapReduce* for video transcoding based on our performance results and evaluation of the output.

## 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 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 in real world scenarios.

Of all of our test files, the two largest scale best overall. Test files 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 (Figure 4).

## Does the ‘chunking’ degrade the visual quality of the encoding?

It was clear from the quantitative analysis data (Table 4) that the visual quality of the output from our solution differed from that of our reference encoder. In both of the encoding types we tested, it was clear that by breaking the video into chunks we reduced the amount of information that the encoder had available to optimise for quality.

However, our user study (Table 5) showed that this degree of quality degradation was not perceivable and that the video output was perfectly watchable. 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. Many of the candidates commented in their questionnaire on how difficult it was to actually notice any difference at all.

## Is the file compression affected?

It was clear from the data in Table 6 that neither of the encoding types for any of the test files have any significant increase or decrease in compression.

## 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 (Table 3). In the best case the performance increase, if we include the time to *Demux* and *Merge* the file, was 10.2 times faster than that of the reference transcoder when using 19 identical machines.

It is clear that the overall performance increase is reduced because of 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 same input is processed several times with different desired output settings, and the *Demux* cost can be spread across each of the executions, especially when the input is large and well suited toward scalability.

## Conclusion

We have successfully verified the feasibility of employing *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. To summarise, our results indicate that *MapReduce*, whilst having originally had a data-processing oriented background, provides a good platform for distributed video transcoding.

# References

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

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

# Results Data Appendix

**Table 1.** Test Input Files.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **#** | **File Size[[5]](#footnote-5)** | **Video pixels/sec** | **Audio samples/sec** | **Length (sec)** | **Approx.**  **bit rate** | **Type (Video/Audio)** |
| 1 | 4.6MB | 2.4M pixels/sec x 4[[6]](#footnote-6) | 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[[7]](#footnote-7) | 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[[8]](#footnote-8) | 8284 | 6103 kbit/s | H.264/AAC+AC3 |
| 8 | 10.52GB | 49.9M pixels/sec (‘HD’) | 288Khz | 9701 | 8670 kbit/s | H.264/HEAAC |

**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 |

## Overall Performance Increase Data

**Table 3.** Speed up vs Reference Transcoder

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **#** | **File Size** | **1 Machine** | **5 Machines** | **11 Machine** | **15 Machines** | **19 Machines** |
| 1 | 4.6MB | 0.2x | 0.2x | 0.2x | 0.2x | 0.2x |
| 2 | 183MB | 0.8x | 1.8x | 2.1x | 2.3x | 2.2x |
| 3 | 237MB | 1.0x | 2.6x | 2.9x | 3.1x | 3.1x |
| 4 | 364MB | 0.9x | 2.9x | 4.1x | 4.6x | 4.6x |
| 5 | 798MB | 1.1x | 3.7x | 5.2.x | 5.9x | 6.1x |
| 6 | 1.54GB | 0.9x | 2.8x | 3.9x | 4.4x | 4.6x |
| 7 | 6.32GB | 0.9x | 3.3x | 5.2x | 6.3x | 6.9x |
| 8 | 10.52GB | 1.0x | 4.0x | 6.9x | 8.9x | 10.2x |

Note: Tests were performed 3 times, and an average taken.

## Machine Scalability Results





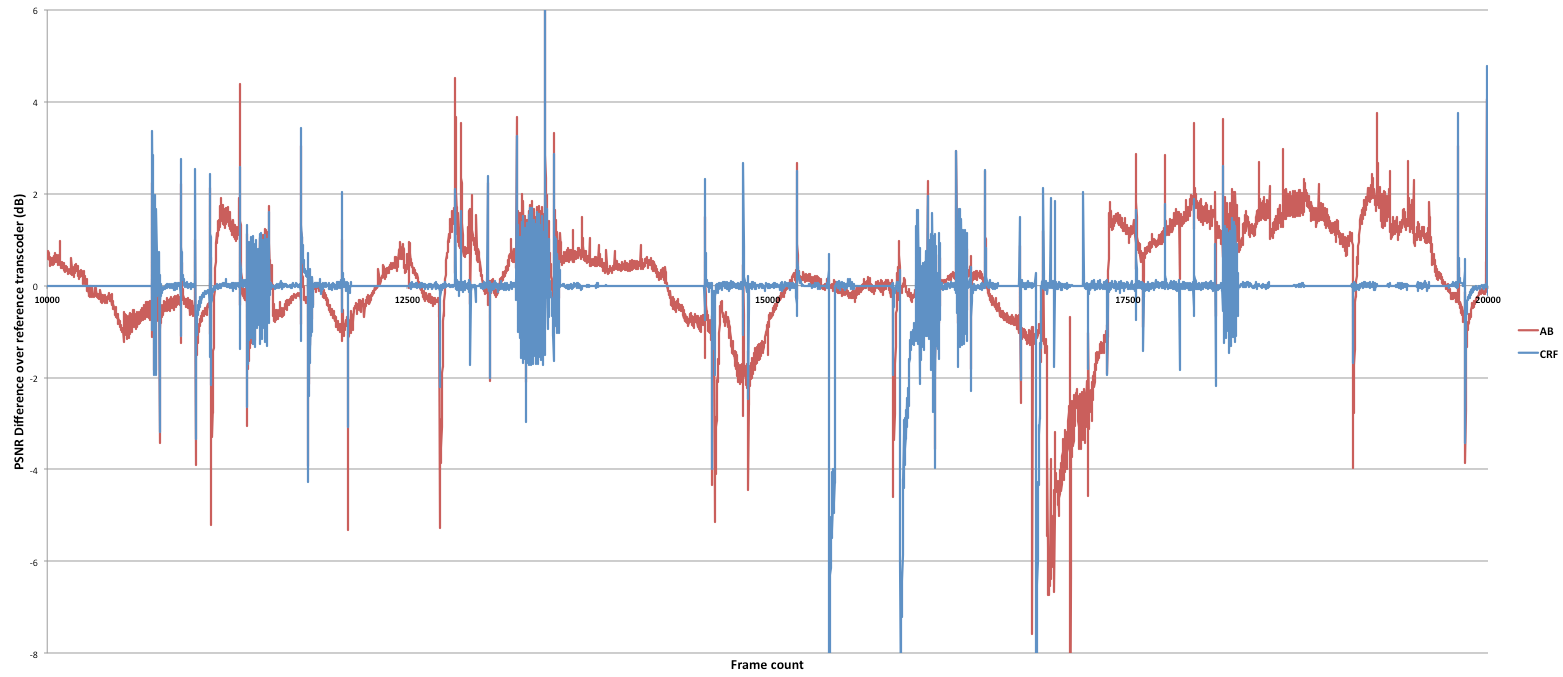
**Figure 4a/b.** Performance increase over a single machine as the number of *Map Task Slots* in the cluster increases.

N.B. *The y-axis scale differs between the graphs*.

## Visual Quality: Quantitative (PSNR)

**Table 4.** 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[[9]](#footnote-9)* | | | *N/A18* | | | *N/A18* | | *N/A18* | |
| 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 |



**Figure 5.** PSNR difference over reference transcoder for a 10,000 frame sample in Test file 4**.**

## Visual Quality: User Study Results

**Table 5.** 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 | |

Note: 16 candidates scored each video clip out of a maximum of 5. The T-value is based on a two-tailed T-test, and both values indicate that there is no significant difference between clips.

## Output File Size Comparisons

**Table 6.** 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% |

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”) [↑](#footnote-ref-1)
2. <http://jacobian.org/writing/web-scale/> [last accessed: 19/04/12] [↑](#footnote-ref-2)
3. <http://hadoop.apache.org/> [last accessed: 17/04/12] [↑](#footnote-ref-3)
4. <http://www.xen.org/> [last accessed: 12/04/12] [↑](#footnote-ref-4)
5. All file size units are presented using SI decimal prefixes, e.g. MB = 1,000,000 bytes (not 1,048,576 bytes). [↑](#footnote-ref-5)
6. Test file 1 has multiple video streams, effectively increasing its length without adding extra audio. [↑](#footnote-ref-6)
7. Test file 5 has more audio than the other files to test performance when many audio streams are present. [↑](#footnote-ref-7)
8. Test file 7 contains a 6-channel (5.1 Dolby Digital) stream at a sample rate of 48Khz per channel. [↑](#footnote-ref-8)
9. 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. [↑](#footnote-ref-9)