
**Universidade do Estado do Rio de Janeiro**

Centro de Tecnologia e Ciências

Instituto de Matemática e Estatística

Felipe Schimith Batista

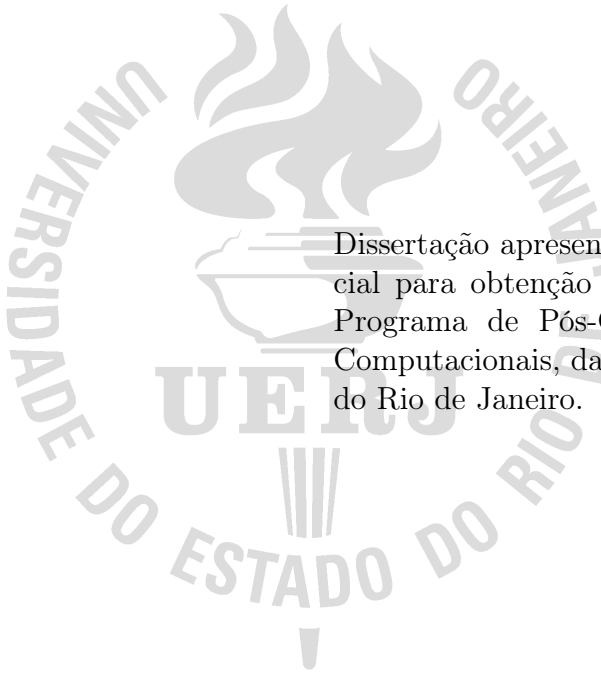
# Computação distribuída aplicada ao processamento de imagens de minérios utilizando o Hadoop

Rio de Janeiro

2017

Felipe Schimith Batista

# Computação distribuída aplicada ao processamento de imagens de minérios utilizando o Hadoop




Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Orientadores: Prof. Dr. Guilherme Lucio Abelha Mota
Prof. Dr. Gilson Alexandre Ostwald Pedro da Costa

Rio de Janeiro

2017

CATALOGAÇÃO NA FONTE
UERJ / REDE SIRIUS / BIBLIOTECA CTC-A

B333 Batista, Felipe Schimith

Computação distribuída aplicada ao processamento de imagens de minérios utilizando o Hadoop / Felipe Schimith Batista. - 2017.

116 f. :il.

Orientadores: Guilherme Lucio Abelha Mota e Gilson Alexandre Ostwald Pedro da Costa.

Dissertação (Mestrado Ciências Computacionais) - Universidade do Estado do Rio de Janeiro, Instituto de Matemática e Estatística.

1. Processamento de imagens - Teses. 2. Programação paralela (Computação) - Teses. 2. I. Mota, Guilherme Lucio Abelha. II.Costa, Gilson Alexandre Ostwald Pedro da Costa. III. Universidade do Estado do Rio de Janeiro. Instituto de Matemática e Estatística. IV.Título.

CDU 004.932



_____ _____

Assinatura Data

Felipe Schimith Batista

# Computação distribuída aplicada ao processamento de imagens de minérios utilizando o Hadoop

Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Aprovada em 29 de Março de 2017.

Banca Examinadora:

---

Prof. Dr. Guilherme Lucio Abelha Mota (Orientador)
Instituto de Matemática e Estatística - UERJ

---

Prof. Dr. Gilson Alexandre Ostwald Pedro da Costa (Orientador)
Instituto de Matemática e Estatística - UERJ

---

Prof. Dr. Otávio da Fonseca de Martins Gomes
Centro de Tecnologia Mineral

---

Dr. Rodrigo da Silva Ferreira
IBM Research Brazil

---

Prof. Dr. Leandro Augusto Justen Marzulo
Instituto de Matemática e Estatística - UERJ

Rio de Janeiro

2017

# DEDICATÓRIA

Dedico esse estudo aos meus pais, por me fazerem entender que educação é a base de tudo, pelo incentivo e por todo o esforço que fizeram para que eu chegasse até aqui.

À Aline que me fez enfatizar esse entendimento, esteve comigo nos momentos mais difíceis desta trajetória e me deu forças para continuar.

## AGRADECIMENTOS

A construção desta dissertação é fruto de um trabalho coletivo e só foi possível com apoio, orientação, amizade e dedicação de várias pessoas a quem serei eternamente grato:

Aos meus orientadores, Dr. Guilherme Lucio Abelha Mota e Dr. Gilson Alexandre Ostwald Pedro da Costa, exemplo de profissionais, pela confiança depositada quando eu mesmo tinha dúvidas se iria conseguir, pelos ensinamentos valiosos na construção deste trabalho, pela paciência e incentivo em todos os momentos.

Ao Dr. Otávio da Fonseca de Martins Gomes pela disponibilização de dados essenciais para o estudo, pelas inestimáveis contribuições na área de mineralogia, durante a realização do estudo e na banca de qualificação.

Ao aluno de graduação Renan Bides por ter me auxiliado na instalação do *cluster*.

Aos professores e pesquisadores da pós-gradução do IME/UERJ, por todo conhecimento transmitido nessa trajetória.

Aos colegas de trabalho e também amigos, pelo apoio e compreensão em todas as fases do curso.

À Deus, por ter me possibilitado viver esse desafio!

## RESUMO

BATISTA, Felipe Schimith **Computação distribuída aplicada ao processamento de imagens de minérios utilizando o Hadoop.** 116 f. Dissertação (Mestrado em Ciências Computacionais) - Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2017.

O desenvolvimento de soluções de processamento paralelo baseado no *framework* Hadoop tem sido amplamente utilizado como uma alternativa eficiente para o processamento de grandes volume de dados. O uso de ferramentas de busca tornou-se ainda mais atraente e difundido na área de tecnologia com a metodologia de MapReduce. Entretanto, podemos adequar essa arquitetura para processamento de imagens, com a finalidade de obter ganhos no tempo e volume de processamento com o modelo de distribuição dos conteúdos do *Hadoop Distributed File System* (HDFS). Nesse contexto, o objetivo desse estudo foi desenvolver uma arquitetura capaz de distribuir o processamento de imagens de partículas de minério em uma plataforma que pode ser configurada em uma infraestrutura física ou virtualizada. Foi realizada a adaptação do algoritmo de processamento de imagens com a criação de funções de *parser* do tipo DataFile do Hadoop para o tipo Mat do OpenCV, para assim possibilitar o processamento das imagens utilizando a linguagem de programação C++ . Os algoritmos foram transformados em bibliotecas compartilhadas para serem distribuídos nos nós de processamento, seguindo os padrões de desenvolvimento do MapReduce for C (MR4C). Os experimentos foram feitos em um *cluster* composto de por dez nós de processamento, cada um com oito processadores Intel(R) Xeon(TM) CPU 2.80GHz e 8 GB de memória. Os experimentos foram feitos empregando diversas configurações, alterando a memória máxima alocada, número de *cores*, número de nós de processamento, número de tarefas, tamanho das imagens e também o número de imagens. Os resultados foram discutidos e analisados com o objetivo de destacar os ganhos e limitações da arquitetura apresentada.



**Palavras-chave:** Hadoop. MR4C. Segmentação de fissuras. crescimento de regiões.

## ABSTRACT

BATISTA, Felipe Schimith **Distributed computing applied to mineral's image processing using Hadoop**:. 2017. 116 f. Dissertação (Mestrado em Ciências Computacionais) - Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2017.

The development of parallel processing solutions based on the Hadoop framework has been widely used as an efficient alternative for large volume data processing. The use of search tools has become even more attractive and widespread in the area of technology with the methodology of MapReduce. However, we can tailor this architecture for image processing in order to obtain gains in time and processing volume with the Hadoop Distributed File System (HDFS) content distribution model. In this context, the objective of this study was to develop an architecture capable of distributing the image processing of mineral particles in a platform that can be configured in a physical or virtualized infrastructure. The image processing algorithm was adapted with the creation of parser functions of the DataFile type from Hadoop to the OpenCV Mat type, in order to allow the image processing using the C++ programming language. The algorithms were transformed into shared libraries to be distributed on the processing nodes, following the development standards of the MapReduce for C (MR4C) API. The experiments were done in a cluster consisting of ten processing nodes, each with Intel (R) Xeon (TM) CPU 2.80GHz and 8GB memory. The experiments were done using different configurations, changing the maximum allocated memory, number of cores, number of processing nodes, number of tasks, size of image and number of images. The results where discussed and analyzed with the aim of highlighting the possible gains and limitations of the proposed architecture.



**Keywords:** Hadoop. MR4C. Crack segmentation. Region growth.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## SUMÁRIO

# INTRODUÇÃO

A lixiviação bacteriana, ou biolixiviação, é um processo que pode ser usado para dar viabilidade econômica ao aproveitamento de rejeitos e minérios com baixo teor pela indústria. Entretanto, esse processo tem algumas limitações, como o tamanho das partículas, que tende a restringir as interações entre a solução de lixiviação e os minerais de interesse. Uma possibilidade de contornar esta limitação corresponde à utilização de técnicas de britagem (WILLS, 2011) para gerar fissuras nas partículas e assim aumentar a exposição do minério.

A avaliação do grau de exposição mineral produzido na britagem pode ser realizada a partir de técnicas de processamento e análise de imagens. GOMES et al. (2014) efetuaram a análise microestrutural das partículas minerais submetidas à britagem através de imagens obtidas por um Microscópio Eletrônico de Varredura (MEV), com resolução de 0,5 $\mu$m/pixel. Na rotina de aquisição de imagens de uma amostra foram gerados 1400 frames com resolução de 2000 $\times$ 1725 pixels cada. Devido à impossibilidade de obter garantia de que todas partículas estejam completamente inseridas em algum frame, a análise de cada partícula de forma isolada requer que os frames sejam agregados formando um mosaico. Um mosaico composto por 1400 frames produz uma imagem em torno de 60.000 $\times$ 45.000 pixels. A partir do mosaico, que pode conter de centenas a milhares de partículas, as partículas minerais são separadas em recortes, cada um contendo somente uma partícula. Além do grande volume de dados, a complexidade do processamento aumenta de acordo com o tipo de algoritmo empregado para segmentação das fissuras em cada recorte.

O procedimento de segmentação de fissuras proposto em (GOMES et al., 2014), e utilizado nesta pesquisa, é baseado no algoritmo de crescimento de regiões (ADAMS; BISCHOF, 1994). Segundo GOFMAN (2006), a implementação deste algoritmo em sua versão original possui ordem de complexidade O($n^3$). Além disso, a implementação do algoritmo emprega processamento sequencial, que tem se mostrado incapaz de fazer frente ao volume de dados envolvido e, tampouco, à necessidade de repetição de execuções para a determinação dos valores ideais para os parâmetros do algoritmo. O método sequencial consiste na execução de uma série de instruções que processam um segmento de dados

em um único processador e uma única unidade de memória. Existe um fluxo único de instruções para processamento de um fluxo único de dados. Esse tipo de arquitetura opera somente um dado a cada instrução.

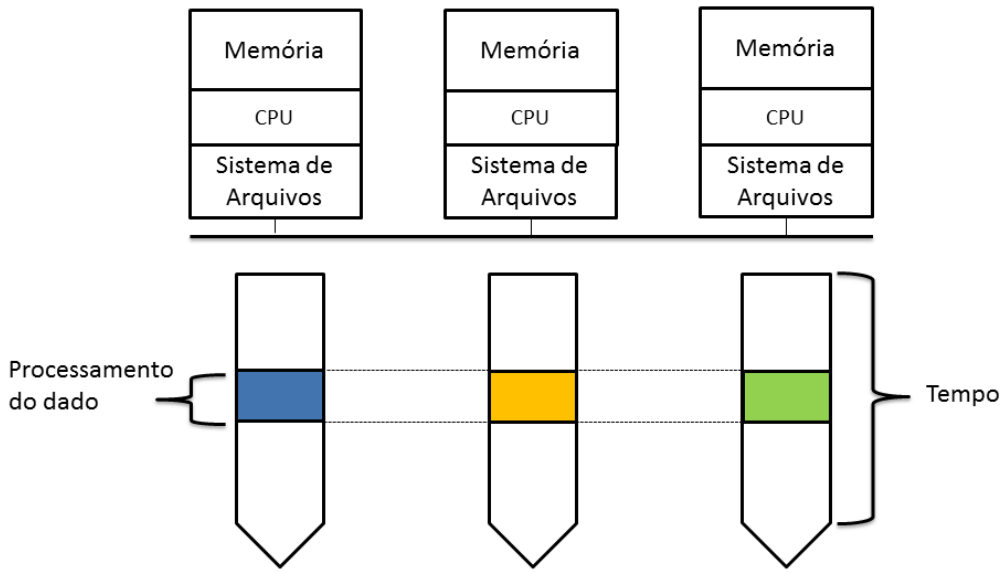
O processamento de grande volume de dados traz desafios para a área de tecnologia da informação. Uma das formas de diminuir o tempo de processamento dos algoritmos é empregar técnicas de processamento paralelo.

O processamento paralelo com memória compartilhada assume que todos os elementos de processamento do sistema possuem visão global da memória e, portanto, diferentes *threads* ou processos podem compartilhar dados através de acessos à memória. Já no modelo com memória distribuída, o compartilhamento de dados deve ser feito explicitamente através de mensagens. Este último modelo é o mais utilizado em *clusters* de computadores, embora seja possível emular uma memória global única com um impacto nos custos de comunicação, como é o caso de software *Distributed Shared Memory* (DSM) (PATTERSON; HENNESSY, 2006) .

Na literatura o termo programação paralela e distribuída refere-se normalmente à distribuição do controle ao orquestrar a execução de uma aplicação paralela. Nesse sentido, um programa pode ser paralelo e não ser distribuído, quando o controle da execução é centralizado em algum elemento de processamento. Este trabalho está interessado em explorar processamento paralelo com o emprego do Hadoop como mediador para distribuição dos dados. No decorrer deste texto, portanto, o termo processamento distribuído será empregado no contexto de distribuição de dados e não de controle.

A Figura 1 ilustra a abordagem de processamento distribuído, onde cada nó de processamento possui sua própria unidade de memória, CPU e sistema de arquivos. Em cada um, dados localmente disponíveis, representados nas cores azul, amarelo e verde, são processados de forma independente. Essa arquitetura se adapta perfeitamente ao problema de interesse deste trabalho, pois nele não há dependências de dados, portanto, não é preciso esperar para que um dado especifico seja processado para que seja iniciado o processamento do dado seguinte.

Figura 1 – Processamento Distribuído.




Fonte: O autor, 2017.

O *framework* Hadoop (BENGFORT; KIM, 2016) é uma das alternativas que possibilita o desenvolvimento de métodos escaláveis de processamento distribuído, capaz de armazenar e processar grande volume de dados. Ele pode ser configurado sobre uma infraestrutura física de *clusters* ou também em servidores em nuvem. O Map Reduce for C (MR4C) agrega ao Hadoop a possibilidade do uso de código nativo da plataforma básica do sistema operacional dos nós do *cluster*, viabilizando o uso de linguagens compiladas, como o C++ , e permitindo através do executável o uso de bibliotecas, como a de processamento de imagens OpenCV, por exemplo.

Considerando essas alternativas, o objetivo desse estudo é desenvolver uma arquitetura, utilizando o Hadoop, capaz de distribuir o processamento de um conjunto de imagens, no caso, imagens de MEV de minérios submetidos a procedimentos de britagem.

A implementação dessa abordagem demandou as seguintes atividades:

- definição da arquitetura da solução;
- instalação e configuração de um servidor virtualizado para fazer a prova de conceito da solução proposta;

- instalação e configuração dos serviços do HDFS, do YARN, do MR4C, OpenCV e todos os seus pré-requisitos;
- adaptação do MR4C para suportar o desenvolvimento utilizando o OpenCV;
- transformação do algoritmo de segmentação de fissuras proposto em (GOMES et al., 2014) em biblioteca dinâmica;
- desenvolvimento de *parsers* de tipo que possibilitem a transformação do dado de um formato Hadoop para um formato OpenCV;
- desenvolvimento de um algoritmo para execução da função de segmentação de fissuras de forma distribuída - através de processamento paralelo com memória distribuída;
- desenvolvimento de um programa para importação das imagens no Hadoop e chamada do algoritmo de segmentação de fissuras de forma distribuída;
- desenvolvimento de um programa para reduzir o problema de balanceamento de carga do MR4C;
- adequação da alocação de recursos dos serviços do Hadoop.

Este trabalho encontra-se organizado da seguinte forma. O Capítulo 1 faz uma revisão bibliográfica das soluções disponíveis na literatura que podem ser utilizadas para processamento de imagens fazendo uso do *framework* Hadoop. O Capítulo 2 trata da fundamentação teórica. O Capítulo 3 apresenta o contexto do estudo para a presente pesquisa. O Capítulo 4 aborda o método distribuído de segmentação de fissuras e a adequação do algoritmo de crescimento de regiões utilizando a arquitetura proposta. O Capítulo 5 descreve o procedimento experimental, apresenta e analisa os resultados. No Capítulo 5.14, são apresentadas as considerações finais e sugestões para trabalhos futuros.

# 1 REVISÃO BIBLIOGRÁFICA

O presente capítulo apresenta algumas soluções disponíveis na literatura que podem ser utilizadas para processamento distribuído de imagens fazendo uso do *framework* Hadoop. O Hadoop, que será descrito na Seção 2.4, permite a utilização do modelo de programação MapReduce em projetos que se enquadram no âmbito da computação distribuída e processamento de dados em larga escala em *clusters* de computadores.

## 1.1 Hadoop Image Processing Interface

Hadoop Image Processing Interface (HIPI) (ARSH; BHATT; KUMAR, 2016) é uma biblioteca de processamento de imagens projetada para ser usada com a estrutura de programação distribuída do Apache Hadoop MapReduce. O HIPI facilita o processamento de imagens com programas paralelos de MapReduce normalmente executados em um *cluster*. Ele fornece uma solução na forma de armazenar uma grande coleção de imagens no Hadoop Distributed File System (HDFS) e disponibilizá-la para processamento distribuído eficiente. Ele tem os seguintes objetivos:

- fornecer uma biblioteca aberta, extensível para aplicações de processamento de imagens e visão computacional utilizando o Hadoop;
- tornar eficiente o armazenamento de imagens em aplicações de MapReduce;
- permitir a seleção com base nas propriedades das imagens;
- prover uma interface intuitiva para operações baseadas em imagens;
- fazer o tratamento de distribuição e balanceamento de carga de forma transparente para o usuário.

Para alcançar os objetivos, o HIPI utiliza um tipo de dado proprietário chamado HIPI Image Bundle (HIB). O HIB é composto por um arquivo de dados contendo imagens concatenadas e um arquivo de índice contendo informações sobre o deslocamento das imagens em termos do sistema do bloco (*bundle*). Isso é feito para possibilitar o melhor

uso dos recursos do Hadoop tornando o processamento de um conjunto de imagens mais eficiente (SWEENEY C., 2011).

No início da presente pesquisa, quando examinava-se as possíveis plataformas a serem utilizadas, a HIPI limitava seu uso à linguagem Java, além de não haver suporte à biblioteca OpenCV, nem tampouco suporte à imagens TIF. Apesar de ter sido recentemente resolvidas, tais limitações levaram à decisão de descartar o HIPI como plataforma base na presente pesquisa.

## 1.2 **InterImage Cloud Platform**

O InterIMAGE Cloud Platform, ou InterCloud, é uma plataforma distribuída para análise de imagens de sensoriamento remoto baseada em objetos, ele foi inspirado no sistema InterIMAGE que é um sistema baseado em conhecimento que segue a abordagem Geographic Object-Based Image Analysis (GEOBIA) (FERREIRA, 2015). GEOBIA é um subcampo da Geographic Information Science (GIScience) dedicado ao desenvolvimento de métodos automatizados para interpretar imagens de sensoriamento remoto e gerar novas informações geográficas em formatos compatíveis com os Sistemas de Informação Geográfica (SIG) (BLASCHKE; LANG; HAY, 2008). O InterCloud foi concebido para possibilitar o processamento de grandes volumes de de dados em *cluster*, utilizando a plataforma Hadoop. É possível executar o InterCloud em uma infraestrutura de nuvem, proporcionando todas as vantagens de provisionamento de infraestrutura como serviço (FERREIRA et al., 2017).

O InterCloud utiliza os componentes de armazenamento e processamento de dados distribuído do Hadoop, sendo que seus operadores são criados em Pig Latin (OLSTON et al., 2008), uma linguagem de programação voltada à expressão de fluxos de dados que combina a consulta declarativa de alto nível, como o SQL, e a programação processual de baixo nível como MapReduce, utilizando User Defined Functions (UDF). Desta forma é possível combinar as operações nativas do Pig com os operadores específicos do domínio do problema (FERREIRA et al., 2017).

Os métodos propostos em (FERREIRA et al., 2017), implementados no InterCloud,

deram suporte ao desenvolvimento de um método de segmentação distribuída, apresentada em (HAPP et al., 2016).

O método inicialmente produz recortes, *tiles*, de imagens de sensoriamento remoto, que são distribuídos pelas unidades de processamento de um cluster e segmentados independentemente. O principal problema atacado em (HAPP et al., 2016) foi a eliminação de artefatos (segmentos com bordas retas), decorrentes das segmentações independentes dos *tiles*. Para tanto, estratégias de pós-processamento são propostas, que basicamente agrupam os segmentos que tocam as bordas dos *tiles* de forma a re-segmentar as regiões da imagem cobertas por estes segmentos.

O método proposto por (HAPP et al., 2016) tem, portanto, o intuito de segmentar imagens de grandes dimensões, dividindo-as em pequenos recortes, o que não se aplica ao presente trabalho, uma vez que, como será visto adiante, imagens menores serão submetidas ao processamento distribuído e não existe neste caso o problema de segmentos com artefatos introduzidos pelo processo.

## 1.3 **Hadoop Streaming**

O Apache Hadoop fornece uma API para MapReduce que permite que se escreva um programa usando funções que não estejam na linguagem Java, utilizando JNI (Java Native Interface) (LIANG, 1999). O assim chamado Hadoop Streaming (WHITE, 2012) usa os canais de comunicação padrão do Unix como interface entre Hadoop e o programa. Assim, o programador pode usar qualquer linguagem que também utilize o canal padrão do sistema operacional para fazer a leitura da entrada de dado padrão e fazer a escrita para a saída padrão a partir do seu programa MapReduce. *Streaming* é uma técnica naturalmente adequada para o processamento de texto e quando usado em modo texto, ele tem uma visão orientada a linha de dados (PERERA; GUNARATHNE, 2013).

O presente estudo se diferencia do Hadoop Streaming pelo fato de possibilitar o processamento de imagens e também possibilitar o uso da biblioteca OpenCV. O uso do Hadoop Streaming em conjunto com o OpenCV é viável, mas carece de disponibilidade de código fonte. Atualmente, esta facilidade pode ser usada somente na infraestrutura da

nuvem da Universidade de Prairie View A&M (EPANCHINTSEV; SOZYKIN, 2015).

## 1.4 **Hadoop Pipes**

Hadoop Pipes é o nome da interface C++ para Hadoop MapReduce. Ao contrário do Hadoop Streaming, que usa a entrada e saída para se comunicar com o código MapReduce, Pipes usa *sockets* (STEVENS, 1998), comunicação cliente servidor utilizando como referência o nome da máquina e a porta do serviço, como o canal através do qual o TaskTracker, o gerenciador de recursos e tarefas da primeira versão do Hadoop (APACHE, 2009), se comunica com o processo em execução Map ou Reduce em C++ . O Hadoop Pipes não utiliza o JNI. Ao contrário da interface Java, as chaves e valores na interface do C++ são *buffers* de bytes, representadas como objetos da classe *string* da biblioteca Standard Template Library (STL) (NELSON, 1995). Isso torna a interface mais simples, embora adicione a carga maior sobre o desenvolvedor que terá que gerenciar manualmente a conversão de tipos (WHITE, 2012). O sobretrabalho de controle das conversões de tipos foi a principal razão que desmotivou o uso do Hadoop Pipes nesse trabalho.

## 1.5 **Hadoop Libhdfs**

Hadoop libhdfs é uma biblioteca baseada no Java Native Interface (JNI) e que fornece uma API disponível para a linguagem C para acessar o Hadoop Distributed File System (HDFS). Ela compreende um subconjunto da API para manipular o sistema de arquivos HDFS (Apache, 2015). Por ser uma biblioteca para um sistema de arquivos, a libhdfs carece de recursos de gerência de aplicação, fato que desmotivou seu uso no presente trabalho.

# 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo descreve os principais fundamentos teóricos relevantes para a compressão do presente trabalho. Nele são abordados os processos de biolixiviação, a microscopia eletrônica de varredura e a segmentação de imagens digitais, parte do escopo do problema estudado, além do Hadoop, o MR4C e a biblioteca OpenCV, infraestrutura de software empregada na solução proposta.

## 2.1 Processo de Biolixiviação

Historicamente a indústria mineral tem sofrido o impacto das crescentes restrições ambientais e da indisponibilidade de minérios de alto teor, que estão cada vez mais escassos. Com a necessidade de manter a produção, perante à crescente demanda, a indústria investe no desenvolvimento de tecnologias e processos mais eficientes e sustentáveis. Um dos exemplos de técnica é a lixiviação, a extração ou solubilização dos constituintes químicos de um mineral pela ação de um fluido. A lixiviação viabilizou o uso pela indústria de minérios com teor reduzido mantendo a rentabilidade econômica (DAMASCENO, 2006).

A lixiviação bacteriana, ou biolixiviação, é um processo alternativo capaz de aproveitar rejeitos e minérios de baixo teor. A bilolixiviação pode ser utilizada no processamento de agregados e partículas grandes, reduzindo os custos relacionados à britagem. Ela é atualmente empregada na extração de metais de base e também de metais preciosos.

O processo de biolixiviação é realizado por bactérias que promovem a solubilização de determinados componentes presentes em uma amostra mineral (LIMA URGEL DE ALMEIDA, 2001). Apesar das vantagens econômicas e ambientais, esse processamento de partículas grandes tem algumas limitações, sendo que apenas os grãos na superfície destas partículas serão expostos à solução de lixiviação (GOMES et al., 2014).

Uma alternativa que pode aumentar a exposição mineral, sobretudo dos minerais com valor econômico, é o uso de métodos de britagem para segmentação de fissuras, com o objetivo de acelerar o processo de biolixiviação reduzindo seu custo financeiro e ambiental. A análise de imagens pode ser empregada para fazer a medição da exposição mineral, assim auxiliando na identificação do melhor método de britagem. A partir dessa medição

é possível a avaliação de algumas combinações de diferentes métodos de britagem empregados sequencialmente, com o intuito de aumentar, a partir da geração de microfissuras, a exposição dos minerais com valor econômico à solução de lixiviação.

## 2.2 Microscópio Eletrônico de Varredura

O Microscópio Eletrônico de Varredura (MEV) pode ser utilizado para a análise de características micro e nanoestruturais, e química elementar de objetos sólidos. Possibilita análises de superfície fraturada como a avaliação do tamanho de partículas e a porcentagem de fase em microestruturas. Em vez de utilizar fótons empregados na microscopia ótica , um MEV utiliza feixes de elétrons (DEDAVID; GOMES; MACHADO, 2007).

O MEV constrói as imagens através da sincronização da varredura do feixe de elétrons com os muitos sinais que vêm da interação entre o feixe de elétrons e a amostra. Os pixels apresentam intensidade proporcional ao sinal medido pelos seus sensores. Um dos sensores mais comuns utilizados é o de Back-Scattered Electrons (BSE), detector de elétrons retroespalhados, que pode fornecer, por exemplo, informações de topografia e contraste de número atômico (GOMES; PACIORNIK, 2012).

## 2.3 Segmentação de Imagens Digitais

A segmentação consiste na divisão de uma imagem em diferentes objetos ou partes de objetos de interesse. O resultado da segmentação é um conjunto de segmentos, que representam conjuntos de pixels contíguos.

As principais técnicas de segmentação se baseiam em duas propriedades básicas da variação das intensidades dos pixels (níveis de cinza numa imagem monocromática): a descontinuidade e a similaridade.

A descontinuidade está relacionada com mudanças bruscas nos níveis de cinza e a similaridade está relacionada com regiões uniformes em termos da intensidade dos pixels.

Técnicas que se baseiam na descontinuidade são principalmente usadas para a detecção das bordas de objetos.

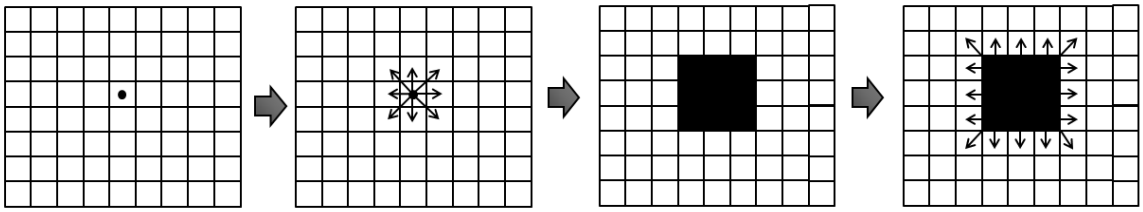
As principais abordagens de segmentação que se baseiam na similaridade são: a limi-

arização; a divisão e fusão de regiões; e o crescimento de regiões (GONZALEZ; WOODS, 2000). No presente estudo, somente a segmentação por crescimento de regiões será abordada.

### 2.3.1 Crescimento de Regiões

As técnicas de segmentação por crescimento de regiões têm como objetivo particionar imagens em regiões homogêneas, de acordo com algum critério de homogeneidade. São procedimentos iterativos que agrupam pixels em regiões que crescem pela inclusão de pixels vizinhos ou pela fusão de regiões vizinhas.

Figura 2 – Crescimento de regiões pelo agrupamento de pixels



Fonte: O autor, 2017.

As técnicas de crescimento de regiões geralmente iniciam o crescimento a partir de sementes, pixels ou conjuntos de pixels que representam as regiões iniciais. A escolha das sementes geralmente é feita baseando-se na natureza do problema e no tipo de dado que se tem disponível (GONZALEZ; WOODS, 2000).

A cada iteração do processo de crescimento as regiões se fundem com regiões ou com pixels vizinhos que sejam semelhantes aos pixels que compõem as regiões. Para que a fusão aconteça ela deve atender a um determinado critério de similaridade.

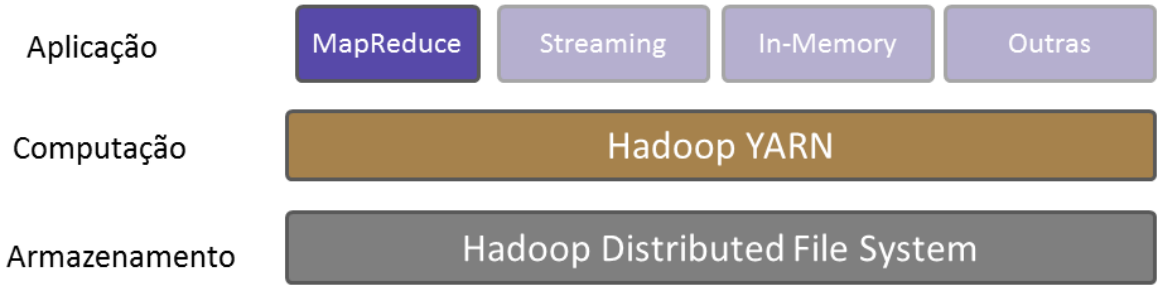
A condição de parada do processo é geralmente quando nenhum par de vizinhos na imagem atende ao critério de similaridade (GONZALEZ; WOODS, 2000).

As técnicas de segmentação por crescimento de regiões geralmente apresentam um alto custo computacional. Um dos desafios da análise de imagens em grandes volumes é a redução do tempo de processamento, o que pode alcançado com a utilização de tecnologias que permitem o processamento paralelo.

## 2.4 Hadoop

A biblioteca de software Apache Hadoop é um *framework* que permite o processamento distribuído de grandes conjuntos de dados em *clusters* de computadores. Ele é projetado para escalar recursos de armazenamento e processamento a partir de um único servidor para potencialmente milhares de máquinas. Em vez de confiar em hardware para proporcionar alta disponibilidade, a biblioteca em si é concebida para detectar e tratar falhas na camada de aplicação, de modo a fornecer um serviço altamente disponível de um conjunto de computadores, cada um dos quais pode ser propenso a falhas (APACHE, 2016).

Figura 3 – Arquitetura do Hadoop.



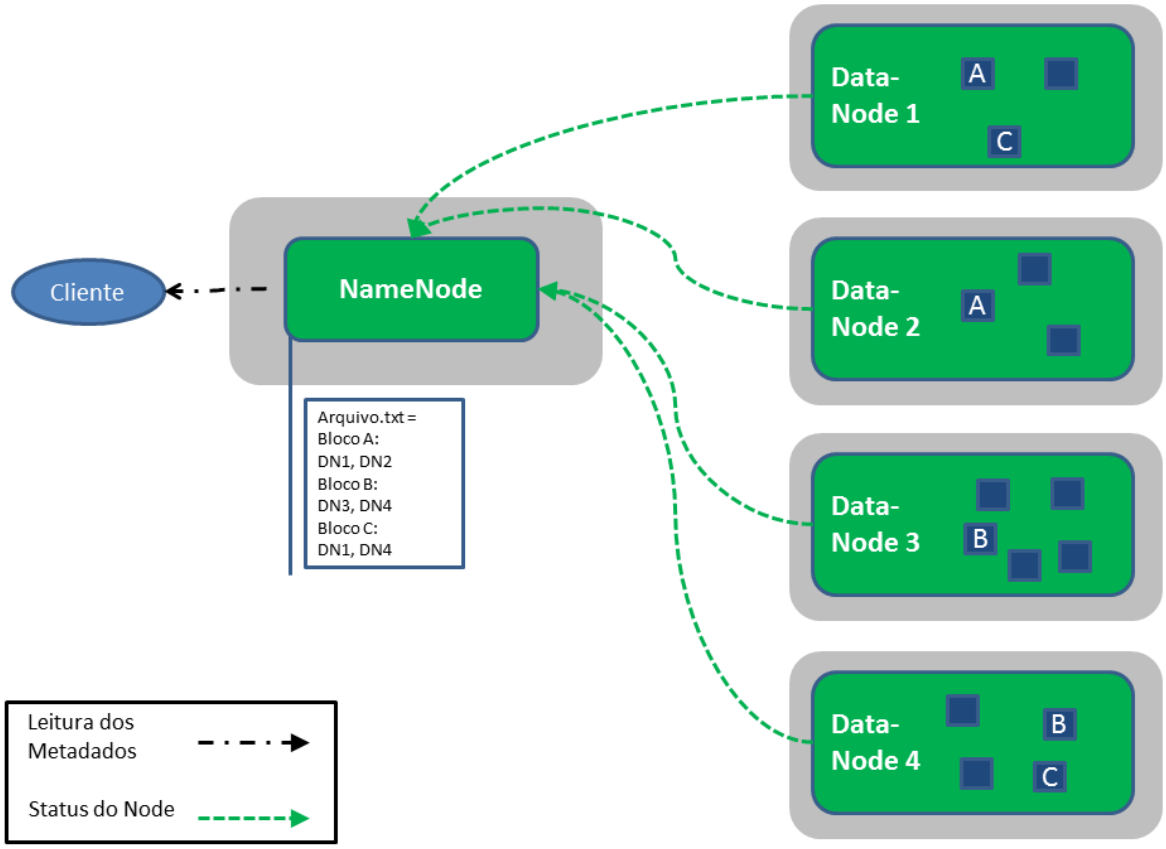

Fonte: Adaptado de WHITE, 2012, p79.

A Figura 3 apresenta em alto nível a arquitetura do Hadoop. Ela pode ser dividida em três principais camadas: armazenamento, computação e aplicação. A camada de armazenamento faz a gestão dos conteúdos e é representada pelo *Hadoop Distributed File System* (HDFS). A camada de computação faz a distribuição do processamento e é representada pelo *Hadoop YARN*. A camada de aplicação faz uso dos recursos de computação e armazenamento e pode ser representada pelo MapReduce. Em suma, os conteúdos importados no Hadoop são distribuídos em blocos no HDFS, a aplicação utiliza o YARN para alocar os recursos de processamento mais próximos dos blocos dos conteúdos a serem processados, mais especificamente o MapReduce divide o processo em duas fases, a de mapeamento do dado (Map) e redução do dado mapeado (Reduce). As subseções a seguir descrevem cada um desses componentes.

### 2.4.1 Hadoop Distributed File System (HDFS)

O *Hadoop Distributed File System* (HFDS) é a componente de gerenciamento de arquivos do Hadoop. O HDFS gerencia a estrutura de arquivos e diretórios, sendo que os principais componentes são o *NameNode* e o *DataNode*, representados na Figura 4.

Figura 4 – Arquitetura do HDFS.




Fonte: Adaptado de Apache, 2013.

Ao importar um conteúdo para o HDFS, o conteúdo é geralmente dividido em blocos de 64 MB ou 128 MB, cada bloco é registrado pelo NameNode e armazenado em pelo menos um DataNode. Os NameNodes são responsáveis por registrarem as informações dos conteúdos como permissões, data de modificação, data de acesso, estrutura de armazenamento e cotas de espaço em disco. Entretanto, quando um conteúdo com tamanho menor que um bloco for importado, o HDFS não aloca espaço extra como em sistemas de arquivos tradicionais. Sendo assim, quando um bloco tiver metade do tamanho, ele

precisa apenas de metade do espaço na unidade local (SHVACHKO et al., 2010).

Os blocos de um conteúdo importado no HDFS são replicados nos DataNodes de acordo com o fator de replicação configurado. No exemplo da Figura 4, o HDFS está configurado com fator de replicação dois, sendo que cada bloco do *Arquivo.txt* está presente em dois DataNodes, por exemplo o bloco A, está presente no DataNode1 e no DataNode2. Caso o fator de replicação seja maior que um, é possível recuperar um conteúdo após a perda de um nó, sendo o NameNode responsável por manter o fator de replicação do bloco. Quando uma aplicação deseja ler um conteúdo, ela vai utilizar a referência como uma estrutura de diretórios comum. Internamente essa requisição chegará para o NameNode que identificará os DataNodes onde os blocos do conteúdo foram armazenados (SHVACHKO et al., 2010).

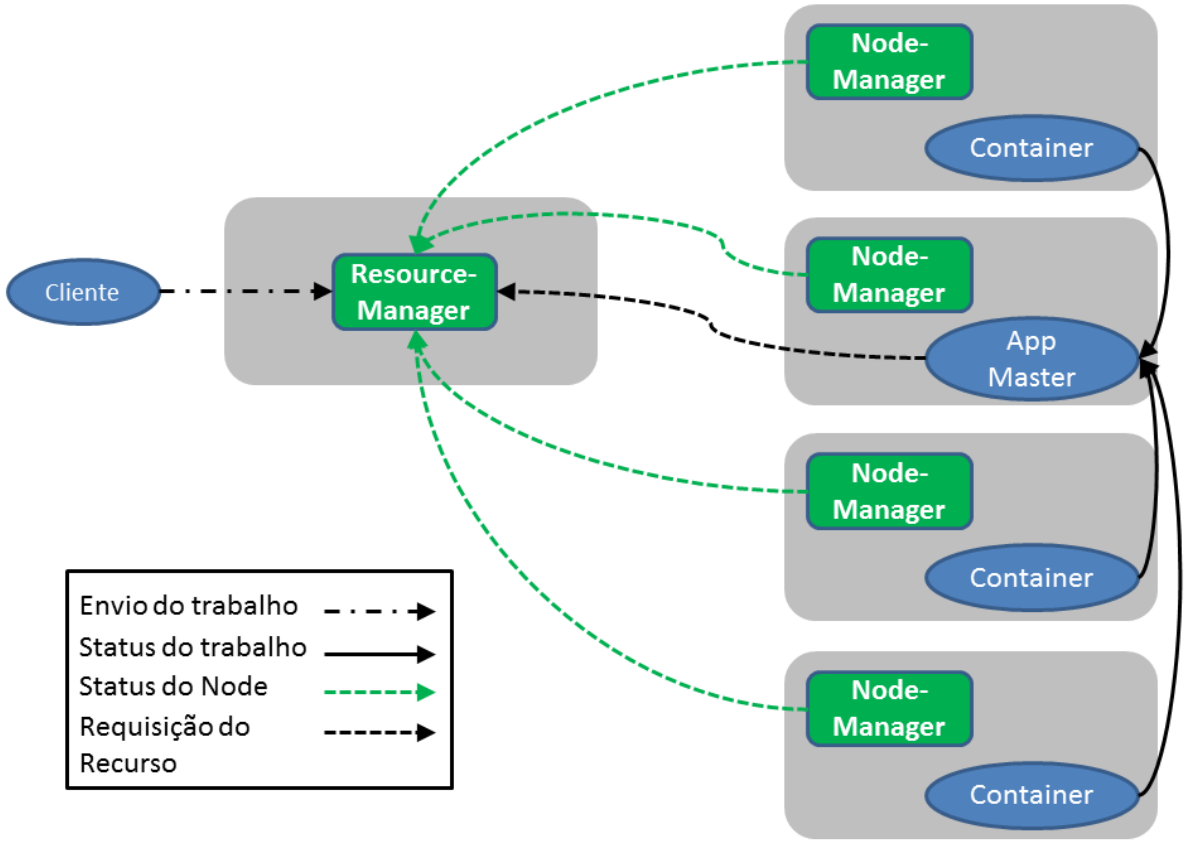
Os DataNodes representam cada bloco em dois arquivos no sistema de arquivos nativo do nó local. O primeiro arquivo contém os metadados do bloco e dados de verificação e o segundo arquivo os dados em si. Cada DataNode gerencia os seus blocos e envia um relatório para o NameNode (APACHE, 2014). Ele também faz chamadas de conexão com o NameNode para confirmar que está operando, sendo que o intervalo padrão dessas chamadas é de três segundos. Se o NameNode não receber o sinal de conexão do DataNode em dez minutos, ele considera o DataNode fora de serviço e inicia a revisão das réplicas de blocos hospedados por esse DataNode. Por exemplo, na Figura 4, se o DataNode2 parar de responder por mais de dez minutos, o NameNode irá replicar o bloco A em outro DataNode que estiver respondendo. Essas chamadas de conexões também enviam a capacidade de armazenamento total, fração de armazenamento em uso e o número de transferências de dados em andamento, que servem para o balanceamento de carga do que é gerenciado pelo NameNode (SHVACHKO et al., 2010).

Normalmente os blocos de dados são armazenados no disco rígido (HD) ou unidade de estado sólido (SSD) dos nós dos DataNodes. Quando um conteúdo é requisitado, os blocos de dados são carregados no cache e por padrão o HDFS também mantém os blocos frequentemente acessados. Entretanto é possível requisitar o HDFS para manter os blocos em cache, aumentando o desempenho de leitura (WHITE, 2012).

### 2.4.2 Hadoop YARN

O *Hadoop YARN* é o componente de gerenciamento de recursos de CPU, memória, rede e disco dos nós de processamento que fazem parte do *cluster* Hadoop, sendo também responsável pelo agendamento e monitoramento do trabalho. Os principais componentes do Hadoop YARN são o *ResourceManager, ApplicationMaster e NodeManager* representados na Figura 5.

Figura 5 – Arquitetura do YARN.




Fonte: Adaptado de Apache, 2016.

A sequência de atividades de alto nível efetuadas pelos componentes do YARN é explicitada na Figura 5. Primeiramente, a aplicação cliente faz a requisição do *Job* ou trabalho para o ResourceManager, que é responsável pelo gerenciamento dos recursos (CPU, memória, disco e rede) de todas as aplicações que requisitam o *cluster* Hadoop. Toda aplicação Hadoop utiliza um *Job* para instanciar as tarefas, que são executadas em

paralelo, sendo que o *Job* é a execução principal da aplicação. O Hadoop distribui os conteúdos pelas tarefas alocadas, por exemplo, se tivermos dez imagens de 30 MB cada uma e três tarefas, duas tarefas ficarão com três imagens para processar e uma ficará com quatro imagens para processar. Ele leva em consideração onde o dado se encontra ao fazer essa divisão, diminuindo a necessidade de tráfego de rede.

O ResourceManager solicita a alocação de recursos para cada aplicação em execução e gerencia as filas de processamento. Para isso, o ResourceManager utiliza o *scheduler* ou agendador que é um planejador de atividades. Ele requisita a alocação de recursos como memória, CPU, disco e rede de uma aplicação para o ApplicationMaster usando o conceito de container. De acordo com o volume de dado a ser processado, o ResourceManager utiliza um ou mais *ResourceRequests* para cada atividade contendo o número de containers, número de unidades de processamento ou *cores*, memória por container, preferências de localidade e prioridade da requisição (MURTHY et al., 2014).

Após o ResourceManager solicitar alocação dos recursos para a execução do trabalho, o ApplicationMaster fica responsável por negociar todos os recursos requisitados pelo ResourceManager e supridos pelos NodeManagers. O ResourceManager coordena a execução da aplicação no *cluster* Hadoop, gerenciando o status dos recursos e monitorando o progresso do trabalho. Para informar seu status e atualizar o andamento das atividades, o ApplicationMaster faz requisições de conexão periódicas com o ResourceManager. Tecnicamente o ApplicationMaster também é executado em um container (VAVILAPALLI et al., 2013).

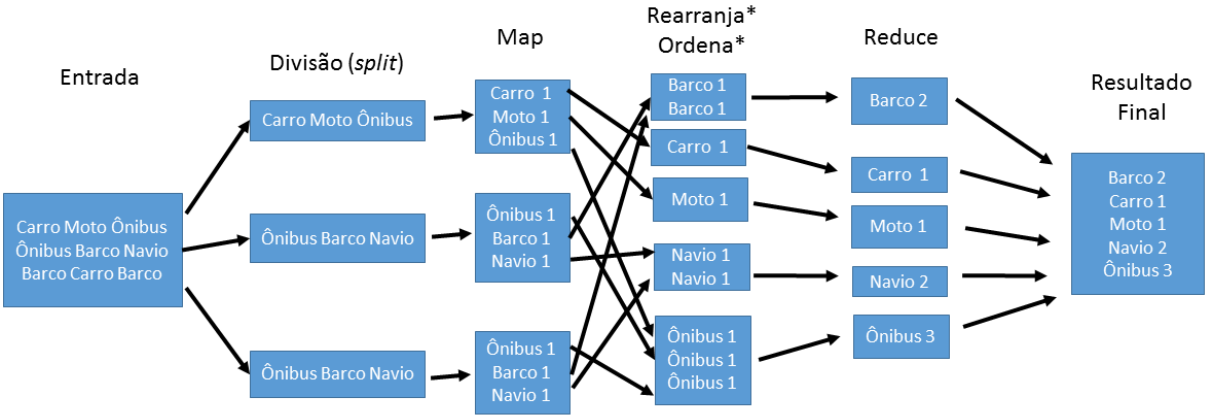
Os containers são onde as tarefas das aplicações serão executadas, o NodeManager é responsável por gerenciar os recursos alocados nos containers como número de *cores*, memória, disco, rede e relata status dos recursos para o ApplicationMaster. Ao criar o container, o NodeManager copia para o armazenamento local todas as dependências necessárias, como arquivos de metadados, executáveis e bibliotecas. Durante toda a fase de execução da tarefa, o NodeManager faz requisições de conexão periódicas com o ResourceManager para informá-lo de seu status. Ao final da execução da tarefa, o ApplicationManager e ResourceManager informam para o NodeManager executar a desalocação dos

recursos dos containers e excluir as dependências das aplicações (VAVILAPALLI et al., 2013).

### 2.4.3 Modelo MapReduce

O modelo de processamento MapReduce consiste em duas etapas distintas. O primeiro passo chamado de *Map* é a fase de processamento paralelo, em que os conteúdos importados no HDFS são divididos em blocos, distribuídos em nós de processamento e podem ser trabalhados de forma independentemente, como apresentado na Figura 6.

Figura 6 – Exemplo de funcionamento do MapReduce.




Fonte: Adaptado de MINER; SHOOK, 2012, p16.

A função Map executa a filtragem e classificação onde mapeia os blocos do conteúdo formando uma coleção de tuplas $(chave, valor)$. Utilizando o exemplo da Figura 6, teríamos no primeiro bloco as tuplas $(Carro, 1)$, $(Moto, 1)$ e $(Onibus, 1)$.

O segundo passo é a fase de redução que executa uma operação de resumo das tuplas com mesma chave que foram processadas na fase de Map. O *Reduce* produz uma saída única que representa o processamento do conteúdo. Essa natureza simples e bastante restrita do modelo de programação se presta a implementações muito eficientes e de grande escala em milhares de servidores que podem ser executados em *commodities* de hardware de baixo custo (PERERA; GUNARATHNE, 2013).

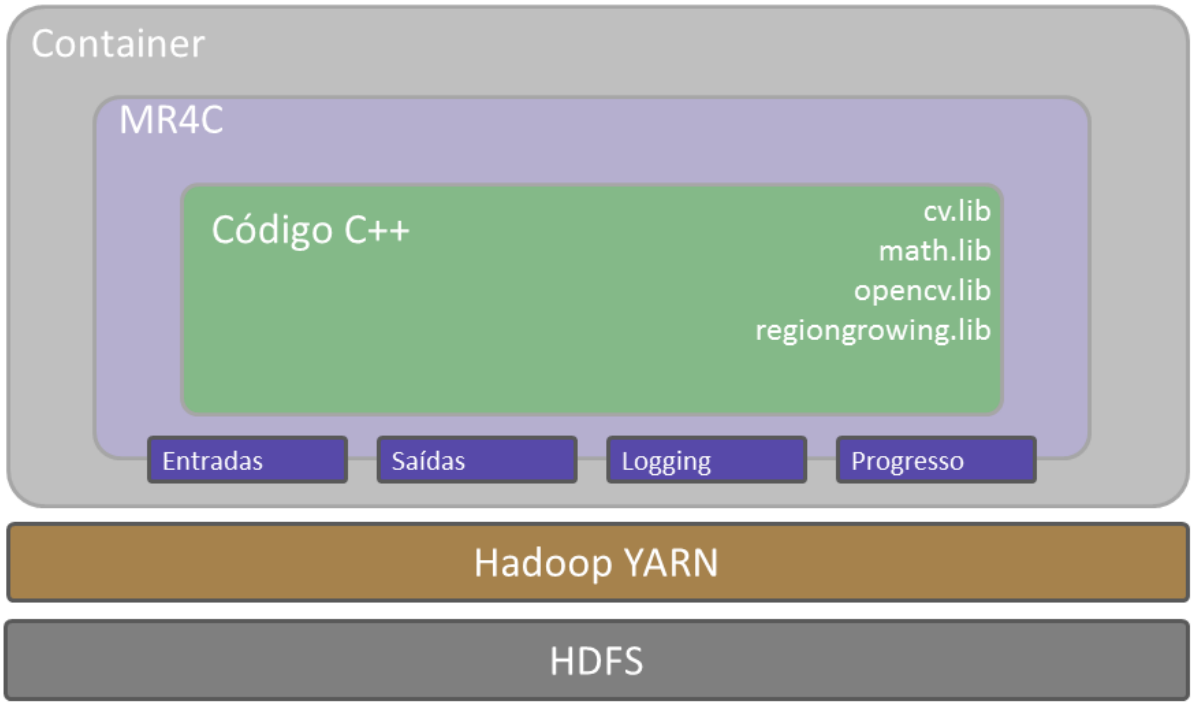
## 2.5 MR4C

O *Map Reduce for C* (MR4C) é uma API baseada em C++ que foi desenvolvida utilizando o *Java Native Interface* (JNI) para possibilitar que um software escrito originalmente nas linguagens C ou C++ possa utilizar a infraestrutura do HDFS. O MR4C foi concebido originalmente para facilitar a grande escala de processamento de imagens de satélite e da ciência de dados geoespaciais. Hoje é um projeto opensource do Google Inc (KENNEDY-BOWDOIN, 2015). Essa API possibilitou o uso do gerenciamento de recursos como também agendamento e monitoramento de trabalho disponível pelo Hadoop YARN.

A divisão e alocação de recursos podem ser configuradas com ferramentas baseadas em Hadoop YARN ou no nível de *cluster* para *MapReduce version* 1 (MRv1).

O MR4C foi desenvolvido para possibilitar a manipulação de dados do HDFS em conjunto com o poderoso ecossistema de bibliotecas de processamento de imagem desenvolvidas em C++ , conforme Figura 7.

Figura 7 – Arquitetura do MR4C.




Fonte: Adaptado de KENNEDY-BOWDOIN, 2015.

A Figura 7 apresenta a arquitetura do MR4C, que utiliza os containers do Hadoop YARN para processar os blocos dos conteúdos localmente no HDFS. O conjunto dos algoritmos necessários para o processamento de dados, representados em verde, são transformados em objetos compartilhados nativos. Dessa forma, após serem replicados para os nós de processamento encapsulados em containers, eles podem ser utilizados como dados do sistema de arquivos local ou qualquer identificador de recursos uniforme (URI).

Já os conjuntos de dados de entrada e saída como também a extensão do formato dos dados a serem processados são configurados no arquivo *JavaScript Object Notation* (JSON). O fluxo de trabalho pode ser construído e testado em uma máquina local usando exatamente a mesma interface utilizada no *cluster* (KENNEDY-BOWDOIN, 2015).

### 2.5.1 Conjuntos de Dados (Datasets)

MR4C usa um modelo de *Datasets* para agrupar todos os dados relevantes e torná-los disponíveis para o algoritmo. Tradicionalmente, pensamos em conjuntos de dados como arquivos, e disponibilizamos eles usando caminhos. Com MR4C, podemos trabalhar com arquivos individuais, ou também referenciando um diretório que contém muitos arquivos e coletivamente agrupá-los como um conjunto de dados (KENNEDY-BOWDOIN, 2015).

### 2.5.2 Identificador de elementos (keyspace)

O *keyspace* é um índice de elementos únicos no conjunto de dados. Cada chave refere-se a um arquivo sem ter que manter o controle do caminho que ele se encontra. Isso pode ser especialmente útil quando estamos operando em um *cluster* onde os arquivos não estão no mesmo local (KENNEDY-BOWDOIN, 2015).

### 2.5.3 Configuração

O MR4C exige um arquivo de configuração no formato JSON. Este arquivo de configuração pode manter o controle de parâmetros de entrada que possibilitam serem alterados sem a necessidade de recompilar o algoritmo novamente. Em suma, um arquivo de configuração deve conter os tipos de objetos a serem processados, o nome da classe do algoritmo

de processamento, a localização dos Datasets de entrada no HDFS, a localização da saída dos Datasets no HDFS e os parâmetros de entrada do algoritmo.

## 2.6 **OpenCV**

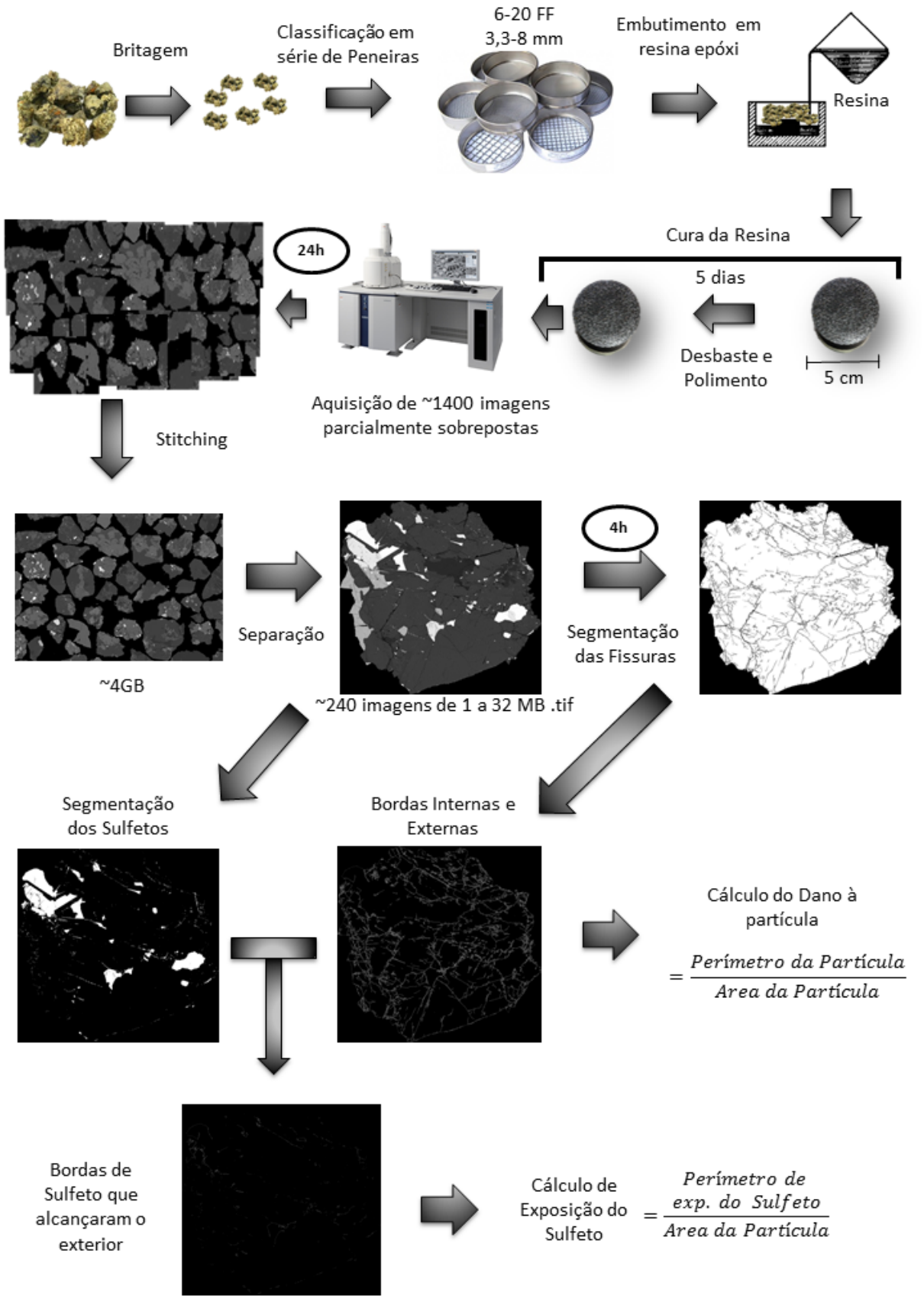
OpenCV é uma biblioteca de visão computacional escrita em C e C++ que pode ser executada em diversas plataformas e possui APIs para diferentes linguagens. Ela foi projetada com foco em eficiência computacional em aplicações de tempo real e pode tirar vantagem de multiprocessadores. A OpenCV fornece uma infraestrutura simples de usar contando com mais de 500 funções como por exemplo processamento de imagens (LAGANIÈRE, 2011).

A classe *Mat*, especificamente, representa uma imagem no formato de matriz sendo que os elementos das coordenadas $< x, y >$ não precisam ser em si números simples. Ele pode ser uma lista de tipos predefinidos.

# 3 CONTEXTO DO ESTUDO

O método desenvolvido na presente pesquisa emprega o algoritmo proposto por GOMES et al. (2014) para a caracterização de danos a partículas e superfície exposta com o intuito de avaliar a exposição do minério de diferentes métodos de britagem de minério. Para tanto, os autores desenvolveram um algoritmo de segmentação de fissuras suficientemente sensível para detectar fissuras com largura inferior a um pixel e capaz de lidar com os tons de cinza das imagens obtidas pelo MEV. O algoritmo baseia-se no método de crescimento de regiões, usando, especificamente, o gradiente de intensidade e a intensidade relativa como fator de comparação. Utilizou-se de fato uma versão desse algoritmo reescrita em C. A implementação foi realizada para o projeto de pesquisa número `E-26/110.124/2014`, intitulado "Computação de alto desempenho aplicada ao bioprocessamento de minérios e rejeitos", financiado através do Edital Faperj `41/2013`. A Figura 8 ilustra o procedimento de avaliação da exposição dos minerais de interesse após a utilização de algum método de britagem.

Figura 8 – Avaliação do grau de exposição.

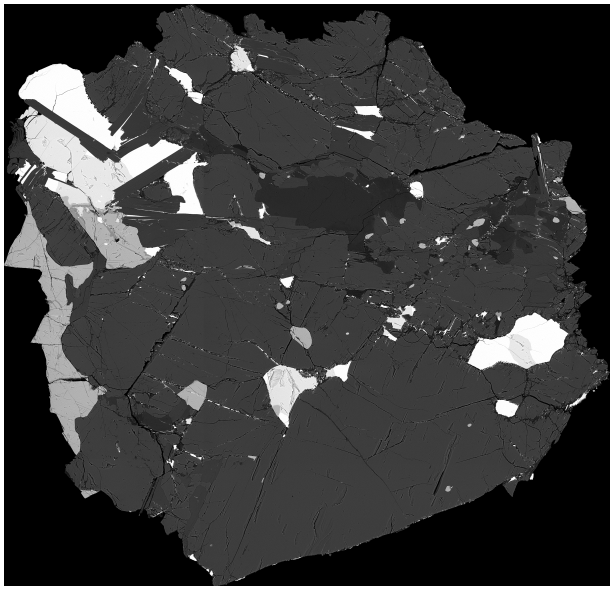



Fonte: O autor, 2019.

O processo se inicia a partir do recebimento das amostras dos rejeitos de cada método de britagem, elas são submetidas ao procedimento de separação em uma série de peneiras para obtenção de partículas de 3,3 mm a 8 mm de diâmetro. As partículas são colocadas em um cilindro e é feito o embutimento em resina epóxi, seguido do desbaste e polimento do corpo de prova. Essa etapa dura em torno de cinco dias.

A aquisição das imagens é feita utilizando o detector de elétrons retroespalhados do MEV. Após a obtenção das imagens, foi utilizado um algoritmo de *stitching* para gerar um mosaico de todos os recortes removendo as sobreposições. Em seguida, as partículas presentes são segmentadas, para que cada uma delas seja analisada em separado. A Figura 9 apresenta um exemplo de amostra de partícula.

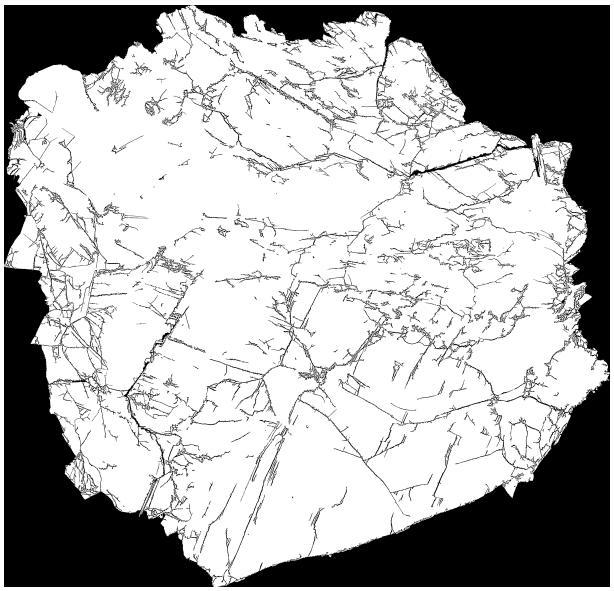
Figura 9 – Amostra de parícula.



Fonte: O autor, 2019.

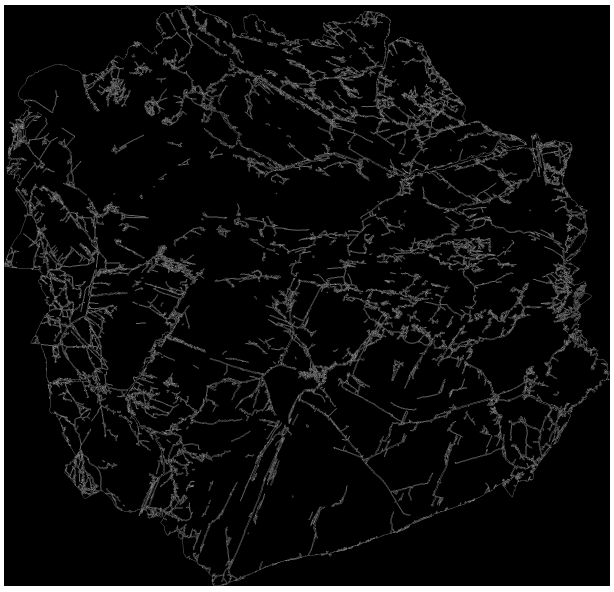
Para o cálculo de dano à partícula é necessário submeter as imagens das partículas ao algoritmo de segmentação de fissuras (Figura 10), e depois ao algoritmo de identificação de bordas internas e externas (Figura 11). O cálculo de dano à partícula é obtido a partir do perímetro da partícula sobre a área da partícula.

Figura 10 – Segmentação de fissuras.



Fonte: O autor, 2019.

Figura 11 – Identificação das bordas.

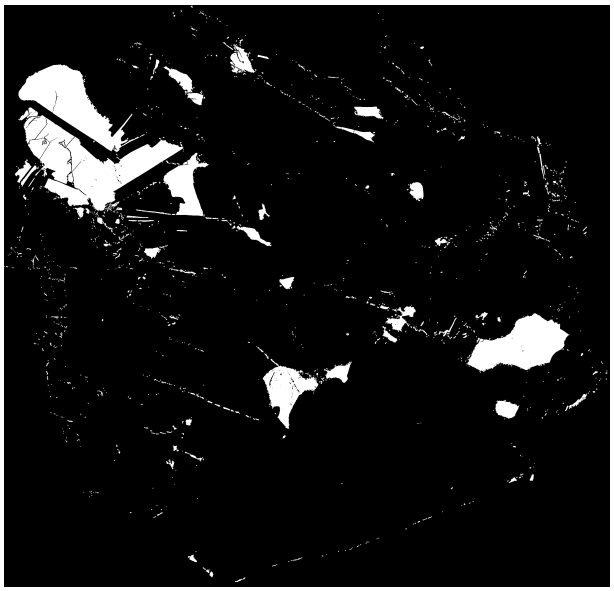


Fonte: O autor, 2019.

Para o cálculo de exposição do sulfeto é necessário submeter as imagens das partículas ao algoritmo de segmentação dos sulfetos (Figura 12), e utilizar o resultado do algoritmo de identificação de bordas internas e externas (Figura 11) aplicado no resultado do algoritmo

de segmentação de fissuras.

Figura 12 – Segmentação de sulfetos.



Fonte: O autor, 2019.

A partir desses dois passos mencionados acima, o resultado é utilizado para identificação das bordas de sulfeto que alcançaram o exterior (Figura 13). O resultado da identificação das bordas de sulfeto que alcançaram o exterior, é utilizado para o cálculo de exposição do sulfeto que é obtido a partir do perímetro exposto de sulfeto sobre a área da partícula.

Figura 13 – Sulfetos que alcançaram o exterior.

Fonte: O autor, 2019.

Segundo o projeto de pesquisa número `E-26/110.124/2014`, o algoritmo de segmentação de fissuras demanda em torno de quatro horas para processar uma amostra com 1400 imagens. Sendo que em alguns casos, após a imagem de resultado do processamento ser analisada pelo pesquisador, é identificada a necessidade de ajuste dos parâmetros de entrada do algoritmo, sendo preciso uma nova execução. Esse processo se repete até que se obtenha-se o resultado esperado.

## 3.1 **Base de dados**

A base de imagens utilizada no presente trabalho foi gerada durante o projeto de pesquisa `(Ciência sem Fronteiras 233182/2012)`, intitulado "Microtomografia aplicada à mineralogia de processo", realizado na *Université de Liege* (Bélgica) , que estudou o efeito de diferentes métodos de britagem na geração de fissuras nas partículas a fim de aumentar a exposição mineral e potencialmente a extração dos metais por biolixiviação. No estudo foram utilizados o britador de mandíbula e dispositivos não-convencionais que envolvem diferentes mecanismos de fratura, tais como prensa de rolos de alta pressão (HPGR) e fragmentação eletrodinâmica Selfrag (Selfrag, 2017). O primeiro processo empregou so-

mente o britador de mandíbula, o segundo utilizou britador de mandíbula em conjunto com a fragmentação eletrodinâmica (Selfrag). Já o terceiro processo empregou o britador de mandíbula e a prensa de rolos de alta pressão (HPGR) sob 20 Bar e o quarto processo utilizou o britador de mandíbula e a prensa de rolos de alta pressão (HPGR) sob 50 Bar. Das quatro bases disponíveis para a execução dos experimentos, foi escolhida uma de forma arbitrária . Ela é relativa aos rejeitos de minérios que passaram pelo segundo processo de britagem que utilizou o britador de mandíbula e a fragmentação eletrodinâmica Selfrag.

A aquisição das imagens foi feita utilizando o detector de elétrons retroespalhados do MEV modelo LEO S440. As imagens de recortes da seção transversal da amostra foram geradas com uma resolução de 0,5$\mu$m/pixel. Para uma dada amostra, a rotina de aquisição fornece 1400 imagens, formando uma grade com $35 \times 40$ recortes cada uma delas com resolução de $2000 \times 1726$ pixels. A sobreposição entre os recortes consecutivos é de cerca de 10% do comprimento de campo em ambas as direções (GOMES et al., 2014).

Uma vez obtidas as imagens dos recortes, foi utilizado um algoritmo de *stitching* para gerar um mosaico de todas as partículas removendo as sobreposições. O algoritmo é capaz de reconstruir grandes imagens a partir de um número arbitrário de recortes de imagens. Para tanto, faz uso da transformada Fourier para a obtenção da melhor sobreposição em termos da medida de correlação cruzada (GONZALEZ; WOODS, 2000).

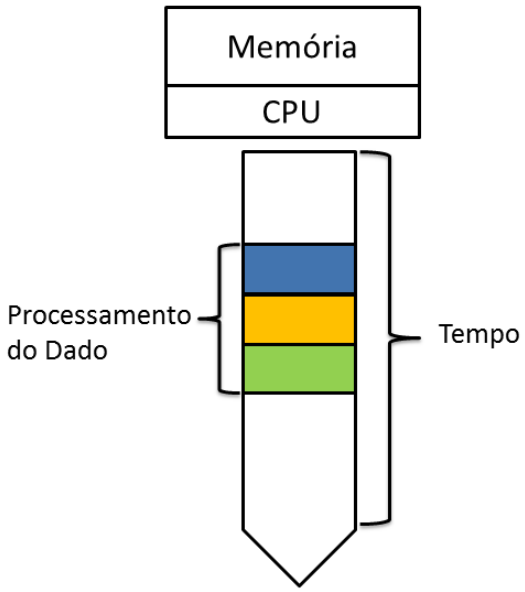
Após a geração do mosaico, as partículas presentes na amostra são separadas. Um algoritmo identifica e remove a representação da resina epóxi na imagem e separa as partículas. A partir da amostra empregada no presente trabalho foram obtidas 240 imagens de partículas entre 1 MB e 30 MB em formato TIF.

## 3.2 Algoritmo Sequencial de Segmentação de Fissuras

O algoritmo original de segmentação de fissuras (GOMES et al., 2014) realiza o processamento de forma sequencial. Esse método processa todas as partículas utilizando um único processador. Fluxos únicos de instruções para o processamento de um único segmento de dados, correspondente à imagem de uma partícula, são representados nas

cores azul, amarelo e verde na Figura 14. Esse tipo de arquitetura é o mais simples, pois opera somente um segmento por vez, contudo, por não se valer de nenhum paralelismo explícito, é lento.

Figura 14 – Processamento Sequencial.




Fonte: O autor, 2019.

O algoritmo de segmentação de fissuras primeiramente faz uma varredura na imagem marcando todos os pixels com intensidade igual a zero. As sementes para o crescimento de regiões são inicializadas, marcando como semente todos os pixels nas bordas dos conjuntos de pixels com intensidade igual a zero.

Após as inicializações e pré-definições de sementes e fissuras, é executado o crescimento de regiões, analisando os pixels vizinhos das sementes que atendem aos critérios descritos a seguir.

- C1: Intensidade menor ou igual a um limiar $ti$;
- C2: Diferença entre a intensidade do pixel e a mediana de uma janela de dimensões $pm \times pm$, centrada no pixel, maior ou igual a um limiar $td$;
- C3: Gradiente de intensidade entre a semente e o pixel vizinho menor ou igual a um limiar $tg$.

Os pixels que atendem aos critérios são marcados como fazendo parte de fissuras. Os pixels recém agregados são usados como sementes na próxima iteração do crescimento de regiões.

O critério de parada do crescimento é que nenhum novo pixel seja marcado como fissura em uma iteração.

A seguir estão descritos os parâmetros de entrada e saída e variáveis do algoritmo sequencial, como também, em português estruturado, sua representação em Algoritmo 1.

(i) $I$: imagem de intensidade (monocromática)

(ii) $ti$: limiar máximo de intensidade

(iii) $td$: limiar mínimo para a diferença de um pixel e a mediana da vizinhança

(iv) $pm$: dimensão de uma vizinhança, $pm \times pm$

(v) $tg$: limiar de magnitude do gradiente

O item listado a seguir descreve a saída do Algoritmo 1.

(i) $F$: imagem binária que identifica as fissuras, de dimensões iguais a $I$. As células de $F$ que representam os pixels de $I$ que fazem parte de fissuras possuem o valor 1.

Os itens listados a seguir descrevem as variáveis do Algoritmo 1.

(i) $l$, $lv$: coordenadas linha de um pixel

(ii) $c$, $cv$: coordenadas coluna de um pixel

(iii) $S$, $S_0$: conjuntos de sementes para o crescimento de regiões

**Algoritmo 1:** Algoritmo de segmentação de fissuras

**Entrada:** $I, pi, pm, pn, tg$

**início**

# Inicializa Regiões

**para** *cada pixel* $I(l,c)$ **faça**

**se** $I(l,c) = 0$ **então**

$F(l,c) \leftarrow 1$

**senão**

$F(l,c) \leftarrow 0$

**fim**

**fim**

# Inicializa Sementes para o Crescimento de Regiões

$S \leftarrow$ *tuplas com as coordenadas das células com valor* 1 *em: F - erosão(F)*

# Procedimento de Crescimento de Regiões

**enquanto** $S \neq \{\}$ **faça**

$S_0 \leftarrow \{\}$

**para** *cada semente* $\{l,c\} \in S$ **faça**

**para** *cada pixel* $I(lv,cv)$ *vizinho de* $I(l,c)$ **faça**

**se** $((I(lv,cv) < ti)$ ***ou*** $(mediana(I(lv - \frac{pm-1}{2} :$

$lv + pm/2, cv - pm/2 : cv + pm/2)) - I(bv,cv) \geq td))$ ***e***

$(gradiente(I(l,c), I(lv,cv)) \leq tg)$ **então**

$F(lv,cv) \leftarrow 1$

$S_0 \leftarrow S_0 + \{lv,cv\}$

**fim**

**fim**

**fim**

$S \leftarrow S_0$

**fim**

**fim**

**Saída:** F

# 4 MÉTODO

O método distribuído de segmentação de fissuras ora apresentado foi concebido para se beneficiar do gerenciamento de arquivos do HDFS, da gerência dos recursos de processamento provida pelo YARN e da facilidade para rodar código nativo do sistema operacional dos nós de processamento, do MR4C. Mais especificamente, o HDFS distribui as imagens pelos nós de processamento, o MR4C encapsula o processamento sequencial de análise de uma partícula e instancia e informa ao sistema distribuído os parâmetros de execução, enquanto o YARN possibilita a execução e gerencia toda a alocação de recursos.

## 4.1 Distribuição das imagens com o HDFS

O HDFS promove a distribuição dos blocos dos dados em nós de processamento e os blocos são gerenciados pelos DataNodes. A configuração do tamanho do bloco pode ser alterada e seu valor deve ser definido baseando-se no tamanho do dado a ser processado. Entretanto, uma imagem pode ter tamanho maior do que o bloco, fato que o HDFS trata de forma transparente ao usuário. Nesta proposta optou-se pela utilização de um bloco de 32 MB, tamanho superior que a maior imagem da base de dados definida para ser utilizada.

É importante frisar que ao utilizar a arquitetura proposta, uma imagem será processada por somente uma tarefa . Isso se dá pelo fato de não ter havido alteração no algoritmo sequencial de segmentação de fissuras para fazer com que este passe a tirar proveito de paralelismo ao nível da *thread*, o que poderia possibilitar a segmentação de uma partícula valendo-se de mais de um *core*.

Outro ponto a ser citado, é o fato de o MR4C distribuir o processamento das imagens em tarefas de forma sequencial, em ordem crescente de nome de imagem. Sendo que o YARN ficará responsável pela definição da localização onde a tarefa será instanciada.

Esta forma de distribuição das imagens pelas tarefas do MR4C pode causar inconvenientes. Em situações onde o tamanho das imagens diverge, isto pode gerar problemas de balanceamento de carga. Para minimizar o problema de balanceamento de carga em situações onde o tamanho das imagens diverge, foi criado um programa que as renomeia

de forma a promover uma melhor disposição de carga por tarefa. O programa de balanceamento, executado antes do envio das imagens ao HDFS, lista as imagens no diretório de entrada por ordem de tamanho e as renomeia, adicionando um número identificador no início do nome das mesmas. Este processo se repete até que todas as imagens no diretório de entrada sejam renomeadas. Para que haja o balanceamento, esse número identificador se inicia em um e é incrementado até atingir o número de tarefas a ser utilizado na configuração da aplicação. Ao exceder esse limite, o identificador inicia-se em um novamente.

## 4.2 **Encapsulamento do processamento com MR4C**

Para adequar o algoritmo de segmentação de fissuras à arquitetura distribuída, foram necessárias adaptações para proporcionar a execução em nós de processamento de forma distribuída, possibilitando um aumento na performance em relação a sua versão sequencial. A adaptação do algoritmo de segmentação de fissuras consistiu em separar todos os includes de bibliotecas e definições de funções em um arquivo de *header*. Após a criação do *header*, este juntamente com a respectiva implementação foram compilados utilizando o parâmetro fPIC, que possibilita que a biblioteca gerada não dependa de estar localizada em um endereço específico (biblioteca compartilhada). Dessa forma, o código acessa todos os endereços de constantes através de uma *global offset table* (GOT) ou tabela de deslocamento global. O operador dinâmico resolve as entradas GOT quando o programa é iniciado, sendo que o operador que realiza o algoritmo de *region growing* torna-se acessível através do sistema operacional (GNU, 2008). Essa maneira de compilar é determinante, pois viabiliza a execução do algoritmo nos nós de processamento onde o código será requisitado.

Foi desenvolvida uma aplicação para execução da função de segmentação de fissuras de forma distribuída. Na aplicação foram implementadas as funções para fazer a conversão dos objetos do tipo DataFile do Hadoop para uma matriz do tipo Mat do OpenCV e vice-versa. Além disto, foi implementada uma estrutura de repetição que faz a leitura do arquivo JSON e obtém cada imagem do HDFS, utilizando as funções de conversão de

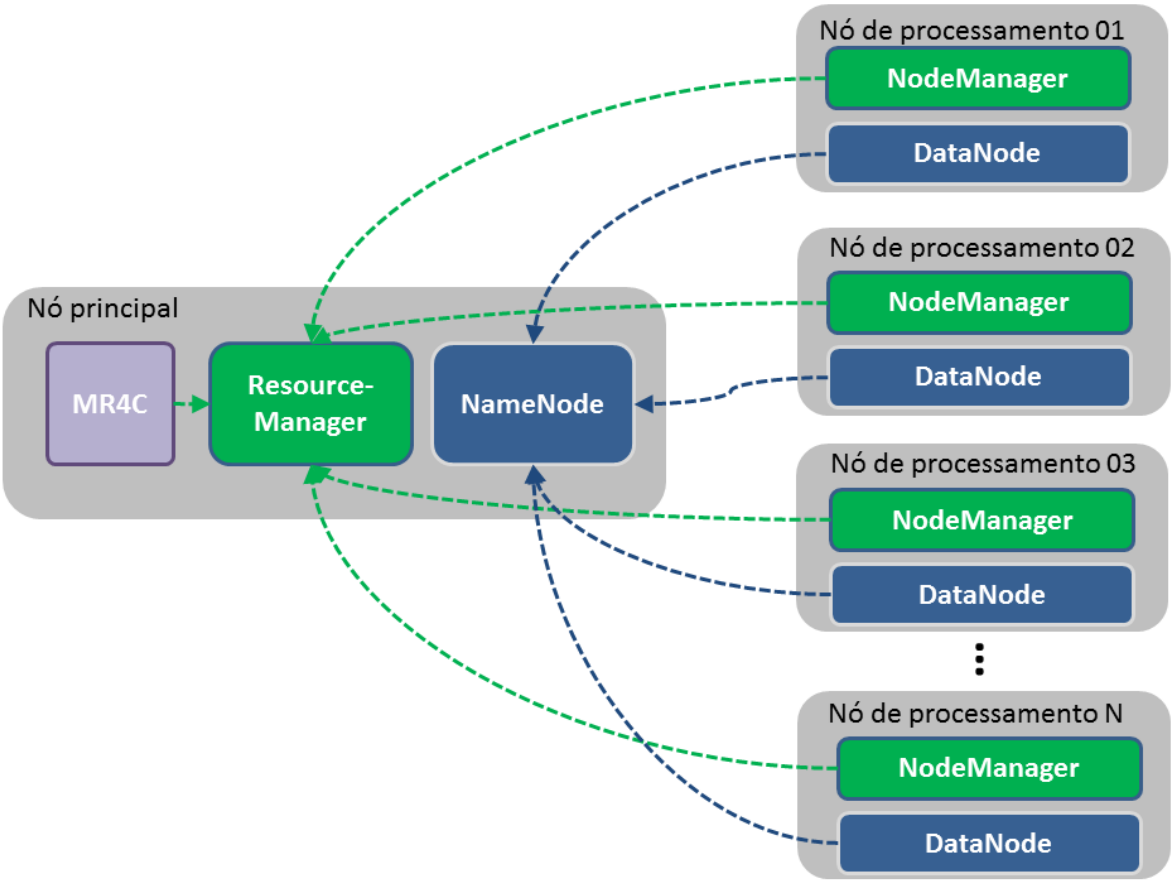
tipo para transformar a imagem com formato DataFile do Hadoop para a matriz Mat do OpenCV. Em seguida, faz a chamada da função de segmentação de fissuras submetendo cada imagem e, a partir do resultado obtido, chama novamente a função de conversão para transformar o objeto para DataFile do Hadoop, possibilitando o armazenamento da imagem processada no formato TIF no HDFS. O arquivo JSON citado acima foi utilizado como base de referência para os parâmetros de entrada da aplicação, como, por exemplo, o local onde o conjunto de imagens a serem processadas foi armazenado na estrutura de diretórios do HDFS como também a extensão do formato do dado.

## 4.3 **Distribuição e Execução do processamento com YARN**

Na abordagem proposta, o YARN é empregado para alocar e gerenciar os recursos de hardware para a execução do algoritmo distribuído de segmentação de fissuras. O YARN faz a requisição dos recursos dinamicamente aos nós de processamento, fazendo com que a escolha do nó que irá processar seja em função da localização dos blocos das imagens. Esta abordagem explora características importantes do Hadoop: a distribuição dos dados em blocos e a computação através de muitos nós de processamento (SHVACHKO et al., 2010). Trata-se de uma arquitetura robusta e facilmente escalável, pois opera com unidades de processamento independentes, cuja quantidade pode ser incrementada sob demanda (SHVACHKO et al., 2010).

A arquitetura representada na Figura 15, é composta por um nó principal de gerenciamento e os outros N nós de processamento.

Figura 15 – Configuração dos serviços.




Fonte: O autor, 2017.

O nó principal deve possuir os recursos de gerenciamento do Yarn, o ResourceManager, e do HDFS, o NodeManager, juntamente com o MR4C. Na infraestrutura utilizada na presente pesquisa, esta separação se mostrou importante para possibilitar uma melhor estabilidade no tempo de resposta. Porém, não é mandatório que se dedique um nó para os serviços de gerenciamento. Já os nós de processamento possuem o recurso de processamento do YARN, o NodeManager, e o recurso de armazenamento do HDFS, o DataNode.

Ao prover mais um servidor ao *cluster*, o mesmo deve ser configurado com os serviços indicados para um nó de processamento, sendo que é comum utilizar somente um nó principal.

Em cada nó de processamento, dados localmente disponíveis, são processados de forma

independente. Assim, o Hadoop transfere o processamento para o nó de processamento onde o dado é armazenado, fazendo com que não haja necessidade de transferência de grande volume de dados na rede. Ele parte do princípio que os blocos das imagens já estão distribuídos pelos DataNodes do HDFS. Desta forma, as partículas podem ser processadas de forma paralela, possibilitando a diminuição do tempo de processamento.

Nesse contexto, o MR4C possibilita a definição da alocação de recursos do *cluster* que a aplicação irá utilizar. Neste caso, o volume de recursos é definido no momento em que a aplicação é instanciada. São disponibilizados três parâmetros para configuração: o número de tarefas, a memória por tarefa e o número de *cores* por tarefa.

A memória por tarefa deve ser configurada para viabilizar a execução da maior imagem da base de dados que irá processar, visando sempre o limite mínimo e assim possibilitar um melhor uso dos recursos.

O número de *cores*, também definido por tarefa, deve ser configurado levando em conta o tipo de algoritmo a ser utilizado. Caso o algoritmo esteja preparado para utilizar mais de uma *thread* no processamento de um dado, esse valor deve ser definido seguindo o limite dos recursos do *cluster*. Caso o algoritmo não esteja preparado para a utilizar mais de uma *thread* no processamento, a configuração ideal é um *core* por tarefa.

Já o número de tarefas é limitado pelo número de dados a serem processados, onde não é possível processar um dado em mais de uma tarefa. A base de dados será dividida pelo número de tarefas e essas tarefas serão processadas em paralelo, tendo como limitador os recursos disponíveis do *cluster*.

É importante destacar que as definições dos parâmetros da aplicação MR4C influenciam diretamente no desempenho da execução da aplicação e, configuradas da forma correta, podem trazer diminuição de tempo de processamento. Entretanto, para utilizá-los de forma ideal, os recursos do *cluster* devem ser providos a fim de não gerar limitações na execução da aplicação.

## 4.4 Referências para definição dos recursos de um *clusters* utilizando a arquitetura proposta

Para fazer a definição ideal de recursos de um *cluster* a partir da arquitetura apresentada, é importante levar em consideração os recursos que mais influenciam no processamento. São eles a quantidade de memória, o número de *cores* e o número de nós de processamento.

A configuração ideal de memória para cada nó de processamento pode ser obtida analisando o valor mínimo da memória máxima requisitada por tarefa necessária para o processamento da maior imagem da base de dados. O valor da memória obtido deve ser multiplicado pelos número de imagens a que se deseja processar de forma paralela. O resultado desse valor deve ser disponibilizado pelos nós do *cluster*. Deve-se enfatizar que um limite superior de paralelismo em um nó de processamento é dado pela razão da memória disponível no nó pela memória utilizada na configuração da tarefa.

Deve-se levar em conta adicionalmente a memória utilizada pelo sistema operacional. Como também o fato de do nó de processamento ter memória múltipla do valor a ser alocado por tarefa mais a memória mínima necessária para o sistema operacional.

O número de *cores* deve ser igual ou maior ao número de tarefas a que se deseja processar em paralelo por nó de processamento, adicionado, ao menos, um *core* dedicado ao sistema operacional.

Outro ponto é que o número de nós de processamento deve ser o maior possível, até limite do número de imagens utilizada na base de dados. Por exemplo, dada uma base de dados composta por vinte imagens, sendo que a maior imagem precisa de 2 GB para ser processada, e é necessário processar pelo menos dez imagens em paralelo. Caso tenhamos cinco nós de processamento e o mínimo de memória necessária para o sistema operacional é 1 GB, então o valor da memória ideal para cada nó de processamento seria de pelo menos 5 GB. Neste exemplo, para que seja possível atingir a meta de dez imagens sendo processadas de forma paralela, também é necessário que o número de tarefas seja igual a dez, ou múltiplo de dez. Por sua vez, o número de nós por *core* é de no mínimo três, sendo um para o sistema operacional e dois para processar tarefas do Hadoop.

# 5 PROCEDIMENTO EXPERIMENTAL, RESULTADOS E ANÁLISE

Os experimentos realizados têm como objetivo avaliar a escalabilidade e ganhos de desempenho com a distribuição do processamento e paralelismo utilizando a arquitetura proposta no presente trabalho.

Serão utilizadas duas métricas para a avaliação dos resultados, o tempo de execução e o *speedup* (KENT; WILLIAMS, 1992). O tempo de execução consiste em medir o período temporal entre o início da execução da aplicação até o término da mesma. Esta métrica é obtida através da função *time* (UBUNTU, 2010b). O *speedup* consiste na razão entre os tempos de processamento da versão sequencial da segmentação de fissuras e o da versão distribuída, proposta nesta dissertação. Além disto, foi utilizado como parâmetro de comparação o *speedup* teórico para o caso linear. Nos experimentos onde foram utilizados como referência, também foi apresentada a expressão do cálculo para a obtenção do caso linear.

As seções a seguir apresentam o procedimento experimental e em seguida são fornecidos e discutidos em separado os resultados de cada um dos experimentos. Por fim, é feita a análise das regularidades nos resultados obtidos nos diversos experimentos.

## 5.1 Procedimento Experimental

O algoritmo de segmentação de fissuras com uso do *region growing* foi executado no Hadoop, variando-se o número de *cores*, a memória máxima por tarefa, o número de tarefas, o número de imagens, o número de cores por tarefa e o número de nós de processamento. Visa-se à obtenção de configurações otimizadas, à avaliação da distribuição do processamento e à análise de ganhos e limitações do método desenvolvido, além de sua escalabilidade. Nesta sessão, serão apresentadas as justificativas que levaram o planejamento deste conjunto de experimentos.

A influência do número de *cores* por tarefa deve ser verificada. Entretanto, é sabido que o algoritmo de segmentação de fissuras executado em cada nó do *cluster*, na versão avaliada nestes experimentos, não se valeu de nenhum paralelismo ao nível de *thread*, tendo sido mantido de forma sequencial no Hadoop. Todavia, é necessário identificar

efeitos adversos do aumento do número de *cores* que tende, no caso, a reservar recursos sem que se tire proveito destes, limitando sobremaneira os recursos disponíveis para as demais tarefas. A fim de estudá-los de forma a obter as melhores configurações para os experimentos seguintes, será utilizado um único nó de processamento forçando o aumento da concorrência por recursos à medida que o número de *cores* por tarefa é incrementado e varia-se o número de tarefas.

Assim como a influência do número de *cores*, a influência da memória máxima por tarefa também deve ser verificada. Entretanto, é sabido que os nós do *cluster* tem uma quantidade limitada de memória. Assim, considera-se como situação ideal utilizar o mínimo de memória possível para possibilitar que mais tarefas possam compartilhar a memória nos nós. Todavia, é necessário identificar efeitos adversos do aumento da memória máxima a fim de estudá-los de forma a obter melhores configurações para os experimentos seguintes. Neste caso, será utilizado um único nó de processamento para aumentar a concorrência de recursos a medida que a memória máxima por tarefa é incrementada.

Não só a influência do número de *cores* e memória, mas também a influência do número de nós de processamento deve ser verificado. É sabido que o Hadoop foi projetado para possibilitar a escalabilidade partindo de um único nó de processamento para milhares. Todavia, é necessário identificar efeitos adversos do aumento do número de nós de processamento. Neste caso, será reservado um nó exclusivamente para o gerenciamento, e como o *cluster* utilizado nos experimentos tem dez nós (conforme descrito na próxima seção), é realizada a variação de um a nove no número de nós de processamento. O nó de gerenciamento se faz necessário para evitar a sobrecarga causada pelos serviços de gerenciamento do Hadoop YARN em nós que executem simultaneamente o ResourceManager, o NodeManager, o NameNode e o DataNode.

Também a influência do número de imagens deve ser verificada. É sabido que o Hadoop pode ser, em alguns casos, uma alternativa para processar grandes volumes de dados estruturados, semi-estruturados e não-estruturados. Contudo, é necessário identificar efeitos adversos do aumento do número de imagens a fim de estudá-los de forma a obter variação do tempo de processamento em relação ao aumento do volume de dados. Neste

caso, serão utilizadas três configurações de nós de processamento para também avaliar a escalabilidade com o aumento do volume de dados.

Assim como as configurações por tarefa, a influência do número de tarefas também deve ser verificada. Os efeitos adversos do aumento de tarefas precisam ser avaliados de forma a obter a variação do tempo de processamento em relação ao número de tarefas. Com este experimento, será avaliado o número de tarefas que podem ser processadas em paralelo a partir de uma determinada configuração de memória.

A influência do tamanho da imagem deve ser verificada, é sabido que a granularidade de imagens por número de tarefa do MR4C é igual a um. No entanto, é necessário identificar efeitos adversos do aumento do número de nós de processamento ao processar uma imagem maior que o bloco do HDFS. Assim, espera-se que seja possível avaliar a viabilidade de obter-se a redução do tempo de processamento com a demanda de mais de um bloco por imagem.

O tempo de execução do algoritmo sequencial utilizando a mesma imagem processada pelo Hadoop no experimento anterior deve ser verificado. É sabido que o Hadoop foi desenvolvido para processar grande volume de dados, entretanto é necessário obter esse tempo de execução para servir como base para identificar o *overhead* ao comparar-se o processamento sequencial com o processamento distribuído no Hadoop.

Assim como a influência do número de imagens, a influência do número de imagens em grande escala também deve ser verificada. É necessário identificar efeitos adversos do aumento do número de imagens em grande escala a fim de estudá-los de forma a obter a variação do tempo de processamento em relação ao aumento do volume de dados. Neste caso, será utilizada uma configurações de nove nós de processamento e quantidades distintas de imagens.

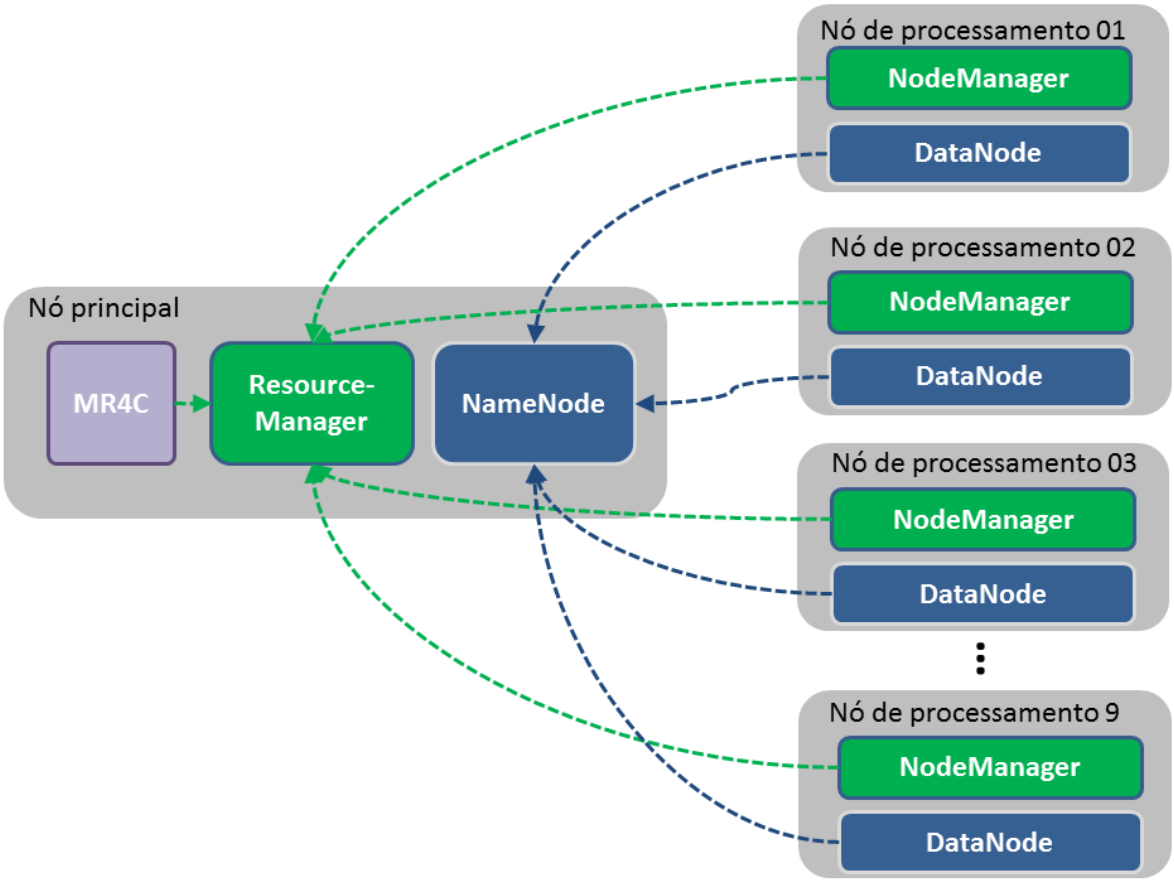
Além disto, o tempo de execução do algoritmo sequencial processando a base de imagens Selfrag deve ser verificado. Para tanto, é necessário obter o tempo de execução para servir como base de comparação com o tempo obtido utilizando o algoritmo distribuído com o Hadoop numa situação real.

## 5.2 Infraestruturada do *cluster* utilizado nos experimentos

Os experimentos foram realizados em um *cluster* composto de 10 nós. Por sua vez, cada nó do *cluster* possui dois processadores Intel®Xeon-E5345, com 4 *cores* cada um, frequência de *clock* de $2,33$ GHz, memória RAM com 8 GB DDR2 667MT/s *dual channel*. O nós do *cluster* são interconectados através de uma rede ethernet Gigabit.

No *cluster*, o Hadoop foi instalado e configurado a partir da distribuição Cloudera, versão 5.9, que incorpora a versão 2.6 do Hadoop, o MR4C e o OpenCV, juntamente com todos os pré-requisitos de cada produto mencionado. A arquitetura do *cluster* é representada na Figura 16, dos dez nós disponíveis reservou-se um como nó principal de gerenciamento e os outros nove são nós de processamento.

Figura 16 – Configuração dos serviços.




Fonte: O autor, 2017.

O nó principal possui os recursos de gerenciamento do YARN, o ResourceManager, e

do HDFS, o NodeManager, juntamente com o MR4C. Já os nós de processamento possuem o recurso de processamento do YARN, o NodeManager, e o recurso de armazenamento do HDFS, o DataNode.

## 5.3 **Base de dados utilizada nos experimentos**

A partir de imagens de partículas de minérios de sulfeto de cobre, submetidas ao britador de mandíbula e posteriormente ao mecanismo de fragmentação eletrodinâmica Selfrag, foram constituídos diversos conjuntos de imagens a serem utilizados nos experimentos. A Tabela 1 apresenta como foram criadas as bases de dados usadas nos experimentos.

Tabela 1 – Base de dados utilizada nos experimentos

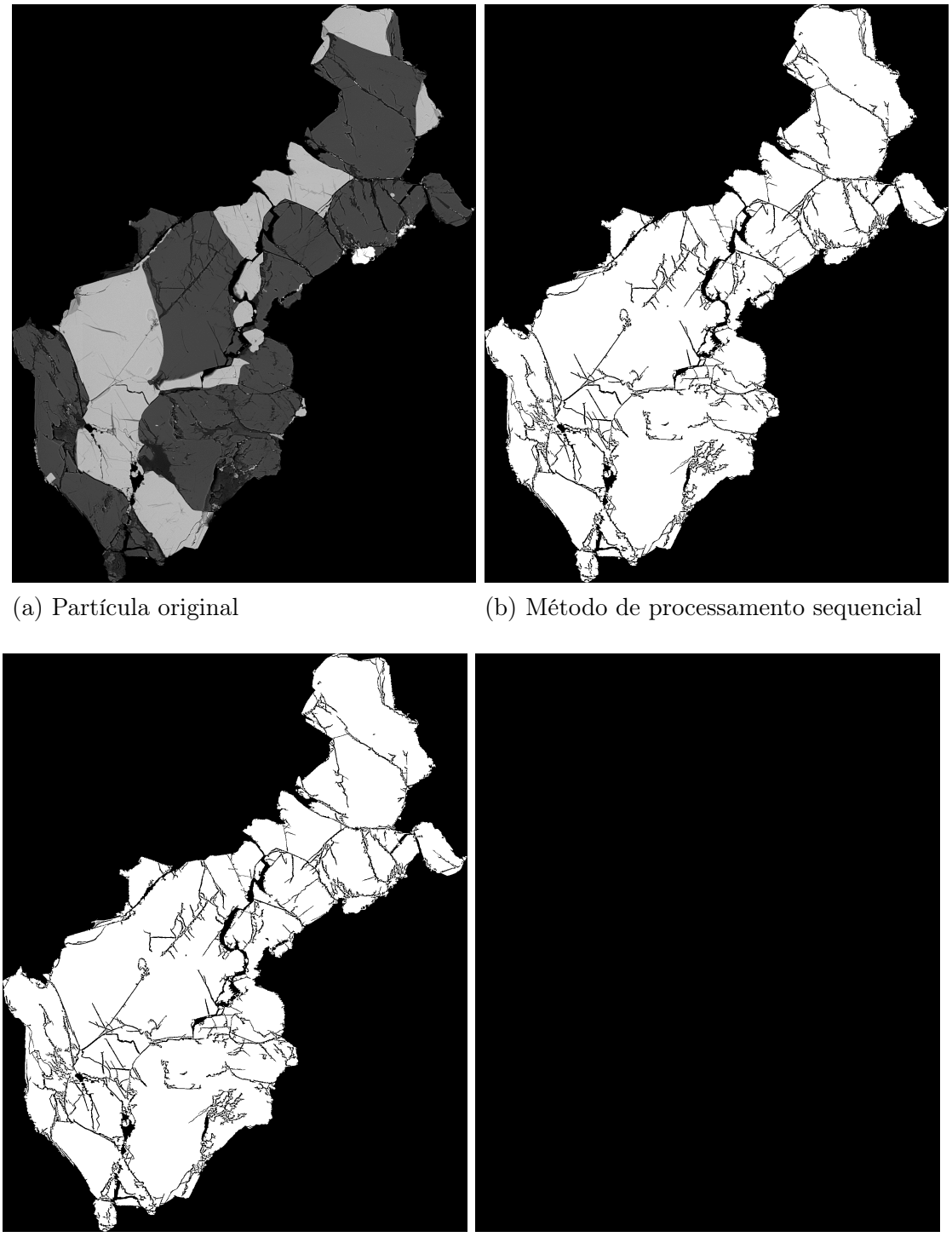
| **Código da base de dados** | **Número de Imagens** | **Tamanho aproximado por imagem em Pixels** | **Tamanho aproximado por imagem compactada em MB** | **Tamanho aproximado por imagem descompactada em MB** | **Nome das imagens** |
|---|---|---|---|---|---|
| Selfrag_20 | 20 | 1200 x 1200 a 2000 x 2000 | 0.6 a 1.4 | 1 a 2.5 | Cu1_20_Selfrag_0218.tif, Cu1_20_Selfrag_0221.tif, Cu1_20_Selfrag_0036.tif, Cu1_20_Selfrag_0219.tif, Cu1_20_Selfrag_0217.tif, Cu1_20_Selfrag_0151.tif, Cu1_20_Selfrag_0002.tif, Cu1_20_Selfrag_0035.tif, Cu1_20_Selfrag_0093.tif, Cu1_20_Selfrag_0220.tif, Cu1_20_Selfrag_0216.tif, Cu1_20_Selfrag_0215.tif, Cu1_20_Selfrag_0200.tif, Cu1_20_Selfrag_0212.tif, Cu1_20_Selfrag_0214.tif, Cu1_20_Selfrag_0213.tif, Cu1_20_Selfrag_0209.tif, Cu1_20_Selfrag_0211.tif, Cu1_20_Selfrag_0134.tif, Cu1_20_Selfrag_0207.tif, |
| Selfrag_221 | 221 | 1200 x 1200 a 9000 x 9000 | 0.6 a 30 | 1 a 76 | Todas as imagens de Selfrag: Cu1_20_Selfrag_0001.tif a Cu1_20_Selfrag_0221.tif |
| Selfrag_40_0001 | 40 | 8700 x 8700 | 30 | 76 | 40 cópias de Cu1_20_Selfrag_0001.tif |
| Selfrag_80_0001 | 80 | 8700 x 8700 | 30 | 76 | 80 cópias de Cu1_20_Selfrag_0001.tif |
| Selfrag_120_0001 | 120 | 8700 x 8700 | 30 | 76 | 120 cópias de Cu1_20_Selfrag_0001.tif |
| Selfrag_180_0031 | 180 | 4000 x 5700 | 10 | 14 | 180 cópias de Cu1_20_Selfrag_0031.tif |
| Corte_Mosaico | 1 | 10000 x 8000 | 72 | 105 | Corte do mosaico teste4GB.tif com oito particulas |
| Selfrag_180_0001 | 180 | 8700 x 8700 | 30 | 76 | 180 cópias de Cu1_20_Selfrag_0001.tif |
| Selfrag_360_0001 | 360 | 8700 x 8700 | 30 | 76 | 360 cópias de Cu1_20_Selfrag_0001.tif |
| Selfrag_720_0001 | 720 | 8700 x 8700 | 30 | 76 | 720 cópias de Cu1_20_Selfrag_0001.tif |

Fonte: O autor, 2017.

## 5.4 **Validação do modelo distribuído**

A Figura 17 apresenta os resultados fornecidos pelas abordagens sequencial e distribuída para a imagem Cu1_20_Selfrag_0220.tif apresentada na Figura 17(a). Ao fazer a comparação dos resultados obtidos no processamento das imagens utilizando o método sequencial, Figura 17(b), e o método distribuído de segmentação de fissuras, Figura 17(c), foi possível observar que não houve diferença nos resultados obtidos. Para comprovar que os resultados são idênticos, foi efetuada a subtração da imagem de resultado gerada pelo método sequencial pela imagem gerada pelo método distribuído, Figura 17(d).

Figura 17 – Comparação do resultado do processamento das partículas utilizando o método sequencial e o método distribuído de segmentação de fissuras.



(a) Partícula original

(b) Método de processamento sequencial

(c) Método de processamento distribuído

(d) Subtração da imagem b pela imagem c

Fonte: O autor, 2017.

## 5.5 Influência do número de *cores* por tarefa

Este experimento visa avaliar a influência do número de *cores* por tarefa no desempenho do procedimento de segmentação de fissuras com uso do *region growing*. Para tanto, será utilizado apenas um nó de processamento no *cluster*, enquanto, tarefas e *cores* são variadas a partir do produto cartesiano entre $\{1, 2, 3, 4\}$ *cores* e $\{1, 2, \ldots, 20\}$ tarefas. Esta variação de número de *cores* por tarefa e número de tarefas foi realizada conforme a Tabela 2.

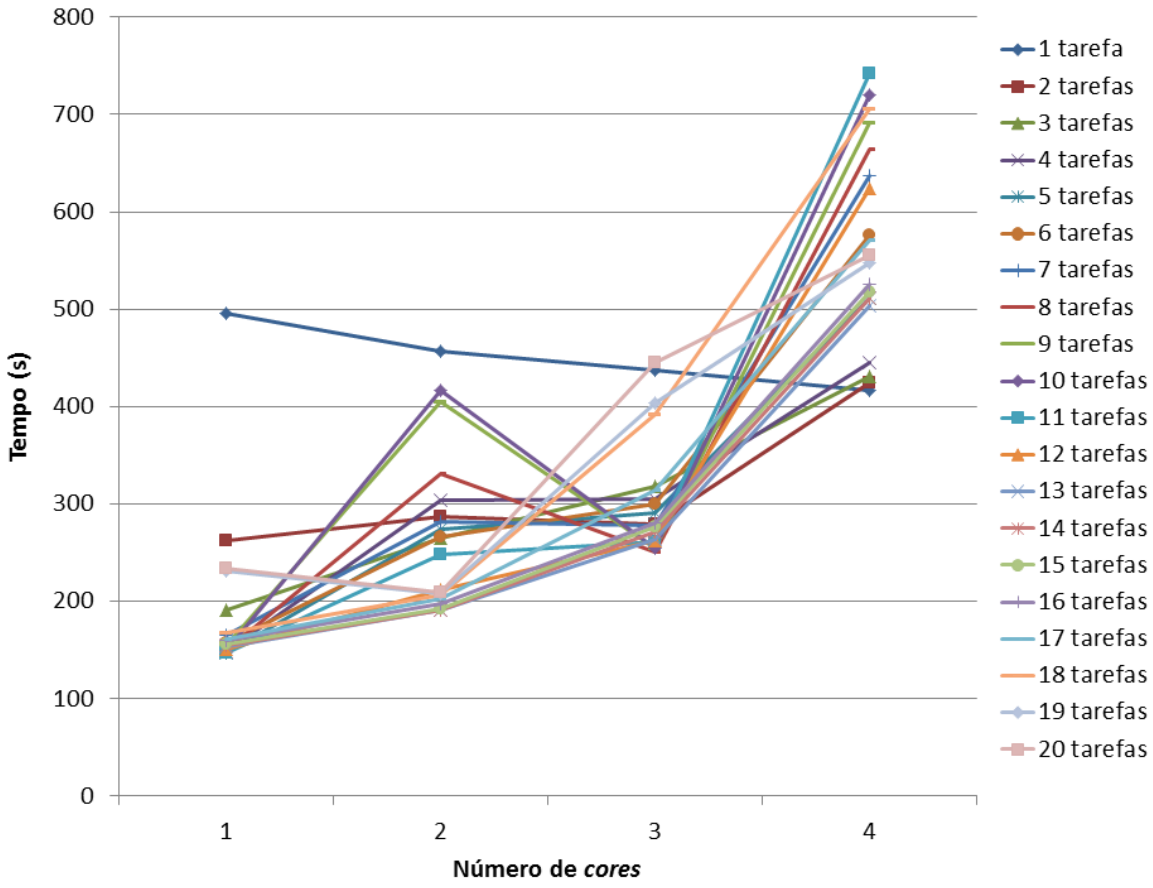
Tabela 2 – Variação de *cores* por tarefa

| Descrição | Base de dados | *cluster* | Tarefa | | |
|---|---|---|---|---|---|
| | | Numero de nós | Número de tarefas | Memória máxima | Número de *cores* |
| Variação do número de *cores* por tarefa | Selfrag_20 | 1 | 1 a 20 | 1.5 GB | 1, 2, 3, 4 |

Fonte: O autor, 2017.

Foram realizadas quatro repetições para cada configuração, no total de 320 execuções. Os tempos médios obtidos nas quatro execuções de cada uma das configurações são apresentados na Figura 18.

Figura 18 – Variação do número de *cores* por tarefa




Fonte: O autor, 2017.

Analisando os resultados apresentados na Figura 18 nota-se que para a maioria dos casos um *core* por tarefa forneceu o menor tempo de execução. Como exceções estão os experimentos realizados para uma tarefa, 19 e 20 tarefas para os quais quatro, dois e dois *cores* forneceram respectivamente os menores tempos de execução.

Uma justificativa para o aumento do tempo pode ser dada partindo do princípio que o ResourceManager só aloca recursos disponíveis no *cluster*, assim sendo, caso a aplicação tenha sido configurada com mais recursos do que o *cluster* disponibiliza, algumas tarefas só iniciarão com a liberação de recursos de uma tarefa já finalizada. Por exemplo, dado um experimento com a configuração de quatro tarefas utilizando quatro *cores*, em um *cluster* com um nó de processamento possuindo oito *cores*, no máximo duas tarefas serão processadas de forma paralela. Pelo menos duas tarefas terão que aguardar as primeiras

serem encerradas e os recursos liberados, para que elas possam alocar os recursos e iniciar o processamento.

Em decorrência dos resultados apresentados na Figura 18 nos experimentos posteriores será utilizado apenas um *core* por tarefa.

## 5.6 **Influência da memória máxima definida por tarefa**

Este experimento visa avaliar a influência da memória máxima por tarefa no desempenho do procedimento de segmentação de fissuras com uso do *region growing*. Para tanto será utilizado apenas um nó de processamento no *cluster*, enquanto número de tarefas e memória máxima são variadas a partir do produto cartesiano entre $\{1.5, 2, 2.5, 3\}$ GB de memória máxima e $\{1, 2, \ldots 20\}$ tarefas. Esta variação de memória máxima por tarefa e número de tarefas foi realizada conforme a Tabela 3.

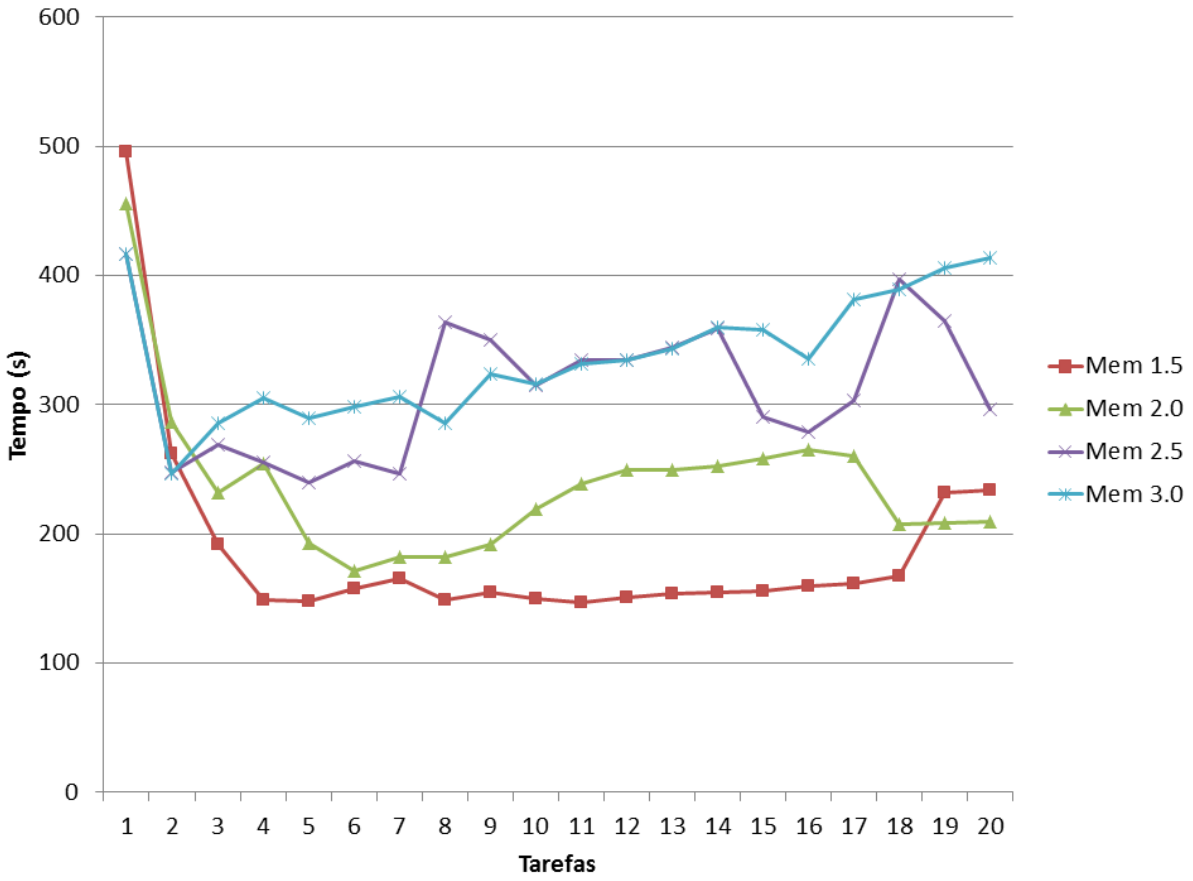
Tabela 3 – Variação do limite de memória

| Descrição | Base de dados | *cluster* | Tarefa | | |
|---|---|---|---|---|---|
| | | Numero de nós | Número de tarefas | Memória maxima | *cores* |
| Variação do limite de memória | Selfrag_20 | 1 | 1 a 20 | 1.5 GB, 2 GB, 2.5 GB, 3GB | 1 |

Fonte: O autor, 2017.

Foram feitas quatro repetições utilizando cada configuração, no total de 80 execuções. Os resultados foram apresentados na Figura 19.

Figura 19 – Variação do limite de memória por tarefa




Fonte: O autor, 2017.

Analisando os resultados apresentados na Figura 19 nota-se que, para a maioria dos casos, a diminuição da a memória máxima proporcionou menor tempo de execução. Como exceções estão os experimentos realizados para duas tarefas, 8, 9, 18, 19 e 20 tarefas para os quais alguns valores de memória máxima causaram um comportamento distinto. Assim sendo, em geral, a redução da memória máxima diminuiu o tempo de execução, pois possibilitou um melhor uso dos recursos do nó de processamento.

Em decorrência dos resultados apresentados na Figura 19 ficou definido que, para cada experimento seguinte, fosse feita uma pré-avaliação para identificar a menor memória alocada que possibilite a execução sem que ocorra erro por insuficiência de memória.

## 5.7 **Influência do número de nós de processamento**

Este experimento visa avaliar a influência do número de nós de processamento no desempenho do procedimento de segmentação de fissuras com uso do *region growing*. Para tanto serão utilizados de um a nove nós de processamento no *cluster*. Em todas as configurações de nós, serão executadas simultaneamente duas aplicações. A primeira aplicação utiliza 25 tarefas com 3 GB para segmentar as 25 maiores imagens da base de dados Selfrag_221. A segunda aplicação utiliza 20 tarefas de 1,5 GB para processar as demais imagens. Esta variação de nós de processamento foi realizada conforme a Tabela 4.

Tabela 4 – Variação do número de nós de processamento

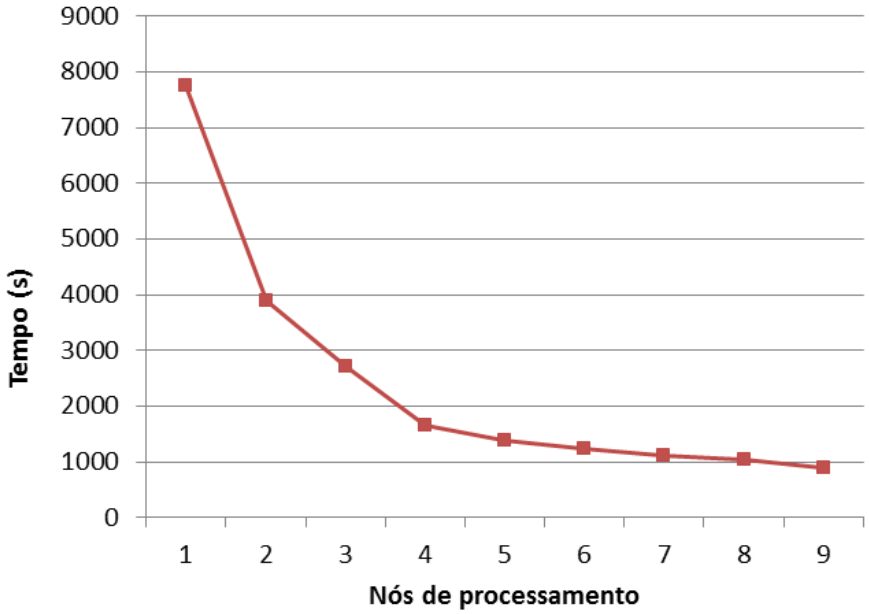
| Descrição | Base de dados | *cluster* | Tarefa | | |
|---|---|---|---|---|---|
| | | Numero de nós | Número de tarefas | Memória maxima | *cores* |
| Variação do número de nós de processamento | Selfrag_221 | 1,2,3,4,5, 6,7,8,9 | 25 tarefas processando 25 imagens de 10 a 30 MB + 20 tarefas processando imagens de 1 a 10 MB | Memória de 3GB para 25 tarefas + Memória de 1.5GB para 20 tarefas | 1 |

Fonte: O autor, 2017.

Neste experimento, a justificativa para a execução simultânea de duas aplicações executando grupos de imagens de tamanho distintos se dá no fato de possibilitar uma melhor utilização dos recursos de memória do *cluster*. Para tanto, é necessário definir parâmetros de memória diferentes para conjuntos de imagens com faixas de tamanhos distintos. Foram feitos alguns experimentos preliminares para definir as configurações mínimas de memória máxima capazes de processar cada grupo de imagens. Os tempos de execução obtidos para a execução concorrente de duas aplicações, sendo a primeira destinada a segmentar as 25 imagens com maiores dimensões, havendo uma tarefa por imagem e 3 GB por tarefa, e a segunda responsável por processar as demais imagens da base, instanciando-se

20 tarefas e 1,5 GB por tarefa, são apresentados na Figura 20.

Figura 20 – Variação do número de nós de processamento

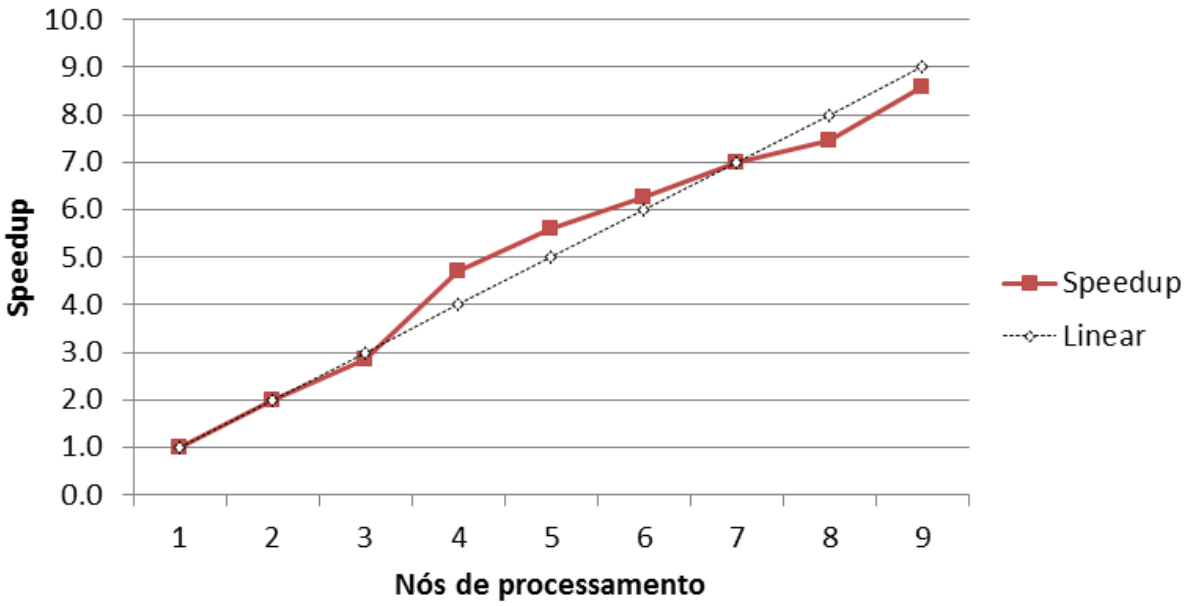



Fonte: O autor, 2017.

Analisando os resultados apresentados na Figura 20 nota-se que o aumento do número de nós de processamento proporcionou menor tempo de execução. Através destes resultados foi possível constatar que na faixa de número de nós avaliada a proposta demonstrou boa escalabilidade.

A Figura 21 apresenta o *speedup* usando como referência para o cálculo o tempo de execução obtido para o *cluster* com apenas um nó de processamento.

Figura 21 – *Speedup* variando o número de nós de processamento




Fonte: O autor, 2017.

Avaliando os resultados apresentados na Figura 21 destacou-se o fato dos resultados apresentarem um *speedup* quase linear, atingindo um valor de 8.6 fazendo uma comparação com a execução utilizando um e nove nós de processamento. Entretanto, apesar do ganho obtido ser bem próximo ao do caso linear, deve-se lembrar que dentre os *cores* dos nós de processamento, no limite máximo destas configurações de aplicação, estarão sendo executadas no máximo quatro tarefas em paralelo em cada nó de processamento. Assim, é provável que ao menos três *cores* estejam fora de uso.

## 5.8 Influência do número de imagens

Este experimento visa avaliar a influência do número de imagens no desempenho do procedimento de segmentação de fissuras com uso do *region growing*. Para tanto, será utilizado uma tarefa por imagem na base de dados, cada uma delas com 3 GB de memória máxima e um *core*. Neste experimento serão utilizadas as bases de dados Selfrag_40_0001, Selfrag_80_0001, Selfrag_120_0001 e 1, 4 e 8 nós. Deve-se lembrar que estas bases de dados compreendem 40, 80 e 120 cópias da imagem Cu1_20_Selfrag_0001.tif, fator que

tende a eliminar a influência de possíveis problemas de desbalanceamento de carga. É preciso mencionar que esta imagem em formato TIF compactado com o algoritmo LZW ocupa 30 MB. Esta variação de número de imagens e nós de processamento foi realizada conforme a Tabela 5.

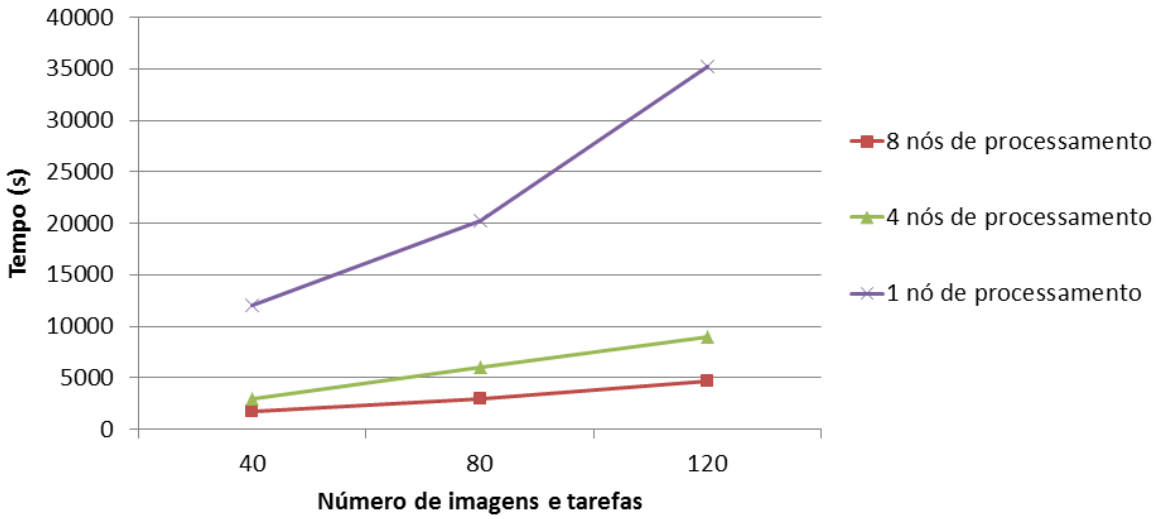
Tabela 5 – Variação do número de imagens

| Descrição | Base de dados | *cluster* | Tarefa | | |
|---|---|---|---|---|---|
| | | Número de nós | Número de tarefas | Memória maxima | *cores* |
| Variação do número de imagens | Selfrag_40_0001, Selfrag_80_0001, Selfrag_120_0001 | 1, 4 e 8 | Igual ao número de imagens | 3 GB | 1 |

Fonte: O autor, 2017.

Em decorrência dos requisitos de memória para o processamento desta imagem, foram feitos alguns experimentos preliminares para definir a menor memória máxima capaz de processar uma imagem com esse tamanho. Assim, as tarefas foram configuradas com 3GB de memória máxima. Os tempos de execução obtidos para as diversas variantes são apresentados na Figura 22.

Figura 22 – Tempo de execução variando os números de imagens e de nós de processamento no *cluster*, enquanto, o número de tarefas equivale ao número de imagens.

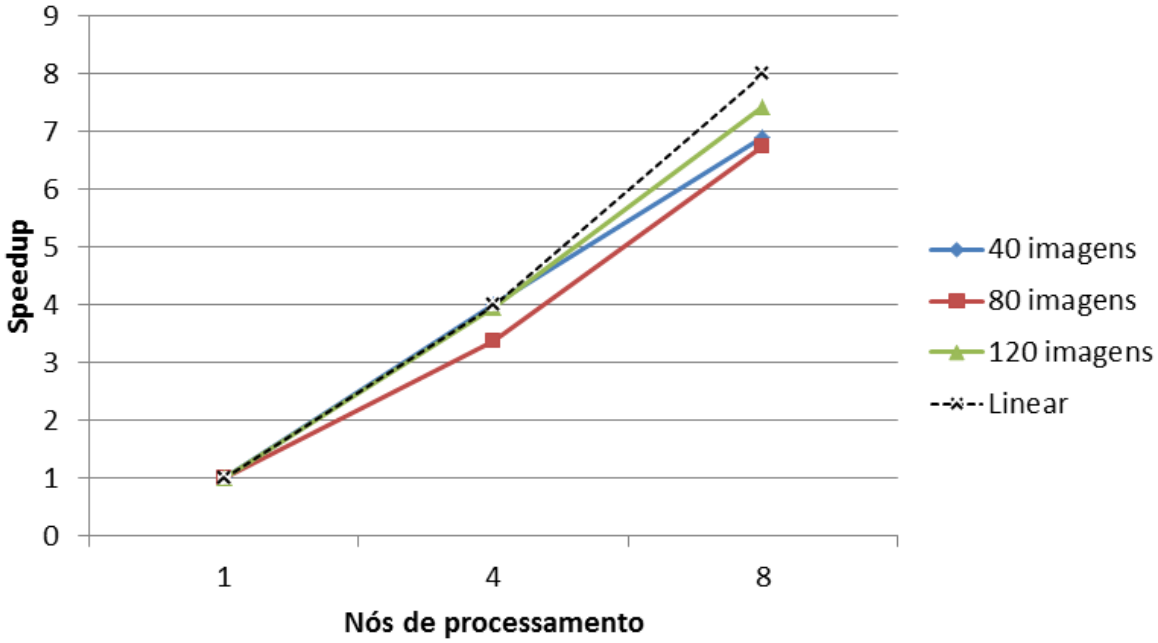



Fonte: O autor, 2017.

Analisando os resultados apresentados na Figura 22 foi verificado que a proposta apresenta boa resposta ao aumento da escala do problema.

A Figura 23 apresenta o *speedup* usando como referência os resultados para um nó de processamento no *cluster*, variando-se o número de imagens e o número de nós de processamento, enquanto, o número de tarefas permanece igual ao número de imagens.

Figura 23 – *Speedup* usando como referência os resultados para um nó de processamento no *cluster*, variando-se o número de imagens e o número de nós de processamento, enquanto, o número de tarefas permanece igual ao número de imagens




Fonte: O autor, 2017.

Avaliando os resultados apresentados na Figura 23 observa-se que a solução proposta responde bem ao aumento da escala do problema. Além disto, os valores de *speedup* obtidos para os valores analisados pode se aproximam do caso linear. No caso do processamento da base de dados Selfrag_120_0001, utilizando oito nós de processamento foi obtido um *speedup* de 7,4 enquanto o valor para o caso hipotético linear seria de 8.

## 5.9 Influência do número de tarefas

Este experimento visa avaliar a influência do número de tarefas no desempenho do procedimento de segmentação de fissuras com uso do *region growing*. Para tanto, serão utilizados $\{9, 18, 36, 45, 90, 180\}$ tarefas, enquanto o número de nós de processamento, *cores* e memória máxima permanecerão inalterados. Esta variação de número de tarefas foi realizada conforme a Tabela 6.

Tabela 6 – Descrição do experimento para avaliação da variação do número de tarefas.

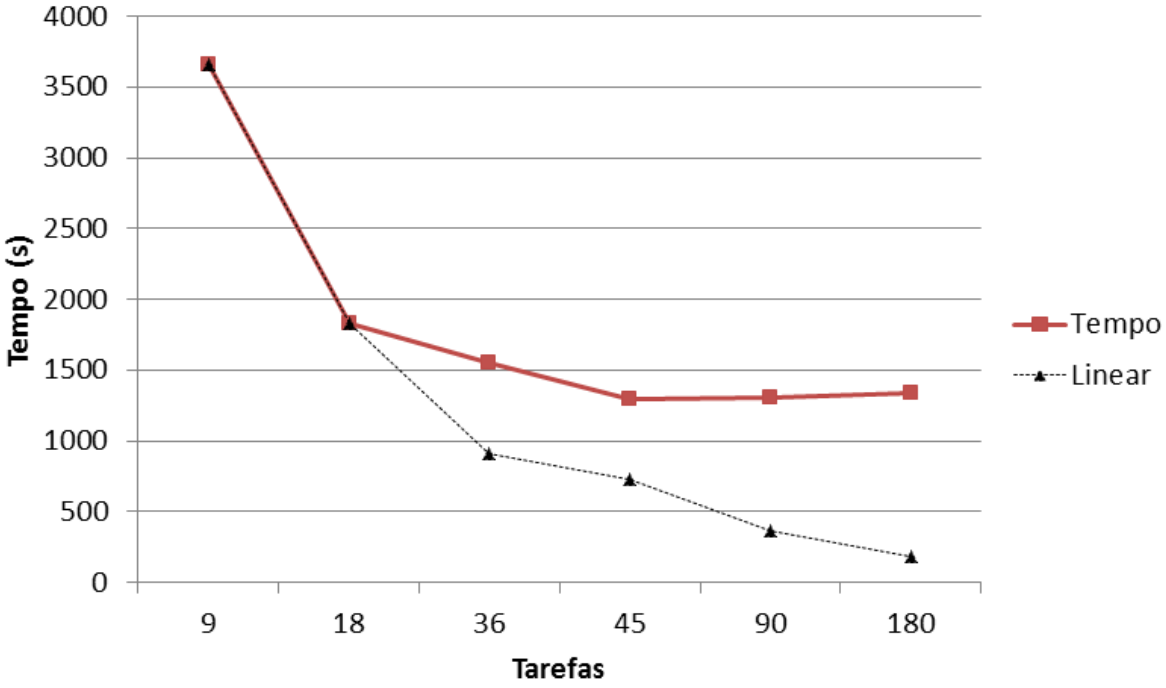
| Descrição | Base de dados | *cluster* | Tarefa | | |
|---|---|---|---|---|---|
| | | Numero de nós | Número de tarefas | Memória maxima | *cores* |
| Variação do número de tarefas | Selfrag_180_0031 | 9 | 9, 18, 36, 45, 90, 180 | 2.5 GB | 1 |

Fonte: O autor, 2017.

Neste experimento, foi utilizada a base de dados Selfrag_180_0031. Todas as imagens da referida base de dados são cópias de uma imagem de aproximadamente 10 MB. Em decorrência disso, foram feitos alguns experimentos preliminares para definir a menor memória máxima capaz de processar uma imagem com esse tamanho. Assim, as tarefas foram configuradas com 2,5 GB de memória máxima. Os resultados foram apresentados na Figura 24.

Analisando os tempos de execução apresentados na Figura 24, nota-se que, dentre as configurações avaliadas neste experimento, o menor tempo de processamento foi obtido para 45 tarefas.

Figura 24 – Comparação do tempo de execução para a variação do número de tarefas ao processar a base de dados Selfrag_180_0031 em um *cluster* com 9 nós com o tempo de execução teórico linear. Cada tarefa instanciada requer 2,5 GB de memória e um core.




Fonte: O autor, 2017.

Como referência para a avaliação do tempo de execução, utilizou-se o tempo de execução relativo linear teórico. Seu valor corresponde à razão do número de tarefas do tempo base pelo número de tarefas de cada execução, sendo em seguida este valor multiplicado pelo resultado do tempo de execução do tempo base. Com essa métrica, ao avaliar os resultados apresentados na Figura 23, observa-se que o resultado obtido escalou linearmente para 9 e 18 tarefas.

Esse comportamento pode ser explicado pelo fato de os nós de processamento possuírem somente 8 GB de memória RAM, sendo que os recursos do Hadoop estão configurados para utilizar até 7.5 GB de memória, dedicando pelo menos 512 MB para os serviços do sistema operacional. Em contra partida, cada tarefa foi configurada com 2.5 GB de memória, o que torna possível a execução de até 3 tarefas em paralelo por nó de processamento. Assim, utilizando nove nós de processamento, é factível haver até 27 tarefas executando

em paralelo.

Fazendo o uso da linha de raciocínio apresentada anteriormente, é possível constatar que foi obtido um *speedup* linear igual a 2.0 com 18 tarefas e a partir de 36 tarefas, os resultados ficaram abaixo do linear e tenderam a um *speedup* de 2.8.

O raciocínio apresentado pôde ser comprovado analisando os *logs* de execução do YARN. Portanto, se for configurado o número de tarefas maior que o limite de recursos de hardware, as tarefas que excederem ao limite de recursos terão que aguardar a liberação de recursos para poderem ser instanciadas.

## 5.10 **Influência do tamanho da imagem**

Este experimento visa avaliar a influência do tamanho da imagem no desempenho do procedimento de segmentação de fissuras com uso do *region growing*. Para tanto, serão utilizados $\{1, 3, 4, 6, 7, 9\}$ nós de processamento, enquanto o número de tarefas, *cores* e memória máxima permanecerão inalterados. Esta variação de nós de processamento foi realizada conforme a Tabela 7.

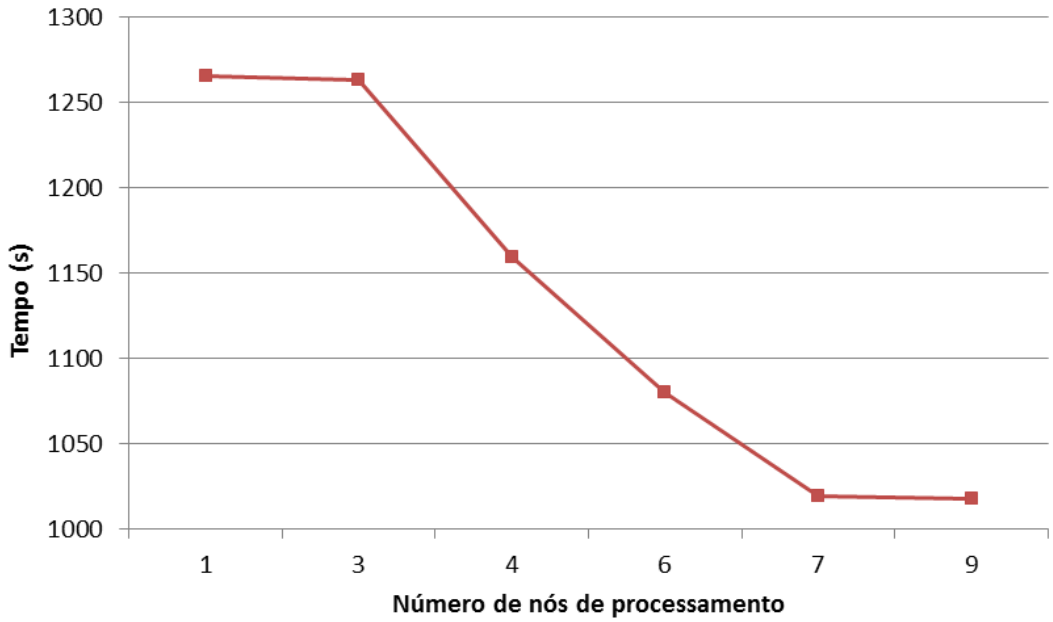
Tabela 7 – Variação de nós de processamento com imagem maior que o bloco HDFS

| Descrição | Base de dados | *cluster* | Tarefa | | |
|---|---|---|---|---|---|
| | | Numero de nós | Número de tarefas | Memória maxima | *cores* |
| Experimento com uma imagem maior do que o bloco do HDFS | Corte_Mosaico | 1, 3, 4, 6, 7 e 9 | 1 | 3 GB | 1 |

Fonte: O autor, 2017.

Neste experimento, foi utilizada a base de dados Corte_Mosaico. A imagem da referida base de dados tem de aproximadamente 72 MB, o que ocuparia três blocos do HDFS, dado que foi configurado com 32 MB. Em decorrência disso, foram feitos alguns experimentos preliminares para definir a menor memória máxima capaz de processar uma imagem com esse tamanho. Neste caso, a tarefa foi configurada com 3GB de memória máxima. Os tempos de execução obtidos foram apresentados na Figura 25.

Figura 25 – Tempo de execução obtido variando-se o número de nós de processamento para uma imagem maior que um bloco do HDFS

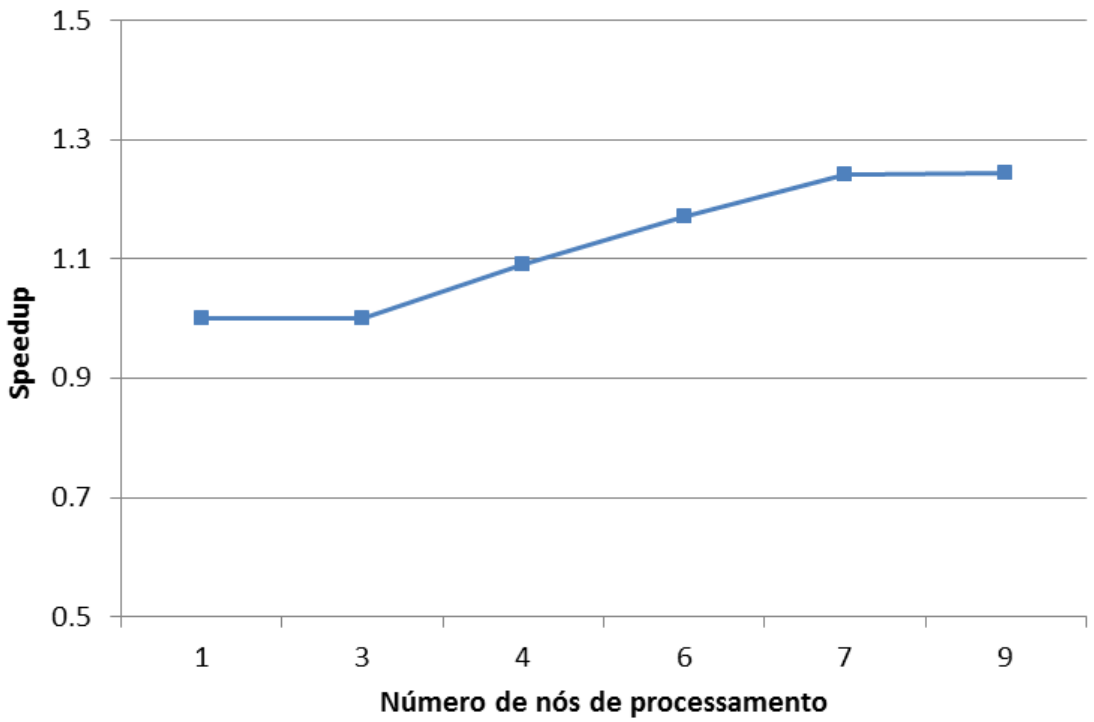



Fonte: O autor, 2017.

Analisando os resultados apresentados na Figura 25 nota-se o aumento do número de nós de processamento até o limiar de sete nós gerou uma pequena diminuição do tempo de execução.

Os *speedups*, usando como referência o resultado para um *cluster* com um nó somente, obtidos para diferentes números de nós são apresentados na Figura 26.

Figura 26 – *Speedup* variando o número de nós de processamento com uma imagem maior que um bloco do HDFS




Fonte: O autor, 2017.

Avaliando os resultados apresentados na Figura 26 observou-se o fato dos resultados apresentarem um *speedup* muito abaixo do valor linear teórico, atingindo um valor de 1.2, fazendo uma comparação com a execução utilizando um e nove nós de processamento. Neste caso, os valores dos *speedups* parecem não recompensar a inclusão de mais nós de processamento.

## 5.11 Execução sequencial do corte do mosaico

Este experimento visa avaliar o tempo gasto na execução do corte do mosaico no procedimento de segmentação de fissuras com uso do *region growing* sequencial diretamente pelo sistema operacional sem o uso do Hadoop. Para tanto, poderá ser utilizado apenas um nó de processamento do *cluster*. Além disso, é preciso mencionar que o algoritmo sequencial não fornece meios de configurar o número de *cores* e memória. Este experimento

foi realizado conforme a Tabela 8.

Tabela 8 – Execução do algoritmo sequencial sem Hadoop

| Descrição | Base de dados | *cluster* | Tarefa | | |
|---|---|---|---|---|---|
| | | Numero de nós | Número de tarefas | Memória maxima | *cores* |
| Execução do algoritmo sequencial sem o Hadoop | Corte_Mosaico | 1 | Não se aplica | - | - |

Fonte: O autor, 2017.

Neste experimento, foi utilizada a base de dados Corte_Mosaico. A imagem da referida base de dados tem de aproximadamente 72 MB. Utilizando o algoritmo sequencial, o tempo de execução foi de 976 segundos, o que equivale a 16 minutos e 16 segundos.

Ao fazer uma comparação do tempo de execução do algoritmo sequencial com o algoritmo distribuído, usando a configuração da Tabela 7, foi identificado que o algoritmo sequencial processa a imagem em 42 segundos a mais que o algoritmo distribuído, com nove nós de processamento, e 289 segundos a mais, com somente um nó de processamento. Nesta comparação de resultados, foi possível constatar que o Hadoop, teve um *overhead* de 4% e 23% respectivamente.

O *overhead* já era esperado, devido ao fato do Hadoop ser desenvolvido para processar processar grande volume de dados e para isso necessita de uma gama de serviços de gerenciamento de tarefas e armazenamento de dados que, por sua vez, podem ser a justificativa do *overhead* no processamento de uma única imagem.

## 5.12 **Influência do número de imagens em grande escala**

Este experimento visa avaliar a influência do número de imagens em grande escala no desempenho do procedimento de segmentação de fissuras com uso do *region growing*. Para tanto, será utilizada apenas uma configuração com nove nós de processamento no *cluster*, número de tarefas igual ao número de imagens, sendo que a memória máxima será sempre 3 GB e o número de cores permanecerá igual a um. O número de imagens será variado

de acordo com os conteúdos das bases de dados Selfrag_180_0001, Selfrag_360_0001, Selfrag_720_0001. Esta variação de número de imagens em grande escala foi realizada conforme a Tabela 9.

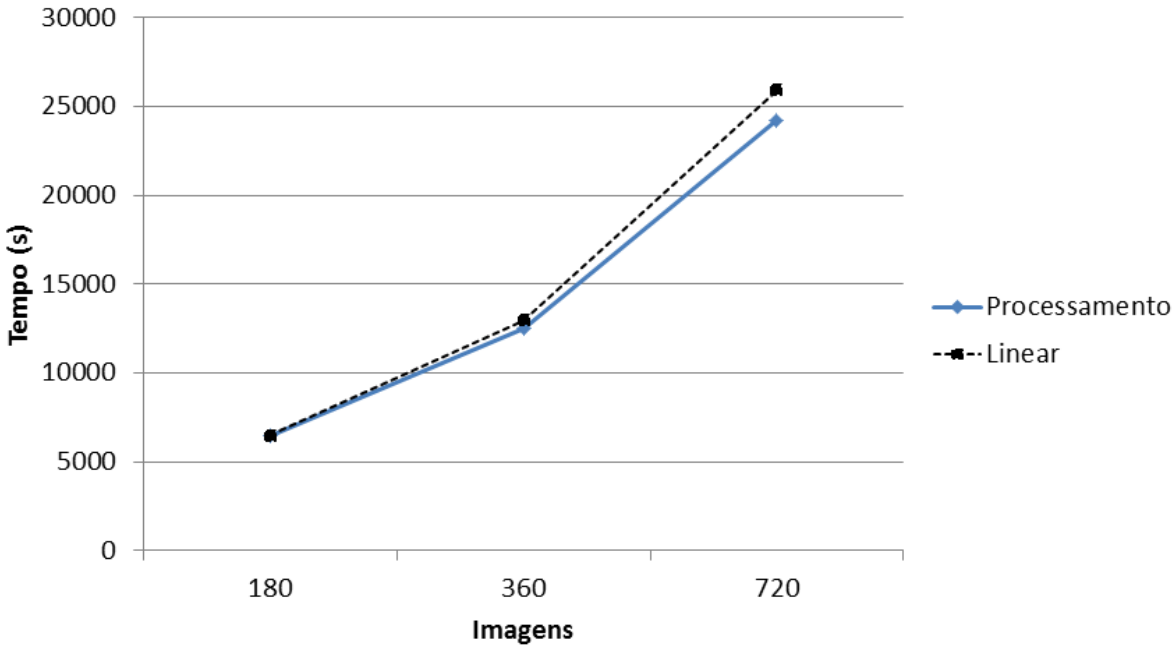
Tabela 9 – Variação do número de imagens em grande escala

| Descrição | Base de dados | *cluster* | Tarefa | | |
|---|---|---|---|---|---|
| | | Numero de nós | Número de tarefas | Memória maxima | *cores* |
| Variando o número de imagens em grande escala | Selfrag_180_0001, Selfrag_360_0001, Selfrag_720_0001 | 9 | Igual ao número de imagens | 3 GB | 1 |

Fonte: O autor, 2017.

Todas as imagens de cada uma das bases processadas são uma cópia de maior imagem Selfrag que possui 30 MB compactada. Em decorrência disso, foram feitos alguns experimentos preliminares para definir valor mínimo memória máxima capaz de viabilizar o processamento de uma imagem com esse tamanho. Assim, as tarefas foram configuradas com 3GB de memória máxima. A Figura 27 apresenta o crescimento do tempo de execução previsto para o caso linear utilizando como referência o resultado obtido no processando de 180 imagens, além dos tempos de execução obtidos para o experimento apresentado na Tabela 9.

Figura 27 – Variação do número de nós de imagens




Fonte: O autor, 2017.

Na Figura 27, o cálculo da projeção do crescimento linear foi feito a partir da média de tempo de execução da base de dados Selfrag_180_0001. Obteve-se trinta e seis segundos de média para execução de uma imagem. O valor foi multiplicado pelo número de imagens das outras bases de dados e foi gerado um fator linear de crescimento baseado do tempo de processamento de Selfrag_180_0001. Comparando-se ambas as curvas, foi possível constatar os tempos obtidos pelo *cluster* crescem numa ordem bem próxima ao linear.

## 5.13 **Execução sequencial da base Selfrag**

Este experimento visa medir os tempos de execução do algoritmo de segmentação para a base Selfrag promovidos pelo cluster Hadoop e de forma direta pelo sistema operacional. Para tanto, será utilizado apenas uma configuração com um nó de processamento no *cluster*, sendo que o algoritmo sequencial não fornece meios de configurar o número de *core* e memória. Esta execução foi realizada conforme a Tabela 10.

Tabela 10 – Execução sequencial da base Selfrag

| Descrição | Base de dados | *cluster* | Tarefa | | |
|---|---|---|---|---|---|
| | | Numero de nós | Número de tarefas | Memória maxima | *cores* |
| Algoritmo sequencial sem o Hadoop | SelFrag_221 | 1 | Não se aplica | - | - |

Fonte: O autor, 2017.

Neste experimento, foi utilizada a base de dados Selfrag_221. As imagens da referida base de dados variam de aproximadamente 1 a 32 MB. No algoritmo sequencial, não é possível fazer a definição dos recursos a serem utilizados. Contudo, o tempo de execução foi de 17259 segundos, o que equivale a 4 horas, 47 minutos e 39 segundos.

Ao fazer uma comparação do tempo de execução do algoritmo sequencial com o algoritmo distribuído usando a configuração da Tabela 4, foi identificado que o Hadoop, com nove nós de processamento, obteve um tempo de execução que equivale a 5.2% do tempo que o algoritmo sequencial, representando um *Speedup* de 19.1. O *Speedup* reflete o fato do Hadoop ser desenvolvido para processar grande volume de dados.

## 5.14 **Análise das regularidades nos resultados obtidos**

Nos experimentos apresentados nesse capítulo, foi possível identificar os comportamentos a partir das diversas variações de configuração do método distribuído de segmentação de fissuras com uso do *region growing*. Ao fazer a variação do número de *cores* por tarefa pôde-se observar que na maioria dos casos os menores tempos de processamento foram obtidos pra um *core* por tarefa.

Assim como a influência do número *cores*, ao fazer a variação da memória máxima definida por tarefa, foi possível constatar que deve-se alocar a memória mínima possível para processar a maior imagem da base de dados, possibilitando assim um melhor uso dos recursos do *cluster*. Portanto, é recomendado utilizar a menor configuração de memória máxima possível que viabilize a execução, sem que ocorra erro por indisponibilidade de memória suficiente.

Em contra partida, com a influência do número de *cores* e memória por tarefa, a variação do número de nós de processamento apresentou um ganho do tempo de execução de forma quase linear. Também, ao fazer a variação do número de tarefas foi possível constatar que o aumento do número de tarefas proporcionou uma diminuição do tempo de execução de forma praticamente linear até atingir o limite de recursos do *cluster*. Observou-se que para esta aplicação o limite do uso dos recursos de processamento era sete *cores* por nó. Contudo, devido à escassez de memória, o aproveitamento destes recursos ficou muito abaixo deste valor.

Ao executar o processamento de uma imagem com tamanho maior que um bloco do HDFS, foi possível obter um resultado mais próximo da execução do algoritmo sequencial diretamente pelo sistema operacional. Assim, o *overhead* estimado ficou entre 4% para 9 nós e 23% para um nó somente no *cluster*. Foi possível também notar que, neste caso, houve uma diminuição pequena do tempo de processamento ao aumentar o número de nós de processamento.

Com o processamento de um volume maior de imagens, além de constatar o potencial em lidar com tal volume, foi possível obter resultados que escalam de forma aproximadamente linear quando comparado com a execução de um volume menor de imagens e tarefas.

Ao utilizar o método proposto com uma base de dados real (Selfrag_221) e nove nós obteve-se *speedup* de 19.1 tomando-se como referência a execução do procedimento sequencial diretamente pelo sistema operacional. Este valor está abaixo do potencial estimado de capacidade de processamento no *custer*, obtido multiplicando-se 7 *cores* por nó (permanecendo um reservado para o sistema operacional) e 9 nós, totalizando 63. A principal razão para este desempenho não ter sido atingido corresponde provavelmente à limitação de memória dos nós de processamento.

# CONCLUSÃO E PRÓXIMOS PASSOS

O presente estudo desenvolveu uma arquitetura, utilizando o Hadoop, capaz de paralelizar e distribuir o processamento de imagens de minérios. Ela foi implementada com o foco em processamento distribuído utilizando o Hadoop juntamente com a linguagem C++ . Para tornar possível, fez-se uso do MR4C e foi necessário o desenvolvimento de funções para possibilitar o uso da biblioteca de processamento de imagens OpenCV com o MR4C.

A análise experimental demonstrou que as configurações do ambiente e os recursos alocados para cada tarefa do MR4C influenciaram diretamente no tempo de execução do método distribuído de segmentação de fissuras. Os resultados demonstraram o potencial da utilização da arquitetura definida e apresenta uma alternativa ao algoritmo proposto em (GOMES et al., 2014).

Identificou-se, nos experimentos executados, um *speedup* de 19.1 comparando a utilização o método sequencial com o método distribuído com nove nós de processamento e duas aplicações MR4C rodando em paralelo.

Assim, foi possível reduzir em até 94% o tempo de processamento obtido pelo método sequencial.

É importante ressaltar que os ganhos de desempenho computacional medidos neste estudo foram obtidos utilizando infraestrutura com baixo recurso de memória em cada nó de processamento. Ficou constatado que, neste *cluster*, a memória é um gargalo e limita a utilização de todos os recursos de *cores*. Apesar dessa limitação, a arquitetura apresentada visou explorar ao máximo os recursos do *cluster* e a capacidade de processamento paralelo do Hadoop.

Dois outros experimentos foram executados com o intuito de avaliar o potencial de processamento paralelo, utilizando como base de dados os mil e quatrocentos frames das imagens parcialmente sobrepostas ($Cu1\_20\_Selfrag0001.tif$ a $Cu1\_20\_Selfrag1400.tif$), cada um com aproximadamente de 1992 x 1726 pixels, e tamanho de 3.28M. Com dessa base de dados, foi possível configurar tarefas com meio GB de memória, desta forma a memória não gerou limitação.

O primeiro experimento utilizou o método sequencial que processou a base de dados

descrita acima em 4 horas, 57 minutos e 30 segundos. Já o segundo experimento utilizou o método distribuído, com nove nós de processamento e 63 tarefas, mantendo-se um *core* dedicado para o sistema operacional. Esse processou a base de dados em 7 minutos e 49 segundos. Um *speedup* de 38.0 comparado com a utilização o método sequencial.

Desta forma, foi possível instanciar todas as tarefas de uma só vez, o mesmo foi comprovado através de análise utilizando o *htop* (UBUNTU, 2010a) em todos os nós de processamento. O resultado obtido foi de até 97% de redução do tempo comparado com o método sequencial. Porém, como nos frames as partículas estão partidas, o resultado não pode ser utilizado para fins de avaliação de exposição do mineral.

A solução proposta têm, portanto, potencial para obter resultados ainda melhores com uma infraestrutura adequada para a arquitetura proposta nesta dissertação.

Como trabalhos futuros, identificamos as seguintes oportunidades:

- adaptar o algoritmo de crescimento de regiões para processar de forma paralela dentro de uma tarefa MR4C;
- desenvolver uma opção para incluir uma lista de configurações de parâmetros de entrada para o algoritmo de segmentação de fissuras;
- adequar a infraestrutura do *cluster* para possibilitar um melhor *speedup* com a arquitetura proposta nesta dissertação;
- incluir os outros algoritmos que fazem parte da avaliação da exposição mineral fazendo também o uso dos recursos de *Reduce* do método *MapReduce*;
- encontrar uma maneira de fazer o *upload* e submeter para processamento no Hadoop as imagens de forma "sob demanda", a medida que elas são geradas no MEV, fazendo o uso das 24 horas da fase de aquisição para o processamento das imagens;
- desenvolver um método automático para o cálculo de dano a partícula e o cálculo de exposição do sulfeto a partir das imagens obtidas diretamente do MEV.

# REFERÊNCIAS

ADAMS, R.; BISCHOF, L. Seeded region growing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 16, n. 6, p. 641–647, Jun 1994. ISSN 0162-8828.

APACHE. *TaskTracker*. Apache Inc, 2009. Disponível em: <https://wiki.apache.org/hadoop/TaskTracker>.

Apache. *Apache Hadoop*. 2013. Acessado em: 11 nov. 2016. Disponível em: <http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html>.

APACHE. *HDFS Federation*. 2014. Acessado em: 08 novembro de 2016. Disponível em: <http://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/Federation.html>.

Apache. *Apache Hadoop*. 2015. Acessado em: 21 set. 2016. Disponível em: <https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-hdfs/LibHdfs.html>.

Apache. *Apache Hadoop*. 2016. Acessado em: 11 nov. 2016. Disponível em: <https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html>.

APACHE. *Welcome to Apache Hadoop*. 2016. Acessado em: 11 outubro de 2016. Disponível em: <http://hadoop.apache.org/>.

ARSH, S.; BHATT, A.; KUMAR, P. Distributed image processing using hadoop and hipi. In: *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. [S.l.: s.n.], 2016. p. 2673–2676.

BENGFORT, B.; KIM, J. *Analítica de Dados com Hadoop: uma introdução para cientistas de dados*. 1. ed. São Paulo, SP: Novatec Editora Ltda., 2016. 346 p. ISBN 978-85-7522-521-9.

BLASCHKE, T.; LANG, S.; HAY, G. *Object-Based Image Analysis: Spatial Concepts for Knowledge-Driven Remote Sensing Applications*. Berlin Heidelberg: Springer, 2008. 77 p. (Lecture Notes in Geoinformation and Cartography).

DAMASCENO, E. C. *Disponibilidade, suprimento e demanda de minérios para metalurgia*. [S.l.]: CETEM, 2006. (Série estudos e documentos).

DEDAVID, B. A.; GOMES, C. I.; MACHADO, G. *Microscopia eletrônica de varredura : aplicações e preparação de amostras*. 1ª. ed. Av. Ipiranga, 6681 - Prédio 33 Caixa Postal 1429 90619-900 Porto Alegre, RS - BRASIL: EDIPUCRS, 2007. Volume 1. 60 p. Disponível em: <http://www.pucrs.br/edipucrs/online/microscopia.pdf>.

EPANCHINTSEV, T.; SOZYKIN, A. Processing large amounts of images on hadoop with opencv. In: _____. *CEUR Workshop Proceedings*. [S.l.]: CEUR-WS, 2015. v. 1513, p. 137–143.

FERREIRA, R. da S. *InterIMAGE Cloud Platform: The Architecture of a Distributed Platform for Automatic, Object-Based Image Interpretation*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, RJ, April 2015.

FERREIRA, R. S. et al. A set of methods to support object–based distributed analysis of large volumes of earth observation data. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, IEEE, v. 10, p. 681–690, 2017.

GNU. *Options for Code Generation Conventions*. GNU, 2008. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Code-Gen-Options.html>.

GOFMAN, E. Developing an efficient region growing engine for image segmentation. In: *Proceedings of the International Conference on Image Processing, ICIP 2006, October 8-11, Atlanta, Georgia, USA*. [s.n.], 2006. p. 2413–2416. Disponível em: <http://dx.doi.org/10.1109/ICIP.2006.312949>.

GOMES, O. d. F. M.; PACIORNIK, S. Multimodal microscopy for ore characterization. *Scanning Electron Microscopy.*, v. 1, p. 313–334., 2012. Disponível em: <http://www.intechopen.com/books/scanning-electron-microscopy/multimodal-microscopy-for-ore-characterization>.

GOMES, O. da F. M. et al. Characterization of particle damage and surface exposure of a copper ore processed by jaw crusher, hpgr and electro-dynamic fragmentation. 2014. Disponível em: <http://hdl.handle.net/2268/168686>.

GONZALEZ, R.; WOODS, R. *Processamento de imagens digitais*. [S.l.]: Edgard Blucher, 2000. ISBN 9788521202646.

HAPP, P. N. et al. A cloud computing strategy for region–growing segmentation. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, IEEE, v. 9, p. 5294–5303, 2016.

KENNEDY-BOWDOIN, T. *MapReduce for C: Run Native Code in Hadoop*. 2015. Acessado em: 11 outubro de 2016. Disponível em: <https://opensource.googleblog.com/2015/02/mapreduce-for-c-run-native-code-in.html>.

KENT, A.; WILLIAMS, J. *Encyclopedia of Microcomputers: Volume 10 - Knowledge Representation and Reasoning to The Management of Replicated Data*. [S.l.]: Taylor & Francis, 1992. (Microcomputers Encyclopedia). ISBN 9780824722791.

LAGANIÈRE, R. *OpenCV 2 Computer Vision Application Programming Cookbook*. [S.l.]: Packt Pub Limited, 2011. Recommended: for OpenCV support. ISBN 1849513244.

LIANG, S. *The Java Native Interface: Programmer's Guide and Specification*. [S.l.]: Addison-Wesley, 1999. (Addison-Wesley Java series). ISBN 9780201325775.

LIMA URGEL DE ALMEIDA, e. a. *Biotecnologia Industrial*. [S.l.]: Editora Edgard Blücher Ltda, 2001. Volume III. 485 p.

MINER, D.; SHOOK, A. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2012. ISBN 1449327176, 9781449327170.

MURTHY, A. et al. *Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2*. [S.l.]: Pearson Education, 2014. (Addison-Wesley Data & Analytics Series).

NELSON, M. *C++ Program Guide to Standard Template Library*. 1st. ed. Foster City, CA, USA: IDG Books Worldwide, Inc., 1995. ISBN 1568843143.

OLSTON, C. et al. Pig latin: A not-so-foreign language for data processing. ACM, New York, NY, USA, 2008. Disponível em: <http://infolab.stanford.edu/~olston/publications/sigmod08.pdf>.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. ISBN 978-01-2370490-0.

PERERA, S.; GUNARATHNE, T. *Hadoop MapReduce Cookbook*. Livery Place 35 Livery Street Birmingham B3 2PB, UK.: Packt Publishing Ltd., 2013. 300 p.

Selfrag. *Selfrag*. 2017. Acessado em: 03 mar. 2017. Disponível em: <http://www.selfrag.com/>.

SHVACHKO, K. et al. The hadoop distributed file system. In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. Washington, DC, USA: IEEE Computer Society, 2010. (MSST '10). Disponível em: <http://pages.cs.wisc.edu/~akella/CS838/F15/838-CloudPapers/hdfs.pdf>.

STEVENS, W. *UNIX Network Programming*. [S.l.]: Prentice Hall PTR, 1998. (UNIX Network Programming, v. 1). ISBN 9780134900124.

SWEENEY C., L. L. A. S. L. J. Hipi: A hadoop image processing interface for image-based mapreduce tasks. *University of Virginia*, 2011. Disponível em: <http://homes.cs.washington.edu/~csweeney/papers/undergrad_thesis.pdf>.

UBUNTU. *Htop*. Ubuntu, 2010. Disponível em: <http://manpages.ubuntu.com/manpages/precise/man1/htop.1.html>.

UBUNTU. *Time*. Ubuntu, 2010. Disponível em: <http://manpages.ubuntu.com/manpages/precise/man1/time.1.html>.

VAVILAPALLI, V. K. et al. *Apache Hadoop YARN: Yet Another Resource Negotiator*. New York, NY, USA: ACM, 2013. (SOCC '13). Disponível em: <http://www.socc2013.org/home/program/a5-vavilapalli.pdf>.

WHITE, T. *Hadoop: The Definitive Guide*. 3ª. ed. [S.l.]: O'Reilly Media / Yahoo Press, 2012. Volume 3. 623 p.

WILLS, B. *Wills' Mineral Processing Technology: An Introduction to the Practical Aspects of Ore Treatment and Mineral Recovery*. Elsevier Science, 2011. Disponível em: <https://books.google.com/books?id=tQj4zuW2VL0C>.

**APÊNDICE A:** Resultados dos experimentos

Tabela 11 – Média em segundos do resultado da variação do número de *cores* por tarefa

| Base de dados | Nós de Processamento | Tarefas | Memória | Média para cada configuração | | | |
|---|---|---|---|---|---|---|---|
| | | | | 1 *core* | 2 *cores* | 3 *cores* | 4 *cores* |
| Selfrag_20 | 1 | 1 | 1.5 | 496 | 456.75 | 437 | 416.25 |
| Selfrag_20 | 1 | 2 | 1.5 | 262.5 | 286.75 | 278.75 | 424.25 |
| Selfrag_20 | 1 | 3 | 1.5 | 191.5 | 264.5 | 318 | 431.5 |
| Selfrag_20 | 1 | 4 | 1.5 | 148.5 | 303.75 | 305.25 | 444.75 |
| Selfrag_20 | 1 | 5 | 1.5 | 147.5 | 274 | 290.75 | 511.75 |
| Selfrag_20 | 1 | 6 | 1.5 | 157.25 | 265.75 | 299.5 | 576.5 |
| Selfrag_20 | 1 | 7 | 1.5 | 165 | 282.25 | 278 | 636.5 |
| Selfrag_20 | 1 | 8 | 1.5 | 149 | 331 | 250.25 | 664.25 |
| Selfrag_20 | 1 | 9 | 1.5 | 154.25 | 405 | 254.75 | 692 |
| Selfrag_20 | 1 | 10 | 1.5 | 150 | 416 | 254.75 | 720 |
| Selfrag_20 | 1 | 11 | 1.5 | 147.25 | 248.5 | 261.25 | 742.5 |
| Selfrag_20 | 1 | 12 | 1.5 | 150.5 | 211.25 | 262 | 623.5 |
| Selfrag_20 | 1 | 13 | 1.5 | 154 | 190.5 | 264 | 503.25 |
| Selfrag_20 | 1 | 14 | 1.5 | 154.75 | 191.5 | 271 | 510.75 |
| Selfrag_20 | 1 | 15 | 1.5 | 155.5 | 192.25 | 277.25 | 517.75 |
| Selfrag_20 | 1 | 16 | 1.5 | 159.5 | 197.25 | 280 | 526 |
| Selfrag_20 | 1 | 17 | 1.5 | 161 | 203.25 | 313.75 | 570.75 |
| Selfrag_20 | 1 | 18 | 1.5 | 167.75 | 207 | 392.5 | 705.5 |
| Selfrag_20 | 1 | 19 | 1.5 | 231.75 | 207.75 | 403.5 | 548 |
| Selfrag_20 | 1 | 20 | 1.5 | 233.5 | 209.5 | 445.5 | 555 |

Fonte: O autor, 2017.

Tabela 12 – Média em segundos do resultado da variação da memória por tarefa

| Base de dados | Nós de Proc. | Tarefas | *core* | Média para cada configuração | | | |
|---|---|---|---|---|---|---|---|
| | | | | Mem 1.5 | Mem 2.0 | Mem 2.5 | Mem 3.0 |
| Selfrag_20 | 1 | 1 | 1 | 496 | 456 | 416.25 | 416 |
| Selfrag_20 | 1 | 2 | 1 | 262.5 | 287 | 247 | 246.5 |
| Selfrag_20 | 1 | 3 | 1 | 191.5 | 231.5 | 268.5 | 285.5 |
| Selfrag_20 | 1 | 4 | 1 | 148.5 | 254.5 | 255.5 | 305.25 |
| Selfrag_20 | 1 | 5 | 1 | 147.5 | 192.25 | 239.75 | 289.25 |
| Selfrag_20 | 1 | 6 | 1 | 157.25 | 170.75 | 256.5 | 298.25 |
| Selfrag_20 | 1 | 7 | 1 | 165 | 181.75 | 246.25 | 306.5 |
| Selfrag_20 | 1 | 8 | 1 | 149 | 181.75 | 363.75 | 285.5 |
| Selfrag_20 | 1 | 9 | 1 | 154.25 | 191.75 | 350 | 323.5 |
| Selfrag_20 | 1 | 10 | 1 | 150 | 218.75 | 314.75 | 315.5 |
| Selfrag_20 | 1 | 11 | 1 | 147.25 | 238.75 | 334 | 331.75 |
| Selfrag_20 | 1 | 12 | 1 | 150.5 | 249.25 | 334 | 334.5 |
| Selfrag_20 | 1 | 13 | 1 | 154 | 249 | 343.75 | 343.5 |
| Selfrag_20 | 1 | 14 | 1 | 154.75 | 252.75 | 358.75 | 360 |
| Selfrag_20 | 1 | 15 | 1 | 155.5 | 258.5 | 290.5 | 358.25 |
| Selfrag_20 | 1 | 16 | 1 | 159.5 | 265.5 | 278.75 | 335.5 |
| Selfrag_20 | 1 | 17 | 1 | 161 | 260.25 | 302.75 | 381.25 |
| Selfrag_20 | 1 | 18 | 1 | 167.75 | 207.5 | 397 | 389 |
| Selfrag_20 | 1 | 19 | 1 | 231.75 | 208.75 | 364.5 | 405.25 |
| Selfrag_20 | 1 | 20 | 1 | 233.5 | 209 | 296.75 | 413.75 |

Fonte: O autor, 2017.

Tabela 13 – Média em segundos do resultado da variação do número de nós de processamento

| Base de dados | *core* | Nós de Proc. | Aplicação 1 | | Aplicação 2 | | Média |
|---|---|---|---|---|---|---|---|
| | | | Tarefas | Memória | Tarefas | Memória | |
| Selfrag_20 | 1 | 1 | 20 | 1536 | 26 | 3048 | 7740 |
| Selfrag_20 | 1 | 2 | 20 | 1536 | 26 | 3048 | 3900 |
| Selfrag_20 | 1 | 3 | 20 | 1536 | 26 | 3048 | 2700 |
| Selfrag_20 | 1 | 4 | 20 | 1536 | 26 | 3048 | 1644 |
| Selfrag_20 | 1 | 5 | 20 | 1536 | 26 | 3048 | 1382 |
| Selfrag_20 | 1 | 6 | 20 | 1536 | 26 | 3048 | 1238 |
| Selfrag_20 | 1 | 7 | 20 | 1536 | 26 | 3048 | 1108 |
| Selfrag_20 | 1 | 8 | 20 | 1536 | 26 | 3048 | 1040 |
| Selfrag_20 | 1 | 9 | 20 | 1536 | 26 | 3048 | 901 |

Fonte: O autor, 2017.

Tabela 14 – Média em segundos do resultado da variação do número de imagens

| Base de dados | Nós de processamento | *cores* | Memória | Tarefas | Média |
|---|---|---|---|---|---|
| Selfrag_40_0001 | 1 | 1 | 3 | 40 | 12000 |
| Selfrag_80_0001 | 1 | 1 | 3 | 80 | 20280 |
| Selfrag_120_0001 | 1 | 1 | 3 | 120 | 35220 |
| Selfrag_40_0001 | 4 | 1 | 3 | 40 | 3000 |
| Selfrag_80_0001 | 4 | 1 | 3 | 80 | 6000 |
| Selfrag_120_0001 | 4 | 1 | 3 | 120 | 8940 |
| Selfrag_40_0001 | 8 | 1 | 3 | 40 | 1740 |
| Selfrag_80_0001 | 8 | 1 | 3 | 80 | 3005 |
| Selfrag_120_0001 | 8 | 1 | 3 | 120 | 4740 |

Fonte: O autor, 2017.

Tabela 15 – Média em segundos do resultado da variação do número de tarefas

| Base de dados | Nós de processamento | *core* | Memória | Tarefas | Média |
|---|---|---|---|---|---|
| Selfrag_180_0031 | 9 | 1 | 2.5 | 9 | 3659.5 |
| Selfrag_180_0031 | 9 | 1 | 2.5 | 18 | 1828 |
| Selfrag_180_0031 | 9 | 1 | 2.5 | 36 | 1552 |
| Selfrag_180_0031 | 9 | 1 | 2.5 | 45 | 1300 |
| Selfrag_180_0031 | 9 | 1 | 2.5 | 90 | 1307.5 |
| Selfrag_180_0031 | 9 | 1 | 2.5 | 180 | 1339 |

Fonte: O autor, 2017.

Tabela 16 – Média em segundos do resultado da variação do número de nós processando uma imagem maior que o bloco HDFS

| Base de dados | Nós de processamento | *core* | Memória | Tarefas | Média |
|---|---|---|---|---|---|
| Corte_Mosaico | 1 | 1 | 3 | 1 | 1265.5 |
| Corte_Mosaico | 3 | 1 | 3 | 1 | 1263.5 |
| Corte_Mosaico | 4 | 1 | 3 | 1 | 1159.5 |
| Corte_Mosaico | 6 | 1 | 3 | 1 | 1080 |
| Corte_Mosaico | 7 | 1 | 3 | 1 | 1019.5 |
| Corte_Mosaico | 9 | 1 | 3 | 1 | 1017.5 |

Fonte: O autor, 2017.

Tabela 17 – Média em segundos do resultado da variação do número de imagens em grande escala

| Imagens | Nós de processamento | *core* | Memória | Tarefas | Média |
|---|---|---|---|---|---|
| Selfrag_180_0001 | 9 | 1 | 2816 | 180 | 6480 |
| Selfrag_360_0001 | 9 | 1 | 2816 | 360 | 12464 |
| Selfrag_720_0001 | 9 | 1 | 2816 | 720 | 24169 |

Fonte: O autor, 2017.

**APÊNDICE B:** Instalação e configuração do *cluster*

**Definicao dos servicos do Hadoop**

No principal

HDFS:

hadoop-hdfs-namenode

hadoop-hdfs-secondarynamenode

hadoop-hdfs-datanode

MapReduce:

hadoop-mapreduce-historyserver

Yarn:

hadoop-yarn-nodemanager

hadoop-yarn-proxyserver

hadoop-yarn-resourcemanager

Nos de Processamento:

HDFS:

hadoop-hdfs-datanode

Yarn:

hadoop-yarn-nodemanager

**Configuracao do bashrc**

```
\#-------------------------ANT----------------------
export ANT_HOME=/usr/local/ant
export PATH=$PATH:${ANT_HOME}/bin
\#-------------------------ANT----------------------
\#-------------------------JAVA---------------------
export JAVA_HOME=/usr/lib/jvm/jdk1.7.0_55
\#-------------------------JAVA---------
```

```
\#----------------------MR4C-----------------
export CPLUS_INCLUDE_PATH=/home/node1/MR4C/gdal-1.10.1
export C_INCLUDE_PATH=/home/node1/MR4C/gdal-1.10.1
export MR4C_HOME=/usr/local/mr4c
\#----------------------MR4C-----------------
\#----------------------HADOOP---------------
export HADOOP_MAPRED_HOME=/usr/lib/hadoop-mapreduce
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
export HADOOP_CLASSPATH=
$HADOOP_CLASSPATH:/usr/share/java/slf4j-simple.jar
\#----------------------HADOOP---------------
```

**Instalação do Java**

Faca o download da versao

```
$1.7.0_55$ do java
```

Descompacte em uma pasta temporaria

```
scp /home/frontend/Projeto_Felipe/Hadoop/jdk-7u55-linux-x64.tar.gz
node1@node3:/home/node1/Hadoop
tar -xvf jdk-7u55-linux-x64.tar.gz
```

Mova a pasta descompactada para /usr/lib/jvm

```
mv ~/Hadoop/jdk1.7.0_55 /usr/lib/jvm/jdk1.7.0_55
```

Execute os seguintes comandos para terminar a instalacao

```
sudo update-alternatives --install "/usr/bin/java"
"java" "/usr/lib/jvm/jdk1.7.0_55/bin/java" 10000
sudo update-alternatives --install "/usr/bin/javac"
"javac" "/usr/lib/jvm/jdk1.7.0_55/bin/javac" 10000
sudo update-alternatives --install "/usr/bin/javaws"
"javaws" "/usr/lib/jvm/jdk1.7.0_55/bin/javaws" 10000
```

```
update-alternatives --display java
update-alternatives --display javac
update-alternatives --display javaws
```

---

**Instalacao do apache ant**

Faca o download da versao

Passo 2: Instalacao do apache ant

```
\#Faca o download da versao 1.8.2 do ant

\#Descompacte em uma pasta temporaria
cd /home/node1/MR4C
tar -zxvf apache-ant-1.8.2-bin.tar.gz

\#Mova para /usr/local/ant
mv apache-ant-1.8.2 /usr/local/ant
cd /usr/local/ant

\#Atribua as variaveis de ambiente
export ANT_HOME=/usr/local/ant
export PATH=$PATH:${ANT_HOME}/bin


\#Faca a seguinte alteracao no arquivo: lib\libraries.properties
m2.url=http://repo1.maven.org/maven2/
\#m2.url=http://ibiblio.org/maven2/
ant -f fetch.xml -Ddest=system
```

---

**Instalacao do apache ivy**

```
\#Faca o download da versao 2.4.0 do ivy

\#Descompacte em uma pasta temporaria
cd /home/node1/MR4C
tar -zxvf apache-ivy-2.4.0-bin-with-deps.tar.gz

\#Mova para /usr/local/ivy
mv /home/node1/MR4C/apache-ivy-2.4.0 /usr/local/ivy

\#copie o arquivo ivy-2.4.0.jar para as seguintes pastas
cp /usr/local/ivy/ivy-2.4.0.jar /usr/local/ant/lib/
cd /usr/local/ivy/src/example/hello-ivy
ant -lib /usr/local/ant/lib

mkdir -p ~/.ant/lib
cp /usr/local/ivy/ivy-2.4.0.jar ~/.ant/lib/ivy.jar
cp /usr/local/ivy/ivy-2.4.0.jar /usr/local/ivy/ivy.jar
cp /usr/local/ivy/ivy-2.4.0.jar /usr/local/ant/lib/ivy.jar
cp /usr/local/ivy/ivy-2.4.0.jar /home/node1/.ant/lib/ivy.jar
cp /usr/local/ivy/ivy-2.4.0.jar /usr/share/ant/lib/ivy.jar
cd /usr/local/ivy/src/example/hello-ivy
ant
```

**Instalacao do make**

```
\#Execute o seguinte comando para instalar

sudo apt-get install make
```

---

**Instalacao do apr**

```
\#Faca o download da versao 1.5.2 do apr

\#Descompacte em uma pasta temporaria
cd /home/node1/MR4C
tar -zxvf apr-1.5.2.tar.gz

\#Execute a instalacao e configuracao
cd apr-1.5.2
./configure
make
make install
```

---

**Instalacao do apr-util**

```
\#Faca o download da versao 1.5.4 do apr-util

\#Descompacte em uma pasta temporaria
```

```
cd /home/node1/MR4C
tar -zxvf apr-util-1.5.4.tar.gz

\#Execute a instalacao e configuracao
cd apr-util-1.5.4
./configure --with-apr=/usr/local/apr/
make
make install
```

---

**Instalacao do apache log4cxx**

```
\#Faca o download da versao 0.10.0 do apache log4cxx

\#Descompacte em ma pasta temporaria
tar -zxvf apache-log4cxx-0.10.0.tar.gz
cd /home/node1/MR4C/apache-log4cxx-0.10.0

scp /home/frontend/Projeto_Felipe/MR4C/log4c/*
node1@node3:/home/node1/MR4C/apache-log4cxx-0.10.0

\#Execute os seguintes comandos:
1. patch -p1 -i cppFolder_stringInclude.patch
2. patch -p1 -i exampleFolder_stringInclude.patch
3. patch -p1 -i log4cxx-build.patch

\#Adicione a linha \#include \<stdio.h\> no arquivo
```

```
./src/examples/cpp/console.cpp

\#execute a instalacao e configuracao
./configure
make
make install
```

---

**Instalacao do Jasson versão 2.2.1**

```
\#Faca o download do jansson-2.2.1.tar.gz

\#Descompacte em uma pasta temporaria:
tar -zxvf jansson-2.2.1.tar.gz
cd jansson-2.2.1

\#Execute a instalacao e configuracao:
./configure
make
make install
```

---

**Instalacao do proj4**

```
\#Faca o download da versao 4.8.0 do proj

\#Descompacte para uma pasta temporaria
```

```
tar -zxvf proj-4.8.0.tar.gz
cd proj-4.8.0

\#Execute a instalacao e configuracao:
./configure
make
make install
```

---

**Instalacao do gdal**

```
\#Faca o download ad versao 1.10.1 do gdal

\#Descompacte em uma pasta temporaria
tar -zxvf gdal-1.10.1.tar.gz
cd gdal-1.10.1

\#Execute a instalacao e configuracao:
./configure
make
make install
```

---

**Instalacao do cppunit**

```
\#Faca o download da versao 1.13.2 do cppunit

\#Descompacte em uma pasta temporaria
```

```
tar -zxvf cppunit-1.13.2.tar.gz
cd cppunit-1.13.2

\#Execute a instalacao e configuracao
./configure
make
make install
```

---

**Instalacao do OpenCV**

```
\#Faca o download da versao 3.1.0 do OpenCV
\#Instale as dependencias necessarias


sudo apt-get install build-essential checkinstall
cmake pkg-config yasm
sudo apt-get install libtiff4-dev libjpeg-dev libjasper-dev
sudo apt-get install libavcodec-dev libavformat-dev
libswscale-dev libdc1394-22-dev
sudo apt-get install libxine-dev libgstreamer0.10-dev
libgstreamer-plugins-base0.10-dev libv4l-dev
sudo apt-get install
python-dev python-numpy
sudo apt-get install libtbb-dev

\#Descompacte em uma pasta temporaria
scp node1@nodeX:/home/node1/MR4C/opencv-3.1.0.tar.gz
node1@nodeX:/home/node1/MR4C
```

```
cd /home/node1/MR4C
tar -xvf opencv-3.1.0.tar.gz
cd opencv-3.1.0/
mkdir release
cd release
mkdir build

\#Setar as variaveis de ambiente e
executar a instalacao e configuracao
cmake -D CMAKE_BUILD_TYPE=RELEASE -D WITH_FFMPEG=OFF -D
CMAKE_INSTALL_PREFIX=/home/node1/MR4C/opencv-3.1.0/release/build
-D WITH_TBB=ON -D BUILD_EXAMPLES=ON ..
make
sudo make install
sudo sh -c 'echo "/usr/local/lib" > /etc/ld.so.conf.d/opencv.conf'
sudo ldconfig
cd bin
./opencv_test_core

sudo apt-get install libopencv-dev python-opencv
```

---

**Instalacao do MR4C**

```
cd /home/node1/MR4C/mr4c

vi /etc/environment
PATH="/usr/local/sbin:/usr/local/bin:
/usr/sbin:/usr/bin:/sbin:/bin:
```

```
/usr/games:/usr/local/games:
/usr/local/ant:/opt/jdk/jdk1.7.0_21/bin"
JAVA_HOME=/opt/jdk/jdk1.7.0_21

Alterado para :
PATH="/usr/local/sbin:/usr/local/bin:
/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:
/usr/local/games:/usr/local/ant:
/usr/lib/jvm/jdk1.7.0_55/bin"
JAVA_HOME=/usr/lib/jvm/jdk1.7.0_55



export ANT_HOME=/usr/local/ant
export PATH=$PATH:${ANT_HOME}/bin

export CPLUS_INCLUDE_PATH=/home/node1/MR4C/gdal-1.10.1
export C_INCLUDE_PATH=/home/node1/MR4C/gdal-1.10.1
export JAVA_HOME=/opt/jdk/jdk1.7.0_21

cd /home/node1/MR4C
 7z x mr4c-master.7z

cp /home/node1/MR4C/mr4c-master /usr/local/mr4c
chmod -R 777 /tmp/jna
cd /usr/local/mr4c
./build_all
./deploy_all
```

---

**Instalacao do cdh_ig_cdh5**

```
\#Fazer o download e adicionar o repositorio do cloudera
cd /etc/apt/sources.list.d/
sudo wget 'https://archive.cloudera.com
/cdh5/ubuntu/trusty/amd64/cdh/cloudera.list' \ -O cloudera.list

cd /home/node1/Hadoop
sudo wget https://archive.cloudera.com/cdh5
/ubuntu/trusty/amd64/cdh/archive.key -O archive.key
sudo apt-key add archive.key

\#Instalar o zookeeper-server e inicializar o servico
sudo apt-get install zookeeper-server

$Fist time$ sudo service zookeeper-server init
sudo service zookeeper-server start

cd /usr/lib/zookeeper/lib
sudo mv slf4j-log4j12-1.7.5.jar slf4j-log4j12-1.7.5_jar
sudo mv slf4j-log4j12.jar slf4j-log4j12_jar


\#Instalar os servicos do Hadoop
sudo apt-get update
sudo apt-get install hadoop-yarn-resourcemanager
sudo apt-get install hadoop-hdfs-namenode
```

```
sudo apt-get install hadoop-hdfs-secondarynamenode

\#Sair do safemode
hadoop dfsadmin -safemode leave

sudo apt-get install hadoop-yarn-nodemanager hadoop-hdfs-datanode

\#Fazer as alteracoes nos arquivos de configuracao do Hadoop
sudo vi /etc/hadoop/conf.cloudera.yarn/yarn-site.xml
<!-- Site specific YARN configuration properties -->
<configuration>
<!-- Site specific YARN configuration properties -->
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services.mapreduce_shuffle.class
    </name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
  <property>
    <name>yarn.app.mapreduce.am.log.level</name>
    <value>INFO</value>
  </property>
</configuration>

 sudo vi /etc/hadoop/conf.cloudera.yarn/core-site.xml
```

```
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://node3:8020</value>
 </property>

 sudo vi /etc/hadoop/conf.cloudera.yarn/hdfs-site.xml
 <configuration>
 <property>
    <name>dfs.namenode.name.dir</name>
    <value>file:///dfs/nn</value>
 </property>
 <property>
    <name>dfs.namenode.servicerpc-address</name>
    <value>node3:8022</value>
 </property>
 <property>
    <name>dfs.https.address</name>
    <value>node3:50470</value>
 </property>
 <property>
    <name>dfs.https.port</name>
    <value>50470</value>
 </property>
 <property>
    <name>dfs.namenode.http-address</name>
    <value>node3:50070</value>
 </property>

sudo vi /etc/hadoop/conf.cloudera.yarn/mapred-site.xml
```

```
  <property>
    <name>mapreduce.jobhistory.address</name>
    <value>node3:10020</value>
  </property>
  <property>
    <name>mapreduce.jobhistory.webapp.address</name>
    <value>node3:19888</value>
  </property>
  <property>
    <name>mapreduce.jobhistory.webapp.https.address</name>
    <value>node3:19890</value>
  </property>
  <property>
    <name>mapreduce.jobhistory.admin.address</name>
    <value>node3:10033</value>

 sudo vi /etc/hadoop/conf.cloudera.yarn/topology.map
 <node name="node3" rack="/default"/>


sudo vi /etc/hadoop/conf.cloudera.hdfs/core-site.xml

 <property>
    <name>fs.defaultFS</name>
    <value>hdfs://node3:8020</value>
  </property>
```

```
sudo vi /etc/hadoop/conf.cloudera.hdfs/hdfs-site.xml

    <property>
    <name>dfs.namenode.servicerpc-address</name>
    <value>node3:8022</value>
  </property>
  <property>
    <name>dfs.https.address</name>
    <value>node3:50470</value>
  </property>
  <property>
    <name>dfs.https.port</name>
    <value>50470</value>
  </property>
  <property>
    <name>dfs.namenode.http-address</name>
    <value>node3:50070</value>
  </property>

  sudo vi /etc/hadoop/conf.cloudera.hdfs/topology.map
<node name="node3" rack="/default"/>


  sudo vi /etc/hadoop/conf.cloudera.mapreduce/core-site.xml
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://node3:8020</value>
  </property>
```

```
sudo vi /etc/hadoop/conf.cloudera.mapreduce/hdfs-site.xml
  <property>
    <name>dfs.namenode.servicerpc-address</name>
    <value>node3:8022</value>
 </property>
 <property>
    <name>dfs.https.address</name>
    <value>node3:50470</value>
 </property>
 <property>
    <name>dfs.https.port</name>
    <value>50470</value>
 </property>
 <property>
    <name>dfs.namenode.http-address</name>
    <value>node3:50070</value>
 </property>

 sudo vi /etc/hadoop/conf.cloudera.mapreduce/mapred-site.xml
    <property>
    <name>mapred.job.tracker</name>
    <value>node3:8021</value>
 </property>
 <property>
    <name>mapred.job.tracker.http.address</name>
    <value>node3:50030</value> "era 0.0.0.0:50030"
 </property>
```

```
node1@node3:~$ sudo update-alternatives --display hadoop-conf
hadoop-conf - modo automatico
  o link actualmente aponta para
  /etc/hadoop/conf.cloudera.yarn
/etc/hadoop/conf.cloudera.hdfs - prioridade 90
/etc/hadoop/conf.cloudera.mapreduce - prioridade 91
/etc/hadoop/conf.cloudera.yarn - prioridade 92
/etc/hadoop/conf.empty - prioridade 10
A 'melhor' versao actual e /etc/hadoop/conf.cloudera.yarn.
```

---

```
\#Diretorios de log
/etc/default/hadoop-hdfs-namenode
/etc/default/hadoop-hdfs-secondarynamenode
/etc/default/hadoop-hdfs-datanode
```

**Instalacao nos slaves**

```
Sudo nano ~/.bashrc
\#---java
export JAVA_HOME=/usr/lib/jvm/jdk1.7.0_55
\#---hadoop
export HADOOP_MAPRED_HOME=/usr/lib/hadoop-mapreduce
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
export HADOOP_CLASSPATH=
```

```
$HADOOP_CLASSPATH:/usr/share/java/slf4j-simple.jar
\#---hadoop

\#Instalar e configurar o ntp para sincronizar os nos
sudo apt-get install ntp

sudo nano /etc/ntp.conf
server 2.br.pool.ntp.org
server 1.south-america.pool.ntp.org
server 0.south-america.pool.ntp.org

sudo ntpdate -u ntp.ubuntu.com

sudo hwclock --systohc


http://www.cloudera.com/documentation
/enterprise/latest/topics/
install_cdh_enable_ntp.html
\#install_cdh_enable_ntp

\#Adicionar o repositorio
cd /etc/apt/sources.list.d/
sudo wget 'https://archive.cloudera.com
/cdh5/ubuntu/trusty/amd64/cdh/
cloudera.list' \ -O cloudera.list

cd /home/node1/Hadoop
wget https://archive.cloudera.com
```

```
/cdh5/ubuntu/trusty/amd64/
cdh/archive.key –O archive.key
sudo apt-key add archive.key

http://www.cloudera.com/documentation
/enterprise/latest/topics
/cdh_ig_cdh5_install.html\#topic_4_4

\#Instalar e configurar o zookeeper-server
sudo apt-get update
sudo apt-get install zookeeper-server
(usar aptitude se der pacote quebrado)

sudo vi /etc/zookeeper/conf.dist/zoo.cfg
server.1=node3:2888:3888
server.2=node4:2888:3888
server.3=node5:2888:3888
server.4=node6:2888:3888
server.5=node7:2888:3888
server.6=node8:2888:3888
server.7=node9:2888:3888
server.8=node10:2888:3888
server.9=node1:2888:3888
server.10=node2:2888:3888


sudo service zookeeper-server
init --myid=3
(um numero que nao esteja sendo
```

```
usado em outros nos )
\#Iniciar o zookeeper-server
sudo service zookeeper-server start
//servico do zookeeper nao ta iniciando no
node10 e no node2 fica caindo

\#Instalar os servicos do hadoop
sudo apt-get install hadoop-yarn-nodemanager
sudo apt-get hadoop-hdfs-datanode
sudo apt-get hadoop-mapreduce
sudo apt-get install hadoop-client

http://www.cloudera.com/documentation/
enterprise/latest/topics/
cdh_ig_cdh5_install.html\#topic_4_4

\#Fazer todas as alteracoes de
configuracao no novo diretorio conf.my_cluster
sudo cp -r /etc/hadoop/conf.empty /etc/hadoop/conf.my_cluster
sudo update-alternatives --install
/etc/hadoop/conf hadoop-conf /etc/hadoop/conf.my_cluster 50
sudo update-alternatives --set hadoop-conf
/etc/hadoop/conf.my_cluster


cd  /etc/hadoop/conf.my_cluster/

\#Atribuir as variaveis de ambiente HADOOP_MAPRED_HOME
e JAVA_HOME nos arquivos hadoop-env.sh e yarn-env.sh
```

```
Copiei o hadoop-env.sh do node3 para o slave e setei
o JAVA_HOME e HADOOP_MAPRED_HOME
tanto no hadoop-env.sh quanto no yarn-env.sh

sudo chown root:hadoop hadoop-env.sh

\#Adicionar o hostname no arquivo slaves
sudo nano slaves
node4
node5
node6
node7
node8
node9
node10
node1
node2


\#Adicionar/alterar as
seguintes configuracoes no arquivo core-site.xml
sudo nano core-site.xml
<property>
 <name>fs.defaultFS</name>
 <value>hdfs://node3:8020</value>
</property>
<property>
 <name>hadoop.proxyuser.mapred.groups</name>
 <value>*</value>
```

```
</property>
<property>
 <name>hadoop.proxyuser.mapred.hosts</name>
 <value>*</value>
</property>

\#Adicionar/alterar as
seguintes configuracoes no arquivo hdfs-site.xml
sudo nano hdfs-site.xml
<!-- <property>
  <name>dfs.namenode.name.dir</name>
  <value>file:///var/lib/hadoop-hdfs/cache/hdfs/dfs/name</value>
</property> -->
<property>
  <name>dfs.permissions.superusergroup</name>
  <value>hadoop</value>
</property>
<property>
  <name>dfs.datanode.data.dir</name>
  <value>file:///data/1/dfs/dn,file:///data/2/dfs/dn,
   file:///data/3/dfs/dn,file:///data/4/dfs/dn</value>
</property>
<property>
  <name>dfs.datanode.failed.volumes.tolerated</name>
  <value>3</value>
</property>
<property>
  <name>dfs.datanode.address</name>
  <value>node4:50010</value>
```

```
</property>
<property>
  <name>dfs.datanode.http-address</name>
  <value>node4:50075</value>
</property>


\#Criar os diretorios
definidos nas configuracoes
sudo mkdir -p /data/1/dfs/dn /data/2/dfs/dn
/data/3/dfs/dn /data/4/dfs/dn
sudo chown -R hdfs:hdfs /data/1/dfs/dn /data/2/dfs/dn
/data/3/dfs/dn /data/4/dfs/dn

\#Iniciar o datanode
Sudo service hadoop-hdfs-datanode start
sudo -u hdfs hadoop fs -mkdir /tmp
sudo -u hdfs hadoop fs -chmod -R 1777 /tmp

http://www.cloudera.com/documentation/enterprise
/latest/topics/cdh_ig_hdfs_cluster_deploy.html\#topic_11_2

\#Adicionar/alterar as seguintes configuracoes
no arquivo mapred-site.xml
Sudo nano mapred-site.xml
<property>
  <name>mapreduce.framework.name</name>
  <value>yarn</value>
</property>
```

```
<property>
  <name>mapreduce.jobhistory.address</name>
  <value>node3:10020</value>
</property>
<property>
  <name>mapreduce.jobhistory.webapp.address</name>
  <value>node3:19888</value>
</property>
<property>
  <name>yarn.app.mapreduce.am.staging-dir</name>
  <value>/user</value>
</property>

\#Adicionar/alterar as seguintes configuracoes
no arquivo yarn-site.xml
Sudo nano yarn-site.xml
<property>
  <name>yarn.resourcemanager.hostname</name>
  <value>node3</value>
</property>
<property>
  <name>yarn.nodemanager.local-dirs</name>
  <value>file:///data/1/yarn/local,file:
  ///data/2/yarn/local,file:///data/3/yarn/local</value>
</property>
<property>
  <name>yarn.nodemanager.log-dirs</name>
  <value>file:///data/1/yarn/logs,file:
  ///data/2/yarn/logs,file:///data/3/yarn/logs</value>
```

```
</property>
<property>
  <name>yarn.nodemanager.remote-app-log-dir</name>
  <value>hdfs://node3:8020/var/log/hadoop-yarn/apps</value>
</property>

\#Criar os diretorios definidos nas configuracoes
sudo mkdir -p /data/1/yarn/local /data/2/yarn/local
/data/3/yarn/local
sudo mkdir -p /data/1/yarn/logs /data/2/yarn/logs
/data/3/yarn/logs
sudo chown -R yarn:yarn /data/1/yarn/local
/data/2/yarn/local /data/3/yarn/local
sudo chown -R yarn:yarn /data/1/yarn/logs
/data/2/yarn/logs /data/3/yarn/logs

\#Iniciar o nodemanager
sudo service hadoop-yarn-nodemanager start
http://www.cloudera.com/documentation
/enterprise/latest/
topics/cdh_ig_yarn_cluster_deploy.html
\#topic_11_4

\#Desativar o safemode
hdfs dfsadmin -safemode leave
```

---

**Diminuir o tamanho do Bloco por default**

```
\#Alterar para o tamanho desejado
/etc/hadoop/conf.cloudera.yarn$ sudo vi hdfs-site.xml
<property>
    <name>dfs.blocksize</name>
    <value>67108864</value>
  </property>
```

**Checar a configuracao padrao**

```
hdfs getconf -confKey dfs.blocksize
```

**Verificar o tamanho padrao do bloco do arquivo**

```
hadoop fs -stat %o /regiongrowing/in/Cu1_20_Selfrag_0005.tif
```

**Verificar o Fator de Replicacao**

```
sudo -u hdfs hadoop fs -ls /regiongrowing/in/
Cu1_20_Selfrag_0005.tif
-rw-r--r--   3 node1 supergroup   18786954 2017-02-20 10:45
/regiongrowing/in/
Cu1_20_Selfrag_0005.tif
```

O numero 3 representa o fator.

**Desativar os nos de processamento**

```
cd /etc/hadoop/conf.cloudera.yarn

\#Definir o arquivo de exclusao no mapred-site.xml
```

```
sudo vi mapred-site.xml
<property>
  <name>mapred.hosts.exclude</name>
  <value>/etc/hadoop/conf.cloudera.yarn/mapred.exclude</value>
</property>

\#Definir o arquivo de exclusao no hdfs-site.xml
sudo vi hdfs-site.xml
<property>
  <name>dfs.hosts.exclude</name>
  <value>/etc/hadoop/conf.cloudera.yarn/hdfs.exclude</value>
</property>

\#Criar o arquivo mapred.exclude com os hostnames
dos nos que se quer desativar
sudo vi mapred.exclude
node10
node9
node8
// colocar os hostnames dos nos que se quer desativar

\#Criar o arquivo hdfs.exclude com os hostnames
dos nos que se quer desativar
sudo vi hdfs.exclude
node10
node9
node8
// colocar os hostnames dos nos que se quer desativar
```

```
\#Remover os nos que se quer desativar do arquivo slaves
sudo vi slaves
node3
node4
node5
node6
node7
node1
node2
//remover os hostnames dos nos que se quer desativar


\#Para aplicar as mudancas
hdfs dfsadmin -refreshNodes

\#para mostrar os nos que estao ativados ou nao
hdfs dfsadmin -report


\#Para reativar o no, basta fazer o processo inverso

https://pravinchavan.wordpress.com/2013/06/03
/removing-node-from-hadoop-cluster/
```

**Configurar o timeout**

```
sudo vi mapred-site.xml
```

```
<property>
  <name>mapreduce.task.timeout</name>
  <value>1800000</value>
</property>
```

**Definir a versao Cloudera**

```
sudo wget 'https://archive.cloudera.com/cdh5
/ubuntu/trusty/amd64/cdh/cloudera.list' \ -O
/etc/apt/sources.list.d/cloudera.list
Alterar o arquivo para:
\# Packages for Cloudera's Distribution for
Hadoop, Version 5, on Ubuntu 14.04 amd64
deb [arch=amd64] http://archive.cloudera.c:
wqom/cdh5/ubuntu/trusty/amd64/cdh trusty-cdh5.9.0 contrib
deb-src http://archive.cloudera.com/cdh5
/ubuntu/trusty/amd64/cdh trusty-cdh5.9.0 contrib
---Salvar e executar o comando abaixo:
sudo apt-get update
```

**Script de sequência para iniciar todos os serviços**

```
\#!/bin/bash

\#Zookeeper
ssh node1 sudo service zookeeper-server start
ssh node2 sudo service zookeeper-server start
sudo service zookeeper-server start
ssh node4 sudo service zookeeper-server start
```

```
ssh node5 sudo service zookeeper-server start
ssh node6 sudo service zookeeper-server start
ssh node7 sudo service zookeeper-server start
ssh node8 sudo service zookeeper-server start
ssh node9 sudo service zookeeper-server start
ssh node10 sudo service zookeeper-server start
\#HDFS
ssh node1 sudo service hadoop-hdfs-datanode start
ssh node2 sudo service hadoop-hdfs-datanode start
sudo service hadoop-hdfs-datanode start
ssh node4 sudo service hadoop-hdfs-datanode start
ssh node5 sudo service hadoop-hdfs-datanode start
ssh node6 sudo service hadoop-hdfs-datanode start
ssh node7 sudo service hadoop-hdfs-datanode start
ssh node8 sudo service hadoop-hdfs-datanode start
ssh node9 sudo service hadoop-hdfs-datanode start
ssh node10 sudo service hadoop-hdfs-datanode start
sudo service hadoop-hdfs-namenode start
sudo service hadoop-hdfs-secondarynamenode start
\#MAPREDUCE
sudo service hadoop-mapreduce-historyserver start
\#YARN
ssh node1 sudo service hadoop-yarn-nodemanager start
ssh node2 sudo service hadoop-yarn-nodemanager start
sudo service hadoop-yarn-nodemanager start
ssh node4 sudo service hadoop-yarn-nodemanager start
ssh node5 sudo service hadoop-yarn-nodemanager start
ssh node6 sudo service hadoop-yarn-nodemanager start
ssh node7 sudo service hadoop-yarn-nodemanager start
```

```
ssh node8 sudo service hadoop-yarn-nodemanager start
ssh node9 sudo service hadoop-yarn-nodemanager start
ssh node10 sudo service hadoop-yarn-nodemanager start
sudo service hadoop-yarn-resourcemanager start
sudo service hadoop-yarn-proxyserver start
```

**Script de sequência para parar todos os serviços**

```
\#YARN
sudo service hadoop-yarn-resourcemanager stop
ssh node1 sudo service hadoop-yarn-nodemanager stop
ssh node2 sudo service hadoop-yarn-nodemanager stop
sudo service hadoop-yarn-nodemanager stop
ssh node4 sudo service hadoop-yarn-nodemanager stop
ssh node5 sudo service hadoop-yarn-nodemanager stop
ssh node6 sudo service hadoop-yarn-nodemanager stop
ssh node7 sudo service hadoop-yarn-nodemanager stop
ssh node8 sudo service hadoop-yarn-nodemanager stop
ssh node9 sudo service hadoop-yarn-nodemanager stop
ssh node10 sudo service hadoop-yarn-nodemanager stop
sudo service hadoop-yarn-proxyserver stop
\#MAPREDUCE
sudo service hadoop-mapreduce-historyserver stop
\#HDFS
sudo service hadoop-hdfs-secondarynamenode stop
sudo service hadoop-hdfs-namenode stop
ssh node1 sudo service hadoop-hdfs-datanode stop
ssh node2 sudo service hadoop-hdfs-datanode stop
sudo service hadoop-hdfs-datanode stop
ssh node4 sudo service hadoop-hdfs-datanode stop
```

```
ssh node5 sudo service hadoop-hdfs-datanode stop
ssh node6 sudo service hadoop-hdfs-datanode stop
ssh node7 sudo service hadoop-hdfs-datanode stop
ssh node8 sudo service hadoop-hdfs-datanode stop
ssh node9 sudo service hadoop-hdfs-datanode stop
ssh node10 sudo service hadoop-hdfs-datanode stop
\#Zookeeper
ssh node1 sudo service zookeeper-server stop
ssh node2 sudo service zookeeper-server stop
sudo service zookeeper-server stop
ssh node4 sudo service zookeeper-server stop
ssh node5 sudo service zookeeper-server stop
ssh node6 sudo service zookeeper-server stop
ssh node7 sudo service zookeeper-server stop
ssh node8 sudo service zookeeper-server stop
ssh node9 sudo service zookeeper-server stop
ssh node10 sudo service zookeeper-server stop
```