UNIVERSIDADE FEDERAL DE MATO GROSSO
INSTITUTO DE FÍSICA
PROGRAMA DE PÓS GRADUAÇÃO EM FÍSICA AMBIENTAL

Processamento distribuído de grande volume de dados ambientais

Ricardo Frederico Figueiredo e Salles

Orientador: **Prof. Dr. Josiel Maimone de Figueiredo**

Cuiabá - MT
Fevereiro/2016

UNIVERSIDADE FEDERAL DE MATO GROSSO
INSTITUTO DE FÍSICA
PROGRAMA DE PÓS GRADUAÇÃO EM FÍSICA AMBIENTAL

Processamento distribuído de grande volume de dados ambientais

Ricardo Frederico Figueiredo e Salles

Dissertação apresentada ao Programa de Pós Graduação em Física Ambiental da Universidade Federal de Mato Grosso, como parte dos requisitos para obtenção do título de Mestre em Física Ambiental.

Prof. Dr. Josiel Maimone de Figueiredo

Cuiabá, MT
Fevereiro/2016

FICHA CATALOGRÁFICA

S168p Salles, Ricardo Frederico Figueiredo e.
Processamento distribuído de grande volume de dados ambientais / Ricardo Frederico Figueiredo e Salles. – 2016.
xvi, 49 f.: il.; color.

Orientador: Prof. Dr. Josiel Maimone de Figueiredo.
Dissertação (mestrado) – Universidade Federal de Mato Grosso, Instituto de Física, Pós-Graduação em Física Ambiental, 2016.

Bibliografia: f. 46-49.

1. Hadoop. 2. Spark. 3. Paralelismo. 4. Big Data. 5. Wavelets. 6. Sistemas de processamento distribuído. 7. Precipitação. 8. Satélite I. Título.

CDU – 004.75:504

UNIVERSIDADE FEDERAL DE MATO GROSSO
INSTITUTO DE FÍSICA
Programa de Pós-Graduação em Física Ambiental

FOLHA DE APROVAÇÃO

TÍTULO: PROCESSAMENTO DISTRIBUÍDO DE GRANDE VOLUME DE DADOS AMBIENTAIS

AUTOR: RICARDO FREDERICO FIGUEIREDO E SALLES

Dissertação de Mestrado defendida e aprovada em 24 de fevereiro de 2016, pela comissão julgadora:

Prof. Dr. Josiel Maimone de Figueiredo
Orientador
Instituto de Computação – UFMT

Prof. Dr. Raphael de Souza Rosa Gomes
Examinador Interno
Instituto de Computação – UFMT

Prof. Dr. João Paulo Delgado Preti
Examinador Externo
Instituto Federal de Mato Grosso - IFMT

DEDICATÓRIA

Á toda minha família, a minha esposa Érica Martini Pessoa Figueiredo e ao meu grande avô Professor Benedito Figueiredo.

Agradecimentos

Primeiramente gostaria de agradecer a pessoa a quem tudo devo em minha vida, a minha mãe Olga Maria Figueiredo que sempre me apoiou nas minhas decisões e puxou a minha orelha nas horas que mais precisava.

À minha amada e querida esposa Érica Martini Pessoa Figueiredo por ser essa grande companheira e amiga.

Ao meu Avô Benedito Figueiredo, que como a minha mãe, teve um papel importantíssimo na história da Universidade Federal de Mato Grosso, fazendo com que eu me inspirasse em sua história.

Ao professor, orientador e principalmente grande amigo Dr. Josiel Maimone de Figueiredo, por ser este grande guia para a conclusão deste projeto, no qual sem sua ajuda não seria possível de ser finalizado, pois sempre esteve disposto a me manter focado no verdadeiro caminho a ser trilhado.

À minha irmã a quem eu amo muito Marina Beatriz Figueiredo Salles de Figueiredo por sempre me ajudar quando preciso, por me fazer sorrir e ser esse exemplo de pessoa.

À minha sobrinha Olívia Salles Maimone de Figueiredo por ser este grande exemplo de superação e que hoje é uma das pessoas que eu mais amo na vida.

Ao professor Dr. José de Souza Nogueira (Paraná) pelas ótimas conversas e por sempre ser um grande conselheiro.

Ao professor Dr. Raphael de Souza Rosa Gomes pela disposição de sempre me ajudar quando possível.

Ao PGFA por ter me aceito como mais um "filho"e ao Instituto de Computação pelo apoio indispensável.

A grandiosa Universidade Federal de Mato Grosso, que sempre será mar-

cada em minha memória por toda esta vida, pela importãncia em minha história desde a minha infância. Foi na Biblioteca central que em 1993 tive o primeiro contato com aquilo que mudaria a minha vida para sempre - O computador pessoal.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – CAPES, pelo apoio financeiro.

A todos os meus amigos, colegas e professores do Programa de Pós-graduação em Física Ambiental pelos os ensinamentos e momentos felizes.

A todos vocês, meu MUITO OBRIGADO!

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

*.CSV	Comma-separated values file
CERN	Organização Europeia para a Pesquisa Nuclear
CORE	Núcleo de processador
CPU	Unidade central de processamento
DAG	Directed acyclic graph
FLOP	Floating-point operations per second
GFS	Google File System
GHZ	Gigahertz
GPU	Unidade de Processamento Gráfico
GV	Ground Validation
HDFS	Hadoop File System
HPC	High-Performance Computing
HPC	High-performance computing
ID	Código de identificação
JAR	Arquivo compactado usado para distribuir um conjunto de classes Java
JAVA	Linguagem de programação interpretada orientada a objetos

KNN	k-Nearest Neighbors
MLlib	Biblioteca para machine learning do Apache Spark
NÓ	Um computador em um cluster
PGFA	Programa de Pós-Graduação em Física Ambiental
POSTGRESQL	Sistema gerenciador de banco de dados objeto relacional de código aberto
R PROJECT	*Software* para cálculos estatísticos de código aberto
RDD	Resilient Distributed Datasets
SEBAL	Surface Energy Balance Algorithm for Land
SQL	Structured Query Language
TM	Thermatic Mapper
TRMM	Tropical Rainfall Measuring Mission
VPM	Modelo de fotossíntese da vegetação

RESUMO

SALLES, R. F. F. Processamento distribuído de grande volume de dados ambientais. Cuiabá, 2016, 49 f. Dissertação (Mestrado em Física Ambiental) - Instituto de Física, Universidade Federal de Mato Grosso.

A pesquisa ambiental envolve uma grande diversidade de dados de diferentes origens que quando manipulados demandam a utilização de funções de alta complexidade computacional. Uma função que se enquadra nessas características é a transformada de wavelets, que é a transformação de uma série de dados temporais em diferentes escalas de frequência e tempo. Esse cálculo gera um alto volume de dados e requer elevado poder de processamento e armazenamento, características estas presentes no contexto de *Big Data*. O objetivo deste trabalho é aplicar técnicas de processamento paralelo e distribuído no contexto do cálculo de dados ambientais utilizando os *frameworks Apache Hadoop* e *Apache Spark*. Como metodologia, foi aplicada a transformada de wavelets em um grande volume de dados de precipitação de aproximadamente 15 anos gerados pela missão TRMM - *Tropical Rainfall Measuring Mission*. Foram feitos testes em diferentes configurações e ambientes, comparando os resultados com o processamento sequencial e outra abordagem distribuída.

Palavras-chave: Hadoop, Spark, Paralelismo, Big data, Satélite, Wavelets, Precipitação

ABSTRACT

SALLES, R. F. F. Distributed processing of unstructured data. Cuiabá, 2016, 49 f. Dissertation (Environmental physics master) - Physical Institute, Federal University of Mato Grosso.

The environmental research involves a wide variety of data from different sources that when manipulated require the use of high computational complexity functions. A function that fits on these characteristics is the wavelet transform, which is the transformation of a series of data in different temporal ranges of frequency and time. This calculation generates a high volume of data and requires high processing performance and storage. These characteristics are present in the context of Big Data. The objective of this work is to apply parallel and distributed processing technologies in the context of calculation of environmental data using the Apache Hadoop and Apache Spark frameworks. As a methodology, it was applied the wavelet transform into a large volume of rainfall data of about 15 years generated by TRMM mission - Tropical Rainfall Measuring Mission. Tests were done at different settings and environments, comparing the results with the sequential processing and other distributed approach.

Keywords: Hadoop, Spark, Parallelism, Big data, Satellite, Wavelets, Precipitation

Capítulo 1

INTRODUÇÃO

1.1 PROBLEMÁTICA

O avanço tecnológico vem permitindo a utilização de equipamentos mais rápidos, precisos e de recursos nunca antes imaginados, como por exemplo imagens de satélites, gerando assim um grande e heterogêneo volume de dados. Para um pesquisador, se faz necessário investigar a relação entre diferentes tipos de dados, coletados de diversos tipos de equipamentos, produzindo assim uma enorme e complexa coleção de dados.

No âmbito da física ambiental, é comum buscar relações entre dados de imagens de satélites com dados oriundos de torres micrometeorológicas, criando assim um cenário onde é preciso uma grande capacidade de armazenamento, processamento e disponibilização dos dados. Esses dados além de estarem em diferentes origens, geralmente possuem um grande volume, tornando a sua análise extremamente trabalhosa.

Um exemplo de processamento dispendioso é a utilização do algoritmo SEBAL para o cálculo de balanço de energia em imagens geradas por satélites, para depois ser validado com dados obtidos pela razão de Bowen. Esse cálculo pode levar várias horas quando utilizado o processamento tradicional de dados.

O tempo excessivo dispendido para o processamento tradicional, resulta da utilização de apenas uma unidade central de processamento (CPU), o que culmina muitas vezes, ao não atendimento de forma satisfatória do cálculo de dados, uma vez que os atrasos a projetos de pesquisa são frequentes. Então, a fim de minimizar o tempo de resposta e imprimir confiabilidade aos resultados, adotam-se outros tipos de processamento, entre eles o paralelo ou distribuído.

No processamento paralelo, utiliza-se dois ou mais processadores para a execução de uma tarefa, o que consequentemente diminui o tempo de resposta. A

indústria já vem adotando a utilização de *cores* para possibilitar processamento paralelo em processadores comuns. Infelizmente quando trabalhamos com um grande volume de dados, este modelo acaba não atendendo de forma adequada, pois faz-se necessário mais processadores para uma execução em tempo adequado, elevando a complexidade do ambiente computacional.

No processamento distribuído, utiliza-se dois ou mais computadores para além de processar, armazenar dados distribuidamente em um ambiente. Geralmente são caracterizados como escaláveis (facilidade no aumento ou redução de novos *computadores*) e de baixo custo, pois é possível a utilização de máquinas ordinárias. Esse ambiente possibilita que os dados sejam armazenados em determinados locais e processados em outros.

Para que um contexto seja caracterizado como *Big Data*, o mesmo deve possuir uma enorme e complexa coleção de diferentes tipos de dados que são processados utilizando várias abordagens distribuídas.

1.2 JUSTIFICATIVA

A tendência é que o volume de dados gerados em programas de pesquisa climáticas seja da ordem de *terabytes*, pois as modelagens e simulações geram dados intermediários dessa ordem, além disso, com uma maior acessibilidade e menores preços, a quantidade de sensores utilizados tende a aumentar, fazendo com que no decorrer dos anos o volume de dados aumente. Esse ambiente futuro é caracterizado como *Big Data* e será onipresente.

É de extrema importância para a física ambiental estar preparada para um futuro próximo em que a necessidade de extrair informação de imensos conjuntos de dados será indispensável para a qualidade das pesquisas. Quanto mais variáveis disponíveis, maior a necessidade de relacionar as mesmas para a predição do comportamento climático de determinada região.

1.2.1 Objetivos Gerais

Aplicar tecnologias de processamento distribuído e implementar um algoritmo de cálculo ambiental voltado para o contexto de *Big Data*, possibilitando a melhoria no armazenamento e processamento do crescente conjunto de dados complexos. Ao final do projeto, foi possível classificar vários tipos de configurações (*tuning*) para diferentes tamanhos de saída de dados e níveis de paralelismo, permitindo assim, futuramente, aplicar as técnicas mais eficientes para outros tipos de algoritmos. Também foi disponibilizado um ambiente computacional para

processamento distribuído que futuramente poderá ser utilizado para o cálculo de outros algoritmos não mencionados neste trabalho.

1.2.2 Objetivos Específicos

- Armazenamento e processamento de dados ambientais com possibilidade de escalar para um contexto de *Big Data*;
- Realizar testes de performance da execução do cálculo da transformada de wavelets nos dados gerados pela missão espacial *Tropical Rainfall Measuring Mission (TRMM)* utilizando o processamento distribuído;
- Comparar os resultados de duas abordagens de processamento distribuído e uma sequencial;
- Executar os testes em diferentes cenários de particionamento de dados e identificar os mais eficientes;
- Prover um ambiente distribuído para futuros cálculos distribuídos.

1.2.3 Organização do Trabalho

Este trabalho está organizado da seguinte maneira:

- **Capítulo 02:** Apresenta a fundamentação teórica, abordando temas como processamento paralelo, *Big Data*, transformada de wavelets e TRMM;
- **Capítulo 03:** Apresenta os métodos e processos técnicos a que foram submetidos os dados: Os tipos de dados processados, programação utilizando o *Apache Spark* e os diferentes cenários testados;
- **Capítulo 04:** Apresenta os resultados obtidos, comparando os diferentes cenários de particionamento de dados, as duas abordagens distribuídas e a sequencial;
- **Capítulo 05:** Apresenta as conclusões deste trabalho.

Capítulo 2

REVISÃO BIBLIOGRÁFICA

O contexto de tratamento de dados ambientais tem evoluído para o contexto de *Big Data*. Neste trabalho o enfoque foi utilizar algumas ferramentas para processamento de dados em *Big Data*, assim este capítulo aborda o processamento de dados ambientais, a transformada de wavelets, precipitação, missão espacial TRMM *(Tropical Rainfall Measuring Mission)*, processamento paralelo e/ou distribuído de dados.

2.1 DADOS AMBIENTAIS

São inúmeros os sensores autônomos de coleta de dados utilizados na pesquisa ambiental que em conjunto com um *Data logger* são uma das maiores fontes de obtenção de dados ambientais na forma de séries temporais. A utilização de sensores faz com que a quantidade de dados gerados e manipulados nos estudos cheguem a casa de *terabytes* (FANG et al., 2014).

No contexto da física ambiental, os dados são gerados em intervalos de segundos (CAMPOS, 2015) e cada variável medida representa uma série temporal que será estudada através de métodos de análise de séries temporais.

2.1.1 SÉRIES TEMPORAIS

Um conjunto de dados de séries temporais é composto por observações sobre uma ou mais variáveis ao longo de um determinado período. Exemplos de dados temporais incluem os preços das ações nas bolsas de valores, índice de preços ao consumidor, produto interno bruto, dados micrometeorológicos, entre outros. Como os eventos do passado podem influenciar os eventos do futuro, o tempo é importante em um conjunto de dados em série, pois a ordem cronológica

das observações em um determinado período transmite informações potencialmente importantes (WOOLDRIDGE, 2013).

Uma característica importante das séries temporais que requer muita atenção, principalmente no contexto de *Big Data* é a frequência com que os dados são coletados. Na análise climática, as frequências podem ser de alguns segundos até vários anos. Pode-se por exemplo estimar o saldo de radiação diária, como também a temperatura a cada 30 minutos. No contexto da física ambiental, são comuns as análises de diferentes variáveis em determinado período de tempo. Isto é importante para verificar as similaridades entre diversas variáveis, estudar o comportamento climático e com isso ser capaz de prever valores e períodos sazonais.

2.1.2 TRANSFORMADA DE WAVELETS

A transformada de wavelets permite a análise periódica de eventos em diferentes escalas de variabilidade temporal, sem a necessidade de uma série estacionária (CAMPOS, 2015), tornando essa ferramenta apropriada para a análise de eventos irregularmente distribuídos e de séries temporais que possuam potências não-estacionárias em diferentes frequências (SANTOS et al., 2013).

A transformada de wavelets é uma das mais populares transformações tempo-frequência utilizadas atualmente e é muito utilizada como alternativa a transformada de Fourier, que é baseada na afirmação que toda função real periódica pode ser escrita como uma soma infinita de senos e cossenos. A grande maioria dos dados hidrológicos possuem uma variabilidade aperiódica, fazendo com que a interpretação do espectro de potência de Fourier seja complexa. Nesses casos a utilização da transformada de wavelets é muito eficiente (SANTOS et al., 2013).

A ideia fundamental da transformada de wavelets é que a transformação gera mudanças apenas no tempo e não na forma e consiste na decomposição de uma determinada série em diferentes resoluções, facilitando assim a sua interpretação (CAMPOS, 2015).

Na análise de dados de precipitação é possível utilizar wavelets para verificar a periodicidade e/ou frequência de energia, podendo ser utilizado para comparar biomas, por exemplo (GUARIENTI, 2015).

2.2 PRECIPITAÇÃO

A água é um dos elementos mais abundantes no planeta terra. Uma molécula de água contém um átomo de hidrogênio e dois de oxigênio que são conectados por ligações covalentes, formando as pontes de hidrogênio. Difíceis de serem quebradas, as pontes de hidrogênio são responsáveis por elevar o calor específico e mais ainda o calor latente de fusão e vaporização da água, portanto é necessário um elevado fluxo de calor para mudar a temperatura e muito mais ainda para mudar o estado físico da água. Essa característica faz com que a água seja uma excelente reguladora térmica do ambiente (REICHARDT; TIMM, 2004).

Quando evaporada, a água em forma de vapor eleva-se por convecção e passa a pertencer a um curso de água atmosférico que é formado por grandes massas de ar compostas por vapor de água. Quando resfriadas, as massas de ar precipitam e a água retorna para a superfície terrestre em forma de chuva (GUARIENTI, 2015).

As correntes de ar que transportam vapor de água são diretamente responsáveis na influência de um clima local. O seu comportamento impacta significativamente nas características da maioria dos biomas do planeta, fazendo do seu estudo indispensável na pesquisa climática (WARD ROY; ROBINSON, 1999).

Como a maioria das variáveis climáticas, o estudo do comportamento da precipitação é muito complexo e muitas vezes os dados coletados por aparelhos de estimativa de precipitação (Pluviógrafo e Pluviômetro) não fornecem informações sobre uma região, mas somente sobre as precipitações pontuais. Para a análise de áreas maiores, é comum utilizar dados provenientes de satélites e radares meteorológicos. Um exemplo de satélite meteorológico é o do projeto TRMM (*Tropical Rainfall Measuring Mission*).

2.2.1 MISSÃO ESPACIAL TRMM

O *Tropical Rainfall Measuring Mission (TRMM)*, é uma missão conjunta entre a Nasa e a agência *Japan Aerospace Exploration* (JAXA) para estudar precipitação voltada para a pesquisa climática (TRMM, 2015), oficialmente finalizado em 15 de abril de 2015. Lançado em novembro de 1997, o TRMM produziu mais de 17 anos de dados científicos valiosos. O conjunto de dados TRMM tornou-se o padrão para medir precipitação e levou a pequisa que melhorou a compreensão da estrutura dos ciclones tropicais e a evolução, propriedades do sistema convectivo, relações entre relâmpago e tempestade, modelagem de clima e tempo e os impactos humanos sobre precipitação pluviométrica. Os dados também apoia-

ram aplicações operacionais, tais como o monitoramento de inundações, secas e previsão de tempo (TRMM, 2015).

Os dados gerados pelo projeto TRMM são os mais confiáveis quando comparados com outros satélites meteorológicos (BARRERA, 2005). Os dados gerados representam os valores de precipitação de estações micrometeorológicas da superfície terrestre e são validados por *Ground Validation* (GV) (WOLFF et al., 2005).

O satélite tem como órbita uma área entre as latitudes 35ºN e 35ºS e apenas dentro dessa área ele é capaz de medir os índices de precipitação e observar as influências dessas no clima global (KUMMEROW et al., 2000). É uma ótima opção para monitorar regiões de difícil acesso, como as grandes áreas oceânicas.

A figura 1 mostra a área de abrangência do satélite do projeto TRMM:

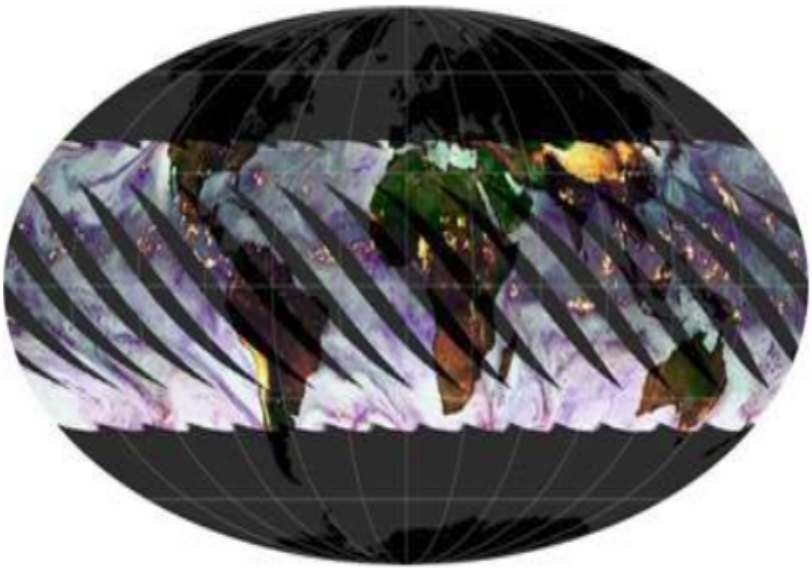


Figura 1: Órbita do satélite da missão TRMM (TRMM, 2015).

A utilização dos dados da missão TRMM permite estimar o índice de precipitação de grandes áreas, o que não é possível em outras abordagens de coleta de dados. A qualidade dos dados de precipitação são imprescindíveis para o êxito das pesquisas meteorológicas e a falta de dados para a análise de precipitação em grandes áreas eleva a demanda pela utilização de imagens de satélites artificiais (MACEDO, 2013).

2.3 PROCESSAMENTO PARALELO

O processamento paralelo é um tipo de cálculo em que muitos processos são efetuados simultaneamente, operando no princípio de que grandes problemas muitas vezes podem ser divididos em partes menores que consequentemente podem ser resolvidos ao mesmo tempo (ALMASI; GOTTLIEB, 1989).

A indústria computacional tem aceitado que o futuro aumento do desempenho será mais pelo acréscimo do número dos processadores e do número de *cores* por processador do que pela construção de um processador de *core* único (KUMAR et al., 2008).

De um modo geral, as abordagens de processamento paralelo requerem um administrador para orquestrar as unidades de processamento, sincronizando as entradas e saídas de dados. Na figura 2 é possível identificar o *dispatcher* (administrador), que recebe as solicitações de tarefas de um determinado cliente e distribui as tarefas entre diferentes *servers* (unidades de processamento), retornando o resultado para o cliente solicitante. Toda esta operação está resguardada por políticas de tolerância a falhas que garantem a eficácia da operação (GOMES, 2015).

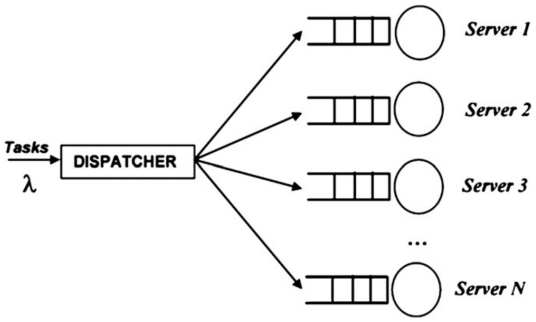


Figura 2: Modelo de servidor distribuído (SEMCHEDINE et al., 2011).

2.3.1 *CLUSTERS*

Um *Cluster* é um conjunto de computadores homogêneos, que interligados através de redes de alta velocidade, tem como objetivo processar um determinado problema simultaneamente. Geralmente são utilizados para executar uma grande quantidade de algoritmos complexos em um grande volume de dados (VALENTINI et al., 2011).

Cada computador em um *Cluster* é chamado de nó e trabalha de maneira independente para um objetivo em comum, fazendo com que a representação de todo o sistema seja única. Cada nó tem um papel específico no ambiente e pode possuir diferentes requisitos de *hardware* (VALENTINI et al., 2011).

Por possuírem características escaláveis, os *Clusters* são muito utilizados por sistemas que requerem processamento em tempo real, pois permitem a adição, remoção e manutenção de recursos ou nós sem a necessidade de interromper o funcionamento do ambiente (GOMES, 2015).

A figura 3 ilustra a arquitetura de um *Cluster* computacional tradicional:

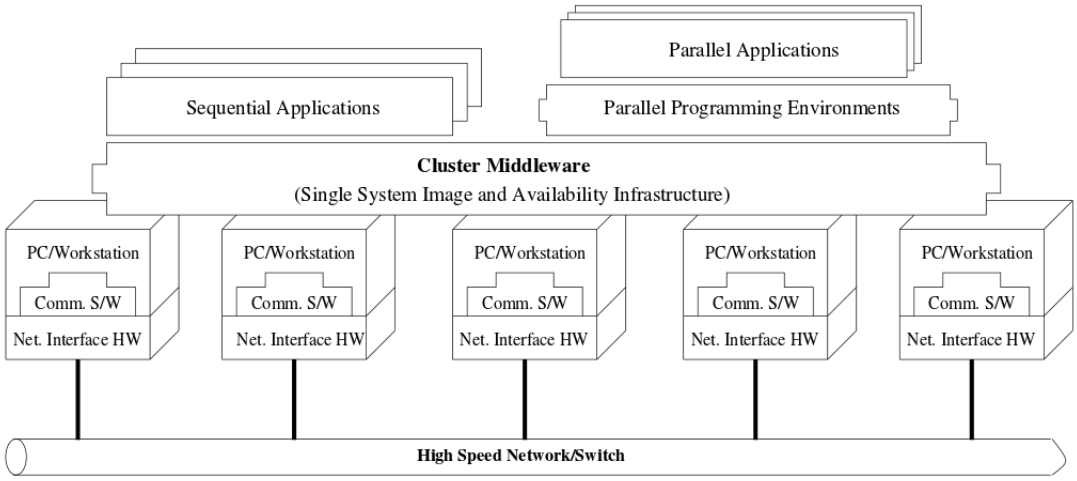


Figura 3: Arquitetura de um Cluster computacional (BAKERY; BUYYAZ, 1999).

Em alguns casos, as tarefas requerem grandes recursos de computação, como no processamento de modelos estatísticos complexos (STORLIE et al., 2014). Para solucionar esses problemas utiliza-se o HPC (*High-performance computing*), que são supercomputadores ou *Clusters* de vários computadores que possuem um alto desempenho computacional se comparado com os computadores comuns e sua eficiência é medida pela quantidade de operações de pontos flutuantes por segundo (FLOPS) (TOP500, 2015).

2.3.2 *GRID COMPUTING*

Semelhante a um *Cluster*, o *Grid Computing* é um conjunto de computadores heterogêneos interligados, porém não estão sujeitos a um controle centralizado, fazendo com que seja possível utilizar diferentes protocolos e interfaces de uso geral para fornecer diferentes tipos de serviços (VAQUERO et al., 2008).

Um *Grid Computing* também é escalável, sendo possível então adicionar e remover recursos sem a interrupção do serviço. Permitem também o compartilhamento de recursos de *hardware* e *software* entre diferentes nós da *Grid*, fazendo desse tipo de paralelismo o mais utilizado em programas de pesquisa (TORKESTANI, 2012). O CERN (*Conseil Européen pour la Recherche Nucléaire*) é um grande exemplo de *Grid Computing* (GOMES, 2015).

2.4 *BIG DATA*

Nos programas de pesquisa climática, são inúmeras as pesquisas realizadas sobre tratamento de imagens, por exemplo a análise de temperatura e albedo da superfície (BIUDES MARCELO S. E MACHADO, 2015), estimativa de produção primária bruta (GPP) de floresta de transição Amazônia-cerrado pelo modelo de fotossíntese da vegetação (VPM) através de imagens do *Thermatic Mapper (TM) Landsat 5* (DANELICHEN et al.,) e o *Tropical Rainfall Measuring Mission (TRMM)*. Geralmente as imagens e/ou conjunto de dados possuem mais de 1 *gigabyte* e quase sempre se faz necessário associar informações encontradas nas imagens com outras variáveis oriundas de sensores micrometeorológicos. Assim, há uma tendência de obtenção de mais dados e difusão de sensores, com isso, pode-se afirmar que esse contexto é classificado como *Big Data*.

Muitos autores definem que *Big Data* é um contexto que possui características baseadas em 5 V's (Ishwarappa; ANURADHA, 2015):

1. Volume: A quantidade dos dados é da ordem de *terabytes*, com tendência a *zetabytes*. Como exemplo temos os vários centros de pesquisa no mundo que possuem um enorme volume de dados arquivados, mas sem a capacidade de processá-los. Os benefícios de extrair informação desses dados é uma das principais características da análise de *Big Data*.

2. Velocidade: Refere-se a criação e transmissão dos dados, o que demanda também alta velocidade no processamento, armazenamento e análise. Todos os dias são gerados dados em estações micrometeorológicas, satélites, ou até mesmo dados colhidos em campo.

3. Variedade: A maioria dos dados gerados não são estruturados, portanto é extremamente difícil o armazenamento desses dados em bancos de dados relacionais. Lidar com uma variedade de dados estruturados, não estruturados e complexos aumenta o desafio de armazenamento e análise de *Big Data*. Por esses exemplos percebe-se que os dados utilizados na PGFA podem ser classificados como dados não-estruturados, pois não possuem uma estrutura definida e são variados como números, documentos, textos, vídeos e imagens.

4. Veracidade: Quando se está trabalhando com um grande volume de dados e com demanda de alta velocidade, não é possível realizar análises precisas em tempo real. Isso ocorre porque é preciso realizar controles e processamentos sobre a qualidade dos dados capturados que irão variar muito.

5. Valor: Obter acesso aos dados da *Big Data* torna-se inútil se não for possível converter esses dados em valores, o que para um ambiente de pesquisa significa gerar resultados com qualidade e com impactos sociais.

A Figura 4 demonstra algumas caracterísicas gerais do *Big Data*:

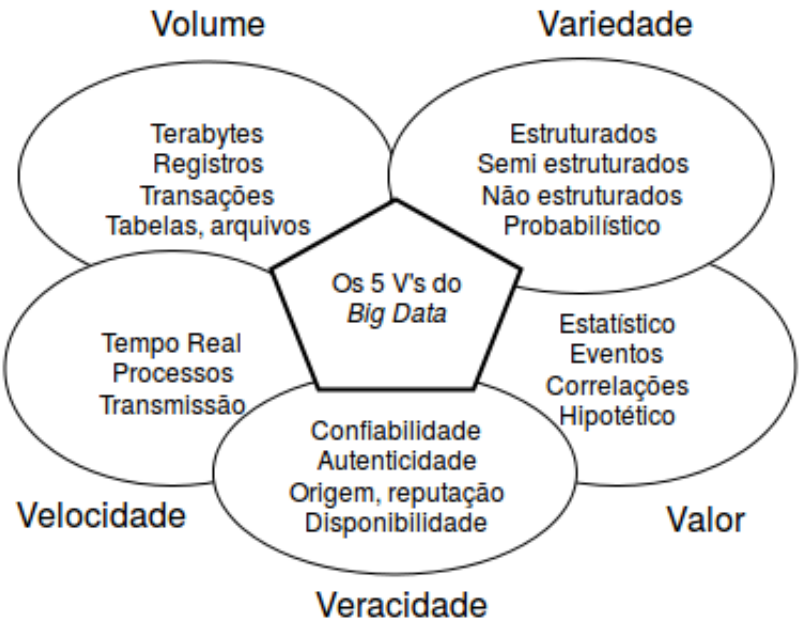


Figura 4: Representação dos 5 V's do *Big Data*(Ishwarappa; ANURADHA, 2015).

Para solução dos problemas gerados por um contexto Big Data, é preciso alcançar elevado poder computacional tanto em processamento quanto em armazenamento. O imenso volume de dados a ser analisado é complexo e difícil de

ser tratado, armazenado, analisado e consultado por meios convencionais. Os conjuntos de dados superam a capacidade de armazenamento de um único computador. Assim, somente um ambiente distribuído pode auxiliar nessas demandas (CAMPOS, 2015).

Devido ao grande volume de tarefas necessárias para processar *Big Data*, o modelo tradicional de processamento, onde existe apenas um processador que processa uma sequência de dados, não atende as necessidades de maneira adequada, portanto é necessário adotar a utilização de processamento paralelo.

2.4.1 O MODELO *MAPREDUCE*

O *MapReduce* é um modelo de programação para o processamento de grandes volumes de dados. Ele foi apresentando à comunidade pelos pesquisadores do *Google* Jeffrey Dean e Sanjay Ghemawat no ano de 2004. No modelo proposto, o usuário especifica uma função *map* que processa conjunto de dados chave/valor para gerar um outro conjunto intermediário de dados chave/valor, e uma função *reduce* que combina todos os dados intermediários através da mesma chave (DEAN; GHEMAWAT, 2004). Esse modelo foi criado devido a necessidade de se processar grandes volumes de dados, que no modelo tradicional se tornava impraticável, gerando uma alta demanda por computação distribuída e paralela. O modelo abstrai do usuário a complexidade do gerenciamento do paralelismo, tolerância a falhas, distribuição dos dados entre os nós e o balanceamento do processamento.

Suponha que a entrada de dados para uma função *map* seja *O que sabemos é uma gota o que ignoramos é um oceano* e deseja-se agrupar e contar o número de palavras do mesmo. Cada palavra no conjunto de dados é definida como uma chave e o seu valor será o número de representatividade da palavra (nesse caso 1). Dados intermediários são criados e ao final é apresentado o resultado da soma dos valores de todas as chaves para chegar ao total de palavras encontradas. A Tabela 1 ilustra este procedimento:

Tabela 1: Exemplo de um processo *MapReduce* (DEAN; GHEMAWAT, 2010).

Entrada *map*	**Saída *map***	**Entrada *reduce***	**Saída *reduce***
o que sabemos é uma gota o que ignoramos é um oceano	o 1 que 1 sabemos 1 é 1 uma 1 gota 1 o 1 que 1 ignoramos 1 é 1 um 1 oceano 1	o 1 1 que 1 1 sabemos 1 é 1 1 uma 1 gota 1 ignoramos 1 um 1 oceano 1	o 2 que 2 sabemos 1 é 2 uma 1 gota 1 ignoramos 1 um 1 oceano 1

Ao final do processo tem-se a quantidade total de ocorrências de cada palavra no conjunto de dados de entrada. A Figura 5 expressa com mais detalhes toda a operação:

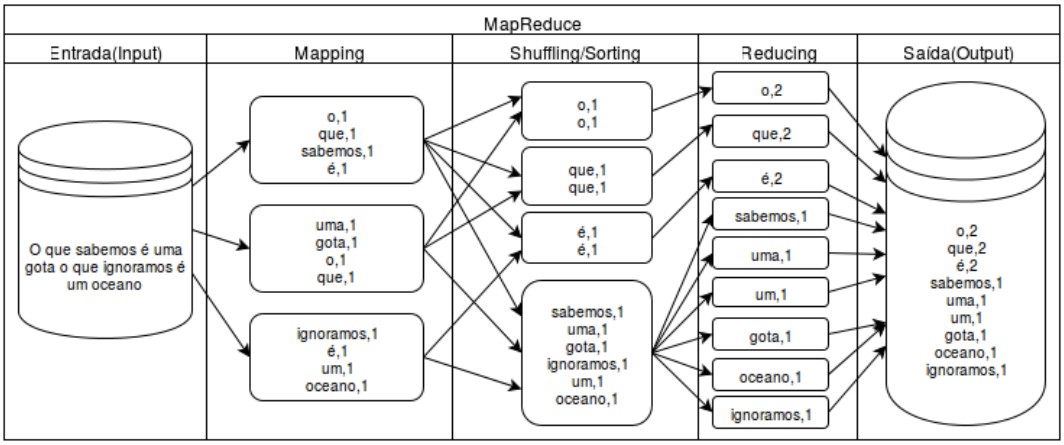


Figura 5: Diagrama exemplificando uma operação *MapReduce*.

A função *map* é acionada para cada linha do arquivo de entrada e tem-se como saída da função um par de dados (chave,valor), como podemos ver na Tabela 2:

Tabela 2: Exemplo de chave/valor (DEAN; GHEMAWAT, 2004).

Chave(k)	**Valor(v)**
gota	1

Neste caso o valor definido pelo usuário foi "1". Na próxima fase, *shuffling/sorting*, os dados são ordenados de acordo com suas chaves e enviados para

a próxima fase chamada *reducing*, que é representada pela função *reduce* que foi implementada pelo usuário. Neste exemplo os dados são agrupados e contabilizados pela soma dos valores. Na Figura 5 é possível identificar características de paralelismo e distribuição de dados. Para tornar isso possível em um grande volumes de dados, foi necessário a implementação de um sistema de arquivos distribuídos, o *GFS - Google File System*.

Outra característica do *MapReduce* é a proposta de utilização de *Clusters* com *hardware* de baixo custo e altamente escalável, que em conjunto com um controle de falhas proporcionam alta performance e robustez.

2.4.2 *GOOGLE FILE SYSTEM*

O GFS (*Google File System*) é um sistema de arquivos distribuído e escalável para grandes volumes de dados distribuídos (GHEMAWAT et al., 2003) que fornece tolerância a falhas enquanto é executado em *hardware* de baixo custo. Disponibilizado pelo *Google* em 2003, foi arquitetado e implementado para suprir a grande demanda de rápido crescimento do processamento de dados. O GFS possui todas as características de um sistema de arquivo distribuído: performance, escalabilidade, confiabilidade e disponibilidade e foi concebido levando em consideração quatro fatores que puderam ser observados nos produtos da empresa:

1. Todo *Hardware* é suscetível a falhas: A falha de componentes é mais comum do que uma exceção. O sistema de arquivos consiste de dezenas ou até centenas de nós de armazenamento e processamento, construído com *hardware* de baixo custo e é acessado por inúmeros clientes. O sistema deve garantir que mesmo em caso de falha, o processamento possa ser realizado;

2. Os arquivos são grandes por padrão: Ao invés de assumir que os arquivos são pequenos e apenas uma parte seja grande, assume-se a ocorrência de arquivos cujos tamanhos são da ordem de *terabytes*;

3. A maioria dos arquivos são modificados acrescentando novos dados, ao invés dos mesmos serem reescritos;

4. Projetar o sistema de arquivos e também os aplicativos que o utilizam permite maior flexibilidade.

Os arquivos são divididos em blocos chamados de *Chunks* que possuem o tamanho de 64 *megabytes*, o que é muito maior que o bloco de um sistema de arquivos comum. Cada *Chunk* é armazenado e replicado em diferentes nós no

Cluster, permitindo o processamento distribuído e provendo alta disponibilidade, pois se um nó que possuir o *Chunk 01* falhar, por exemplo, o mesmo *Chunk 01* poderá ser acessado em outro nó no qual ele tenha sido replicado anteriormente.

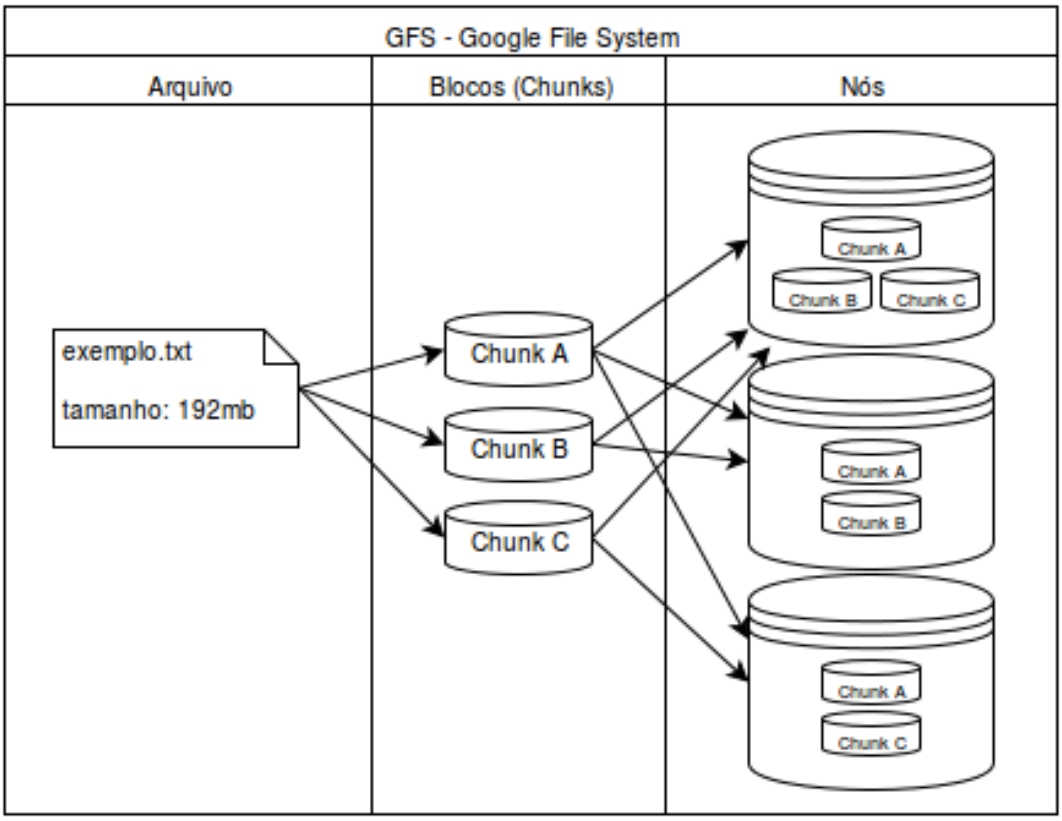


Figura 6: Exemplo de replicação de *chunks* do GFS (GHEMAWAT et al., 2003).

A estratégia de replicação dos blocos pode ser customizada definindo o número de vezes em que os blocos são replicados em todo o *Cluster*. Na Figura 6 o *Chunk A* foi replicado 3 vezes, enquanto que os *Chunks B e C* foram replicados apenas 2 vezes.

Baseando-se no *MapReduce* e no *Google File System*, a comunidade *Apache* desenvolveu o *framework Apache Hadoop*.

2.4.3 *APACHE HADOOP*

A biblioteca de software *Apache Hadoop* é um *framework* que permite o processamento distribuído de grandes conjuntos de dados em todo um aglomerado de computadores comuns que usam os modelos de programação simples. Ele é projetado para escalar a partir de um único servidor para milhares de máquinas, cada um oferecendo processamento e armazenamento local. 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 no topo de um conjunto de computadores, onde cada um dos quais pode ser propenso a falhas (HADOOP, 2015).

Os principais módulos do projeto são:

1. *Hadoop MapReduce*;
2. *Hadoop Distributed File System (HDFS)*: Um sistema de arquivos distribuídos baseado no *GFS*;
3. *Hadoop YARN*: Um *framework* para agendamento de tarefas e administração de recursos do *Cluster*.

2.4.4 Arquitetura HDFS

2.4.4.1 *NameNode e DataNode*

O HDFS possui uma arquitetura *master/slave*. Consiste de um único *NameNode*, um servidor master que administra o *namespace* do sistema de arquivos e regula o acesso dos clientes aos arquivos. Todavia, existe um *DataNode* por nó do *Cluster* que gerencia o armazenamento conectado ao nó no qual eles são executados. O HDFS disponibiliza um *namespace* de sistema de arquivos e permite que dados de usuários sejam armazenados em arquivos. Internamente um arquivo é dividido em um ou mais blocos que são armazenados em um conjunto de *DataNodes*. O *NameNode* executa operações de *namespace* do sistema de arquivos como abrir, fechar e renomear e também determina o mapeamento dos blocos nos *DataNodes*. Os *DataNodes* são responsáveis por servir requisições de escrita e leitura dos clientes do sistema de arquivo e realizam a criação, exclusão e replicação de blocos através de instruções originadas de um *NameNode*.

A figura 7 demonstra o fluxo de dados em uma arquitetura HDFS:

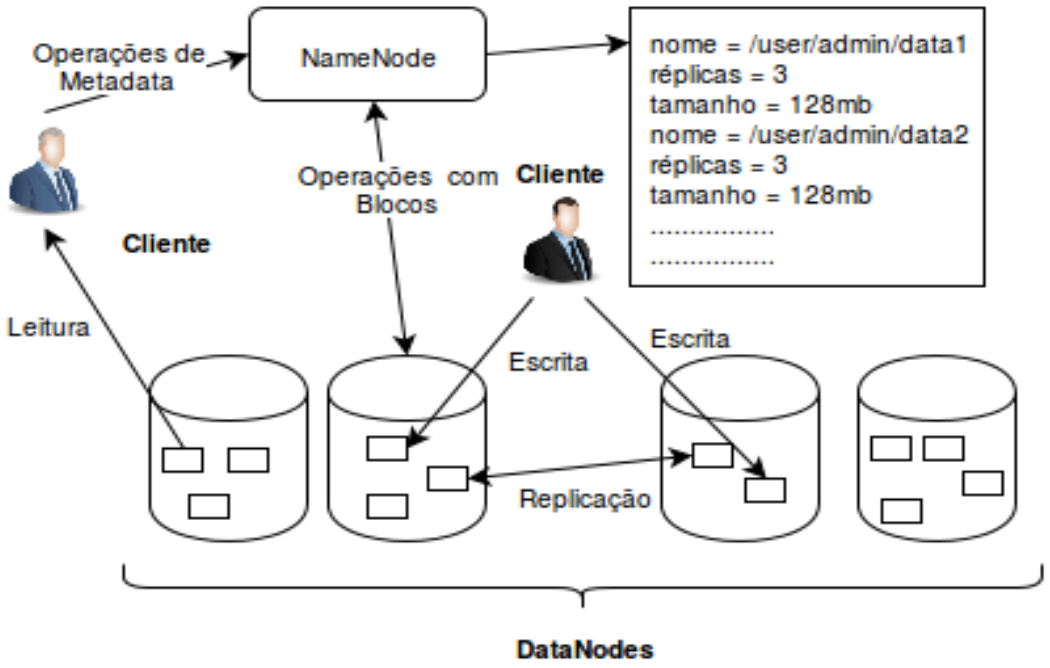


Figura 7: Arquitetura HDFS (HADOOP, 2015).

2.4.4.2 O *Namespace* do sistema de arquivos

O HDFS fornece uma organização de arquivos de forma hierárquica, como ocorre com a maioria dos sistemas de arquivos utilizados. Ao usuário é possível criar diretórios e armazenar arquivos ou outros diretórios dentro dos mesmos e realizar operações comuns como criação, alteração, visualização e exclusão.

O *namespace* é gerenciado pelo *NameNode*. Qualquer tipo de mudança nele é registrado pelo *NameNode*. É possível à aplicação especificar o número de réplicas de um arquivo armazenado no HDFS.

Dessa forma, pode-se definir nos arquivos de configuração o tamanho dos blocos e a quantidade de replicação dos mesmos, o que permite uma maior flexibilidade para trabalhar com arquivos na ordem de *gigabytes* ou maiores, como por exemplo na ordem de *terabytes*. Isto se deve ao fato do HDFS ter sido desenvolvido para ser confiável no armazenamento de grandes arquivos através de vários nós em um grande *Cluster*. A replicação dos blocos de um arquivo é feita para garantir a tolerância a falhas. O *NameNode* é responsável pelas decisões que envolvem a replicação de dados e periodicamente recebe relatórios sobre a saúde dos blocos e da disponibilidade de um nó no *Cluster* (*Heartbeat*). O *Heartbeat* verifica se um determinado *DataNode* está em funcionamento no momento.

O *NameNode* precisa de alta disponibilidade, pois ele sempre será requisitado para a regulação de acesso aos blocos de dados, por isso é possível nas

configurações do HDFS indicar mais de um servidor para o *NameNode*, possibilitando a continuidade do serviço mesmo após a queda de um dos servidores.

Todos os protocolos de comunicação do HDFS estão em camadas no topo de um protocolo TCP/IP. Um cliente estabelece uma conexão com uma porta TCP pré-configurada na máquina onde está localizado o *Namenode*. Os *Datanodes* comunicam-se com o *Namenode* através do protocolo *Datanode* e um RPC (Chamada de procedimento remoto) envolve tanto o protocolo cliente, como o protocolo *Datanode*. Por definição, o *Namenode* nunca inicializará qualquer RPC e somente responderá por requisições de RPC enviadas por *Datanodes* e clientes (HADOOP, 2015).

2.4.5 *Hadoop Yarn*

O *MapReduce* do *Apache Hadoop* foi totalmente remodelado a partir da versão *hadoop-0.23* e passou a chamar-se *MapReduce 2* ou *Yarn* (HADOOP, 2015).

2.4.5.1 O *ResourceManager*

O *ResourceManager* é o Master que orquestra todos os recursos do *Cluster* disponíveis e assim ajuda a gerenciar as aplicações distribuídas em execução no sistema *Yarn*. Ele trabalha em conjunto com os *NodeManagers* de cada nó e os *ApplicationMasters* de cada aplicação (YARN, 2015).

1. *NodeManagers* buscam instruções do *ResourceManager* e gerencia os recursos disponíveis em um nó.

2. *ApplicationMasters* são responsáveis pela negociação de recursos com o *ResourceManager* e por trabalhar em conjunto com os *NodeManagers* para inicializar os *containers*.

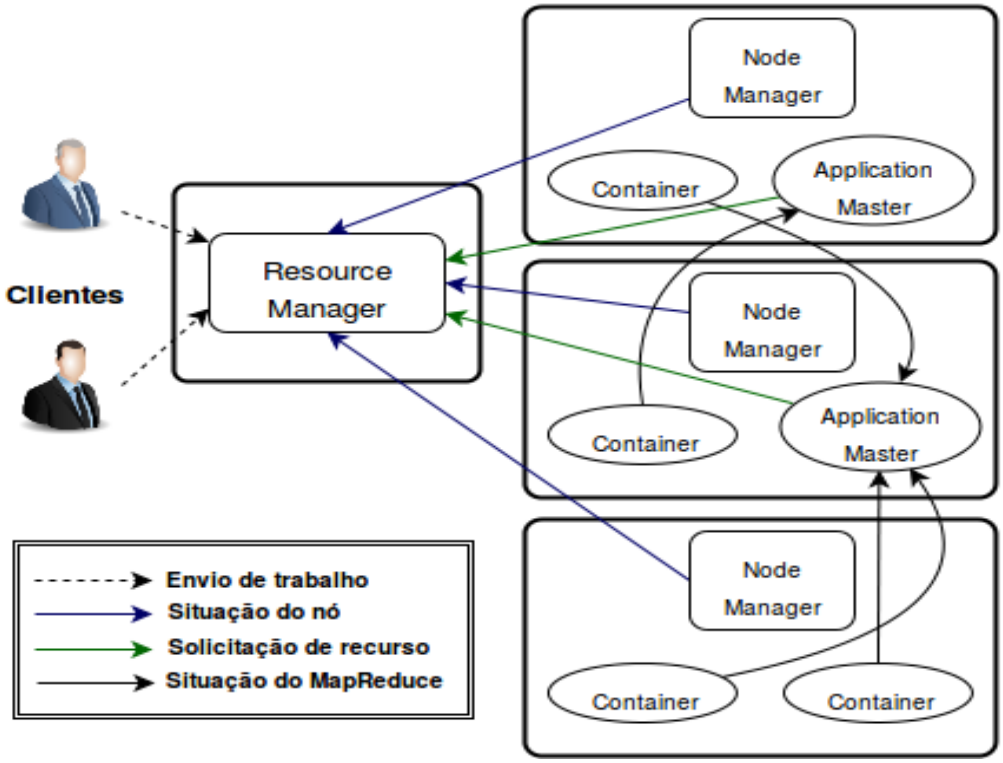


Figura 8: Arquitetura YARN (HADOOP, 2015).

Na Figura 8 fica evidenciado como o Yarn consegue utilizar dados de diferentes *containers* em uma aplicação e como é possível ter aplicações sendo executadas em diferentes nós ao mesmo tempo.

2.4.6 O ecossistema Hadoop

Com o passar do tempo, o processamento de grandes volumes de dados não limitou-se a apenas a utilizar *MapReduce*. Outros propósitos tornaram-se necessários para a comunidade *Big Data* como por exemplo a organização dos dados através da utilização de linguagem *SQL*, o processamento de dados em tempo real, a utilização de algoritmos de estatística como classificação, regressão, Clusterização, etc e até mesmo o processamento de grafos e coleções. Para atender essas necessidades, a comunidade desenvolveu ao longo do tempo ferramentas para supri-las, como por exemplo o *Apache Spark*.

O atual ecossistema *Apache Hadoop* consiste no kernel do Hadoop , MapReduce , o sistema arquivos distribuídos do Hadoop (HDFS) e uma série de projetos relacionados, tais como Apache Hive, HBase e Zookeeper.

Na figura 9 é possível observar os *frameworks* em suas respectivas camadas:

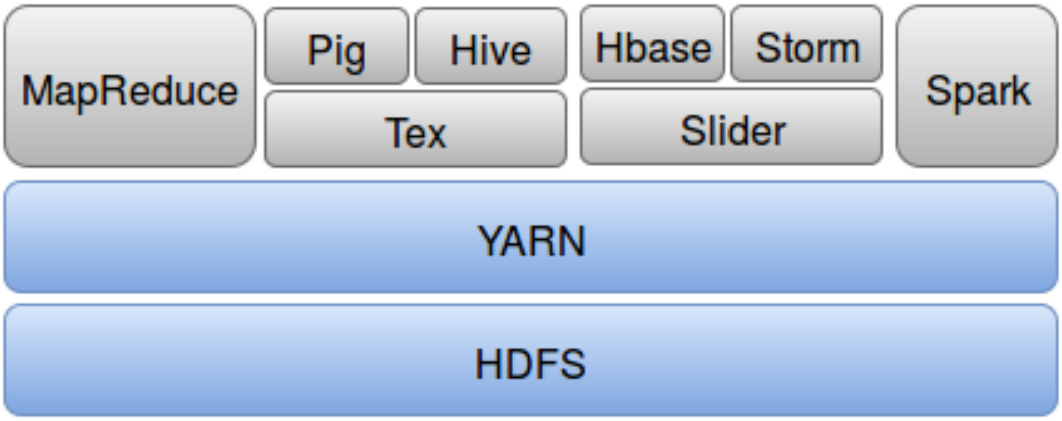


Figura 9: O ecossistema *Apache Hadoop* (HADOOP, 2015).

2.4.7 Apache Ambari

O *Apache Ambari* é um projeto que tem como objetivo tornar a administração de um ecossistema *Apache Hadoop* mais simples através do provisionamento, gestão e monitoramento do *Cluster*. Com ele é possível criar um novo *Cluster*, adicionar novos nós, *frameworks* e monitorar o *Cluster* através de um painel de *dashboard*. Isso tudo pode ser feito através de um ambiente gráfico, utilizando um *browser* da internet (AMBARI, 2015).

2.5 *APACHE SPARK*

O *Apache Spark* é um conjunto de ferramentas de uso geral para a computação em *Cluster*. Com o *Apache Spark* é possível utilizar SQL para o processamento de dados estruturados, *Machine Learning* para o aprendizado de máquina, análise de dados em tempo real e processamento de grafos (SPARK, 2015). Foi desenvolvido por Matei Zaharia em sua tese de doutorado na Universidade da Califórnia, Berkley.

O *MapReduce* apresenta um modelo geral e simples para processamento em lote com tolerância a falhas, entretanto, ele é pouco adequado para muitas aplicações importantes, incluindo algoritmos interativos, consultas interativas e processamento em tempo real. Isto levou ao desenvolvimento de uma vasta quantidade de *frameworks* para estas aplicações (ZAHARIA, 2014).

Foi possível notar que muitos dos aplicativos que não conseguiram implementar o *MapReduce* tem a mesma característica: Todos eles exigem um eficiente compartilhamento dos dados através de cálculos computacionais. Por exemplo: Algoritmos interativos como o *PageRank*, Clusterização *K-means* ou regressão lo-

gística precisam realizar inúmeras consultas a um mesmo subconjunto de dados, compartilhando o resultado entre os processos em execução e a única maneira de fazer isso nos *frameworks* tradicionais é persistindo os dados em uma unidade de armazenamento, como por exemplo o HDFS. Isto acaba causando problemas de performance devido a alta replicação de dados, operações de disco e serialização (ZAHARIA, 2014).

Para contornar esses problemas, foi apresentada uma nova abstração de como os dados são armazenados em memória e em disco, os chamados *Resilient Distributed Datasets - RDD*.

2.5.1 *Resilient Distributed Datasets - RDD*

O *Resilient Distributed Dataset - RDD* é uma extensão do modelo *MapReduce* (ZAHARIA, 2014). O RDD é uma coleção de elementos particionados através dos nós do *Cluster* que podem ser operados em paralelo.

No *Apache Spark*, a tolerância a falhas é garantida graças ao *Directed acyclic graph - DAG*, que é um grafo que representa de forma encadeada e ordenada as transformações e ações do RDD. Em caso de falha, o *Apache Spark* simplesmente processa novamente o RDD baseando-se no seu DAG original. Existem dois tipos de operações com RDDs: *transformações*, que criam um novo RDD a partir de outro existente e *ações* que retornam um valor para o cliente após executar um conjunto de dados. Por exemplo, *map* é uma transformação que é feita em cada elemento do conjunto de dados através de uma função e retorna um novo RDD que representa os resultados e o *reduce* é uma ação que agrega todos os elementos do RDD usando alguma função e retorna o resultado final para um cliente.

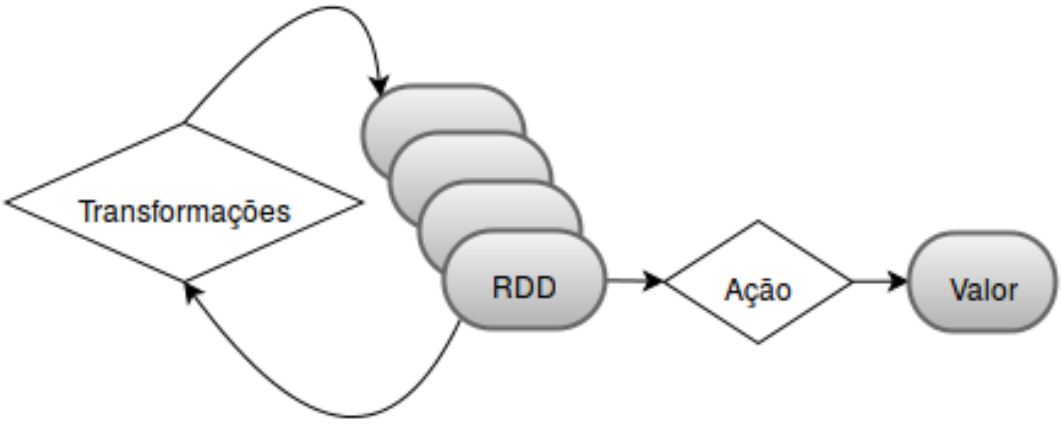


Figura 10: Operações de um RDD (SPARK, 2015).

Todas as transformações no *Apache Spark* são *lazy*, o que significa que as mesmas não são executadas imediatamente. As transformações aplicadas a um conjunto de dados ficam em espera e somente são computadas quando uma ação requer um resultado para retornar a um cliente. Isto permite ao *Apache Spark* uma maior performance, pois é possível perceber que um conjunto de dados criados através de uma função *map* será utilizado em um *reduce* e retornará somente o resultado do *reduce* para o cliente, ao invés de todo o conjunto de dados mapeado.

Por definição, cada transformação em um RDD será executada toda vez que for executada uma ação, entretanto é possível persistir o RDD em memória ou em disco, isso permite manter os elementos no *Cluster* para um acesso muito mais rápido na próxima consulta e é possivel persistir RDDs no disco ou replicar através de vários nós, isso, inclusive é o grande diferencial em relação ao modelo *MapReduce* do *Apache Hadoop* (SPARK, 2015).

A figura 11 ilustra as estratégias de leitura e escrita do *Apache Hadoop* e *Apache Spark*.

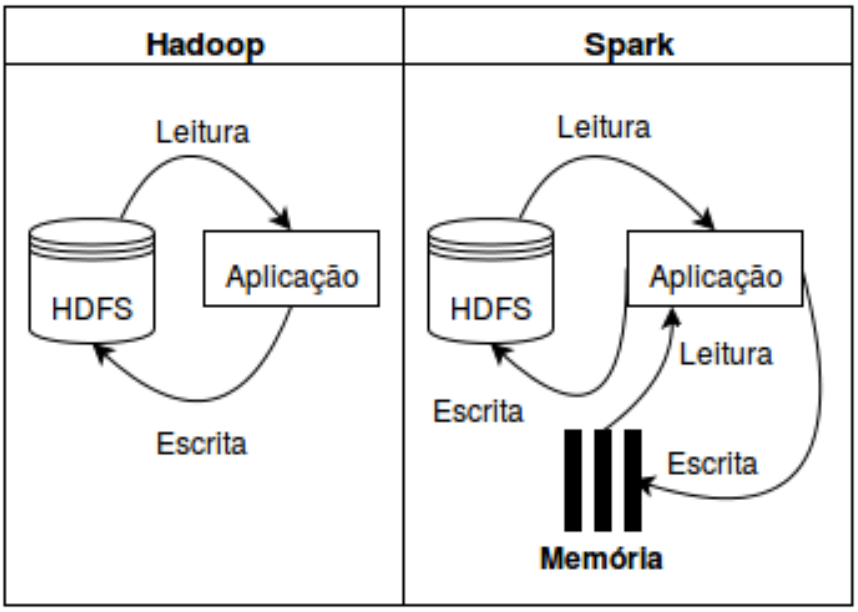


Figura 11: *MapReduce* vs *Apache Spark*.

Os RDDs são coleções particionadas de registros somente para leitura. Os RDDs só podem ser criados através de operações de transformação em dados armazenados ou em outros RDDs. Alguns exemplos de transformação são *map*, *filter* e *join* (ZAHARIA, 2014).

Quadro 3: Exemplos de transformação no *Apache Spark*.

Transformação	**Significado**
map(*func*)	Retorna um novo RDD formado pela passagem de cada elemento da fonte de dados que passou pela função *func*
filter(*func*)	Retorna um novo RDD formado pela seleção dos elementos da fonte através de uma função *func*
join(*outroConjuntoDados, [numTarefas]*)	Retorna a combinação dos elementos de dois RDDs: *((K,V) join (K,W))* retorna *(K, (V, W))*

Fonte: (SPARK, 2015)

Um RDD possui informações necessárias de como ele foi derivado de outros conjuntos de dados para calcular suas partições de dados armazenados (linhagem). Isso é importante para que um programa não faça referência a um RDD que não pode ser reconstruído após uma falha. Os usuários também podem controlar dois aspectos dos RDDs: persistência e particionamento. É possível indicar qual RDD será reutilizado e escolher a melhor estratégia de armazenamento (Memória ou Disco), possibilitando a reutilização do RDD futuramente (ZAHARIA, 2014).

Após definir um ou mais RDDs através de transformações em outros conjuntos de dados, faz-se necessário realizar operações que exportam os valores para a aplicação ou para um sistema de armazenamento. Para isso são utilizadas ações que podem, por exemplo, retornar a quantidade de elementos de um conjunto de dados (*count*), retornar os próprios elementos (*collect*) ou mesmo salvar os dados em um sistema de armazenamento (*save*).

Quadro 4: Exemplos de Ação no *Apache Spark*.

Ação	**Significado**
count()	Retorna a quantidade de elementos de um RDD
collect()	Retorna todos os elementos de um RDD para o driver program. Geralmente é muito útil após um filtro ou outra operação que retorne um conjunto pequeno de dados.
saveAsTextFile(*caminho*)	Escreve em um sistema de arquivos os elementos de um RDD em formato texto.

Fonte: (SPARK, 2015)

Somente as ações geram novos RDDs, pois como dito anteriormente as transformações são do tipo *lazy*, ou seja, só são executadas após uma ação, criando transformações canalizadas (*pipeline*).

2.5.2 Arquitetura do *Apache Spark*

A arquitetura aqui especificada é do *Apache Spark* quando executado em modo *Cluster*. O sistema suporta atualmente três tipos de *Cluster*: *Standalone*, *Apache mesos* e *Hadoop YARN* (SPARK, 2015). Neste projeto o tipo de *Cluster* utilizado foi o *Hadoop YARN*, pois o objetivo era utilizar o *Apache Spark* como ferramenta para o processamento de dados e o *Hadoop HDFS* como responsável pela alta disponibilidade e armazenamento distribuído de dados.

As aplicações do *Apache Spark* executam como um conjunto de processos independentes em um Cluster, coordenados pelo objeto *SparkContext* no seu programa principal, que também é chamado de *driver program*. Para ser executado em um *Cluster*, o *SparkContext* conecta-se a um administrador de *Clusters* (neste caso o *Hadoop YARN*), que aloca os recursos através das aplicações. Quando conectado, *Apache Spark* adquire executores (*worker node*) em nós do *Cluster*, que são processos que computam e armazenam dados para a sua aplicação. Feito isso, ele envia o código da aplicação para os executores e o *SparkContext* envia tarefas para serem executadas pelos executores (SPARK, 2015).

A Figura 12 ilustra o funcionamento da arquitetura do *Apache Spark*:

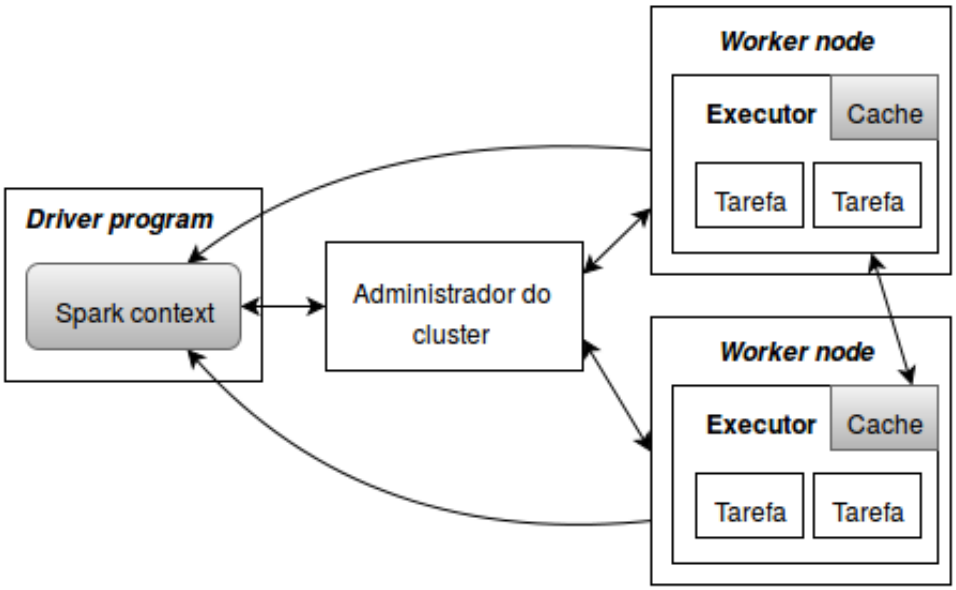


Figura 12: Arquitetura do *Apache Spark* em modo *Cluster* (SPARK, 2015).

O *Apache Spark* permite implementar funcionalidades que somente o *Apache Hadoop* não é capaz de realizar, como por exemplo na interdependência dos dados da série temporal, porém o *Apache Hadoop* fornece auxílio no armazenamento, escalonamento e disponibilização de grandes volumes de dados através do HDFS. Além do mais, o *Apache Spark* possui bibliotecas nativas para cálculos estatísticos e análise de dados em tempo real, que são imprescindíveis em um ambiente de pesquisa ambiental.

Capítulo 3

MATERIAL E MÉTODOS

Para substituir o modelo tradicional de processamento sequencial pelo processamento distribuído e melhorar a eficiência na programação de algoritmos voltados para o processamento de dados ambientais e temporais, utilizou-se do *framework Apache Spark* para os cálculos de precipitação e do *Apache Hadoop* para o armazenamento dos dados e resultados, e como o administrador do *cluster*.

Todo este ecossistema *Hadoop* está sendo administrado através do *Apache Ambari*.

3.1 AMBIENTE DE TRABALHO

O ambiente consiste de um *cluster* com 20 máquinas físicas e 3 máquinas virtuais utilizando o sistema operacional *CentOS 7*.

As 20 máquinas físicas exercem o papel de *DataNodes*, ou seja, são responsáveis pelo armazenamento e processamento distribuído e possuem um processador AMD A10 com 4 núcleos e 1,4 GHZ de clock, 4 GB de memória RAM e 200 GB de espaço em disco. Essas máquinas físicas utilizam placas de rede de 10/100 Mbps conectadas a uma rede de topologia estrela com 2 *switches* de 1 GB.

O ambiente físico está localizado em um laboratório do Instituto de Computação da Universidade Federal de Mato Grosso, como pode ser visto na figura 13:

Figura 13: Laboratório das máquina físicas do *cluster*.

As 3 máquinas virtuais estão localizadas em um servidor e possuem 4GB de memória RAM e 45GB de espaço em cada disco, sendo responsáveis por hospedar os diversos serviços de gerenciamento do ambiente distribuído como:

1. *NameNode* e *Secondary NameNode*, que são responsáveis pelos metadados do HDFS;

2. O *Spark History Server*, que é responsável pelo *Apache Spark*;

3. O *Apache Ambari Server* que é um facilitador de administração para um *cluster Apache Hadoop*.

As máquinas virtuais estão em um servidor que utiliza uma placa de rede física 10/100/1000 Mbps, com *switches* virtuais configurados para permitir a comunicação das máquinas virtuais com a rede do laboratório das máquinas físicas.

As versões utilizadas no projeto foram a do *Apache Hadoop*, do *Apache Spark* .

Os testes utilizando os *frameworks Apache Hadoop* versão 2.7.1, *Apache Spark* versão 1.4.1 e *Apache Ambari* versão 2.1.2 foram realizados no ambiente descrito anteriormente. Já o teste sequencial foi feito em uma das máquinas físicas, utilizando a linguagem de programação *java* para processar os dados armazenados em um servidor *PostgreSQL* configurado localmente. As tabelas foram criadas com todos os índices necessários para uma melhor performance de leitura dos dados.

3.2 ORGANIZAÇÃO DOS DADOS

O conjunto de dados de séries temporais da missão TRMM foram organizados em um arquivo de extensão *.csv (*Comma-separated values*) armazenado no HDFS com o seguinte formato: data, latitude, longitude, Precipitação e Identificação da localização. Para cada linha ou registro do arquivo, foram incluídas três novas identificações: mensal, semestral e anual para otimizar a pesquisa pelas séries originais (CAMPOS, 2015).

O Id. mensal é composto pelo ano, mês e Id. da localização. O Id. do semestre é composto pelo ano, semestre (1 ou 2) e Id. da localização e o Id. anual é composto pelo ano e Id. da localização. A Tabela 5 exemplifica esses registros:

Tabela 5: Registro adaptado do TRMM.

Data	Lat.	Long.	Prec.	Id.	Id. Mês	Id. Semestre	Id. Ano
1998-01-03	-19.625	-53.875	9.902	22	19980122	1998122	199822

Os locais utilizados para o processamento pertencem a 4 diferentes biomas: Amazônia, Cerrado, Pantanal e Mata atlântica, como observado na Figura 14:

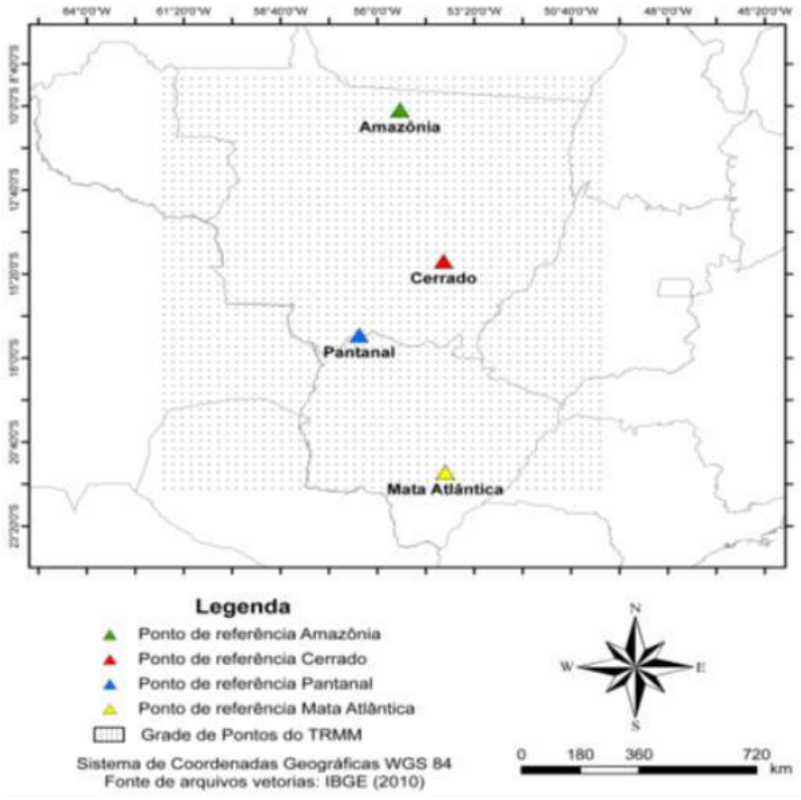


Figura 14: Localização das áreas de estudo (IBGE, 2015).

3.3 IMPLEMENTAÇÃO DOS ALGORITMOS

O *Apache Spark* permite a implementação em 3 linguagens diferentes: Scala, Java e Python. Para este projeto a linguagem escolhida foi a Scala pelo maior suporte da comunidade e maior utilização pelos projetos de pesquisas sobre *Apache Spark*.

Scala é uma linguagem de programação que unifica a programação orientada a objeto e a funcional. É uma linguagem orientada a objeto onde cada valor é um objeto em si e seus tipos e comportamentos são descritos como classes. A linguagem foi desenvolvida para operar perfeitamente com Java.

No objeto *main* do projeto, um *Spark Context* é instanciado para dar início à aplicação. É no *Spark Context* que todas as operações tem início e fim, e todas as aplicações em um ambiente *Apache Spark* sempre deverão ter um *Spark Context* instanciado.

Algoritmo 1 Instanciado o *Spark Context*.

```
1: val sparkConf = new SparkConf().setAppName("Programa")
2: val sc = new SparkContext(sparkConf))
```

Na linha 1 do algoritmo 1, a classe *SparkConf* é instanciada e é enviado como parâmetro do construtor da classe *SparkContext*. É na classe *SparkConf* que são definidas inúmeras configurações para a aplicação a ser executada, como por exemplo o nome do programa que poderá ser enviado via parâmetro pelo comando *spark-submit* (SPARK, 2015).

3.3.1 Fluxo de informação

A Figura 15 demonstra o fluxo de transformada das wavelets. O usuário utiliza o comando *spark-submit* para enviar um pacote *jar* para o *cluster YARN* passando também parâmetros para configuração e sintonia (*tuning*) do ambiente e da aplicação em si. Ao receber a solicitação, o *Spark* através do *cluster YARN* instancia um *Spark Context* que fará a solicitação de leitura do arquivo (passado por parâmetro pelo *spark-submit*) localizado no *HDFS* para carregar um RDD.

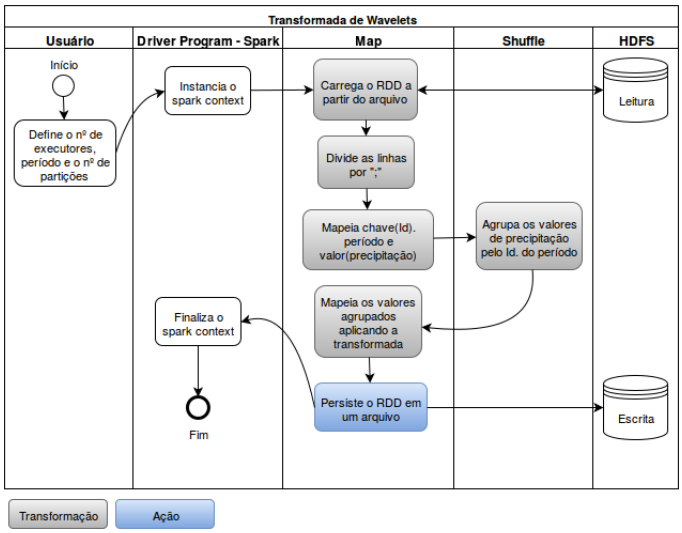


Figura 15: Fluxo da transformada de wavelets.

Na etapa *map*, o arquivo é dividido entre ';' e mapeados com o valor da chave (*key*) igual ao ID do período selecionado (mês, semestre ou ano enviado por parâmetro) e o valor representado pelo índice de precipitação. Na etapa *shuffle* os dados são agrupados pela chave e logo após é aplicada a transformada de wavelets nos agrupamentos de precipitação. Esta é a etapa mais dispendiosa, pois no agrupamento os dados são reorganizados por todo o *cluster*.

Após o mapeamento dos valores com a aplicação da transformada de wavelets, o RDD representativo do resultado obtido é persistido em forma de texto no HDFS. Para persistir os dados da wavelet foi utilizado o tipo de dado *org.apache.spark.mllib.linalg.Vector* da biblioteca MLlib do *Apache Spark*, que permite a serialização de dados que é um processo de converter um determinado objeto em bytes, para que o mesmo possa ser persistido em memória ou em arquivos.

A Figura 16 exemplifica um registro de saída da transformada de wavelets, onde no primeiro item temos o Id. do período selecionado e o segundo item é um *vector* serializado que pode ter N elementos, variando de acordo com a quantidade de itens da transformada. Neste exemplo obtem-se o retorno de apenas três elementos.

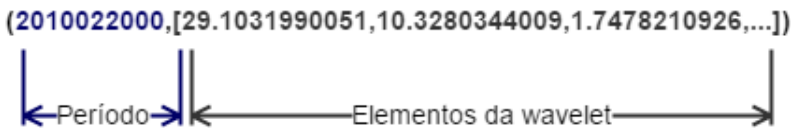


Figura 16: Exemplo de uma wavelet com 3 elementos.

A entrada de dados para a realização dos cálculos tem aproximadamente 1.555,4 *megabytes* de tamanho e o volume dos dados de saída variam de acordo com o período calculado: O cálculo mensal tende a ter menos elementos na transformada do que os períodos semestrais e anuais, portanto o volume de dados tende a ser maior nos períodos anuais e semestrais e menor no período mensal.

A linguagem Scala tem como principal característica a facilidade na execução de comandos em cadeia. O algoritmo 2 apresenta a codificação do fluxo apresentado anteriormente:

Algoritmo 2 Transformada de wavelets em scala.

```
1: wavelet = sc.textFile(OrigemHDFS)
2: wavelet.map(line =>line.split(";"))
3: wavelet.map(vals => (vals(periodo).toInt,vals(3).toDouble)
4: wavelet.groupByKey(numParticoes)
5: wavelet.mapValues(list => createWavelet(list,periodo))
6: wavelet.saveAsTextFile(DestinoHDFS)
```

Cada linha no algoritmo 2 representa uma etapa:

1. A variável *wavelet*, que representa o RDD, é carregada com os valores do arquivo no HDFS;
2. É feito um *map* dividindo o arquivos entre *";"*;
3. Os dados são mapeados em chave (ID. período) e valor (Precipitação);
4. Executa o agrupamento dos dados pela chave (id. período). Nesta etapa é possível definir o número de partições no comando *groupByKey* para uma melhor utilização dos nós e dos *cores* do processador de cada nó do *cluster*;
5. Os valores de precipitação de cada Id. período são mapeados e neste momento é aplicada a função que realiza a transformada de wavelets (*createWavelet*);
6. O resultado obtido é persistido em outro local do HDFS.

O DAG da aplicação demonstra claramente a linhagem de transformações e ação do RDD. A aplicação foi dividida em dois estágios e enumerados para indicar as estapas do algoritmo, como pode ser visto na Figura 17.

No primeiro estágio é feito o *map* dos dados para que no segundo estágio os dados sejam distribuídos através de *shuffle* (*groupByKey*) ,processados (*mapValues*) e persistidos no HDFS (*saveAsTextFile*).

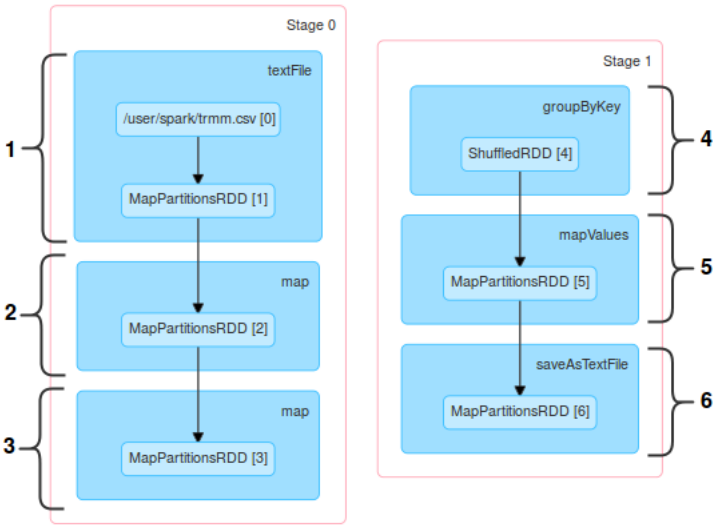


Figura 17: Estágios e etapas do DAG da transformada de wavelets.

Programas feitos para serem executados de maneira distribuída muitas vezes apresentam gargalos de processamento por qualquer tipo de recurso no *cluster*: CPU, rede, memória, etc. Para evitar problemas de performance, é necessário definir a sintonia (*tuning*) do ambiente a ser utilizado. Isso pode ser feito através de *tuning* de memória e/ou níveis de paralelismo, que podem ser definidos no *Apache Spark* através do reparticionamento dos dados. Para identificar gargalos e consequentemente melhorar a performance do ambiente distribuído, os testes realizados levaram em consideração o período dos dados, número de executores (*nós*), número de núcleos (*cores*) de cada processador e o número de partições a serem processadas. A Tabela 6 especifica quais os testes realizados para os dados mensais, semestrais e anuais:

Tabela 6: Configurações para os testes.

Executores	Núcleos	Total (Exec x núcleo)	Partições				
			1x	2x	3x	5x	8x
20	1	20	20	40	60	100	160
	2	40	40	80	120	200	320
	3	60	60	120	180	300	480

A primeira coluna representa o número executores (nós) do *cluster* do processo, a segunda indica o número de *cores* utilizados por cada processador de cada executor, a terceira totaliza a quantidade de *cores* do processo e a quarta

a quantidade de partições do *groupByKey*. O número de partições vai variar de acordo com o número total de *cores* do processo.

A proporção de cores/*partições* dos testes foram definidos utilizando sequencia de Fibonacci, para que fosse possível identificar sensíveis diferenças de tempo entre cada tipo de cenário.

Para auxiliar na identificação dos testes mais eficientes (velocidade de processamento), foi utilizada a Análise de variância (ANOVA), que é uma técnica estatística que basicamente verifica se existe uma diferença significativa entre média de populações (SOKAL ROBERT; ROHLF F, 1997). Para definir a maior diferença significativa entre os diferentes cenários testados e assim auxiliar na análise dos gráficos de resultados, foi utilizado o teste de *Tukey*.

Todos os cálculos estatísticos presentes no trabalho foram feitos utilizando o *software* R para cálculos estatísticos.

Capítulo 4

RESULTADOS

A aplicação de algoritmos distribuídos em dados de precipitação permitem diversas análises e o enfoque principal foi o de verificar o comportamento do processamento dos dados perante as variações na quantidade de dados e da infraestrutura computacional. Assim, a transformada de wavelets, que é uma função que requer complexos e volumosos cálculos computacionais, foi utilizada nos testes.

A tabela 7 demonstra todos os detalhes de entrada e saída de dados em nível de tamanho em disco, quantidade de registros e elementos da matriz de cada wavelet.

Tabela 7: Dados de entrada e saída

	Mensal	Semestral	Anual
Tamanho do arquivo de entrada	1.555,4 MB		
Quantidade de registros de entrada	15.176.868		
Tamanho do arquivo de saída	3.788,8 MB	7.270,4 MB	8.089,6 MB
Quantidade de registros de saída	501.222	85.702	44.150
Quantidade de elementos da wavelet	448	4.914	11.315

Ao analisar a tabela 7, fica claro que o tamanho da saída da wavelet varia de acordo com o período. Períodos menores (mensal) tendem a ter mais registros e menos elementos da matriz da wavelet e períodos maiores (anual) possuem menos registros, mas um número muito maior de elementos da matriz da wavelet, fazendo com que até o tamanho em disco seja muito maior.

Os testes foram realizados levando em consideração cada período, comparando com o método de processamento tradicional (sequencial), *MapReduce* do

Apache Haddop e o *Apache Spark*.

Obviamente o tempo de processamento distribuído tende a ser menor que o sequencial, porém vale ressaltar que a comparação do processamento distribuído com o tradicional, realizada neste trabalho, leva em consideração que o algoritmo utilizado para a transformada de wavelets é o mesmo para ambos. As ferramentas utilizadas no processamento distribuído orquestram de forma autônoma os processos em execução, tratam das possíveis falhas de hardware e fornecem um ambiente computacional escalonável, abstraindo essas responsabilidades do usuário desenvolvedor.

A figura 18 mostra o comparativo de velocidade de processamento do período mensal entre as duas abordagens distribuídas (*Apache Hadoop* e *Apache Spark*) e a sequencial:

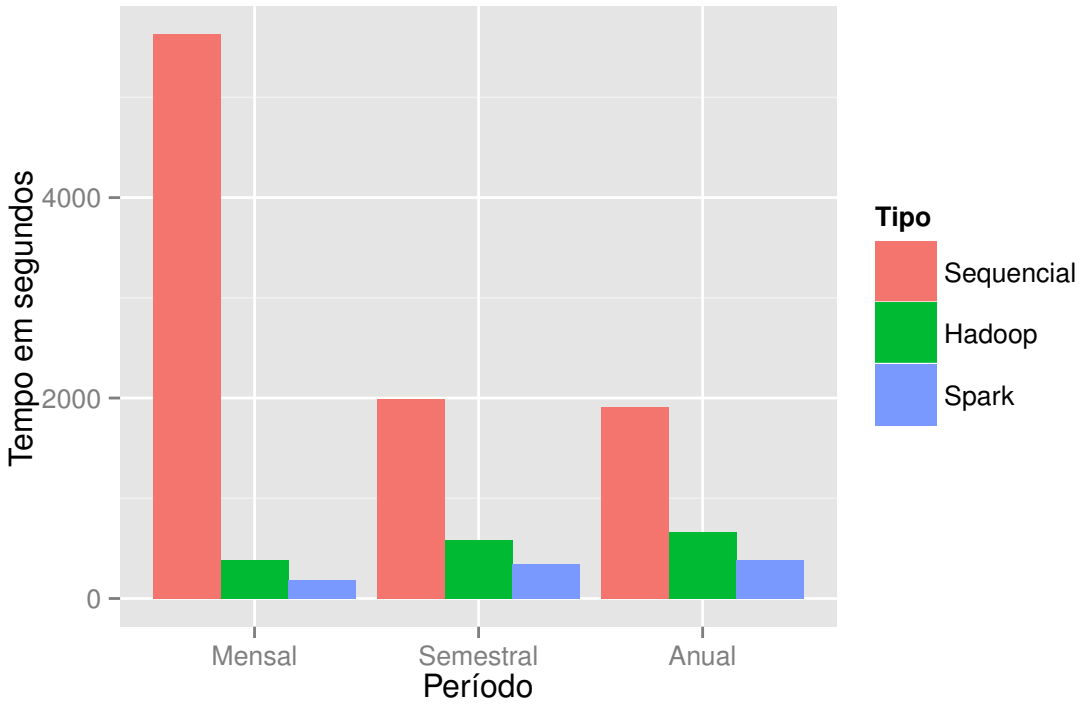


Figura 18: Comparativo sequencial x hadoop x spark.

O quadro 8 apresenta os resultados de velocidade de processamento dos períodos em segundos:

Quadro 8: Resultados por período (em segundos).

Período	**Sequencial**	**Apache Hadoop**	**Apache Spark**
Mensal	5628 s	382 s	181 s
Semestral	1992 s	587 s	346 s
Anual	1906 s	668 s	380 s

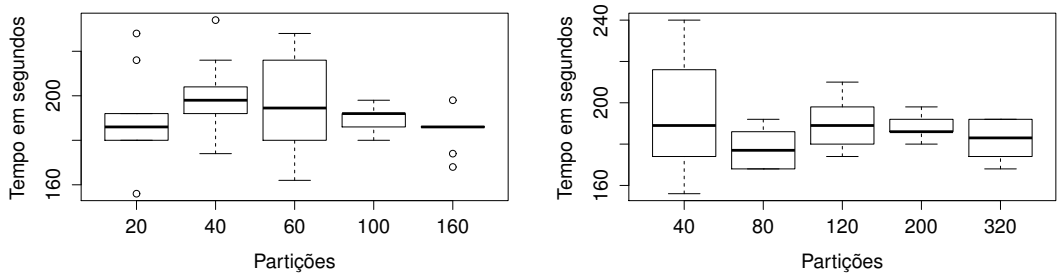
Com isso, é possível destacar que:

- No período mensal o *Apache Spark* foi aproximadamente 31 vezes mais rápido que o sequencial e 2,1 vezes mais rápido que o *Apache Hadoop*.
- No período semestral o *Apache Spark* foi aproximadamente 5,75 vezes mais rápido que o sequencial e 1,7 vez mais rápido que o *Apache Hadoop*.
- No período anual o *Apache Spark* foi aproximadamente 5 vezes mais rápido que o sequencial e 1,75 vez mais rápido que o *Apache Hadoop*.

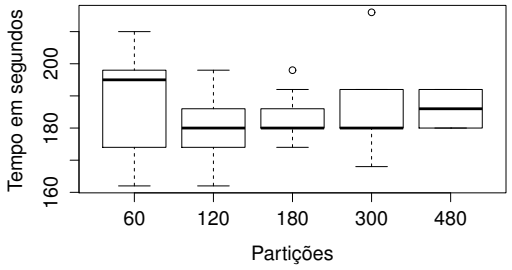
O ganho de tempo de processamento no *Apache Spark* em relação ao processamento sequencial é inversamente proporcional ao tamanho do período selecionado. Isso se deve ao fato de que períodos maiores necessitam de uma movimentação maior de dados no *cluster* na etapa *shuffle*, afinal o tamanho do arquivo de saída é muito maior nos períodos mais longos, como visto na tabela 7 e também pela menor paralelização do processamento, pois o cálculo é aplicado ao conjunto de dados do período (períodos maiores geram um conjunto maior de dados de precipitação). Outro fato a ser citado é que períodos mais longos possuem agrupamentos de precipitação maiores, fazendo com que o cálculo computacional seja mais dispendioso.

O próximo passo foi analisar o desempenho da transformada de wavelets nos diferentes períodos e configurações de *tuning*.

4.1 PERÍODO MENSAL



(a) (b)



(c)

Figura 19: Teste de performance com 20 (a), 40 (b) e 60 (c) *cores*.

No processamento com 20 *cores* (figura 19(a)), houve a presença de três *outliers*, provavelmente devido a performance de rede. A medida que foi aumentando o número de partições, o tempo de processamento elevou entre 40 e 100 partições e reduziu com 160 partições. A medida que é aumentado o número de partições, o tamanho dos arquivos de *shuffle* diminuem, porém a concorrência de processamento em cada máquina aumenta por causa do número reduzido de *cores*. O tempo de processamento torna-se mais estável com um número maior de partições, pois o fluxo de tarefas na rede é reduzido.

Os testes em 19(b) e 19(c) apresentaram uma maior dispersão dos resultados com baixa simetria no particionamento de 1/1 (core:partição). Houve também uma redução do tempo de processamento quando foi de 1/2, pois com mais partições na mesma máquina foi possível conseguir mais paralelismo devido a um número maior de *cores* por máquina, mas a medida que foi aumentando o número de partições, o comportamento foi como do teste 19(a) devido ao aumento de concorrência em cada nó envolvido no processo. A dispersão foi diminuindo

com o aumento de número de partições.

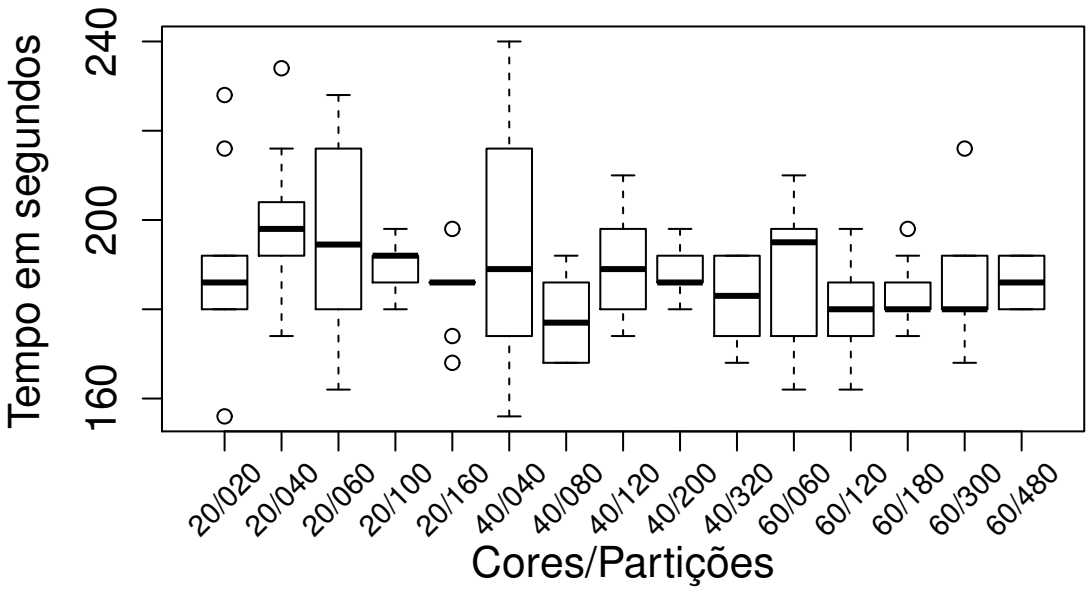


Figura 20: Todos os testes mensais.

A figura 20 exibe todos os testes mensais. Ao visualizar o comportamento dos testes como um todo, percebe-se uma menor dispersão no tempo de processamento quando aumenta-se o número de *cores* e partições.

No teste de ANOVA, o valor p foi de $0,0289$, indicando que existem diferenças significativa entre as amostras. No teste de Tukey, o valor p ajustado indica que os testes com maior diferença significativa foram 20/40 e 40/80 e ao analisar o gráfico da figura 20, é possível afirmar que o teste mais eficiente foi o 40/80 (proporção de 1/2).

O período mensal, entre todos os períodos testados, é o que apresentou o menor arquivo de saída do estágio 2 e portanto teve o menor tempo de processamento de todos. Processos em *batch* (lote) terão um tempo de processamento menor quando bem particionados, pois a estabilidade será maior, mesmo com a ligeira queda do tempo de processamento.

4.2 PERÍODO SEMESTRAL

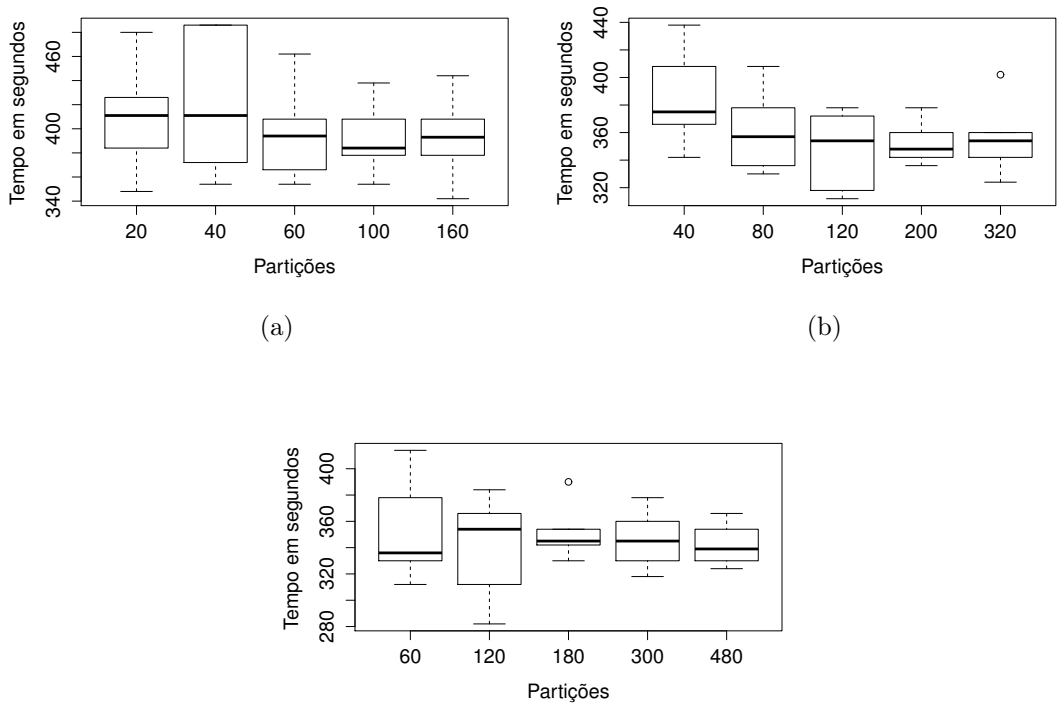


Figura 21: Teste de performance com 20 (a), 40 (b) e 60 (c) *cores*.

Em todos os testes foi perceptível a redução do tempo de processamento e menor dispersão ao aumentar o número de *cores* e partições. À medida que aumenta-se o número de partições, o tempo de processamento cai proporcionalmente (figuras 21(a), 21(b) e 21(c)).

O tempo de processamento é relativamente maior do que no período mensal, devido ao tamanho maior do arquivo de saída (aproximadamente 1,9 vezes) e ao decréscimo do paralelismo da execução do algoritmo, pois a quantidade de registros de saída é bem menor (aproximadamente 5,84 vezes).

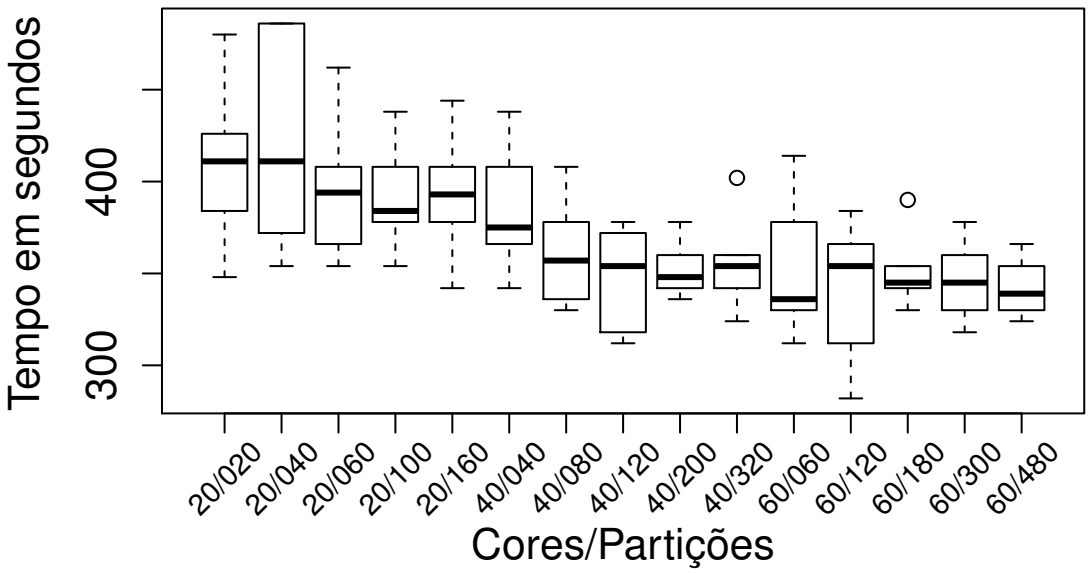
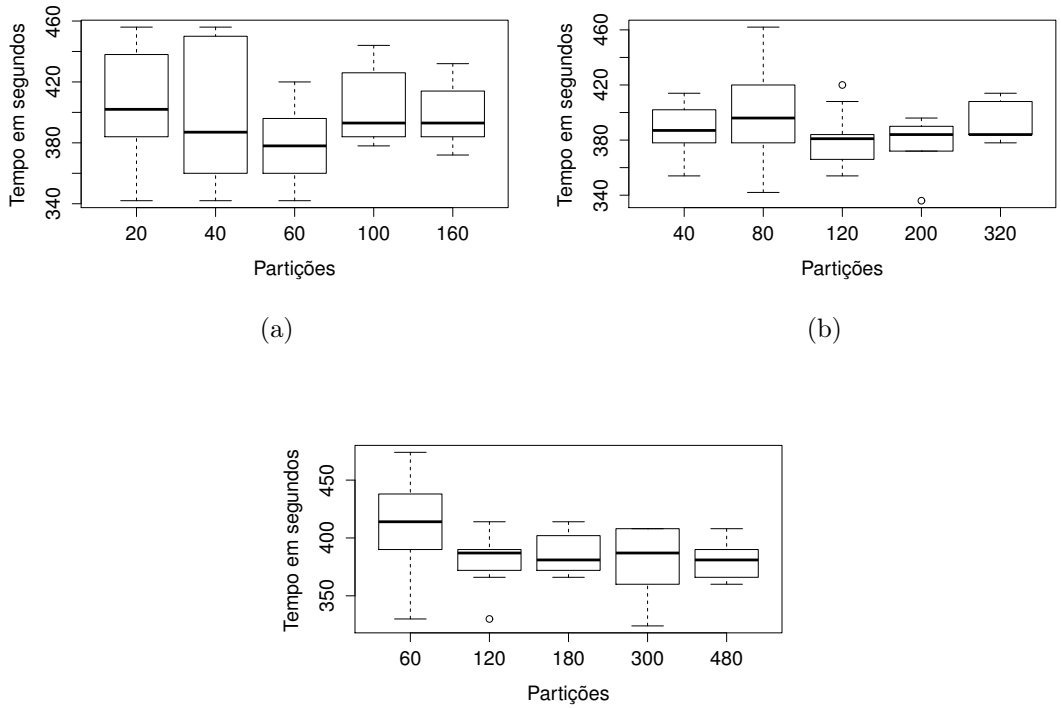


Figura 22: Todos os testes semestrais.

A figura 22 exibe todos os testes semestrais. O período semestral apresentou a maior queda no tempo de processamento ao aumentar o número de partições e *cores*. Isso se deve ao fato de que houve uma queda acentuada no número de registros na saída do arquivo (de 501.222 para 85.702), com um número não tão elevado de elementos da matriz da wavelet (4.914) quando comparado com o período anual (11.315), permitindo o melhor particionamento dos dados, pois foi possível uma menor fragmentação dos blocos de partição.

No teste de ANOVA, o valor p foi de $1,02^{-12}$, indicando que existem diferenças significativa entre as amostras. No teste de Tukey, o valor p ajustado indica que os testes com maior diferença significativa foram 20/20 e 40/120 e ao analisar o gráfico da figura 20, é possível afirmar que o teste mais eficiente foi o 40/120 (proporção de 1/3).

4.3 PERÍODO ANUAL



(a) (b)

(c)

Figura 23: Teste de performance com 20 (a), 40 (b) e 60 (c) *cores*.

O tempo de processamento do período anual apresentou o mesmo comportamento (figuras 23(a), 23(b) e 23(c)) em relação aos testes anteriores. Assim como o período semestral, a saída de dados do período anual é grande (proporção de 2 semestres para um ano, enquanto que o mês é de 6 para 1 semestre e 12 para um ano). Este foi o período que gerou o maior arquivo de saída dentre todos os outros períodos, porém com uma quantidade menor de registros por saída, fazendo com que o tempo de processamento seja mais alto que os outros períodos, pois o nível de paralelismo é reduzido.

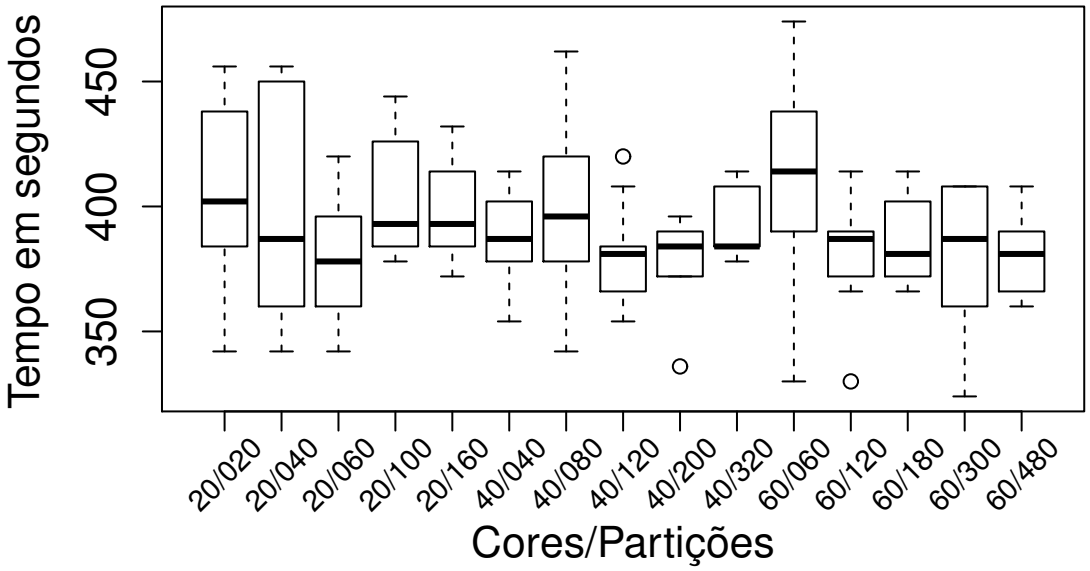


Figura 24: Todos os testes anuais.

A figura 24 exibe todos os testes anuais. A velocidade do processamento apresentou menor dispersibilidade para todos os cenários, não reduzindo o tempo de processamento na mesma proporção de aumento de partições e *cores*, como ocorrido no período semestral. A quantidade de registro gerados no período anual (44.150) é menor do que do período semestral (85.702), mas o número de elementos da matriz da wavelet é muito maior do que do semestral, fazendo com que os dados não sejam adequadamente particionados, como ocorrido no semestral, aumentando o tempo de *shuffle* a medida que aumenta o número de partições.

No teste de ANOVA, o valor p foi de 0.147, indicando que existem diferenças significativas entre as amostras. No teste de Tukey, o valor p ajustado indica que os testes com maior diferença significativa foram 60/60 e 20/60 e ao analisar o gráfico da figura 20, é possível afirmar que o teste mais eficiente foi o 20/60 (proporção de 1/3).

Capítulo 5

CONSIDERAÇÕES FINAIS

Este trabalho processou um algoritmo da transformada de wavelets em dados ambientais utilizando processamento distribuído e armazenou seus resultados no mesmo contexto. Nos melhores cenários, o ganho de velocidade foi de até 31 vezes em comparação com o processamento sequencial, demonstrando que a utilização deste tipo de processamento proporciona um aumento na eficiência do cálculo de grandes volumes de dados utilizando complexos algoritmos. Esta eficiência fica ainda mais evidenciada ao trabalhar em um contexto de dados na casa dos *terabytes*, contexto este presente ao trabalhar com imagens de satélites, cálculo de fractais, simulações de clima, método de covariância de vórtices turbulentos (*Eddy Covariance*) e em séries temporais de grandes períodos com alta precisão.

A transformada de wavelets gerou saídas de dados de diferentes tamanhos para os períodos mensal, semestral e anual, bem como diferentes níveis de paralelismo. Em conjunto com os dissemelhantes testes de *tuning*, será possível determinar quais configurações serão as mais eficientes para outros tipos de algoritmos.

Identificou-se comportamentos diferentes no processamento de dados, provando que o *tuning* do sistema distribuído é extremamente importante, influenciando positivamente na qualidade do processo e negativamente quando configurado de maneira errônea. Variáveis como tamanho do arquivo de saída, número de execuções (processos em lote) e nível de paralelização do cálculo requerem diferentes tipos de *tuning*. Para cálculos com maior paralelismo (mensal), a melhor proporção foi de 1/2 (*cores*/partições) e para os cálculos de menor paralelismo (semestral e anual), a melhor proporção foi de 1/3.

Os testes foram focados na performance do processamento do algoritmo de transformada de wavelets em diferentes ambientes e/ou situações para atingir

a excelência no que diz respeito a sintonia de um ambiente distribuído (*tuning*), que é diretamente relacionado ao desempenho do ambiente.

Como comprovado nos testes, o *tuning* varia de acordo com o tamanho do arquivo a ser processado, a quantidade de transformações envolvendo a etapa *map*, a quantidade de dados trafegado na etapa de *shuffle* e na saída do mesmo. Processos com uma quantidade elevada de dados de saída (Período Anual) tendem a ser mais velozes ao aumentar o número de partições, pois assim tem-se um maior nível de paralelismo.

O cálculo da transformada de wavelets gerou diferentes tamanhos de arquivos de saída. O menor período (mensal) gerou um tamanho de arquivo menor, porém com uma quantidade maior de registros, por outro lado, o maior período (anual) gerou um tamanho de arquivo maior, mas com uma menor quantidade de registros. Isto fez com que o nível de paralelismo fosse reduzido do menor para o maior período.

5.1 CONTRIBUIÇÕES

O trabalho contribuiu, de um modo geral, com a implementação de uma rotina mais eficiente de transformada de wavelets, através do processamento distribuído, com isso auxiliando a pesquisa e a análise de dados de séries temporais. Os testes em diferentes ambientes contribuiram também na configuração de um ambiente distribuído para o processo de outros tipos de dados e algoritmos de apoio a pesquisa ambiental.

Outras contribuições que podem ser citadas são:

- Configuração e disponibilização de um ecossistema *Apache Hadoop* para ser utilizado para outros tipos de cálculos paralelos ambientais.
- Melhor entendimento de como particionar os processos, para uma maior eficiência no processamento paralelo utilizando o *Apache Spark*.
- Ganhos de performance no processamento de dados, o que poderá ser ainda mais útil ao trabalhar com maiores volumes de dados (*terabytes*).
- Um ambiente escalável que permite várias configurações diferentes para determinados objetivos, desde cálculos de transformada de wavelets com diferentes granularidades temporais.

5.2 TRABALHOS FUTUROS

- Melhoria no algoritmo da transformada de wavelet, para possibilitar uma maior paralelização do cálculo e permitir processar wavelets de períodos maiores.
- Implementação de novos algoritmos para o cálculo de *Big Data*, envolvendo imagens de satélites, *Eddy Covariance*, fractais, entre outros.
- Integração entre estações micrometeorológicas com o ambiente distribuído através da biblioteca de *streaming* para a análise de dados em tempo real.
- Testes de performance com outros tipos de dados, ambiente e configurações, para auxiliar na implementação de novos algoritmos.
- Utilizar GPU's (Unidade de processamento gráfico) ao invés de CPU's para o processamento distribuído. Por terem sido desenvolvidas com o objetivo de calcular gráficos 3D complexos e por possuírem mais *cores* do que uma CPU comum, uma GPU pode realizar operações mais velozes do que uma CPU.
- Análise de tráfego de rede na etapa de *shuffle*, pois foram identificados *outliers* que até este momento estão relacionados fluxo de dados na rede interna.
- Este projeto também possibilitará a implementação e adaptação de outros algoritmos com ênfase no cálculo de dados ambientais, mesmo que não tenham sido mencionadas neste trabalho.
- Pesquisar sobre técnicas de processamento utilizadas no Programa de Pós-Graduação em Física Ambiental para a adaptabilidade de execução no *Apache Spark*.

REFERÊNCIAS

ALMASI, G. S.; GOTTLIEB, A. *Highly Parallel Computing*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1989. ISBN 0-8053-0177-1. Citado na página 8.

AMBARI, A. *Ambari* -. 2015. Disponível em: <https://ambari.apache.org/>. Citado na página 20.

BAKERY, M.; BUYYAZ, R. Cluster computing at a glance. v. 1, p. 3–47, 1999. Disponível em: <http://academic.csuohio.edu/yuc/hpc00/lect/chapter-B1.pdf>. Citado 2 vezes nas páginas X e 9.

BARRERA, D. F. Precipitation estimation with the hydroestimator technique: its validation against raingauge observations. 2005. Citado na página 7.

BIUDES MARCELO S. E MACHADO, N. G. Temperatura e albedo da superfÍcie por imagens tm landsat 5 em diferentes usos do solo no sudoeste da amazÔnia brasileira. v. 12, p. 169–183, 2015. Disponível em: <http://ojs.c3sl.ufpr.br/ojs/index.php/revistaabclima/article/view/40128>. Citado na página 10.

CAMPOS, G. F. d. S. *Adaptação de algoritmos de processamento de dados ambientais para o contexto de Big Data*. Dissertação, 2015. Citado 4 vezes nas páginas 4, 5, 12 e 28.

DANELICHEN, V. H. M.; BIUDES, M. S.; VELASQUE, M. C. S.; MACHADO, N. G.; GOMES, R. S. R.; VOURLITIS, G. L.; NOGUEIRA, J. S. Estimating of gross primary production in an amazon-cerrado transitional forest using MODIS and landsat imagery. v. 87, n. 3, p. 1545–1564. ISSN 0001-3765. Citado na página 10.

DEAN, E.; GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. 2004. Disponível em: <http://www.csee.umbc.edu/~nicholas/676/papers/MapReduce.pdf>. Citado 3 vezes nas páginas XII, 12 e 13.

DEAN, J.; GHEMAWAT, S. MapReduce: a flexible data processing tool. v. 53, n. 1, p. 72, 2010. ISSN 00010782. Disponível em: <http://portal.acm.org/citation.cfm?doid=1629175.1629198>. Citado 2 vezes nas páginas XII e 13.

FANG, W.; SHENG, V. S.; WEN, X.; PAN, W. Meteorological data analysis using MapReduce. v. 2014, p. 1–10, 2014. ISSN 2356-6140, 1537-744X. Disponível em: <http://www.hindawi.com/journals/tswj/2014/646497/>. Citado na página 4.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The google file system. In: *ACM SIGOPS operating systems review*. ACM, 2003. v. 37, p. 29–43. Disponível em: <http://dl.acm.org/citation.cfm?id=945450>. Citado 3 vezes nas páginas X, 14 e 15.

GOMES, R. S. R. *GeneSE : Um ambiente computacional para cálculo de balanço de energia da superfície utilizando imagens de satélites*. Tese, 2015. Citado 3 vezes nas páginas 8, 9 e 10.

GUARIENTI, G. S. S. *Desenvolvimento de uma técnica computacional de processamento espaço-temporal aplicada em séries de precipitação*. Dissertação, 2015. Citado 2 vezes nas páginas 5 e 6.

HADOOP, A. *Apache™ Hadoop®!* 2015. Disponível em: <https://hadoop.apache.org/>. Citado 6 vezes nas páginas X, 16, 17, 18, 19 e 20.

IBGE. *IBGE :: Instituto Brasileiro de Geografia e Estatística*. 2015. Disponível em: <http://www.ibge.gov.br/home/>. Citado 2 vezes nas páginas X e 28.

Ishwarappa; ANURADHA, J. A brief introduction on big data 5vs characteristics and hadoop technology. v. 48, p. 319–324, 2015. ISSN 18770509. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S1877050915006973>. Citado 3 vezes nas páginas X, 10 e 11.

KUMAR, R.; MARINOV, D.; PADUA, D.; PARTHASARATHY, M.; PATEL, S.; ROTH, D.; SNIR, M.; TORRELLAS, J. Parallel computing research at illinois the UPCRC agenda. 2008. Disponível em: <http://www.researchgate.net/profile/Maria_Garzaran/publication/265821281_Parallel_Computing_Research_at_Illinois_The_UPCRC_Agenda/links/54b67ac30cf2bd04be3214cf.pdf>. Citado na página 8.

KUMMEROW, C.; SIMPSON, J.; THIELE, O.; BARNES, W.; CHANG, A. T. C.; STOCKER, E.; ADLER, R. F.; HOU, A.; KAKAR, R.; WENTZ, F.; others. The status of the tropical rainfall measuring mission (TRMM) after two years in orbit. v. 39, n. 12, p. 1965–1982, 2000. Citado na página 7.

MACEDO, M. J. H. *Aplicações De Redes Neurais Artificiais e Satélite TRMM na Modelagem Chuva - Vazão da Bacia Hidrográfica do Rio Paraguaçu/BA*. Tese, 2013. Citado na página 7.

REICHARDT, K.; TIMM, L. C. *Solo, planta e atmosfera: conceitos, processos e aplicações*. [S.l.]: Editora Manole, 2004. Citado na página 6.

SANTOS, C. A. G.; FREIRE, P. K. d. M. M.; TORRENCE, C. A transformada wavelet e sua aplicação na análise de séries hidrológicas. v. 18, n. 3, p. 271–280, 2013. Disponível em: <https://www.abrh.org.br/sgcv3/UserFiles/Sumarios/dc22420acb362b36c1c143d3afc3a4cf_a73a4d5319dd5e0ca893196240745bb3.pdf>. Citado na página 5.

SEMCHEDINE, F.; BOUALLOUCHE-MEDJKOUNE, L.; AÏSSANI, D. Review: Task assignment policies in distributed server systems: A survey. *J. Netw. Comput. Appl.*, Academic Press Ltd., London, UK, UK, v. 34, n. 4, p. 1123–1130, jul. 2011. ISSN 1084-8045. Disponível em: <http://dx.doi.org/10.1016/j.jnca.2011.01.011>. Citado 2 vezes nas páginas X e 8.

SOKAL ROBERT, R.; ROHLF F, J. *Biometry: the principles and practice of statistics in biological research*. 3º edição. ed. [S.l.]: W. H. Freeman and Company, 1997. Citado na página 33.

SPARK, A. *Apache Spark™*. 2015. Disponível em: <http://spark.apache.org/>. Citado 8 vezes nas páginas X, 20, 21, 22, 23, 24, 25 e 29.

STORLIE, C.; SEXTON, J.; PAKIN, S.; LANG, M.; REICH, B.; RUST, W. Modeling and predicting power consumption of high performance computing jobs. 2014. Disponível em: <http://arxiv.org/abs/1412.5247>. Citado na página 9.

TOP500. *Top 500 Supercomputers*. 2015. Disponível em: <http://www.top500.org/>. Citado na página 9.

TORKESTANI, J. A. A new approach to the job scheduling problem in computational grids. v. 15, n. 3, p. 201–210, 2012. ISSN 1386-7857, 1573-7543. Disponível em: <http://link.springer.com/10.1007/s10586-011-0192-5>. Citado na página 10.

TRMM. 2015. Disponível em: <http://trmm.gsfc.nasa.gov/>. Citado 3 vezes nas páginas X, 6 e 7.

VALENTINI, G. L.; LASSONDE, W.; KHAN, S. U.; MIN-ALLAH, N.; MADANI, S. A.; LI, J.; ZHANG, L.; WANG, L.; GHANI, N.; KOLODZIEJ, J.; LI, H.; ZOMAYA, A. Y.; XU, C.-Z.; BALAJI, P.; VISHNU, A.; PINEL, F.; PECERO, J. E.; KLIAZOVICH, D.; BOUVRY, P. An overview of energy efficiency techniques in cluster computing systems. v. 16, n. 1, p. 3–15, 2011. ISSN 1386-7857, 1573-7543. Disponível em: <http://link.springer.com/10.1007/s10586-011-0171-x>. Citado na página 9.

VAQUERO, L. M.; RODERO-MERINO, L.; CACERES, J.; LINDNER, M. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 39, n. 1, p. 50–55, dez. 2008. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1496091.1496100>. Citado na página 10.

WARD ROY, C.; ROBINSON, M. *Principles of Hydrology*. [S.l.]: McGraw Hill Higher Education, 1999. Citado na página 6.

WOLFF, D. B.; MARKS, D. A.; AMITAI, E.; SILBERSTEIN, D. S.; FISHER, B. L.; TOKAY, A.; WANG, J.; PIPPITT, J. L. Ground validation for the tropical rainfall measuring mission (TRMM). v. 22, n. 4, p. 365–380, 2005. Disponível em: <http://journals.ametsoc.org/doi/abs/10.1175/JTECH1700.1>. Citado na página 7.

WOOLDRIDGE, J. M. *Introductory econometrics: a modern approach*. 5th ed. ed. [S.l.]: South-Western Cengage Learning, 2013. ISBN 978-1-111-53104-1. Citado na página 5.

YARN. *Apache Hadoop Yarn*. 2015. Disponível em: <http://hortonworks.com/blog/apache-hadoop-yarn-resourcemanager/>. Citado na página 18.

ZAHARIA, M. A. *An architecture for fast and general data processing on large clusters*. Tese, 2014. Disponível em: <http://digitalassets.lib.berkeley.edu/etd/ucb/text/Zaharia_berkeley_0028E_14121.pdf>. Citado 4 vezes nas páginas 20, 21, 22 e 23.

Apêndice 1: Implementação da Transformada de Wavelets

.

A.1 Exemplo do envio do pacote jar para o cluster YARN utilizando o comando spark-submit

```
sudo -u spark spark-submit --master yarn-cluster --num-
    ↪ executors 20 --executor-cores 3 --conf "spark.
    ↪ executor.memory=2g" --class main.WaveletsTRMM12 /
    ↪ home/spark/SparkWavelets-1.0-SNAPSHOT.jar '/user/
    ↪ spark/trmm.csv' 5 '/user/spark/outwaveletvector' '
    ↪ MensalE20C3' 2 500 1000
```

A.2 Objeto principal WaveletsTRMM

```
package main
import util.Wavelet
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.mllib.util
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.FileSystem
import org.apache.hadoop.fs.Path
import java.util.ArrayList
import scala.collection.JavaConversions._
```

```
import org.apache.spark.mllib.linalg.{ Vector, Vectors }
object WaveletsTRMM {
  def main(args: Array[String]) {
    if (args.length < 6) {
      System.err.println("Erro de parametro. Use:
        ↪ HdfsTextIn, Periodo, HdfsTextOut, Nome do
        ↪ Programa, Num repeticoes, (N) Particoes")
      System.exit(1)
    }
    //opcao 5 - mes
    //opcao 6 - semestre
    //opcao 7 - ano
    val opcao = args(1).toInt
    val repeticoes = args(4).toInt
    var periodo = 28
    opcao match {
      case 5 => periodo = 28
      case 6 => periodo = 182
      case 7 => periodo = 365
    }
    val sparkConf = new SparkConf().setAppName(args(3))
    val sc = new SparkContext(sparkConf)
    for (j <- 5 to args.length - 1) {
      for (i <- 1 to repeticoes) {
        val particoes = args(j).toInt
        val wavelet = sc.textFile(args(0))
        FileSystem.get(new Configuration()).delete(new
          ↪ Path(args(2)), true)
        wavelet.map(line => line.split(";")).map(vals => (
          ↪ vals(opcao).toInt, vals(3).toDouble)).
          ↪ groupByKey(particoes).mapValues(list =>
          ↪ createWavelet(list, periodo)).saveAsTextFile
          ↪ (args(2))
      }
    }
    sc.stop()
```

```
    def createWavelet(list: Iterable[Double], periodo:
      ↪ Integer): Vector = {
      val entrada = list.toArray

      var soma: Double = 0
      for (i <- 0 to entrada.size - 1) {
        soma += entrada(i)
      }
      if (entrada.size != periodo || soma <= 0) {
        return Vectors.dense(0)
      }
      var wavelet = new Wavelet(entrada)
      val result = wavelet.waveletTransform()
      val tamanho = result.length * result(0).length
      var resultado2: Array[Double] = new Array[Double](
        ↪ tamanho)
      var indice = 0
      var matrix = new ArrayList[java.lang.Double]()
      for (i <- 0 to result.length - 1) {
        for (j <- 0 to result(i).length - 1) {
          resultado2(indice) = result(i)(j)
          indice += 1
        }
      }
      return Vectors.dense(resultado2)
    }
  }
}
```

A.3 Classe Wavelet responsável pela transformada

```
package util
import org.jtransforms.fft.DoubleFFT_1D
import scala.collection.JavaConversions._
class Wavelet {
  private var data: Array[Double] = _
  private var mean: Double = _
  private var std: Double = _

  def this(dados: Array[Double]) {
    this()
    this.data = dados
    mean = getMean(data)
    std = getStd(data)
  }

  def waveletTransform(): Array[Array[Double]] = {
    val normal = createNormal()
    val w0 = 6
    val dt = 0.25
    val dj = 0.25
    val s0 = 2 * dt
    val N = normal.length
    var J = (Math.log(N * dt / s0) / Math.log(2)) / dj
    J = fix(J)
    val j = arange(0, J.toInt + 1)
    val s = calc_s(s0, j, dj)
    var k = arange(0, N / 2)
    val wn = calc_wn(k, N * dt)
    k = arange(N / 2, N)
    val wn2 = calc_wn2(k, N, dt)
    val wn_con = concatenate(wn, wn2)
    val hwn = heaviside(wn_con)
    val fft = new DoubleFFT_1D(normal.length)
    val fourier = Array.ofDim[Double](normal.length * 2)
    for (i <- 0 until normal.length) {
```

```
      fourier(2 * i) = normal(i)
    }
    fft.complexForward(fourier)
    val wave = Array.ofDim[Double](J.toInt + 1, N)
    var wavefilha: Array[Double] = null
    for (i <- 0 until s.length) {
      wavefilha = Array.ofDim[Double](hwn.length)
      for (l <- 0 until wavefilha.length) {
        wavefilha(l) = Math.sqrt(2 * Math.PI * s(i) / dt)
          ↪ * (Math.pow(Math.PI, -0.25)) *
          hwn(l) *
          Math.exp(-Math.pow((s(i) * wn_con(l) - w0), 2) /
            ↪ 2)
      }
      val fwf2 = fwf(fourier, wavefilha)
      fft.complexInverse(fwf2, true)
      for (l <- 0 until wavefilha.length) {
        val aux = Math.hypot(fwf2(2 * l), fwf2(2 * l + 1))
        wave(i)(l) = Math.pow(aux, 2)
      }
    }
    wave
  }

  private def fwf(fourier: Array[Double], wavefilha: Array
    ↪ [Double]): Array[Double] = {
    val calc = Array.ofDim[Double](fourier.length)
    var count = 0
    for (i <- 0 until fourier.length) {
      calc(i) = fourier(i) * wavefilha(count)
      if (i % 2 != 0) {
        count += 1
      }
    }
    calc
  }
```

```
private def heaviside(wn: Array[Double]): Array[Double]
  ↪ = {
  val calc = Array.ofDim[Double](wn.length)
  for (i <- 0 until calc.length) {
    calc(i) = if (wn(i) < 0) 0.0 else if (wn(i) > 0) 1.0
      ↪  else 1 / 2
  }
  calc
}

private def concatenate(a: Array[Double], b: Array[
  ↪ Double]): Array[Double] = {
  val calc = Array.ofDim[Double](a.length + b.length)
  for (i <- 0 until a.length) {
    calc(i) = a(i)
  }
  val tam_a = a.length
  for (i <- tam_a until calc.length) {
    calc(i) = b(i - tam_a)
  }
  calc
}

private def calc_wn2(k: Array[Int], N: Int, dt: Double):
  ↪  Array[Double] = {
  val calc = Array.ofDim[Double](k.length)
  for (i <- 0 until calc.length) {
    calc(i) = -(2 * Math.PI * (N + k(i) + 1)) / (N * dt)
  }
  calc
}

private def calc_wn(k: Array[Int], Ndt: Double): Array[
  ↪ Double] = {
  val calc = Array.ofDim[Double](k.length)
  for (i <- 0 until calc.length) {
    calc(i) = (2 * Math.PI * k(i)) / Ndt
```

```
    }
    calc
  }

  private def calc_s(s0: Double, j: Array[Int], dj: Double
    ↪ ): Array[Double] = {
    val calc = Array.ofDim[Double](j.length)
    for (i <- 0 until calc.length) {
      calc(i) = s0 * Math.pow(2, j(i) * dj)
    }
    calc
  }

  private def arange(inicio: Int, fim: Int): Array[Int] =
    ↪ {
    val tam = fim - inicio
    val saida = Array.ofDim[Int](tam)
    for (i <- 0 until tam) {
      saida(i) = i + inicio
    }
    saida
  }

  private def createNormal(): Array[Double] = {
    val normal = Array.ofDim[Double](data.size)
    for (i <- 0 until normal.length) {
      normal(i) = (data(i) - mean) / std
    }
    normal
  }

  private def getMean(dados: Array[Double]): Double = {
    val total = dados.size
    var sum: Double = 0
    for (double1 <- dados) {
      sum += double1
    }
```

```
    sum / total
  }

  private def getStd(dados: Array[Double]): Double = {
    val total = dados.size
    val mean = getMean(dados)
    var sum = 0.0
    var aux: Double = 0.0
    for (double1 <- dados) {
      aux = double1 - mean
      sum += Math.pow(aux, 2)
    }
    Math.sqrt(sum / total)
  }

  private def fix(a: Double): Double = {
    if (a < 0) {
      Math.ceil(a)
    } else {
      Math.floor(a)
    }
  }
}
```