4. The Split&Merge Architecture

Scalable and fault tolerant architectures that allow distributed and parallel processing of large volumes of data in Cloud environments are becoming increasingly more desirable as they ensure the needed flexibility and robustness to deal with large datasets more efficiently. We are interested in architectures that deal with video processing tasks, as they are not fully addressed by existing techniques for high performance video processing. In what follows we discuss the requirements for such an architecture, focusing in the dynamic deployment of additional computer resources, as a means to handle seasonal peak loads.

The first point we need to address is flexibility. We want the ability to deploy the proposed architecture on different commercial Cloud platforms. Therefore, the architecture must be simple, componentized, and able to run on a hybrid processing structure, that combines machines in a private cluster with resources available in the Cloud. With this approach it would be possible, for example, to have servers in a local cluster, as well as instances on a public Cloud environment, e.g. Amazon EC2, simultaneously processing tasks. Alternatively, all the processing can be done on servers in a private cluster, but, using a storage in the Cloud.

To make this possible, all components of the architecture should be service oriented, that is, they must implement web services that allow functional architectural building-blocks to be accessible over standard Internet protocols, independently of platform, and/or programming languages. This is a key feature when dealing with services in the Cloud. In the case of Cloud services provided by Amazon, for example, it is possible to manipulate data stored in Amazon S3, or even to provision resources in the Amazon EC2, enabling the scaling up or down using programs that communicate through REST web services [15]. Thus, an architecture for task processing should provide a service-oriented interface for scheduling and manipulating jobs. The same paradigm must be used to support the communication among internal components. This makes deployment more flexible, facilitates the extension of existing features, and the addition and/or removal of software components, as needed.

If we analyze the Map-Reduce paradigm [10] in its essence, we note that process optimization is achieved by distributing tasks among available computing resources. It is precisely this characteristic that we want to preserve in the proposed architecture. The possibility of breaking an input, and processing its parts in parallel, is the key to reducing overall processing times [31].

With a focus on these issues, the proposed Split&Merge architecture borrows from key Map-Reduce concepts, to produce a simple, and yet general, infrastructure in which to deal with distributed video processing. The intuition behind it is the straightforward split-distribute-process-merge process illustrated in Figure 8. Similarly to Map-Reduce, this architecture efficiently uses available computing resources.

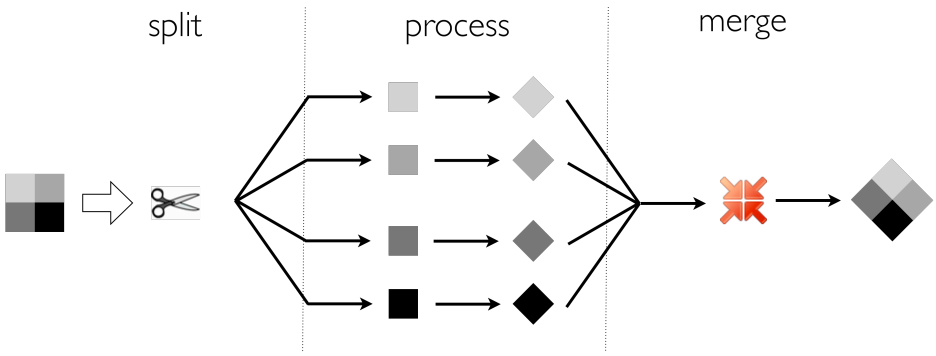

Figure 8. The Split, Process and Merge Concept

It is important to note that the proposed architecture was developed with the care to maintain a choice among techniques used in the split, distribute, process and merge steps, so that their implementation can be switched and customized as needed. That secures flexibility, adaptation, extensibility and the accommodation of different applications. In the case of video processing, it is paramount to allow for a choice among different codecs, containers, audio streams, and video splitting techniques. Let us take the case where the input video has no temporal compression, the MJPEG standard [16, 67] for example, as an illustration. In this case, the split can be performed at any frame. Conversely, cases where the input is a video encoded using only p-frame temporal compression, e.g., H.264 [17] Baseline Profile, we must identify the key-frames before splitting.

The generalization of this idea, lead to an architecture in which it is possible to isolate and diversify the implementation for the split, distribute, process and merge steps.

4.1 The Split&Merge for Video Compression

As discussed in chapter 3, video compression refers to reducing the quantity of data used to represent digital video images, and is a combination of spatial image compression and temporal motion compensation. Video applications require some form of data compression to facilitate storage and transmission. Digital video compression is one of the main issues in digital video encoding, enabling efficient distribution and interchange of visual information.

The process of high quality video encoding is usually very costly to the encoder, and requires a lot of production time. When we consider situations where there are large volumes of digital content, this is even more critical, since a single video may require the server's processing power for long time periods. Moreover, there are cases where the speed of publication is critical. Journalism and breaking news are typical applications in which the time-to- market the video is very short, so that every second spent in video encoding may represent a loss of audience.

Figure 9 shows the speed of encoding of a scene, measured in frames per second, with different implementations of the H.264 compression standard [17]. We note that the higher the quality, i.e., the bitrate of the video output, the lower the speed of encoding.

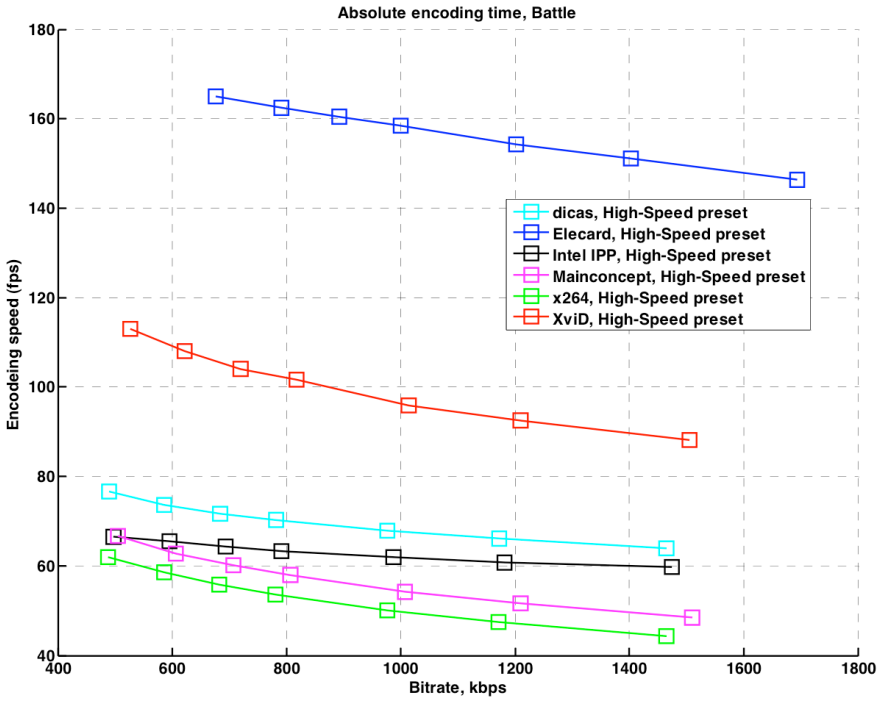

Figure 9. Encoding speed for different H.264 implementations [69]

In order to speed up encoding times, there are basically two solutions. The first one is to augment the investment in encoding hardware infrastructure, to be used in full capacity only at peak times. The downside is that probably the infrastructure will be idle a great deal of the remaining time. The second solution is to try and optimize the use of available resources. The ideal scenario is to optimize resources by distributing the tasks among them evenly. In the specific case of video encoding, one approach is to break a video into several pieces and distribute the encoding of each piece among several servers in a cluster. The challenge of this approach is to split, as well as merge video fragments without any perceptible degradation as a consequence of this process.

As described in the previous session, there are several techniques whose goal is to perform parallel and distributed processing of large volumes of information, such as the Map-Reduce paradigm. However, video files possess characteristics that hinder the direct application of distributed processing techniques, without first adjusting the way they deal with the information contained in the video. Firstly we must remark that a video file is a combination of an audio and a video stream, which should be compressed in separate and using different algorithms. These processes, however, must be done interdependently, so that the final result is

decoded in a synchronized way. This means that the video container must maintain the alignment between the two streams (audio and video) at all times. In addition, a video stream can be decomposed into sets of frames, which are strongly correlated, especially in relation to its subsequent frames. In fact, it is the correlation among the frames that allows the reduction of temporal redundancy in certain codecs [17, 58].

Therefore, without adaptation, the Map-Reduce approach is of very little use to video compression. Firstly, a classical Map Reduce implementation would divide the tasks using a single mapping strategy. Video encoding requires that we use different strategies to compressing video and audio tracks. Secondly, the traditional Map-Reduce approach does not take into consideration the order, much less the correlation of the pieces processed by individual units processing different parts of the video. Video encoding requires that we take into consideration frame correlation, and more importantly, the order of frames.

With a focus on these issues, the proposed architecture provides an infrastructure to deal with video processing. That is, for every video received, it is fragmented, its fragments are processed in a distributed environment, and finally, the result of processing is merged. As in the Map-Reduce, this architecture is able to efficiently use the computing resources available, and furthermore, it allows the use of more complex inputs, or more specific processing.

4.1.1 The Split Step

In what follows we describe a technique for reducing video encoding times, based on distributed processing over cluster or cloud environments, implemented using the Split&Merge architecture and illustrated in Figure 10. The fragmentation of media files and the distribution of encoding tasks in a cluster consists in a solution for increasing the performance of encoding, and an evolution of the simple distribution of single complete video encoding tasks in a cluster or cloud. The idea is to break a media file into smaller files so that its multiple parts can be processed simultaneously on different machines, thereby reducing the total encoding time of the video file. Furthermore, to avoid synchronization problems between audio and video, we must separate the two, so that they can be independently compressed. If processed together, chunks containing both audio and video may generate various

synchronization problems, since audio frames do not necessarily have the same temporal size than video frames. One should thus avoid processing both streams together, for it may generate audible glitches, delays and undesirable effects. Because the overall impact to the performance is very small, the audio stream is processed in one piece (no fragmentation).

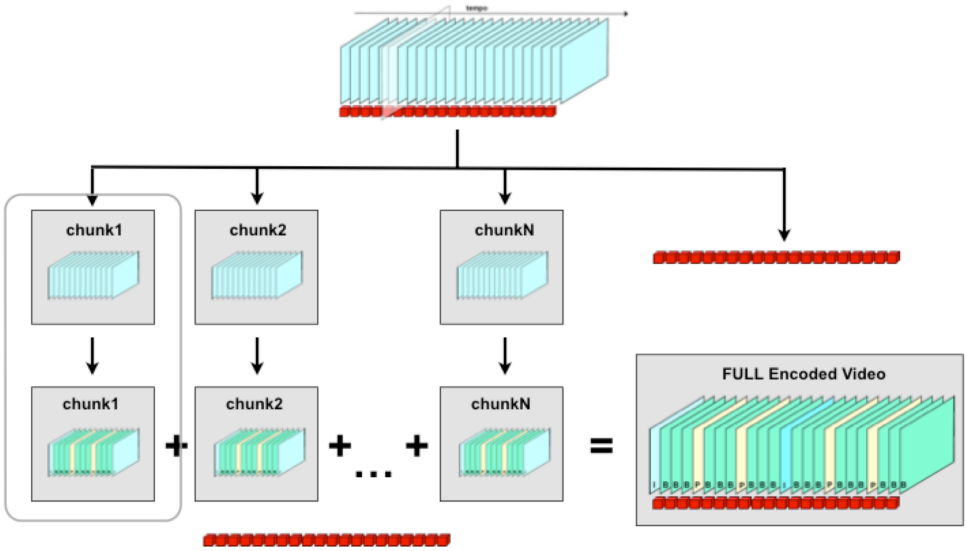

Figure 10. The proposed Split&Merge approach for video compression

The greatest challenge of video processing, differently from text files, is that it is not possible to split a video file anywhere. If the input video already provides some form of temporal compression, then it would be necessary to first identify its key-frames, so that the cuts are made at their exact positions. This is necessary because in the case where there is temporal compression, some frames (b-frames and p-frames) require information existing on key-frames to be decoded. Thus, a separation of video chunks, when there is temporal compression, which isolates a b-frame or p-frame from the key-frame required for its decoding, would derail the process of transcoding. To perform the video stream split, considering a temporal compression in the input, we can use the following algorithm:

```
identify key-frames
open chunk
for each frame in input video
  if frame is key-frame and chunkSize > GOP
        add frame to chunk
        close chunk
        open new chunk
```

```
    end
    add frame to chunk
  end
  close chunk
```

Situations where the original video does not show temporal compression, are special cases where the video can be split at specific frame numbers or at regular intervals. The important point here is to ensure that no frame coexists in more than one chunk, and that no frame is lost. In the cases where there is temporal compression, the duplicated key-frames should be discarded in the merge step.

A key point in the fragmentation of the input video is to determine the size of the chunks to be generated. This decision is closely related with the output that should be generated, that is, the video codec and compression parameters passed to it in the processing step. This is because, after processing, the chunks will present a key-frame in its beginning and in the end. Indiscriminate chunk fragmentation will produce an output video with an excess of key-frames, reducing the efficiency of compression, as key-frame typically contains much more information than a b or p frame. To illustrate this fact, it is frequent the use of 250 frames in between consecutive key-frames (GOP size), for a typical 29.97fps video. Thus, if in the split step chunks are generated with less than 250 frames, the efficiency of the temporal compression of the encoder will be inevitably reduced. A good approach is to perform the split so that the number of chunks generated is equal to the number of nodes available for processing. However, when we use an elastic processing structure, we can further optimize this split, analyzing what is the optimum amount of chunks to be generated, which certainly varies according to the duration of the video, and the characteristics of the input, and output to be produced.

This optimized split would require the implementation of a decision-making algorithm to evaluate the characteristics of input and output, choosing what size of fragment will use resources more efficiently, producing a high quality result and with acceptable response times. The implementation of this algorithm is quite desirable in order to improve the efficiency of the process, however, it is beyond the scope of this work.

When we split a video file into several chunks, or smaller files, we must repair their container, i.e., rewrite the header and trailer, so that the input chunks could be completely decoded during the process step, once important information about video structure is stored inside the header and/or trailer fragments of a video file, depending of the container type. This process can be avoided with a very interesting method. When we refer to split the video, we are actually preparing the data to be distributed in a cluster, and to be processed in parallel. If in the split step, instead of breaking the video file, we just identify the beginning and end points of each chunk, then it would not be necessary to rewrite the container. This greatly reduces the total encoding time. The disadvantage, in this case, is that all nodes must have read access to the original file, which could cause bottlenecks in file reading. Read access can be implemented through a shared file system, as an NFS mount, or even through a distributed file system with high read throughput. The structure bellow exemplifies the output in this split through video marking, which is a data structure containing the chunk marks:

```
[{'TYPE' => 'video', 'WIDTH' => output_video_width, 'HEIGHT' => output_video_height, 'CHUNK_START' => start_chunk_time, 'CHUNK_END' => end_chunk_time, 'CHUNK_ID' => chunk_index, 'ORIGINAL_INPUT_FILE' => input_video_filename }]
```

4.1.2 The Process Step

Once the video is fragmented, the chunks generated should be distributed among the nodes to be processed. In the specific case of video compression, this process aims at reducing the size of the video file by eliminating redundancies. In this step, a compression algorithm is applied to each chunk, resulting in a compressed portion of the original video.

The process of chunk encoding is exactly equal to what would be done if the original video was processed without fragmentation, i.e. it is independent of the split and the amount of chunks generated. However, if the option to mark the points of beginning and end of chunks was used during the split, then the processing step should also have read access to all the original video, and must seek to the position of the start frame, and stop the process when the frame that indicates the end of the chunk is achieved. Using this marking approach in the split

step, the processing step could be simply implemented as exemplified by the following mencoder [19] command and detailed by Table 1:

```
mencoder
   -ofps 30000/1001
   -vf crop=${WIDTH}:${HEIGHT},scale=480:360,harddup
${ORIGINAL_INPUT_FILE}
   -ss ${CHUNK_START}
   -endpos ${CHUNK_END}
   -sws 2
   -of lavf
   -lavfopts
format=mp4,i_certify_that_my_video_stream_does_not_use_b
_frames -ovc x264
   -x264encopts
psnr:bitrate=280:qcomp=0.6:qp_min=10:qp_max=51:qp_step=4
:vbv_maxrate=500:vbv_bufsize=2000:level_idc=30:dct_decim
ate:me=umh:me_range=16:keyint=250:keyint_min=25:nofast_p
skip:global_header:nocabac:direct_pred=auto:nomixed_refs
:trellis=1:bframes=0:threads=auto:frameref=1:subq=6
   -nosound -o $( printf %04u ${CHUNK_ID} ).mp4
```

Table 1. Description of MEncoder parameters used for video chunk encoding

Parameter	***Description***	***Used Value***
`ofps`	Output frames per second: Is the frame rate of the output video	29.97 (The same framerate of the input)
`vf`	Video Filters: a sequence o filters applied before the encoding. (e.g. crop, scale, harddup, etc)	Original video is firstly cropped to get the 4:3 aspect ratio. Then it is scaled to 480x360 spatial resolution. Finally it is submitted to the harddup filter which writes every frame (even duplicate ones) in the output in order to maintain a/v syncronization
`ss`	Seek to a position in seconds	This value is set to the chunk start time, extracted in the split step
`endpos`	Total time to process	This value is set to the chunk end time, extracted in the split step
`sws`	Software Scaler Type: This option sets the quality (and speed, respectively) of the software scaler	2 (bicubic)
`of`	Output format: Encode output to the specified format	lavf
`lavopts`	Options for the lavf output format	Format option sets container to mp4. The `i_certify_that_my_video_stream_does_not_use_b_frames` explicitly defines that the input and ouput video do not present b-frames. The output video codec is set to x264, an implmentation of H.264

x264encopts	Options for the x264 codec (e.g bitrate, quantization limits, key frame interval, motion estimation algorithm and range, among others)	In this example the output video bitrate is set to 280kbps, the GOP size is set to 250 frames, with minimum size of 25 frames, the H.264 profile is set to Baseline with no b-frames
nosound	Discard sound in the encoding process	-

There are several open source tools for video compression, among the most popular, ffmpeg[18, 20] and mencoder[19], which are compatible with various implementations of audio and video codecs. It is possible, for example, to use mencoder to implement the processing step, performing a compression of a high-definition video, generating an output that can be viewed on the Internet, as well as on mobile devices that provide UMTS[21] or HSDPA[22] connectivity. In this case, could be used the H.264 Baseline Profile with 280kbps, and a 480x360 resolution, performing, therefore, an aspect ratio adjustment.

In addition to processing the video chunks, it is also necessary to process the audio stream, which must be done separately during the split step. Audio compression is a simple process, with a low computational cost. The following piece of code exemplifies the audio processing, and Table 2 details it:

```
mencoder
   ${ORIGINAL_INPUT_FILE}
   -ovc raw
   -ofps 30000/1001
   -oac mp3lame
   -af lavcresample=22050,channels=1
   -lameopts cbr:br=32
   -of rawaudio -o audio.mp3
```

Table 2. Description of MEncoder parameters used for audio encoding

Parameter	***Description***	***Used Value***
ofps	Output frames per second: Is the frame rate of the output video	29.97 (30000/1001)
ovc	Ouput video codec: set the codec used in video encoding	The output video codec is set to raw since in this task only audio is considered

oac	Output audio codec: set the codec used in audio encoding	The Lame codec is used to generate an MP3 audio stream
af	Audio format: sets the audio sampling rate, channels, and others	The audio sampling rate is set to 22050Hz, and the stream has only one channel (mono)
lameopts	Options for the Lame codec	The encoder is set for constant bitrate with an average of 32kbps of data rate
of	Output format: Encode output to the specified format	rawaudio

At the end of the processing step, video chunks are compressed, as well as the audio stream. To obtain the desired output, we must merge and synchronize all fragments, thus reconstructing the original content in a compressed format.

4.1.3 The Merge Step

The merge step presents a very interesting challenge, which consists of reconstructing the original content from its parts, so that the fragmentation process is entirely transparent to the end user. Not only the join of the video fragments should be perfect, but also that the audio and video must be fully synchronized. Note that the audio stream was separated from the video before the fragmentation process took place. As compression does not affect the length of the content, in theory after merging the processed chunks, we just need to realign the streams through content mixing.

The first phase of the merge step is to join the chunks of processed video. That can be accomplished easily by ordering the fragments and rewriting the video container. As result, we will have the full compressed video stream, with the same logical sequence of the input. It is done by the identification of chunk index and ordering according to the split order reference, generated at the split step. Using the marking method in the split step, it is not necessary to remove duplicated key-frames, which could appear as consequence of imprecise split that occurs when using some encoding tools where the seek process is based only in timestamp, and not in frame counting, as mencoder, for example. The merge process can be performed using the following operation:

```
mencoder
  ${LIST_OF_VIDEO_CHUNKS}
  -ovc copy
```

```
   -nosound
   -of lavf
   -lavfopts
format=mp4,i_certify_that_my_video_stream_does_not_use_b
_frames
   -o video.mp4
```

Following, we remix the audio stream with the video, synchronizing the contents, and generating the expected output. The remix reconstructs the container by realigning the audio stream with the video stream, and, because the duration of either stream does not change in the compression process, the video content is synchronized in the end.

```
mencoder
   video.mp4
   -audio-demuxer lavf
   -audiofile audio.mp3
   -ovc copy
   -oac copy
   -of lavf
   -lavfopts
format=mp4,i_certify_that_my_video_stream_does_not_use_b
_frames
   -o ${OUTPUT}
```

The combination of the split, process and merge steps, implemented using the proposed architecture, results in a fully parallel and distributed video compression process, where different pieces of content can be simultaneously processed in either a cluster or, alternatively, in a Cloud infrastructure.

4.2 Deployment in The AWS Cloud

In cases where there is a floating demand or services, or when sudden changes to the environment dictate the need for additional resources, the use of public Cloud Computing platforms to launch applications developed using the Split&Merge architecture becomes extremely interesting. The “pay-as-you-go” business model provides a series of advantages: there are no fixed costs, no depreciation, and it does not require a high initial investment. Furthermore, it is totally elastic, i.e., it is possible to add and remove workers at any time, according to demand. If there is no demand, all workers can be turned off, on the fly, by the

master node. Operation costs are thus minimal. Even master nodes may be disconnected, and re-connected, manually, which makes the total cost of operation in idle situations very low. In Figure 11, we illustrate the Split&Merge architecture when used to implement a software application that makes use of a public Cloud infrastructure. Because we opted for the Amazon Web Services (AWS) infrastructure for our experiments, the examples used throughout the text refer to their services. The Split&Merge architecture, however, is general enough to accommodate other choices of cloud service providers.

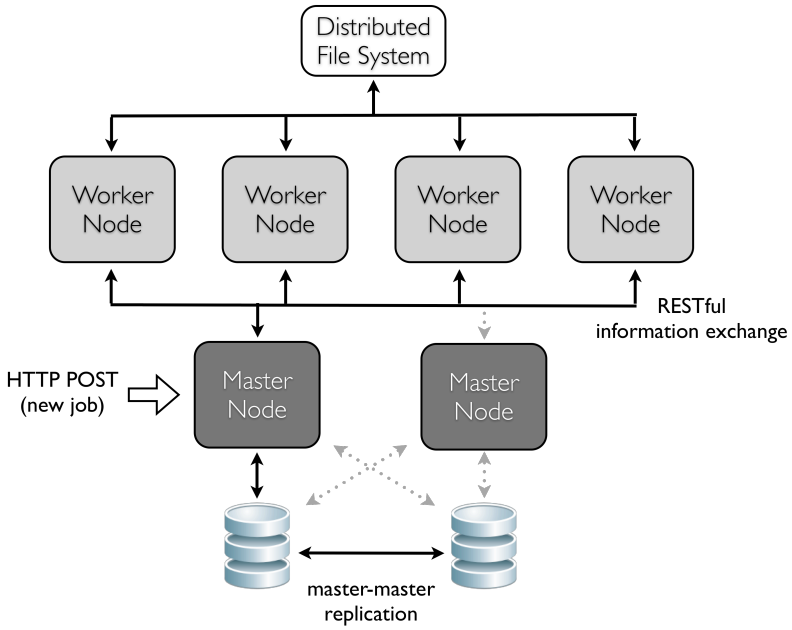

Figure 11. Split&Merge architecture deployed on Amazon Web Services infrastructure

To enable the use of Amazon Web Services to deploy applications using of the proposed architecture, we first need to build an image (AMI) for EC2 instances, one that corresponds to one full installed and configured worker. This way we ensure that new instances can be started in a state of readiness. We also need a separate image for the master node, because it has different software requirements, since it doesn't perform the video encoding itself.

For our storage needs, we use Amazon S3. In this case, redundancy and availability concerns are transparent and delegated to the Cloud provider. We also use Amazon Relational Database Service, a service that implements a simple relational database, used, in our architecture, to store the processing state (e.g. which chunks are already processed, which is processing phase, which nodes are available, queue control, among others).

An important point to consider when making a deployment in a public Cloud service, is data delivery and recovery in the Cloud storage. It is relevant because, in addition to paying for data transfer, the network throughput is limited by bandwidth availability between the destination and origin. This factor can greatly impact the performance of the application in question.

4.3 Fault Tolerance Aspects

To understand how the Split&Merge architecture deals with possible failures in its components, we need to detail the implementation of redundancy mechanisms, component behavior, and information exchange. The first point is the way in which messages are exchanged. We advocate in favor of a service-oriented architecture, based on exchange of messages through REST [15] web services.

The typical Map-Reduce [10] implementation [11] provides a single master node, responsible for the scheduling tasks to worker nodes responsible for doing the processing. Communication between workers and the master node is bidirectional: the master node delegates tasks to workers, and the workers post the execution status to the master. This type of architecture has received severe criticism, as a single failure can result in the collapse of the entire system. Conversely, worker failures could happen without ever being detected.

The Split&Merge architecture tackles this problem by coupling a service to the master node, that periodically checks the conditions of its workers. This ensures that the master node, which controls the entire distribution of tasks, is always able to identify whether a node is healthy or not. This simple mechanism can be further refined as necessary, e.g., adding autonomic features, such as monitoring workers to predict when a particular worker is about to fail, isolating problematic nodes, or rescheduling tasks. Of course, care must be taken to avoid overloading the master node with re-scheduling requests, and additional overhead as the result of the action of recovery and prevention mechanisms.

Another issue addressed by the proposed Split&Merge architecture is related to the fact that in traditional Map-Reduce implementations the master node is a single point of failure. The main challenge in having two active masters is sharing

state control between them. More specifically, state control sharing means that, whenever one of the master nodes delegates a task, it must inform its mirror which task has been delegated and to which worker node(s), so that both are able to understand the processing status post from the worker(s). State sharing can be implemented through several approaches, but our choice was to use master replication using a common database. In addition to the simplicity of the chosen state sharing solution, we also secure control state persistence. In case of failure, we may resume processing of the chunks from the last consistent state.

We must also pay attention to how workers read the input and write processing results, which translates to the problem of ensuring file system reliability. In cases where a private cluster is used, we opted for a shared file system, e.g. NFS, HDFS (that uses a distributed architecture)[23], or MogileFS[24]. They seem a natural choice as Distributed file systems, in general, already incorporate efficient redundancy mechanisms. In cases where the Cloud is used, storage redundancy is generally transparent, which greatly simplifies the deployment architecture. However, we must note that data writing and reading delays in Cloud storage systems are significantly higher, and often depend on the quality of the connection among nodes and servers that store content.

4.4 Performance Tests

In order to validate the proposed architecture we experimented using Amazon's AWS services. We deployed an instance application responsible for the encoding of different sequences of videos, evaluating the total time required for the encoding process, and comparing it with the total time spent in the traditional process, where the video is encoded without fragmentation, i.e. all content is rendered on a single server.

For these tests, we selected sequences of high-definition video, with different durations, and encoded with MJPEG 25Mbps, 29.97fps, and audio PCM/16 Stereo 48kHz. The video output of the compression process was set to be an H.264 Baseline Profile, with 800kbps, 29.97fps, and with a resolution of 854x480 (ED), and audio AAC, 64kbps, Mono, 44100Hz.

To deploy the infrastructure for this case study, we chose AWS instance types m1.small for all servers. The m1.small type is characterized by the following attributes:

1.7 GB memory

1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit)

160 GB instance storage (150 GB plus 10 GB root partition)

I/O Performance: Moderate

In Figure 12 we depict the comparison between total times, measured in seconds, required for the encoding of different video sequences, using the proposed Split&Merge implementation, and using the traditional sequential compression process. In this test scenario, we worked with chunks of fixed size (749 frames), since in MJPEG all frames are key-frames, and with one node per chunk.

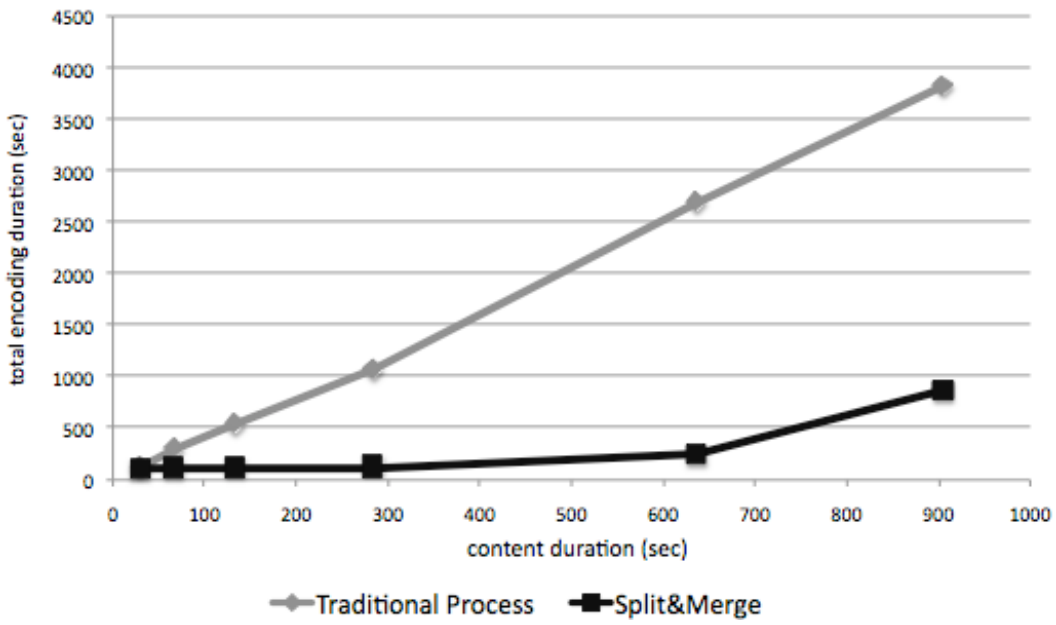

Figure 12. Total Encoding Times for Different Sequence Durations (in sec)

Note that, while the total encoding time using the traditional process, grows linearly with increasing duration of the video input, the Split&Merge, average process times remain almost constant for short duration videos. In fact, the total CPU time consumed, which is the sum of the CPU usage in all nodes, will be greater in the Split&Merge approach, however, the distribution of processing

among several nodes for parallel execution will reduce the total encoding duration. This result is quite significant when one considers videos of short and average duration. In the case when we have a video about 10 minutes long, the total time for encoding using the technique of Split&Merge is equivalent to less than 10% of the total time spend using the traditional process, which is extremely interesting for applications where time to market is vital.

Sports coverage can also benefit from a gain in processing time. If we consider the encoding of soccer matches, where videos are typically two hours long, we could maximize gains by provisioning a greater number of servers in the Cloud. In this case, we are able to reduce the total production time from several hours, to a few minutes. However, as the number of chunks being processed increases, the efficiency of the Split&Merge approach tends to be reduced, as evidenced in Figure 12 (note that there is a significant decay in efficiency for sequences over 700 seconds long. This fact is explained by network input/output rates, i.e., too many concomitant read operations slow down the process).

Whereas, in the limit, the elastic capacity of a public Cloud tends to exceed user's demand, i.e. the amount of resources available is unlimited (including network latency) from users perspective, then we can say that it is possible to encode all the video content of a production studio collection, with thousands of hours of content, in a few minutes, by using the approach of Split&Merge deployed in a Cloud, which certainly would take hundreds of hours using the traditional process of coding in a private cluster.

4.5 A Cost Analysis

Another point worth considering is the monetary cost of the Split&Merge approach when deployed in a public Cloud infrastructure, against the cost of having a private infrastructure dedicated to this task, with dozens of dedicated servers. Taking into account the results of the tests above, and an average production of 500 minutes a day, we have, at the end of one year, an approximate cost of $25,000 using the Amazon AWS platform, with the added advantage of producing all content in a matter of minutes. This value is comparable to the cost of only one high-end single server, around $20,000 in Brazil, including taxes, not considering the depreciation and maintenance, which makes the architecture of

Split&Merge deployed in the public Cloud not only efficient in terms of processing time, but also in terms of deployment and operation costs.

Considering an optimal situation where there are nearly unlimited resources available, it is possible to use the experimental results to predict the total cost and number of nodes needed to encode videos of different categories. Table 3, bellow, compares the traditional encoding process with the proposed Split&Merge approach. In this example we set the total encoding time to 2 minutes, and explore several different scenarios, i.e. advertisements, breaking news, TV shows and movies or sports matches, respectively. We base our calculations in the cost per minute, although Amazon's minimum billable timeframe is the hour. We are taking into consideration scenarios where there is a great number of videos to be processed, so machines are up and in use for well over one hour period.

Table 3. Comparison between the Traditional Encoding Process and The Split&Merge Approach

Input Video Duration	**Traditional Encoding Duration**	**S&M Encoding Duration**	**Number of S&M Nodes**	**S&M Encoding Cost Using EC2 (in US dollar)**
30 sec.	2 min.	2 min.	1	$0.003
5 min.	19 min.	2 min.	10	$0.03
30 min.	112 min.	2 min.	57	$0.16
2 hour	7.5 hour	2 min.	225	$0.63

Note that the Split&Merge approach, when deployed in a public Cloud, reduces the total encoding time for a 2-hour video from 7.5 hours to 2 minutes, with the total processing cost of $0.63. If we extrapolate these numbers for the Super Bowl XLIV [25], it is possible to encode the 3.3 hours match for $1.03, in only 2 minutes, as opposed to 12.2 hours, if we opted for the traditional process.