Configuração de Alto Desempenho para Processamento de Vídeo em um Cluster Hadoop

Rodrigo Oliveira Brito
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Salvador, Brasil
Email: rodrigo.brito@ifba.edu.br

Sandro Santos Andrade, Ph.D
Professor do Departamento de Ciência da Computação no
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Pesquisador do Grupo de Pesquisa em Sistemas Distribuídos,
Otimização, Redes e Tempo Real - GSORT
Salvador, Brasil
Email: sandroandrade@ifba.edu.br

***Resumo*—As novas tecnologias propiciaram à sociedade a produção de uma quantidade massiva de dados, dos quais uma parte significativa é constituída por arquivos de imagem e vídeo. Ambos possuem uma elevada complexidade computacional, necessitando de grande poder de processamento, consumindo grandes fatias de tempo no uso da CPU, e espaço em disco, requerendo muitas vezes uma infraestrutura com custos proibitivos. O Hadoop oferece uma plataforma de Computação Distribuída, independente de uma infraestrutura de *hardware* específica, a qual pode ser configurado para otimizar o processamento de vídeo. Diante disto, quais são os parâmetros mais significativos necessários a obter o melhor desempenho de um *cluster* Hadoop, quando utilizado para processamento de vídeo? Em resposta a este questionamento, foi criada uma aplicação *MapReduce* para extração dos *frames* de um vídeo, aplicação de filtros e criação de um novo vídeo a partir dos *frames* processados com o objetivo de estudar a influência dos parâmetros de configuração do Hadoop no desempenho de um *job MapReduce* para processamento de vídeo. Para efeito de avaliação de desempenho, é feita uma comparação entre os resultados obtidos no processamento distribuído de vídeo no Hadoop, em função de suas configurações, e no MLT, um programa que obtém resultado similar utilizando apenas um único computador.**

Palavras-Chave*—*Big data, Hadoop, Processamento de Vídeo, Processamento Distribuído, Alto Desempenho

I. Introdução

A evolução tecnológica da sociedade humana impulsionou o surgimento de novas formas de comunicação, bem como um número sem precedentes de informações não estruturadas. Tanto aplicações científicas quanto usuários domésticos têm produzido uma quantidade massiva de dados. No que concerne ao poder de processamento e armazenamento, bem como questões éticas que se relacionam com a privacidade do indivíduo frente a questões de segurança nacional e interesse de grandes corporações empresariais, vive-se hoje a era do *Big Data* [1].

Em contrapartida, apesar da capacidade dos dispositivos de armazenamento terem aumentado, falando especificamente dos discos eletromecânicos, a sua taxa de transferência não evoluiu de maneira proporcional. Isto ampliou ainda mais o gargalo de Von Newman — expressão usada para enfatizar a diferença de velocidade entre processadores e os diversos tipos de memórias existentes em um computador — pois, a velocidade das memórias RAM e dos microprocessadores têm crescido em um ritmo constante [1][2]. Uma possível solução para isso seria fragmentar o conteúdo de um arquivo em vários discos e efetuar a sua leitura a partir deles de uma única vez. Isto constitui uma forte motivação à utilização de um sistema distribuído capaz de beneficiar-se da execução de tarefas em paralelo [1].

Referências indicam que o Facebook e o Youtube armazenavam, em 2014, respectivamente, 282 e 504 PB (*petabytes*) [3]. Uma vez que parte significativa dos dados das redes sociais são imagens e vídeos, pode-se inferir que uma parte considerável deste acervo digital é constituído por arquivos multimídia. Logo, faz-se necessário a construção de soluções, as quais além de extrair informações úteis, possam criar maneiras otimizadas para processar e guardá-las [4].

O Google formulou uma solução capaz de fazer frente aos novos desafios proporcionados pela era da informação: surgia então um sistema de arquivos distribuído altamente escalável e um paradigma de programação distribuída chamado de *MapReduce*. Porém, a popularização de tais ideias veio através de uma implementação *open source* conhecida como Hadoop. Atualmente o Hadoop fornece uma plataforma distribuída, segura e confiável para análise de dados, constituindo-se em um verdadeiro ecossistema de *software* [1].

O processamento de imagem, e de modo mais expressivo o de vídeo, possui alta complexidade computacional exigindo um elevado tempo de uso da CPU [5] e a utilização de *hardware* especializado apresenta um custo elevado [6]. Entretanto, a maioria dos formatos de vídeo possui uma estrutura hierárquica de organização de seus dados a qual possibilita que seus constituintes, os *frames*, sejam decodificados arbitrariamente [4][7]. Teoricamente seria extremamente vantajoso armazená-lo no HDFS (*Hadoop Distributed Filesystem*), o sistema de arquivo distribuído do Hadoop, e utilizar a inclinação natural da estrutura de seu formato, a possibilidade de ler seus *frames* independente da posição em que ele se encontra, para processá-lo paralelamente em um *cluster*, reduzindo desta forma o seu tempo de processamento.

Otimizar o desempenho de um sistema distribuído que executa sobre múltiplos sistemas de *sofware* e *hardware* da envergadura do Hadoop é extremamente complexo. Podem existir gargalos de diversas origens, desde componentes físicos, versões de *software* e até mesmo configuração inadequada do *cluster* [8]. Porém, o Hadoop possui uma série de parâmetros

os quais podem ser customizados para extrair o máximo de performance [8]. Deste modo, a sua correta parametrização é um fator crucial, podendo ser a diferença entre a maximização de desempenho ou potencialização e criação de novos gargalos. Assim, este texto tem por objetivo avaliar as configurações dos parâmetros que influenciam na performance do processamento de vídeo, por meio de um *job MapReduce*, em um *cluster* Hadoop.

O restante do trabalho está organizado como segue. Na primeira é feita uma revisão bibliográfica sobre o tema. A subseção 2.1 faz uma explanação a respeito do Hadoop de modo a esclarecer seus principais conceitos tais como o paradigma *MapReduce*, seu sistema de arquivos, o HDFS, e o YARN (*Yet Another Resources Negotiator*). A 2.2, aos principais conceitos referente aos vídeos digitais e a 2.3 aos artigos cuja temática seja semelhante. A seção 3 tem por objetivo explanar todo o processo de construção da presente solução. As seções 4 e 5 são dedicadas as optimizações dos parâmetros de configuração do Hadoop e avaliação experimental da solução proposta. Na 6 é feita a conclusão e a 7 é dedicada as presentes limitações envolvendo este trabalho e a 8 discorre sobre possíveis futuras melhorias.

II. Referencial Bibliográfico

A. HADOOP

Entre os anos de 2003 a 2004 o Google divulgou a concepção do seu sistema de arquivos, o GFS, acrônimo de *Google File System*, e o modelo de programação *MapReduce* [1]. O GFS foi projetado para armazenar arquivos muito grandes bem como resolver problemas de escalabilidade, pois, um cluster GFS pode crescer, em teoria, indefinidamente. Já o novo paradigma de programação, segundo White [1], abstraia as dificuldades de leitura e escrita em discos em um sistema distribuído por meio de um conjunto de chaves e valores.

Estas publicações impulsionaram os desenvolvedores do Apache Nutch, uma *engine* de busca na web cujo código era *open source* e a arquitetura não era escalável, parte do Apache Lucene, a criarem uma implementação *open source* do GFS, o NDFS (*Nutch Distributed File System*). Posteriormente, a maioria de seus algoritmos foi migrada para o paradigma *MapReduce* e estes deveriam executar no NDFS. Em 2006 o Nutch foi emancipado do Lucene e renomeado de Hadoop [1].

O Hadoop fornece uma plataforma *open source*, segura e escalável, que executa em uma estrutura de *hardware* como *commoditie*, isto é, *hardware* oriundo de qualquer fabricante. É constituído basicamente por um sistema de arquivos distribuído (HDFS — *Hadoop Distributed File System*), pelo *MapReduce*, um paradigma de programação dsitribuída, e por uma camada de gerenciamento de recursos entre a aplicação e o HDFS, o YARN (*Yet Another Resources Negotiator*) [1].

De maneira análoga ao GFS, o HDFS é um sistema de arquivos que possui foco na escalabilidade, podendo armazenar arquivos da ordem de centenas de *gigabytes* [1]. Na maioria dos discos da atualidade, a taxa de transferência de arquivos encontra-se na ordem de 100 MB/s. A título de exemplo, a leitura de um arquivo de 1 TB necessitará de aproximadamente duas horas e meia. O HDFS soluciona este problema do seguinte modo [1]: o armazenamento do arquivo de 1 TB é fragmentado em vários discos ao longo do *cluster* de modo a efetuar a leitura paralela do mesmo, a qual é feita, mais ou menos, na mesma velocidade da taxa de transferência do disco. Para que isso seja possível, a leitura de um arquivo no HDFS é feita em blocos de 128 MB cada, o mesmo tamanho *default* de um bloco do HDFS, minimizando desta forma os custos de busca [1].

O principal benefício proveniente da abstração de um sistema de arquivos distribuído em blocos é a possibilidade de ter um arquivo que ocupe um espaço maior que qualquer HD disponível na rede. Além disso, os blocos de um arquivo não precisam ser necessariamente armazenados em um mesmo *node*, o que facilita a leitura paralela dos dados. A consequência negativa é que independentemente do tamanho do arquivo armazenado ele ocupará no mínimo 128 MB, tornando o Hadoop inadequado para lidar com arquivos muito pequenos [1].

Em um *cluster* HDFS os *nodes* podem ser de dois tipos: o *namenode* (*master*) e os *datanodes* (*workers*). Um *namenode* é responsável pelo armazenamento da árvore do sistema de arquivos e conhece exatamente os *nodes* encarregados de todo o trabalho pesado feito no HDFS. Eles armazenam os blocos dos arquivos contidos no HDFS, quando solicitados efetuam a sua busca e se reportam periodicamente ao *namenode* comunicando quais blocos estão armazenando [1].

Com a informação partilhada através de diversos dispositivos ao longo de uma rede, faz-se necessário ponderar possíveis falhas de *hardware* cujas consequências variam de indisponibilidade a perda de dados. A maneira predominante mais comum de lidar com este tipo de questão é a replicação. No HDFS os blocos são replicados através de um determinado número de máquinas do *cluster*. Em caso de indisponibilidade a sua cópia pode ser lida de outra máquina de maneira transparente ao cliente e se o bloco estiver corrompido ou o *node* indisponível, a cópia pode ser replicada ao longo de outras máquinas disponíveis no *cluster*. Assim possíveis falhas dos *nodes* podem ser administradas, garantindo uma alta taxa de disponibilidade do sistema [1].

Um *job MapReduce* é constituído por uma entrada, um programa *MapReduce* e informações de configuração. O Hadoop executa o *job* dividindo-o em dois tipos: um *map* e outro *reduce*. Eles são programados utilizando o YARN, que os executa nos *nodes* do *cluster*. Vale a pena salientar que mais de um *job* do tipo *map* é executado por vez e por *default* apenas um *job* do tipo *reduce*, porém a quantidade de *reducers* pode ser configurada pelo programador. Caso a execução da tarefa falhe, ela é redirecionada imediatamente para outro *node* disponível.

A entrada de um *job MapReduce* é dividida em pedaços de tamanhos fixos denominados *input-split*, os quais possuem o mesmo tamanho de um bloco do HDFS e são processados paralelamente por diferentes *nodes* do *cluster*. Cada uma delas executa as definições *map* customizadas pelo usuário, para cada *input-split*. Preferencialmente cada tarefa *map* é executada no *node* no qual o dado está armazenado de modo a evitar a utilização de sua largura de banda disponível. Uma tarefa *reduce* não aproveita-se deste recurso uma vez que ela é a entrada de todas as outras tarefas *map*. Logo todas as saídas ordenadas da tarefa *map* deverão ser transferidas através da

rede para o *node* no qual a tarefa *reduce* está executando, no caso *default* quando têm-se apenas um único *reduce* [1].

O fluxo completo de um *job MapReduce* é ilustrado na Figura 1 [1]. Os retângulos serrilhados correspondem a um *split*, uma representação lógica de um bloco do *node*, o qual encontra-se em memória. As setas descontínuas indicam o fluxo de dados em um *node* em particular. Cada *split* é lido em um formato serializado adequado a fase *map*, o quadrado em azul com o *tag map*, gerando os pares chave-valor. Ao final da fase *map* os pares chave-valor são ordenados em função da sua chave e a seguir são transferidos, indicado pela seta contínua, para o *node* no qual irá ocorrer a próxima etapa. Pares com chaves idênticas são agrupados — indicado por *merge* — e transferidos para a fase *reduce*, e o resultado é armazenado no HDFS. Frequentemente o resultado de uma computação *MapReduce* será consumido por outras aplicações [9]. Assim faz-se necessário armazená-lo em um formato, o qual possa ser eficientemente consumido, representado pelo *output* HDFS na Figura 1 [1].

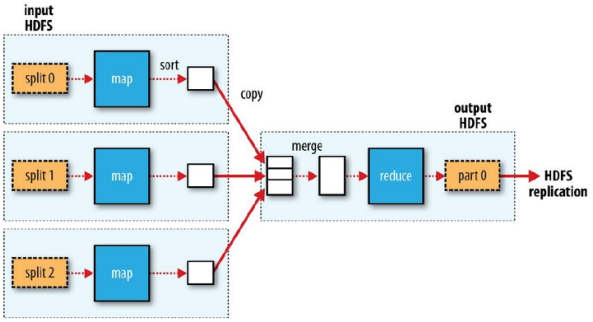

Figura 1: MapReduce Job [1]

A Figura 2 mostra o YARN como a camada intermediária entre a aplicação e o HDFS, exemplificando a divisão em camadas do Hadoop. O YARN fornece APIs para requisição e trabalho com o *cluster*, as quais, em geral, não são utilizadas diretamente pelos desenvolvedores. De maneira alternativa, utilizam APIs de alto nível, a camada *Application* na Figura 2, e o YARN se encarrega das complexidades do gerenciamento de *nodes* em um sistema distribuído [1].

O núcleo do YARN oferta os seguintes recursos: um gerenciador de recursos por *cluster*, um gerenciador de *nodes*, executando em todas os *nodes* do *cluster*, e um monitor de *container*, o qual pode ser compreendido como um conjunto limitado de recursos, tais como CPU e memória, em que uma aplicação específica executa [1].

Um dos aspectos importantes em um sistema distribuído é a utilização eficiente da largura de banda do *cluster*. Deste modo, o YARN permite que aplicações especifiquem localmente os recursos que ela necessita. Caso os recursos não estejam disponíveis, o YARN pode alocar o *container* em outro *node* do *cluster*. Na hipótese do processamento de blocos do HDFS, os *container* são alocados nos *nodes* no qual está o bloco a ser processado. Em caso de indisponibilidade o *container* será executado no *node* onde a réplica do bloco está contida [1]. Isto evita a transmissão desnecessária de dados, economizando desta forma a largura de banda.

O paradigma *MapReduce* pode não ser adequado a todas as situações. então pode-se escrever um código que execute diretamente sobre o YARN. Em muitos casos não é necessário construir uma aplicação YARN do zero pois existe um conjunto de projetos que tornam esta tarefa mais simplificada, tais como Apache Storm, Apache Slider, Apache Twill, *distributed shell application*, entre outros [1], indicados na camada *application* da Figura 2.

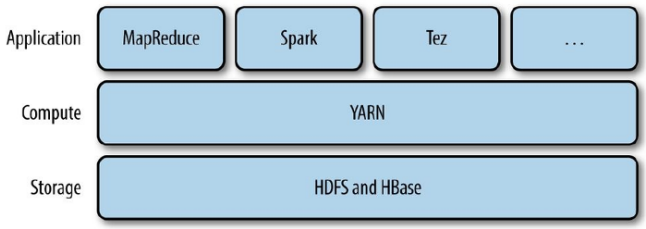

Figura 2: YARN layer [1]

Embora feito em Java, o Hadoop fornece uma API a qual permite escrever programas *MapReduce* em outras linguagens, tais como Ruby e Python, as quais possuem compatibilidade com o *streaming* padrão do Unix [1].

Por *default* o Hadoop não possui tipos primitivos adequados a leitura de vídeos, entretanto o *framework* dá suporte à criação de tipos que possam satisfazer as necessidades do programador [1].

B. FUNDAMENTOS DE PROCESSAMENTO DIGITAL DE VÍDEO

Avanços significativos no meio científico, no campo do processamento digital de imagens, foram produzidos, a partir da década de 60, pelo *Jet Propulsion Laboratory*, um importante centro tecnológico norte-americano da NASA. O advento da necessidade de processar imagens da lua provenientes de sondas espaciais não tripuladas foi o principal catalisador ao uso de computadores para analisá-las. Desta forma, foram lançados os alicerces para o surgimento do processamento digital de imagens, uma nova área do conhecimento que viria a impactar e influenciar diversas outras, tais como telecomunicações, transmissão de sinais de TV, medicina, entre outros [10].

A primeira etapa de um processamento consiste na formação da imagem digital propriamente dita. Tal processo consiste na captura de um sinal ótico, conversão a um sinal elétrico analógico para em seguida convertê-lo a um sinal digital [10]. O sinal analógico é quantizado — atribuição de valores discretos a um sinal que possui amplitudes infinitas — mediante um processo de amostragem fundamentado no Teorema de Nyquist [5]. Este estipula que um sinal analógico pode ser recuperado a partir de valores discretos se tais valores forem amostrados a partir de uma taxa determinada, a qual dever ser o dobro da maior frequência encontrada no sinal a ser convertido. Utilizar esta técnica faz-se necessário para que o modelo matemático binário no qual os computadores da atualidade são baseados possa reconstruir o sinal com a maior aproximação possível [5].

O processo de quantização de um sinal analógico é ilustrado na Figura 3. Os pontos em destaque no sinal analógico senoidal corresponde as amostras, as quais contém todas as informações necessárias a conversão do sinal analógico em digital [11].

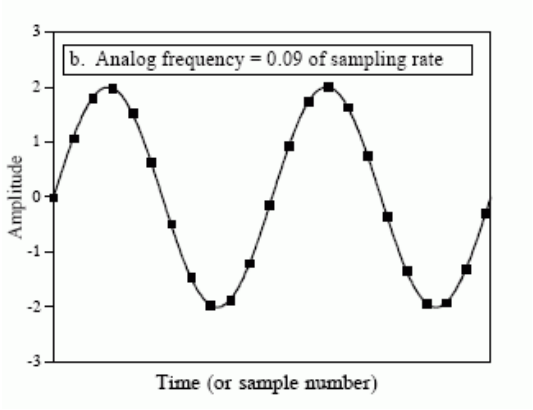

Figura 3: Quantização de um sinal analógico [11]

Uma imagem pode ser representada digitalmente por um conjunto matricial de *pixels*, a menor unidade que constitui uma imagem, o qual é constituído por três componentes, R, G e B (*red, green, blue*), sendo estes responsáveis pelas cores da imagem. Estes componentes são representados por um número inteiro variando de 0 a 255, o que corresponde a um *byte*. Cada *byte* é responsável pela intensidade de cada uma das componentes da cor de um *pixel*. Um *pixel* utilizado para reproduzir uma imagem em tons de cinza pode ser representado por um *byte*. No entanto para expressar cores em *truecolor* de um determinado *pixel*, ou conjunto de *pixels*, uma imagem utilizará três *bytes*. Neste caso, a cor branca será expressa pelas seguintes coordenadas RGB = (255, 255, 255) [12].

Uma sequência de vídeo é formada por um conjunto de imagens, as quais são exibidas em uma taxa determinada, algumas vezes com um número de imagens agrupadas, conforme ilustrado na Figura 4, (GOP – *Group of Pictures*) [7]. Cada imagem, também denominada de *frame*, é composta por vários grupos de blocos (GOB), comumente chamados de fatias. Cada GOB possui um número de macroblocos (MB) e cada MB é constituído de quatro blocos de luminância, 8X8 *pixels* cada, o qual representa a variação de intensidade e dois blocos de crominância, representando a cor da informação, conforme a Figura 4.

Considerando um vídeo com resolução *full HD* de duração de sessenta minutos, reproduzido a uma taxa de vinte e quatro *frames* por segundo, o mesmo será composto por 86400 *frames*. Cada *frame* ocupa em média 2MB (*mega bytes*). Multiplicando a quantidade de *frames* pelo tamanho médio de uma imagem e desconsiderando o volume do seu arquivo de áudio, obtêm-se o tamanho total do vídeo, cerca de 172,8 GB (*gigabytes*), o qual é altamente custoso para armazenamento. Isto constitui-se em uma forte justificativa a utilização de técnicas de compressão de modo a reduzir o seu tamanho.

A maioria dos algoritmos utilizados pelos padrões de codificação mais populares empregam uma combinação de transformada discreta de cosseno, um operação matemática de certa complexidade, e compressão de movimento. As duas técnicas têm por objetivo, respectivamente, reduzir a redundância espacial, semelhanças entre os *pixels* adjacentes de um *frame* e a redundância temporal, similaridade entre os *frames* sucessivos, de modo a reduzir o tamanho ocupado por um vídeo [7].

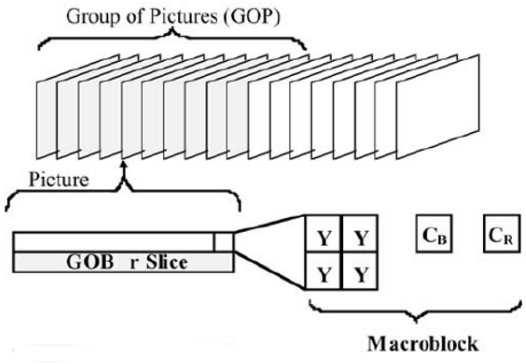

Figura 4: Estrutura de um vídeo digital [7]

Devido a sua natureza multidimensional, o processamento de imagens e vídeo possui características *data intensive*, necessitando de uma quantidade significante de espaço em disco bem como um elevado *quantum* no processador [5]. A título de exemplificação, apenas uma simples rotação de um vídeo HDTV (Figura 5), operação que consiste na modificação das coordenadas x,y dos *pixels* que compõem a imagem, requer taxas de uso da CPU da ordem de múltiplos de 10^{10} operações acumuladas por segundo [13]. Isto acontece porque eles são função de mais de uma variável, as quais são necessárias para determinar as coordenadas de um determinado ponto em uma figura.

Table A. Throughput rates for real-time processing of video in different formats.

Format	Image size (pixels)	Frame rate (Hz)	Throughput (Mpixels/sec)
PAL	720 × 576	25.0	10.4
NTSC	720 × 480	29.97	10.4
HDTV (SMPTE 260M)	1,920 × 1,080	30.0	62.2

Figura 5: Vazão em pixels/segundo [13]

A tabela da Figura 5 fornece o *througput*, vazão, em *Mpixels* por segundo de um processamento de vídeo em tempo real para diferentes tipos de formato [13]. Quanto maior a resolução, neste caso representado pelo formato HDTV de 1920X1080, maior é o *throughput*, em *Mpixels* por segundo e consequentemente maior será a fatia de tempo necessária no processador para que o processamento seja concluído.

C. TRABALHOS RELACIONADOS

No artigo, *Dynamic resource allocation and distributed video transcoding using hadoop cloud computing*, [14] os autores propõem um *software* para *transcoding* de vídeo,

em um ambiente de *cloud computer*, utilizando o paradigma *MapRecduce*, de qualquer formato para MPEG-4.

A proposta divide uma tarefa em dois *jobs MapReduce*. A etapa *Video Sequence Header MapReduce Job* possui por objetivo extrair as informações necessárias para decodificar o vídeo que se encontram armazenada apenas no primeiro bloco que o arquivo ocupa no HDFS. A fase *reduce* não é necessária e as informações são armazenadas no HDFS como um arquivo de texto. A etapa seguinte, *Video Decoder MapReduce Job*, utiliza a saída do *job* anterior para configurar o objeto *Video Decoder* para decodificar cada pedaço do vídeo e escrever os *frames* decodificados em um formato amigável ao Hadoop que possua os pares chave-valor. O *transcoding* dos *frames* para o formato MPEG-4 seria feito pelo Xuggler, uma API *open source* para codificação e decodificação de áudio e vídeo.

Segundo os autores, as técnicas aplicadas ao desenvolvimento de sua solução proporcionariam um sistema de *trancoding* veloz e inteligente devido a alocação dinâmica de recursos proporcionadas pelo ambiente de *cloud computer* [14]. No entanto devido a inexistência de dados referente ao número de *nodes* que compõe o *cluster*, bem como o tamanho do arquivo e o tempo gasto entre as duas fases de decodificação e a transcodificação do vídeo propriamente dita é difícil quantificar sua eficiência, apesar de tais dados serem relegados a trabalhos futuros. Também faltam maiores detalhes de como se daria a alocação dinâmica de recursos.

Em *An approach for fast and parallel video processing on Apache Hadoop clusters* [4], o autor tem como proposta utilizar o poder proporcionado pelo Hadoop para aplicar algoritmo de visão computacional visando detecção de face e rastreamento de movimentos. Esta solução utiliza como principais componentes o Fuse-DFS, um dos subprojetos do Hadoop, e duas bibliotecas de vídeo a OpenCV [15] e FFMPEG [16] bem como sua interface Java, o JavaCV. O HDFS oferece a estrutura de armazenamento enquanto o Fuse-DFS monta a partição distribuída como uma partição local e o JavaCV faz o *port* das outras duas bibliotecas para a linguagem Java.

A solução [4] é descrita com mais detalhes do que a [14], detalhando a quantidade de *nodes* utilizado no experimento, expressando em tabelas as respectivas diferenças de tempo de acordo com a quantidade de *nodes* utilizados. Porém o principal problema é o tamanho dos arquivos de vídeo utilizados no experimento, 2.6 MB, que é menor do que o tamanho padrão de um bloco no HDFS, sendo 64 MB na versão utilizada neste programa. Devido a isto, não é possível garantir que a solução funcionará para um arquivo de tamanho superior a 64 MB.

Uma das maiores dificuldades para processar um arquivo que não possui um tipo *built in* suportado pelo Hadood, que é o caso de um arquivo de vídeo, é que a implementação da classe responsável pela leitura do arquivo deve resolver o problema de quando ele inicia em um bloco e termina em outro. No caso de um arquivo de texto existe uma classe já disponibilizada pelo Hadoop [1]. Como todos os arquivos utilizados possuem 2.6 MB não é possível saber se o autor solucionou este problema.

O artigo *Scalable Traffic Video Analytics using Hadoop MapReduce* [17] tem por intuito oferecer uma solução computacional para análise de *stream* de vídeos em tempo real, de modo a detectar acidentes em rodovias com maior velocidade alertando hospitais e equipes de resgate praticamente no momento de sua ocorrência. Adicionalmente também verifica possíveis congestionamentos e rotas alternativas, focando na solução de problemas em tempo real.

Todas essas soluções têm alguns pontos em comum. O primeiro deles é a necessidade de criar um tipo customizado pois o Hadoop não dispõe nativamente de um *built in* capaz de efetuar a leitura de vídeo. Tanto a solução [14], quanto a [4] utilizam bibliotecas adicionais capazes de trabalhar com vídeo e imagens, respectivamente Xuggler [18] e JavaCV. A [17] não deixa claro qual a *lib* utiliza para atingir este fim.

O segundo ponto é que todas elas precisam decompor o vídeo em *frames* de modo a efetuar suas análises e o fazem de maneira distinta pois, cada um implementou a sua própria solução. De modo semelhante aos anteriores correlatos, este também necessita converter o vídeo em *frames*. Analizou-se a opção de utilizar uma solução análoga, a qual seria utilizar o JavaCV ou Xuggler para escrever uma classe que seria responsável pela leitura do vídeo armazenado no HDFS. No entanto, existe uma *lib open source*, HVPI[19], a qual utiliza o Xuggler[18], que já efetua esta tarefa, diminuindo consideravelmente o tempo de desenvolvimento o que permite ao desenvolvedor focar na solução de seu problema, que no caso seria a adição dos filtros de vídeo.

As soluções apresentadas não abordam as influências das configurações no desempenho de um *job* MapReduce processado em um *cluster* Hadoop. Nestas, o desempenho comparativo é feito pela quantidade de *nodes* utilizados para processar um determinado *job*. De maneira similar, nenhuma aplicação que resolve o mesmo problema de maneira não distribuída é levada em conta, ficando em dúvida o quanto estas abordagens são mais eficientes e adequadas que as soluções não distribuídas existentes no mercado.

Abaixo a tabela I possui a relação dos principais componentes das implementações:

Tabela I: Posicionamento entre as implementações

Componentes	1	2	3	4
JavaCV	**sim**	**não**	**sim**	**não**
Xuggler	**não**	**sim**	**não**	**sim**
Java	**1.7**	**N/I**	**N/I**	**1.7**
N° Nodes	**6**	**N/I**	**N/I**	**9**
Tamanho do Vídeo	**2.6 MB**	**N/I**	**N/I**	**>= 900MB**
Hadoop	**1.04**	**1.04**	**N/I**	**2.72**
FuseDFS	**sim**	**não**	**não**	**não**
MapReduce	**sim**	**sim**	**sim**	**sim**
Config. Hadoop	**não**	**não**	**não**	**sim**

1 An approach for fast and parallel video processing on Apache Hadoop clusters[4]
2 Dynamic r. a. and distributed video transcoding using hadoop cloud computing [14]
3 Scalable Traffic Video Analytics using Hadoop MapReduce [17]
4 Config. de A. Desempenho para P. de Vídeo num Cluster Hadoop

III. MÉTODOS E MATERIAIS

O primeiro passo na construção da solução foi realizar um estudo mais profundo sobre o Hadoop e efetuar a instalação da versão 2.7.2 em um pequeno *cluster* local constituído por três máquinas com as seguintes configurações: O *node*

master, responsável pelo gerenciamento dos demais nós do *cluster*, possui 8 GB de memória RAM, 1TB, e processador Intel Core i5, Sistema Operacional ArchLinux 64 bits com a versão 8 do java instalada; Os demais *nodes*, que constituem-se nos *slaves*, responsáveis pelo armazenamento e envio de informações ao *master node*, possuem respectivamente as seguinte configurações: *notebook* com 1TB, 4GB de RAM, Intel Core i3, S. Operacional Unbutu 14.04 e Java 8; *notebook* com 1TB, 4GB de RAM, intel Core i3, S.O manjaroLinux e Java 8.

Em seguida, testou-se o exemplo do *TeraSort*, um exemplo disponível junto com a instalação do Hadoop, de modo a verificar o seu correto funcionamento. Para criar um tipo customizado de modo a permitir ao Hadoop efetuar o processamento do vídeo, é necessário seguir os seguintes passos [20]:

1) Implementar a classe que será responsável pela serialização dos dados. Esta classe deve estender de *org.apache.hadoop.io.Writable*;
2) Implementar a classe que terá por atribuição gerar os pares chave/valor que serão encaminhados a fases *MapReduce*. No caso da presente implementação os valores correspondem aos *frames* e as chaves a ordem na qual o *frame* é extraído do vídeo. Esta classe deve estender de *org.apache.hadoop.mapreduce.RecordReader*. Os valores gerados serão no formato serializado customizado descrito no ítem anterior. Não é necessário, neste caso, criar um formato especializado para serializar as chaves, pois será utilizado um tipo *built in* disponível no Hadoop;
3) Criar a classe responsável por instanciar a classe customizada descrita anteriormente. Ela deve estender de *org.apache.hadoop.mapreduce.RecordReader*;
4) Criar as classes responsáveis pelas tarefas *map* e *reduce*, as quais devem estender respectivamente das classes *Mapper* e *Reducer*;
5) Escrever a classe que irá salvar o resultado da computação *MapReduce* no HDFS.

Optou-se pela utilização de uma *lib open source*, o HVPI (*Hadoop Video Processing Interface*) [19], que possui a maioria das classes customizadas necessárias ao processamento de vídeo. No entanto, esta biblioteca só aloca um *container*, independente se o arquivo de vídeo ocupa mais de um bloco do HDFS pois, o método *isSplitable()*, da implementação customizada da classe *FileInputFormat* retorna um *false* .

Ao alocar apenas um *container*, o arquivo será processado em um único *node*, e de maneira sequencial, perdendo o paralelismo. Isso acontece por que a compressão de alguns *codecs* de vídeo necessitam da informação do *frame* anterior para decodificar o próximo *frame*, o que seria um problema na criação do tipo customizado, pois inevitavelmente algum *frame* estará na região de fronteira entre os blocos do HDFS. Assim os desenvolvedores teriam que tratar caso a caso cada um dos formatos existentes e para manter a compatibilidade com a maior quantidade de formatos possíveis, eles escolheram esta abordagem. Para contornar esta limitação, o vídeo é previamente fragmentado pelo programa *ffmpeg* de modo a cada fração do arquivo gerada ser menor ou igual a um bloco do HDFS, de forma a eliminar o problema de um *frame* ocupar a fronteiras de blocos contíguos no HDFS.

Foi feito o *download* do HVPI e efetuou-se as configurações na IDE Eclipse, Mars.1 Release (4.5.1), para que a mesma pudesse funcionar. Um dos problemas na sua utilização é a sua dependência ao Xuggler [18], que não possui mais suporte e está sem atualizações. A maneira mais simples de resolver isso sem precisar recompilar o Xuggler foi adicionar o OpenImaj [21], outra *lib open source* para processamento multimídia, ao arquivo de configuração pom.xml do projeto, pois a OpenImaj vem com o Xuggler internamente.

Buscou-se as principais soluções disponíveis que possuíssem um bom *portifolio* de filtros de modo a aplicá-los aos *frames* extraídos. Os principais candidatos foram:

1) O MLT *framework* [22], um *software open source* extremamente completo, utilizado pelo *software* KDEnlive [22], um programa para edição de vídeo;
2) O ImageMagick, um *software* que de modo similar ao o MLT possui versões disponíveis para Java [23];
3) Marvin *Framework* [24], é uma aplicação com arquitetura baseada em *plugins*, novos filtros podem ser adicionados com a criação de novos *plugins*.

Tanto o MLT quanto o ImageMagick precisam do JNI pois não são nativamente compatíveis com Java. O MLT era o preferido devido aos filtros disponíveis. No entanto, em função da alta impedância entre os tipos necessários ao programa o mesmo precisou ser descartado. O ImageMagick seria a segunda opção. No entanto a sua *interface* Java, o Image4j [25] está desatualizado e alguns comandos não possuem compatibilidade com as novas versões do ImageMagick, a qual o Im4java apresenta dependência. Logo o Marvin *Framework* foi o escolhido. Ele é Java nativo e apresenta compatibilidade com os tipos de dados necessários ao processamento dos *frames* e possui um bom acervo de filtros disponíveis como *plugins*, que pode ser expandido.

IV. Otimização das Configurações do Hadoop

Intuitivamente, pode-se inferir que o modo mais simples de aumentar o poder de processamento de um *cluster* Hadoop é através da adição de mais *nodes*. Entretanto, nem sempre existirão *nodes* disponíveis e conhecer os gargalos inerentes a uma aplicação MapReduce, bem como quais os parâmetros que os impactam de maneira significativa, pode ser uma alternativa, sendo perfeitamente possível otimizar o desempenho por meio da customização de parâmetros.

O Hadoop possui um conjunto de arquivos responsáveis pelas suas configurações, dentre os quais podem ser destacados [1]:

1) core-site.xml - Arquivo responsável pelas configurações do núcleo do Hadoop, tais como operações de I/O, tamanho do bloco do HDFS, entre outros;
2) yarn-site.xml - Arquivo responsável pelas configurações do YARN. As informações contidas neste arquivo sobrepõe as configurações dos parâmetros *default*, contidas em yarn-default.xml;
3) mapred-site.xml.template - Armazena as configurações necessárias a execução de um *job* MapReduce, sobrepondo os parâmetros *default* disponíveis;
4) capacity-scheduler.xml - Possui as informações, utilizadas pelo *Capacity Scheduler*, necessárias a alocação de um *job*, de maneira simples e previsível, por

meio do conceito de filas. Inicialmente os *jobs* são alocados em filas, as quais possuem uma certa quantidade de recursos computacionais disponíveis. Estes recursos podem ser alocados para outras filas caso os mesmos não sejam utilizados, provendo elasticidade e alta disponibilidade ao *cluster* [26].

Um parâmetro é configurado de acordo com o seguinte modelo:

```
<property>
    <name>nome.da.propriedade</name>
    <value>valor.da.propriedade</value>
    <description>Campo opcional</description>
</property>
```

Para incrementar a *performance* de um *cluster* deve-se disponibilizar a maior quantidade possível de memória e poder de processamento aos seus *nodes*. Inicialmente é preciso informar as quantidades de memória e núcleos da CPU de cada uma das máquinas, por intermédio das propriedades yarn.nodemanager.resource.memory-mb e yarn.nodemanager.resource.cpu-vcores, disponível no arquivo yarn-site.xml [1].

Por meio dos parâmetros yarn.scheduler.minimum-allocation-mb, yarn.scheduler.maximum-allocation-mb, yarn.scheduler.minimum-allocation-vcores, yarn.scheduler.maximum-allocation-vcores, no arquivo yarn-site.xml, pode-se especificar as quantidades máximas e mínimas, de memória e núcleos, necessárias a alocação de um *container*. Já os parâmetros mapreduce.map.cpu.vcores, mapreduce.reduce.cpu.vcores, mapreduce.map.memory.mb e mapreduce.reduce.memory.mb, discriminam os recursos de um *container* que devem ser disponibilizados a um *job* MapReduce [1].

O modo como os recursos computacionais são alocados em um *container* é indicado em capacity-scheduler.xml, podendo ser especificado de duas formas: *DefaultResourcesCalculator* ou *DominantResourcesCalculator*.

No primeiro caso, *DefaultResourcesCalculator*, apenas a quantidade de memória disponível em um *node* é levada em conta para a alocação de um *container*. Por exemplo, caso um *node* possua 8GB de RAM, e quantidade de memória mínima a ser alocada seja 1GB, o YARN poderia alocar até oito *containers* para este *node*.

Quando os *containers* são alocados de acordo com *DominantResourcesCalculator* parâmetros multidimensionais, tais como memória e CPU, são considerados. Neste caso a quantidade mínima de núcleos de uma CPU, necessário para alocar um *container*, também é ponderada. Caso um *node* com 8GB de RAM, e 4 núcleos, tivesse como quantidade mínimas de recursos para memória e núcleos para a alocar um *container*, respectivamente 4GB e 2 núcleos, o YARN alocaria apenas dois *containers* por *node*.

Para determinar qual das propriedades acima é a mais adequada, deve-se levar em conta os recursos computacionais disponíveis em cada *node*. Quando for necessário maximizar as configurações de um *container*, a opção *DominantResourcesCalculator* é a mais adequada.

Após especificar as informações necessárias a alocação de um *container*, bem como os recursos que serão utilizados por um *job* MapReduce, existem parâmetros adicionais que podem ser setados de modo a reduzir o tempo de processamento em cada uma das etapas de um *job*.

As chaves, emitidas na fase *map*, são previamente ordenadas antes de serem enviadas a fase *reduce*, em um processo conhecido como *suffle*, conforme a Figura 6. Cada tarefa *map* possui um *buffer* no qual armazena o resultado de sua computação antes de escrevê-la em disco. Por *default*, este *buffer* possui 100MB e seu tamanho é definido através da propriedade mapreduce.task.io.sort.mb. Ao atingir um certo percentual, o qual é especificado em mapreduce.map.sort.spill.percent, o seu conteúdo é escrito no disco [1]. Antes da tarefa ser concluída, todo o conteúdo armazenado em disco é ordenado e aglutinado em um único arquivo, indicado na Figura 6 por *merge on disk*. Por padrão, é feito o *merge* de dez *streams* por vez, e esta quantidade pode ser modificada por meio da propriedade mapreduce.task.io.sort.factor [1].

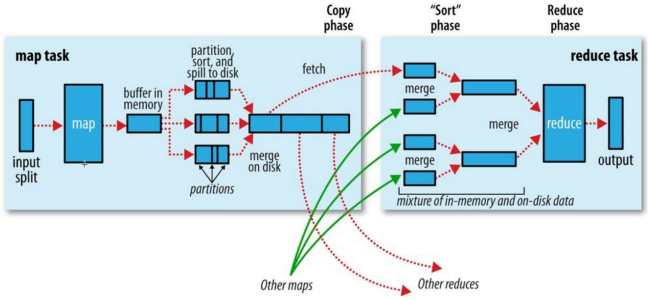

Figura 6: Suffle e ordenação [1]

Diante das operações de escrita em disco, bem como da ordenação dos pares chave-valor a fase de *suffle* é um dos processos onde se concentra um dos maiores gargalos de um *job*. De modo geral, deve-se fornecer ao *suffle* maior quantidade de memória possível, porém as fases *map* e *reduce* devem possuir uma quantidade suficiente de memória para funcionar. A quantidade de memória utilizada por uma JVM, *Java Virtual Machine*, para executar as tarefas de *map* e *reduce* são setadas em mapred.child.java.opts [1].

Para a maioria dos casos, pode-se presumir que o tamanho padrão do *buffer* no processo de *suffle* é suficiente. Isto porque a abstração lógica de um bloco do HDFS, o *input split* possui 128MB, que é relativamente próximo aos 100MB de um *buffer* padrão. Entretanto, para processar arquivos compactados, que é o caso de um arquivo de vídeo, faz-se necessário incrementar o tamanho deste *buffer*.

Em alguns casos pode ser interessante comprimir os dados de saída da fase *map*. Além de ser mais rápido, resultará em economia tanto na utilização de espaço em disco, quanto na largura de banda ao transferí-los ao *node* no qual a tarefa *reduce* será executada. Esta funcionalidade pode ser habilitada setando mapreduce.map.output.compress para *true*. Como consequência negativa, a economia em disco e largura de banda resultante da compactação, o arquivo compactado precisará de mais tempo para ser processado na fase *reduce* [1]. Processos CPU *bounded*, àqueles que requerem um maior tempo de uso

do processador, não serão beneficiados com esta abordagem, sendo mais adequada a processos IO *bounded*.

A propriedade mapreduce.tasktracker.http.threads é responsável pela quantidade de *threads* que enviam os arquivos de saída da fase *map* por meio do protocolo HTTP. Assim que os dados são disponibilizados pela fase *map*, o *reducer* iniciará a sua cópia. Como é necessário copiar os dados de vários *nodes*, o *reducer* executará esta tarefa em paralelo, indicado por intermédio das setas verdes na Figura 6. A quantidade de cópias feitas por um *reducer* é regida por mapreduce.reduce.shuffle.parallelcopies. Seu valor *default* é cinco, significando que um único *reducer* pode efetuar paralelamente a cópia de cinco arquivos de saída de uma tarefa *map* [1].

Adicionalmente, a existem outras propriedades que podem ser modificadas para incrementar o desempenho de um *reducer*. Pode-se, utilizando mapreduce.reduce.input.buffer.percent, aumentar a proporção de memória destinada a manter os arquivos de saída da fase *map* durante a fase *reduce* e até mesmo modificar, por meio do parâmetro mapreduce.reduce.shuffle.input.buffer.percent, o percentual do *buffer* destinado a alocar a saída da fase *map* durante a cópia do *suffle* [1].

O Hadoop fornece suporte a criação de arquivos de configuração customizados, os quais podem ser adicionados em tempo de execução. Isto é interessante quando é necessário optimizar certas configurações do *cluster* para melhorar o desempenho da aplicação. Tal recurso pode ser utilizado, nesta aplicação, através do seguinte comando:

```
1     Configuration conf = new Configuration();
2     conf.addResource(newPath(args[n]));
```

Onde *args[n]*, corresponde ao *path* no qual o arquivo se encontra no *master node*.

De modo adicional, também é possível setar as configurações diretamente a partir do programa, conforme o exemplo:

```
1 conf.set("mapreduce.map.cpu.vcores", "6");
```

Este comando diz para para o Hadoop ignorar o valor da propriedade mapreduce.map.cpu.vcores, definida nos arquivos de configurações dos *nodes*, e substituí-la pelo valor especificado pelo programa. No entanto se o valor especificado for superior aos valores especificados em yarn.nodemanager.resource.cpu-vcores yarn.scheduler.maximum-allocation-vcores, o *job* não será executado.

Em alguns caso é interessante especificar uma *combiner function* de modo a minimizar a quantidade de dados escrita em disco, bem como a que será transferida para o *reducer*. Uma *combiner function* é definida pela implementação de um *Reducer* e a mesma classe responsável pela etapa *Reduce* pode ser utilizada como um *combiner*. Para habilitá-la basta escrever o seguinte código no método responsável pelas configurações:

```
1 job.setCombinerClass(CustomReducer.class);
```

A. IMPLEMENTAÇÃO

A implementação tem por objetivo a adição de filtros aos *frames* de um vídeo através de processamento paralelo, utilizando o paradigma de programação *MapReduce* e o Hadoop como plataforma de computação distribuída e a posterior criação de um novo vídeo, em formato MPEG-4, a partir dos *frames* processados.

De modo a analisar a influência dos parâmetros das configurações no desempenho da aplicação, alguns deles serão customizados antes da execução do programa. Ao término, tem-se por intuito apresentar um gráfico indicando o tempo total de execução da aplicação em função dos parâmetros do Hadoop bem como um comparação entre esses resultados e àqueles obtidos com um comando que obtenha resultado similar utilizando o MLT. Vale ressaltar que o tempo total obtido também leva em consideração a cópia do vídeo criado para o HDFS.

A Figura 7 exemplifica o funcionamento de um *job MapReduce* para processamento de vídeo segundo a solução proposta. A primeira parte do processamento, segundo a abordagem da presente aplicação, depois de fragmentar o arquivo de vídeo e salvá-lo no HDFS, é a extração dos *frames*. Para isso, faz-se necessário a criação de um tipo customizado que terá como função efetuar a leitura do bloco a partir do HDFS, pois nativamente o Hadoop não possui um tipo *built in* que dê suporte a leitura de um arquivo de vídeo [20].

Optou-se por utilizar o HVPI [19], que vem com algumas classes já prontas que realizam as seguintes tarefas:

1) *XugglerReader.java* - decodifica o *stream* de *bytes* do arquivo de vídeo em *frames* para que possa ser gerado os pares chave-valor;
2) *VideoRecordReader.java* - é responsável por gerar os pares chave-valor, necessários ao paradigma *MapReduce*. Cada chave representa a ordem que um determinado *frame* ocupa dentro de uma dada sequência de vídeo. As chaves geradas são únicas porque na fase *reduce* será necessário recriar o vídeo e para isso é necessário ordená-los corretamente;
3) *MRVideoReader.java* - tem um *job MapReduce*, o qual pode ser customizado às necessidades do usuário. Neste caso, a fase *map* é utilizada para adicionar os filtros aos *frames* dos vídeos de modo a aproveitar o paralelismo, conforme indicado na Figura 7.

Para desenvolver este programa, além de customizar o *job MapReduce* disponibilizado pelo HVPI e efetuar as modificações necessária *VideoRecorReader.java*, foi necessário escrever as seguintes classes com as seguintes responsabilidades:

1) *OutputCustomFile.java* - responsável pela instância de VideoRecordWriter.java;
2) *VideoRecorWriter.java* - responsável pela escrita dos dados em disco;
3) *MediaWriter.java* - utiliza os *frames* extraídos na fase *map* para reconstruir o vídeo;
4) *SortingCustomComparator.java* - tem por função ordenar as chaves dos *frames* na ordem crescente ;
5) *FilterEfect.java* - adiciona os filtros a cada um dos *frames* extraídos na fase *map*.

Como a biblioteca HVPI não suporta arquivos maiores que um bloco do HDFS, foi necessário efetuar alguns artifícios para processar arquivos maiores. O primeiro deles foi fragmentar previamente o arquivo de vídeo de modo que cada um

dos arquivos geradas ocupasse no máximo um bloco do HDFS. Posteriormente para que todos os arquivos fossem processados de modo paralelo configuramos o *job* para utilizar múltiplas entradas. A maior desvantagem desta abordagem é a necessidade de digitar cada um dos arquivos que será processado na linha de comando. Em função da baixa complexidade computacional requerida para efetuar o *encode* do áudio com o vídeo, esta etapa é realizada localmente utilizando os comandos do *ffmpeg*.

Após extrair cada um dos *frames*, os filtros são aplicados individualmente a cada um deles durante a fase *map*. Efetuar a adição durante esta etapa é computacionalmente menos custoso pois, a fase *map* é executada em diversos *nodes* do *cluster*. Caso a aplicação dos filtros fosse feita na fase *reduce* a aplicação consumiria muito mais tempo, pois, neste caso existe apenas um único *reduce*. A fase *reduce* praticamente só efetua a iteração sobre os *frames*, emitindo os pares chave-valor para criação de um novo vídeo. Nesta parte as chaves são simplesmente ignoradas e os *frames* são utilizados para recriar o vídeo. As chaves são únicas para cada *frame*, representando a sua correta ordem no vídeo, sendo utilizada após a fase *map* para ordenar os *frames* de maneira correta.

A ordenação padrão do Hadoop não é satisfatória para este caso. Para que a ordem corretas dos *frames* fosse obedecida foi criado um arquivo customizado responsável pala ordenação. Os arquivos de vídeo fragmentados em tamanho correspondente a um bloco do HDFS, são armazenados em um diretório correspondente ao vídeo e cada uma de suas partes é convenientemente numerada em ordem crescente. O nome do vídeo é posteriormente concatenado com o nome da chave gerada. Estes arquivos de vídeo são numerados em ordem crescente, excluindo o número zero, de modo que o término do antecedente coincida com o início do próximo arquivo de vídeo. Isso tem por objetivo gerar uma chave única para cada *frame* de modo a colocá-los na ordem exata em que foram gerados no momento da recriação do vídeo.

As chaves dos *frames* são numerados por ordem de extração e a cada uma delas é adicionado um *token*, o nome do arquivo de vídeo, que corresponde ao arquivo ao qual ele pertence indicando a sua correta ordem. Por exemplo uma chave de número 1148 pertence ao arquivo 1.avi, enquanto a chave 45 representa o quinto *frame* de 4.avi. Embora o número 45 seja menor que 1148, neste caso é exatamente o contrário, pois cada um dos *frames* iniciará com um número que corresponde a ordem relativa entre os vídeos que estão sendo processados. Então a chave 1148 corresponde ao *frame* de número 148 pertencente ao arquivo 1.avi, do mesmo modo 45 corresponde ao quinto *frame* do arquivo 4.avi.

A comparação para determinar a ordem corretas dos *frames* é feita da seguinte maneira: primeiro é feita uma comparação entre os primeiros dígitos do número correspondente as chaves. Caso eles sejam iguais os *frames* pertencem ao mesmo vídeo e é feita uma comparação entre todos os dígitos que compõe as chaves em questão para determinar a ordem correta do *frame*. Caso sejam diferentes os *frames* pertencem a vídeos distintos, sendo necessário comparar apenas os primeiros dígitos.

O novo vídeo é salvo localmente no *namenode*, sendo copiado em seguida para o HDFS conforme ilustrado na Figura 7 . O áudio é adicionado posteriormente por meio do *ffmpeg*, devido a baixa complexidade computacional requerida. Esta é uma das partes que necessita de melhoria pois seria mais interessante salvar o arquivo diretamente no HDFS ao invés de efetuar uma cópia.

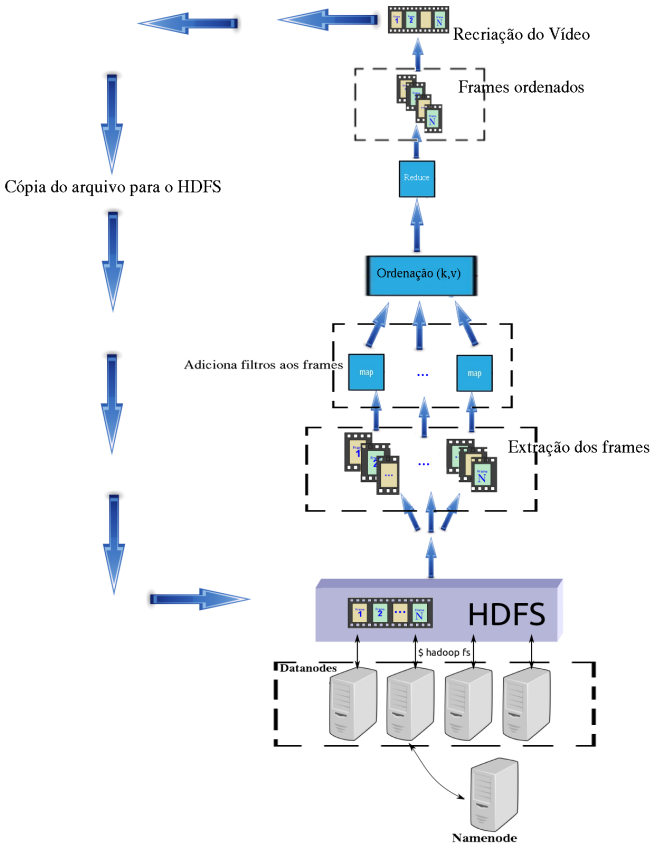

Figura 7: Adição de filtros aos *frames* de um vídeo por meio de um *job MapReduce*

O arquivo de vídeo que se deseja processar deve ser previamente copiado para o HDFS. Com o programa compilado, e no diretório corrente no qual o Hadoop encontra-se instalado executa-se o seguinte comando no terminal do Linux:

```
$ ./bin/hadoop/jar nome-arquivo.jar /saida
    org.marvinproject.image.color.
    nomeDofiltro /entrada/video.avi
```

Onde *entrada* é o diretório no HDFS no qual o arquivo encontra-se armazenado e *saida* é o diretório em que serão armazenados o resultado da computação *MapReduce*, e *nome-filtro* pode ser substituído pelos nomes dos *plugins* do Marvin. Pode-se adicionar vários filtros, bastando para isso colocar o símbolo -"separando cada um deles. Caso o diretório especificado como a saída já exista no HDFS, o Hadoop irá lançar uma exeção do tipo *FileAlreadyExistsException* e finalizar a execução do programa.

Os *plugins* do Marvin devem ser armazenados, antes de executar o comando, no mesmo diretório em que o Hadoop encontra-se instalado no *namenode* e o seu *download* pode ser encontrados no site do desenvolvedor [27].

Para criar um *job MapReduce* as classes devem herdar de Mapper.java e Reducer.java e implementar os seus respectivos métodos abstratos: *public void map(chave, valor)* e *public void reduce(chave, valor)*. A classe *MRVideoReader.java* possui duas *inner classes* estáticas que herdam de *Mapper.java* e *Reducer.java* [1] conforme pode ser visto no trecho de código da implementação da classe *MRVideoReader.java*. Cada uma das classes filhas de *Mapper.java* e *Reducer.java* devem especificar quais são os pares chave-valor que servem como entrada e saída. Os pares chave-valor recebidos como entrada e emitidos como saída das fases *map* e *reduce* são idênticos, correspondendo aos tipos *Text* e *ImageWritable* , conforme a implementação.

O tipo *Text* é um tipo nativo suportado pelo Hadoop, enquanto o valor *ImageWritable* é um tipo customizado escrito pelo time de desenvolvedores do HVPI. A chave é o elemento pelo qual os *frames* serão ordenados. Os *frames* são representados pelo valor *Imagewritable*, que é responsável pela serialização e deserialização dos dados dos *frames* através da rede.

No método *run()* de *MRVideoReader.java* efetua-se algumas configurações do *job*. É nele por exemplo que especifica-se quais classes representam os pares chave-valor como entrada e saída das fases *map* e *reduce*, a classe responsável pela ordenação das chaves, que no caso da presente aplicação também precisou ser customizada, bem como os diretórios de *input*, que neste caso contém o arquivo de vídeo, e o de *output*, o qual armazenará o resultado da computação *MapReduce*, localizados no HDFS. Neste, os parâmetros passados pela linha de comando que contém as informações necessárias a seleção do filtro que será utilizado durante a computação são adicionado:

```
Configuration conf = new Configuration();
conf.set("filter", args[1]);}
```

Depois de setado, o filtro é utilizado na classe *MRVideoReaderMapper.java* no método *public void map(Text key, ImageWritable value, Context context)* através dos seguintes comandos:

```
String filterName = context.getConfiguration()
    .get("filter");
FilterEffect.getInstance().setFilters(
    filterName);
FilterEffect.getInstance().addFilter(value.
    getBufferedImage());
```

A primeira linha do trecho de código acima recupera o tipo do filtro digitado pelo usuário. Na segunda, o tipo, ou tipos, de filtro são setados. Já comando *value.getBufferedImage()* retorna o *frame* sobre o qual será adicionado o filtro.

Num primeiro momento verificou-se que os *frames* extraídos pelo HVPI eram colocados fora de ordem o que comprometia a recriação do arquivo de vídeo. Para resolver este problema foi necessário escrever uma classe que seria responsável pela ordenação e setá-la em *job.setSortComparatorClass()*, na classe *MRVideoReader.java*, no seu método *run()*.

Para que o *job* seja iniciado é necessário setar uma série de parâmetros, tais como os diretórios do HDFS que contém os arquivos que serão processados, o diretório de saída, criado em *run time*, no qual será armazenado os resultados da computação, as classes que contém as implementações Mapper e Reducer utilizadas na execução do *job*, quais os tipos que serão utilizados como pares chave-valor, entre outros. Todas essas configurações, assim como uma diversidade de outros parâmetros, são customizadas no método *run()*, da classe MRVIdeoREader.java.

```
/*
* Implementacao das classes responsaveis pelo
    job mapreduce
*/
public class MRVideoReader extends Configured
    implements Tool{

    public static class MRVideoReaderMapper
        extends Mapper<Text, ImageWritable,
        Text, ImageWritable>{

        public void map(Text key,
            ImageWritable value, Context
            context)throws IOException,
            InterruptedException {

            // Adiciona filtros aos frames de
                v deo
            }
        }

    public static class MRVideoReaderReducer
        extends Reducer<Text, ImageWritable,
        Text, ImageWritable>{

        public void reduce(Text key, Iterator<
            ImageWritable> values, Context
            context) throws IOException,
            InterruptedException {
        //Emite os pares chave-valor

            }
        }
    }

    public static void main(String[] args)
        throws Exception{
         int res = ToolRunner.run(new
             MRVideoReader(), args);
            System.exit(res);
    }
    @Override
    public int run(String[] args) throws
        Exception {
        //Configuracoes do job
    }
}
}
```

Após a fase *reduce*, os *frames* previamente ordenados são utilizados para criar um novo vídeo em formato MPEG-4. Para isso utiliza-se um tipo customizado responsável por esta operação de I/O, que foi escrito exclusivamente para esta finalidade. Segundo o *framework* disponibilizado pelo Hadoop, esta classe deve herdar de *RecordWriter.java*, especificar o par chave-valor, neste caso, *Text-Imagewritable* e implementar os métodos abstratos *public void close(...)* e *public void write(...)*

[20] [28].

Para implementar um *output format* customizado deve-se seguir um padrão do tipo herança/delegação e implementar os métodos das classes disponibilizadas pelo *framework* do Hadoop [28]. Criou-se as classes *OutputCustomFile* e *VideoRecordWriter*, que herdam respectivamente de *OutputFormat* e *RecordWrite*. A implementação do método *getRecordWriter()*, da classe *OutputCustomFile* é responsável pelo retorno da instância da classe *VideoRecordWrite*, a qual efetua a escrita do vídeo. No método *write(...)* da classe *VideoRecordWrite* a criação e persistência do vídeo em disco é efetuada pela classe *frames*, que será utilizado para criar o vídeo.

V. Avaliação Experimental

Para avaliar o desempenho da aplicação utilizou-se um *cluster* composto por nove máquinas, sendo oito *nodes* e um *master node*, com as seguintes configurações:

1) Sistema Operacional ArchLinux 64 bits;
2) Processador Intel i5 (*quad core*);
3) 4 GB de RAM;
4) Java 7;

O arquivo de vídeo utilizado nos testes possui 952,3MB. Este, possui extensão AVI e foi dividido em sete pedaços. Em termos de desempenho, o inconveniente da necessidade de particionar o vídeo antes de salvá-lo no HDFS é que os pedaços dos vídeos não são do mesmo tamanho, resultando em um desbalanceamento de carga nos *nodes*.

O filtro *gray scale* será utilizado como padrão durante a sintonia do *cluster*. A partir da identificação dos valores das configurações que obtiverem o menor tempo de processamento, será avaliada a influência de outros fatores no desempenho da aplicação tais como tipo de filtros e extensão do arquivo. Para efeito comparativo, o tempo de processamento em um único *node*, utilizando os comandos do MLT, por meio dos quais se obtém um resultado equivalente, será considerado o limite a ser superado pela aplicação distribuída.

Inicialmente serão mantidos a maioria das variáveis com seus valores *default* e a partir deste resultado inicial, as grandezas deste parâmetros serão customizadas visando reduzir o tempo final de processamento. Durante a avaliação de desempenho a quantidade dos *nodes* será mantida constante. A maioria das mudanças será realizada em mais de uma variável, visto que muitas propriedades são correlacionadas entre si e o tempo necessário para avaliar cada uma delas individualmente seria muito maior.

Por padrão, devido a grande quantidade pares chave-valor geradas provenientes da descompactação dos *frames* dos arquivos de vídeo, um *combiner* é especificado de modo a minimizar a largura de banda, assim como a quantidade de dados a ser transferida para a etapa *Reduce*.

Alguns dos parâmetros *default* do Hadoop são impeditivos ao início do experimento. Isso se deve aos seus valores iniciais serem superiores aos disponíveis nos *hardwares* dos *nodes*, o que levaria a finalização da tarefa de maneira prematura pelo *application manager*. Por este motivo foi necessário efetuar uma configuração preliminar que contemplasse os requisitos de *hardware* disponíveis nos *nodes* do *cluster*.

Inicialmente as propriedades responsáveis pela definição das quantidades de memória física do *node*, limite máximo e mínimo de memória, e as quantidades de núcleos virtuais disponíveis e os necessários a alocação de um *container* foram modificadas, bem como a limitação da memória virtual dos *containers* e a redução do tempo limite das tarefas, o *time out*, foram modificadas conforme a tabela II. O tempo obtido com estas configurações foi de 2h17min46s (duas horas, dezessete minutos e quarenta e quarenta e seis segundos), correspondente a 8226s (oito mil duzentos e vinte e seis segundos). Para efeitos de comparações no gráfico 8, esta configuração será denominada Configuração 1, seguindo esta ordem para as demais.

Tabela II: Valores customizados da Configuração 1

Propriedade	Valor Default	Valor Customizado	Função
yarn.nodemanager.resource.memory-mb	**8192 MB**	**4096 MB**	**memória fisica**
yarn.scheduler.minimum-allocation-mb	**1024 MB**	**1024 MB**	**qtd mímima alocada**
yarn.scheduler.maximum-allocation-mb	**8192 MB**	**4096 MB**	**qtd máxima alocada**
yarn.nodemanager.resource.cpu-vcores	**8**	**5**	**qtd núcleos virtuais**
yarn.scheduler.minimum-allocation-vcore	**1**	**1**	**qtd mínima alocada**
yarn.scheduler.maximum-allocation-vcores	**32**	**4**	**qtd máxima alocada**
mapreduce.task.timeout	**6000000 ms**	**600000 ms**	**Tempo de resposta**
yarn.nodemanager.vmem-check-enabled	**true**	**false**	**Limite a m. Virtual**

Na Configuração 2 foram considerados os parâmetros da tabela III:

Tabela III: Valores customizados da Configuração 2

Propriedade	Valor Default	Valor Customizado	Função
yarn.nodemanager.resource.memory-mb	**8192 MB**	**4096 MB**	**memória fisica**
yarn.scheduler.minimum-allocation-mb	**1024 MB**	**1024 MB**	**qtd mímima alocada**
yarn.scheduler.maximum-allocation-mb	**8192 MB**	**4096 MB**	**qtd máxima alocada**
yarn.nodemanager.resource.cpu-vcores	**8**	**5**	**qtd núcleos virtuais**
yarn.scheduler.minimum-allocation-vcore	**1**	**1**	**qtd mínima alocada**
yarn.scheduler.maximum-allocation-vcores	**32**	**4**	**qtd máxima alocada**
mapreduce.task.timeout	**6000000 ms**	**600000 ms**	**Tempo de resposta**
mapreduce.map.java.opts	**-Xmx1000m**	**-Xmx3300m**	**heap space map**
mapreduce.reduce.java.opts	**-Xmx1000m**	**-Xmx3300m**	**heap space reduce**
mapreduce.map.memory.mb	**1024**	**4096**	**m. virtual map**
mapreduce.map.memory.mb	**1024**	**4096**	**m. virtual reduce**

As modificações seguem os parâmetros das configurações 1 e 2, adicionando as propriedades mapreduce.map.java.opts, mapreduce.reduce.java.opts, mapreduce.map.memory.mb e mapreduce.reduce.memory.mb . Estas, especificam, respectivamente, os valores do *heap size memory* e a quantidade de memória virtual reservada para as tarefas de *map* e *reduce*. O tempo total o obtido com esta configuração foi de 1h40m35s (uma hora, quarenta minutos e trinta e cinco segundos) ou 6035s (seis mil e trinta e cinco segundos), cerca de 26,63% mais rápido do que a primeira configuração.

Na Configuração 3 foram mantidos todos os valores dos parâmetros anteriores e modificou-se a maneira como um *container* é alocado em um *node*. A alocação é feita de acordo

Tabela IV: Valores customizados da Configuração 3

Propriedade	Valor Default	Valor Atual	Função
yarn.scheduler.capacity.resource-calculator	**Default**	**Dominant**	**Alocação dos containers**

com a propriedade yarn.scheduler.capacity.resource-calculator, no arquivo capacity-scheduler.xml.

Por padrão seu valor é *DefaultResourceCalculator* e utiliza apenas o critério referente a memória para alocar um *container*. A propriedade foi modificada para *DominantResourceCalculator* de modo a considerar também como critério a quantidade de núcleos requeridos por um *job* MapReduce. Como resultado obteu-se 1h35m23s (uma hora, trinta e cinco minutos e vinte e três segundos) ou 5723s (cinco mil setecentos e vinte e três), 30,42% mais eficiente em relação a primeira configuração.

Preservando as alterações efetuadas na configuração anterior, modificou-se os parâmetros referentes às quantidades de núcleos virtuais utilizados em uma tarefa MapReduce. Na tabela V é informados as propriedades modificadas e os seus valores de referência. O tempo obtido foi igual a 1h31mins9s (uma hora, trinta e um minutos e nove segundos), 33,39% inferior ao obtido na primeira tentativa.

Tabela V: Valores customizados da Configuração 4

Propriedade	Valor Default	Valor Atual	Função
mapreduce.map.cpu.vcores	**8**	**4**	**núcleos virtuais map**
mapreduce.reduce.cpu.vcores	**8**	**4**	**núcleos virtuais reduce**

De modo a incrementar os resultados obtidos no teste anterior, na Configuração 5, buscou-se reduzir o tempo de acesso ao disco utilizado pela aplicação. A variável mapreduce.task.io.sort.mb é responsável pelo tamanho do *buffer* utilizado pelas tarefas MapReduce antes de serem escritas em disco. Quando este *buffer* atinge um determinado percentual o seu resultado é escrito em disco e este valor é determinado por mapreduce.map.sort.spill.percent. A tabela VI contém as propriedades e suas respectivas alterações. Com estas modificações o *job* levou 1h25m9s (uma hora, vinte e cinco minutos e nove segundos), 37,89% mais eficiente.

Tabela VI: Valores customizados da Configuração 5

Propriedade	Valor Default	Valor Atual	Função
mapreduce.task.io.sort.mb	**100 mb**	**1024 mb**	**tamanho do buffer**
mapreduce.map.sort.spill.percent	**80%**	**90%**	**percentual do buffer**

A próxima configuração, a de número 6, tem por objetivo reservar mais recursos ao *application master*, um dos *node*, que está configurado em modo *slave*, reservado pelo YARN para gerenciar a aplicação. O tempo total obtido para o processamento do vídeo foi de 1h14010s (uma hora, quatorze minutos e dez segundos), valor 45,09% mais rápido que a o primeiro teste.

O gráfico da Figura 8 ilustra a variação do tempo de processamento em função das modificações de parâmetros das configurações.

Tabela VII: Valores customizados da Configuração 6

Propriedade	Valor Default	Valor Atual	Função
yarn.app.mapreduce.am.resource.mb	**1536 mb**	**3072 mb**	**Memória do A. Master**
yarn.app.mapreduce.am.command-opts	**-Xmx1024m**	**-Xmx3096m**	**Heap da JVM reservada**

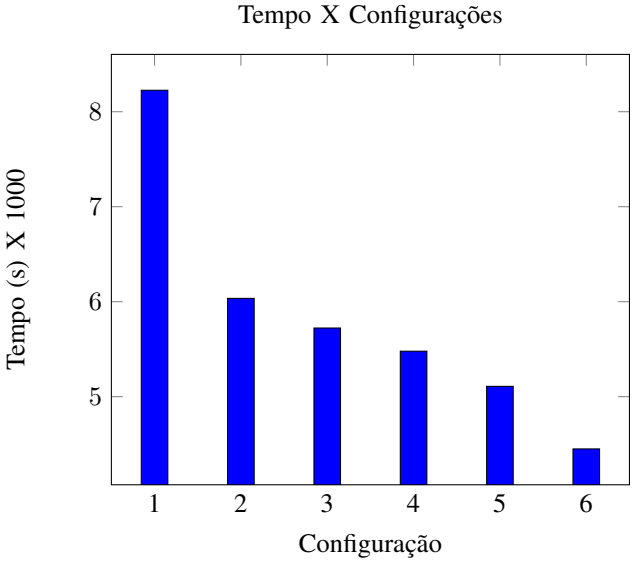

Figura 8: Variação do tempo em função das configurações

De maneira a converter um vídeo em formato *avi* para *mp4* e adicionar o filtro *grayscale* em um único *node* utilizou-se o seguinte comando:

```
$ time input.avi -filter grayscale -consume
    avformat:output.mp4
```

E saída foi:

```
real 30m51.599s
user 105m2.966s
sys  0m42.2
```

Para obter o tempo total consumido pela aplicação não distribuída somamos *user* e *sys*, o qual foi de 1h45min44s (uma hora, quarenta e cinco minutos e quarenta e quatro segundos) ou 6344s (seis mil trezentos e quarenta e quatro segundos). Verificando este valor em relação ao menor tempo obtido pela aplicação distribuída calcula-se que a aplicação distribuída foi 29,85% mais rápida.

Tabela VIII: Tempo obtido pelas aplicações

Tipo de aplicação	Quantidade de nodes	Duração do processo (s)
Single node	**1**	**6.344**
Multinode	**9**	**4.450**

VI. Conclusão

A necessidade de criação de um tipo customizado constitui-se em um dos maiores desafios do desenvolvimento de uma

aplicação *MapReduce* para processamento de vídeos. Em alguns casos os tipos disponibilizados pelo Hadoop são incompatíveis com o qual o programador deseja representar. Uma vez que o Hadoop não oferece suporte nativo ao processamento de vídeo e imagens fez-se necessário escrevê-los, de modo a compatibilizá-lo com o paradigma *MapReduce*. No caso desta aplicação optou-se por utilizar uma biblioteca *open source*, a qual pudesse simplificar o desenvolvimento da aplicação.

Apesar da maioria dos arquivos de vídeo possuir uma estrutura que, teoricamente, facilite o desenvolvimento de aplicações distribuídas, criar um tipo customizado que contemple a grande diversidade de formatos de vídeos não é algo muito simples. Isto porque o programador deve tratar as possibilidades envolvendo a quebra de um *frame* em blocos contíguos do HDFS. Como existe divergência na maneira como alguns formatos de vídeo armazena os seus *frames* em decorrência do tipo de compactação utilizada, é complicado desenvolver uma solução genérica o suficiente que funcione para os vários tipos de formatos disponíveis no mercado. A solução foi particionar o arquivo de vídeo previamente antes de enviá-lo ao HDFS de modo que cada fração do arquivo ocupasse no máximo um bloco, eliminando assim a necessidade de lidar com este problema.

O Hadoop possui uma grande variedade de parâmetros que podem ser personalizados. Contudo, em virtude do pouco tempo disponível, bem como alguns problemas de infraestrutura, tais como indisponibilidade de computadores e instabilidade na rede, não foi possível averiguar a influência de uma maior gama de parâmetros durante um processamento de vídeo em um *cluster* Hadoop, nem efetuar todas as comparações desejadas, tais como diferentes tipos de filtros, se a influência no tipo de formato de vídeo processado é significativa e adição de mais *nodes*. Todavia, o resultado obtido foi de acordo com o esperado, apresentando desempenho ligeiramente superior ao de um arquivo processado em modo não distribuído.

A customização dos parâmetros do Hadoop mostrou-se ser o fator determinante a melhoria de desempenho da aplicação. Somente modificando as propriedades disponibilizadas foi possível otimizar a performance do sistema em 45% em relação as configurações *default* e ser 29,85% superior a um programa similar não distribuído.

Devido a grande diversidade de parâmetros personalizáveis disponíveis no Hadoop, este trabalho não teve por objetivo esgotar todas as possibilidades possíveis, podendo inclusive ser possível otimizar ainda mais o desempenho desta aplicação. Ainda assim, os resultados obtidos evidenciam uma melhoria significativa do desempenho mediante a customização dos parâmetros, sem a necessidade de adição de um novo *node*.

VII. Limitações

O HVPI não foi projetado para lidar com arquivos maiores que um bloco do HDFS, sendo necessário adotar algumas estratégias. Esta biblioteca também é fortemente dependente do Xuggler, que apesar de ser um dos melhores *wrappers* FFMPeg para linguagem Java, é uma *lib deprecated* e não tem atualizações a muitos anos.

A presente aplicação não apresenta suporte a alguns dos filtros disponibilizados pelo Marvin, pois os mesmos são parametrizados, sendo necessário refatorar o código de modo a disponibilizar o suporte.

O vídeo criado não é salvo diretamente no HDFS e sim no disco local no qual a tarefa *reduce* foi executada, sendo necessário copiá-lo para o HDFS.

Devido a limitação da *lib* citada anteriormente, no caso da adição de um novo *node* é necessário particionar novamente o arquivo e depois transferi-lo para o HDFS caso contrário não será possível melhorar o desempenho da aplicação. A quantidade dos pedaços dos vídeos devem ser proporcionais a quantidade de *nodes* disponíveis e cada pedaço deve ser digitado na linha de comando.

VIII. Trabalhos Futuros

As seguintes sugestões poderiam ser levadas em considerações em possíveis trabalhos futuros:

1) A eficiência do sistema poderia ser aumentada mediante a implementação de múltiplos *reducers*, pois nesta aplicação esta etapa não se beneficia do paralelismo do sistema e apenas um *node* é responsável por compactar todos os *frames* em um novo arquivo de vídeo;
2) Averiguar a relevância de outros tipos de filtros, a influência da combinação deles, bem como dos diferentes tipos de formatos, no desempenho do *cluster*;
3) Solucionar a limitação da biblioteca HVPI de modo a dispensar o particionamento prévio do arquivo de vídeo;
4) Criar uma aplicação distribuída para processamento de vídeo em outro paradigma de programação suportado pelo ecossistema do Hadoop e comparar o seu desempenho com o programa atual que utiliza o paradigma *MapReduce*, com o objetivo de definir qual dos modelos é o mais eficiente para processamento de vídeo distribuído;
5) Otimizar a aplicação mediante a adição dos filtros de vídeo diretamente sobre o *stream* de *bytes*, eliminando a necessidade de descompactar e recompactar o arquivo.

Referências

[1] T. White, *Hadoop the definitive Guide*, 4th ed. O'Reilly, 2009, vol. 1, páginas 19-246.

[2] R. R. Linhares, "Contribuição para o desenvolvimento de uma arquitetura de computação própria ao paradigma orientado a notificações," p. 90, 2015.

[3] "How much data does x store," "https://techexpectations.org/tag/how-much-data-does-youtube-store/", (Acessado em: 15 de Setembro, 2016).

[4] H. Tan, "An approach for fast and parallel video processing on apache hadoop clusters," *Researchgate*, pp. 1–6, 2014.

[5] B. Al, *Handbook of Image and Video Processing*. Academic Press, 2000, página 5.

[6] A. Shahriar, "Digital video, concepts, methods and metrics," p. 165, 2014.

[7] S. Yun, Q and S. Hui, Yang, "Multimedia image and video processing," *CRC press*, pp. 19–25, 2001.

[8] J. Shirinivas, "Hadoop performance tunning guide," "http://www.admin-magazine.com/HPC/Vendors/AMD/Whitepaper-Hadoop-Performance-Tuning-Guide", (Acessado em: Julho 07, 2017).

[9] "Yahoo! hadoop tutorial," "https://developer.yahoo.com/hadoop/tutorial/module5.html/", (Acessado em: 15 de Setembro, 2016).

[10] P. I, “Digital image processing algorithms and applicaions,” pp. 1–5, 2000.

[11] “The sampling theorem,” "http://www.dspguide.com/ch3/2.htm", (Acessado em: 10 de Fevereiro, 2017).

[12] C. A. Poynton, “A technical introduction to digital video,” vol. 1, pp. 1–31, 1996.

[13] H. Simon, D, J. Stone, L. Wayne, Cheung, and K. Peter, Y, “Video image processing with the sonic architecture,” *CoverFeature*, p. 51, 2000.

[14] S. B, R and C. Narayanan, Prakash, “Dynamic resource allocation and distributed video transcoding using hadoop cloud computing,” *IEE*, pp. 1–6, 2014.

[15] “Opencv open source computer vision,” "http://opencv.org/", (Acessado em: 15 de Setembro, 2016).

[16] “Ffmpeg a complete, cross-platform solution to record, convert and stream audio and video,” "https://www.ffmpeg.org/", (Acessado em: 15 de Setembro, 2016).

[17] A. Natarajan, Vaithilingam, J. Subbaiyan, and N. Gudivada, Venkat, “Scalable traffic video analytics using hadoop mapreduce,” *AllData*, pp. 1–5, 2015.

[18] “Xuggle,” "http://www.xuggle.com/xuggler/", (Acessado em: 15 de Setembro, 2016).

[19] “Hvpi github repository,” "https://github.com/xmpy/hvpi/", (Acessado em: 15 de Setembro, 2016).

[20] G. Thilina, *Hadoop MapReduce Cookbook V2, Explore the Hadoop MapReduce Ecosystem to Gain Insights from Very Big Data Sets*. Livery Place 35 Livery Street Birmingham B3 2PB, UK: Packt Publisher, 2015, vol. 2, páginas 252-302.

[21] “Openimaj,” "https://openimaj.org/", (Acessado em: 10 de Fevereiro, 2017).

[22] “Mlt framework,” "https://www.mltframework.org/", (Acessado em: 16 de Setembro, 2016).

[23] “Imajmagick,” "https://www.imagemagick.org/script/index.php/", (Acessado em: 10 de Fevereiro, 2017).

[24] A. Guilherme, F. Andrijauskas, and D. Muñoz, “Marvin–a tool for image processing algorithm development,” pp. 5–6, 2008.

[25] “Image4j,” "http://image4j.sourceforge.net/", (Acessado em: 10 de Fevereiro, 2017).

[26] “Understand hadoop capacity scheduller,” "http:/https://br.hortonworks.com/blog/understanding-apache-hadoops-capacity-scheduler/", (Acessado em: 20 de julho, 2017).

[27] A. Guilherme, “Marvin editor,” "http://marvinproject.sourceforge.net/en/download.html", (Acessado em: 10 de Fevereiro, 2017).

[28] J. R. Owens, B. Fenians, and J. Lentz, *Hadoop Real-Word Solutions Cookbook, Realistic, Simple Code Examples to Solve Problems at Scale with Hadoop and Related Tecnologies*. Packt Publisher, vol. 1.