EXTENSÃO DE UM AMBIENTE DE COMPUTAÇÃO DE ALTO DESEMPENHO PARA O PROCESSAMENTO DE DADOS MASSIVOS

LUCAS MIGUEL SIMÕES PONCE

EXTENSÃO DE UM AMBIENTE DE COMPUTAÇÃO DE ALTO DESEMPENHO PARA O PROCESSAMENTO DE DADOS MASSIVOS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: DORGIVAL OLAVO GUEDES NETO

Belo Horizonte

Setembro de 2018

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Ponce, Lucas Miguel Simões

P792e Extensão de um ambiente de computação de alto desempenho para o processamento de dados massivos / Lucas Miguel Simões Ponce. — Belo Horizonte, 2018.
xx, 67 f.: il.; 29 cm.

Dissertação (mestrado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Dorgival Olavo Guedes Neto.

1. Computação – Teses. 2. Computação de alto desempenho. 3. Processamento de dados. 4. Big data. I. Orientador. II. Título.

CDU 519.6*24(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Extensão de um ambiente de computação de alto desempenho para o processamento de dados massivos

LUCAS MIGUEL SIMÕES PONCE

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. DORGIVAL OLAVO GUEDES NETO - Orientador
Departamento de Ciência da Computação - UFMG

PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG

PROF. PHILIPPE OLIVIER ALEXANDRE NAVAUX
Instituto de Informática - UFRGS

Belo Horizonte, 4 de setembro de 2018.

Agradecimentos

Agradeço a minha grande família: pais, avós, tios e primos, sempre presentes em minha vida. Tenho todos com muito carinho e tenho certeza que vocês comemoram comigo essa nova etapa na minha vida. Agradeço aos meus pais e avós pelo carinho e determinação em me proporcionar as condições propícias para que eu pudesse perseguir minhas próprias ambições durante a minha criação. Gostaria de agradecer também à minha noiva, Júlia, pelo carinho e apoio durante esse percurso. Ainda acho que você vai ouvir muito sobre esses meus assuntos que você não entende nada mas que, mesmo assim, me ajuda escutando ;P

Aos meus colegas e amigos: Tiago, pelo companheirismo e pelas série de desafios que traçamos juntos; Gustavo, pelas trocas de ideias e bate-papos que tivemos no laboratório *Speed*; e ao Walter, pelas inúmeras dicas e ajudas durante esse período, um dia chego lá.

Agradeço em especial ao meu orientador, professor Dorgival Guedes, pela confiança depositada e pelos importantes conselhos, pessoais e acadêmicos, durante essa etapa. Sua orientação não só foi fundamental para o término desse trabalho como também contribuiu para o meu caráter científico. E também ao professor Wagner Meira Jr., que como coordenador do projeto EUBra-BIGSEA que estive inserido durante o Mestrado, sempre mostrou interesse e suporte em minhas contribuições.

"Se você quer ir rápido, vá sozinho.
Se quiser ir longe, vá acompanhado"
(Provérbio africano)

Resumo

A computação de alto desempenho (HPC) e o processamento de dados massivos (*big data*) são duas tendências em sistemas de computação para lidar com problemas complexos ou com grande quantidade de dados. Sistemas desenvolvidos com cada uma dessas tendências em mente se especializaram em conjuntos de problemas específicos, com abordagens únicas. Atualmente, tais sistemas estão começando a convergir, muitas vezes forçados pela mistura de domínios de um determinado problema. Por exemplo, a comunidade de HPC tem avaliado a aplicabilidade do COMP Superscalar (COMPSs), um modelo de programação paralela e distribuída originado do mundo de HPC, no contexto de *big data*. Para isso, é necessário integrá-lo a novas funcionalidades usualmente relacionadas a ambientes de *big data*. Este trabalho apresenta nossa contribuição nesse caminho de convergência a fim de processar dados massivos integrando o COMPSs ao HDFS, um dos sistemas de arquivos distribuídos mais utilizado em *big data*, e ao Lemonade, uma ferramenta de análise e mineração de dados desenvolvida na Universidade Federal de Minas Gerais (UFMG). Os resultados mostram que a integração com o HDFS beneficia o COMPSs não só pela abstração de programação, que simplifica o acesso aos dados, mas também pelo aumento de desempenho em execuções que precisam ler grandes volumes de dados, devido à reorganização da transferência de dados pela rede. Além disso, a integração com o Lemonade facilita o uso do COMPSs e sua popularização na área de Ciência dos Dados, fornecendo boas implementações de algoritmos e operações para especialistas do domínio de dados que desejam desenvolver e executar aplicações COMPSs com um nível mais alto de abstração.

Abstract

High performance computing (HPC) and massive data processing (big data) are two trends in computing systems that allow us to deal with complex or large data problems. Systems developed with each of these trends in mind have specialized in sets of specific problems with unique approaches. Currently, such systems are beginning to converge, often forced by the mix of domains in a given problem. For example, the HPC community is evaluating the applicability of COMP Superscalar (COMPSs), a parallel and distributed programming model originated in the HPC world, in the context of big data. For that, is necessary to integrate COMPSs with new functionalities usually found in big data environments. This paper presents our contribution for that convergence path, in order to process massive data in COMPS by integrating it with HDFS, one of the most widely used distributed file systems for big data, and with Lemonade, a data mining and analysis tool developed at Universidade Federal de Minas Gerais (UFMG). The results show that integration with HDFS benefits COMPSs not only by adding a new programming abstraction, which simplifies access to data, but also by improving performance in executions that need to read large volumes of data, due to the reorganization of the data transfer patterns in the network. In addition, Lemonade integration facilitates the use of COMPSs and its popularisation in the Data Science area by providing good implementations of algorithms and operations for data domain specialists who wish to develop and run COMPS applications with a higher level of abstraction.

Lista de Figuras

2.1 Exemplo de um contador de palavras em PyCOMPSs 6
2.2 Esquema de diferença entre blocos físicos e lógicos 12
2.3 Esquema da aplicação Workflow criada a partir do Lemonade 14

3.1 Exemplo de um contador de palavras em PyCOMPSs utilizando o HDFS . 21

5.1 Localidade dos dados: tempo de execução do Grep em função do tamanho do arquivo de entrada 38
5.2 Leitura de arquivos no HDFS: *traces* da transferência de dados durante a execução do WordCount 41
5.3 Leitura de arquivos no HDFS: tempo de execução do Grep (PyCOMPSs) em função do tamanho do arquivo 43

A.1 Testes visuais para verificar as premissas do projeto fatorial 61
A.2 Testes visuais das regressões: distribuição normal dos erros 62
A.3 Testes visuais das regressões: independências dos erros 63

Lista de Tabelas

2.1 Argumentos suportados pela anotação PyCOMPSs 7

4.1 Operações e algoritmos implementados em COMPSs 27

5.1 Planejamento dos experimentos 34
5.2 Configurações dos parâmetros para os projetos fatoriais $2^3 10$ 35
5.3 Resultado do projeto fatorial $2^3 10$ 36
5.4 Localidade dos dados: análise das regressões lineares ($y = \alpha x + \beta$) 37
5.5 Leitura de arquivos no HDFS: relação das configuração dos algoritmos . . 39
5.6 Leitura de arquivos no HDFS: teste pareado Z entre versões em Java (tempo em segundos) 39
5.7 Leitura de arquivos no HDFS: teste pareado Z entre versões em Python (tempo em segundos) 40
5.8 Leitura de arquivos no HDFS: análise das regressões lineares ($y = \alpha x + \beta$) . 44
5.9 Escrita de dados no HDFS: teste pareado Z entre versões em Python (tempos em segundos) 44
5.10 Impacto do Lemonade: teste pareado Z (tempo em segundos) 45

B.1 Métodos da API de integração HDFS 66
B.2 Algoritmos/operações COMPS suportados pelo Lemonade 67

Sumário

Agradecimentos **vii**

Resumo **xi**

Abstract **xiii**

Lista de Figuras **xv**

Lista de Tabelas **xvii**

1 Introdução **1**

1.1 Objetivos 3

1.2 Contribuições 3

1.3 Organização 4

2 Referencial Teórico **5**

2.1 COMPSs 5

2.2 HDFS 10

2.3 Lemonade 13

2.4 Trabalhos Relacionados 17

3 Integração entre o HDFS e o COMPSs **19**

3.1 Abstração de dados 20

3.1.1 Leitura dos dados 20

3.1.2 Escrita dos dados 21

3.2 Abstração com localidade dos dados 22

3.3 Comunicação com o HDFS 23

4 Suporte do Lemonade ao COMPSs **25**

4.1 Algoritmos e Operações 26

4.2 Integração e geração de código 28

5 Avaliação experimental **31**
5.1 Planejamento dos experimentos 32
5.2 Impacto da utilização do HDFS 35
5.2.1 Estudo do impacto da localidade dos dados 35
5.2.2 Estudo do impacto do HDFS na leitura de arquivos 38
5.2.3 Estudo do impacto da escrita de arquivos 44
5.3 Impacto da integração com o Lemonade 45
5.4 Aplicabilidade da solução 46

6 Conclusão **49**
6.1 Trabalhos futuros 50

Referências Bibliográficas **53**

Glossário **57**

Apêndice A Premissas do projeto fatorial e das regressões **59**
A.1 Projeto fatorial 2^kr 60
A.2 Regressão 60

B Funções implementadas **65**
B.1 API de integração com o HDFS 65
B.2 Algoritmos e operações suportadas no Lemonade 65

Capítulo 1

Introdução

O International Exascale Software Project (IESP), um grupo de computação avançada formado para capacitar ciência e engenharia de alto desempenho e dados intensivos, prevê uma série de desafios para os próximos anos em diversas áreas da ciência que não só exigirão sistemas de grande porte que quebrem a barreira do *exaflop*[1], chamado *Exascale*, mas também sejam capazes de processar grandes quantidades de dados, isto é, *big-data*. Na Biologia, pode ser utilizado na modelagem e simulação de sistemas biológicos em múltiplas escalas para uma melhor compreensão de processos como o crescimento celular e o metabolismo. Na Engenharia Aeroespacial, podem ser realizadas simulações de alta fidelidade para reduzir o número de experiências custosas. Nas Ciências Sociais, com cidades inteligentes que visam melhorar a vida dos cidadãos em diferentes níveis como informações de transporte [Reed & Dongarra, 2015].

Nesse contexto, a computação paralela e distribuída tem se mostrado essencial para lidar com esses problemas, seja por problemas complexos ou pelo processamento de grandes volumes de dados. A computação de alto desempenho (HPC) e o processamento de dados massivos (*big-data*) são duas tendências nesses sistemas de computação que estão começando a convergir. As aplicações de alto desempenho em computação são aquelas que geralmente se valem de hardware de alto nível de paralelismo e alto desempenho, incluindo redes de baixa latência, para processar dados normalmente regulares com algoritmos científicos. Por outro lado, cenários convencionalmente denominados de *big data*, que buscam processar grandes volumes de dados de diversos tipos, normalmente não estruturados, utilizam hardware convencional e se valem fortemente de técnicas de paralelismo de dados. Nesse caso, os dados podem ser processados como fluxos individuais e analisados coletivamente em *stream* ou em lote, para a descoberta de conhecimento, sendo a mineração em *big data* uma das

[1] 10^{18} operações por segundo

tarefas chaves em muitos domínios da ciência [Kamburugamuve et al., 2017].

Considerando suas similaridades, começam a surgir propostas para a convergência dessas duas áreas em sistemas de computação [Fox et al., 2016; Reed & Dongarra, 2015; Tejedor et al., 2017]. Ambientes de HPC em geral oferecem interfaces mais adequadas para o processamento de dados regulares, algoritmos científicos baseados em modelos de *bag-of-tasks* ou cálculo matricial. Apesar de seu desempenho nesses cenários, é muitas vezes trabalhoso expressar neles aplicações que manipulam dados irregulares e estruturas de dados complexas. Por outro lado, ambientes de *big data* oferecem boas soluções para tratar tais tipos de dados, bem como para facilitar o desenvolvimento de aplicações pelos especialistas no domínio da aplicação.

No entanto, com tanta diversidade é difícil que uma solução criada em um sistema que se adeque a todos os demais pertencentes a esse nicho específico. Além disso, o nível de integração entre os diferentes sistemas é baixo, criando uma forte barreira para lidar com cenários que combinem *big data* e HPC. Uma outra característica é que em muitos casos o desenvolvimento de uma solução nesses sistemas é usualmente feito pelo especialista que analisará os resultados, mas sim por um cientista de dados. Tais especialistas raramente possuem experiência em programação paralela [Rocha et al., 2016], criando a necessidade de desenvolver plataformas que possam explorar o paralelismo intrínseco à aplicação. Modelos com abstrações simplificadas que favorecem a generalização e produtividade em relação à eficiência da computação têm sido amplamente adotados em ambos os segmentos de pesquisa e da indústria [Hall et al., 2013]. Alguns desses exemplos conhecidos atualmente são o Apache Spark [Zaharia et al., 2012] e COMP Superscalar (COMPSs) [Tejedor & Badia, 2008]. De um lado, um dos *frameworks* de *big data* mais utilizados atualmente, que busca oferecer fácil compreensão para gerenciar e processar aplicações com uma variedade de conjuntos de dados de diversas naturezas; de outro, um *framework* já bem consolidado no mundo de HPC, que busca explorar o paralelismo sem nenhuma nova sintaxe envolvida.

Apesar do Spark prover camadas de abstração para muitos detalhes na programação em sistemas paralelos e distribuídos, ainda assim, é necessário um grande conhecimento de programação e de um aprendizado inicial para entender a nova sintaxe e operadores envolvidos [Agneeswaran et al., 2013]. Um trabalho recente [Conejero et al., 2017] comparou o desempenho e outros fatores qualitativos do modelo de programação de ambos os *frameworks* e, apesar do trabalho indicar o melhor desempenho do COMPSs nos cenários testados, dois dos principais pontos positivos do Spark mencionados foram: a ampla gama das bibliotecas que estão disponíveis em torno desse ambiente, devido à sua popularidade (p.ex., MLlib, GrapX, Streaming e SparkSQL), e uma maior interoperabilidade com outras ferramentas de processamento

de dados massivos[2]. Além disso, os testes realizados utilizaram uma arquitetura de discos compartilhados, uma infraestrutura característica de ambientes de HPC, no qual são usadas várias máquinas com CPUs e memórias RAM independentes, mas armazenam dados em uma série de discos compartilhados entre as máquinas, conectadas através de uma rede rápida (no caso desses teste, foi utilizado uma rede InfiniBand). É comum em cenários de *big data* a utilização de armazenamento de dados distribuídos (*horizontal scaling*) em contraste à arquitetura com discos compartilhados, pois evita a contenção e a sobrecarga do limite de bloqueio da escalabilidade em casos onde não é possível garantir uma rede rápida.

1.1 Objetivos

Nesse contexto, propomos a extensão do COMPSs em duas direções: primeiro, adicionando o suporte ao HDFS, o sistema de arquivos distribuído mais usado para *big data*; segundo, integrando-o a um ambiente de desenvolvimento de aplicações de processamento de dados massivos que reduz a necessidade do usuário conhecer os detalhes de uma linguagem de programação para produzir aplicações *big data*. Com o suporte ao HDFS, pretendemos facilitar a utilização do COMPSs com grandes volumes de dados não estruturados comumente usados e que costumam estar disponíveis em repositórios com HDFS. Com o uso de um ambiente de desenvolvimento visual, pretendemos tornar o COMPSs acessível a um número maior de usuários que normalmente não dominam os detalhes de uma linguagem de programação paralela. Para esse fim, adotamos o ambiente Lemonade (do inglês *Live Environment for Mining Of Non-trivial Amount of Data Everywhere*), uma ferramenta de análise e mineração de dados desenvolvida na Universidade Federal de Minas Gerais (UFMG) [Santos et al., 2017].

1.2 Contribuições

Como principais contribuições deste trabalho, destacamos:

1. **Integração ao HDFS**: Apresentamos uma API de integração entre o COMPSs o HDFS para as linguagens Java e Python que oferece uma abstração dos dados semelhante ao modo convencional de se trabalhar com arquivos em COMPSs. Apresentamos também um estudo experimental avaliando a API desenvolvida.

[2]Exemplo de ferramentas do ecossistema Hadoop: `https://hadoopecosystemtable.github.io`

2. **Criação de uma biblioteca COMPSs**: Apresentamos também uma bibliotecas de funções COMPSs implementadas em Python. Tal biblioteca pode ser utilizada como um módulo externo para auxiliar desenvolvedores e estudantes em COMPSs ajudando a popularização do COMPSs no mundo de *big data*.

3. **Suporte ao Lemonade**: Por fim, apresentamos nossa adição do suporte do Lemonade ao COMPSs. Dessa forma, aplicações COMPSs podem ser criadas e executadas por meio de uma simples conexão visual de blocos de operações.

1.3 Organização

O texto desta dissertação está organizado da forma que segue. O capítulo 2 apresenta os principais conceitos relacionados necessários para o desenvolvimento deste trabalho, bem como os principais trabalhos de pesquisa relacionados. O capítulo 3 apresenta a API de integração do HDFS ao COMPSs, descrevendo a abstração de dados, as opções de leitura e de escrita de dados além de um exemplo do que é necessário alterar em um código existente para poder utilizá-la. No capítulo 4 apresentamos o suporte do Lemonade ao COMPSs, abordando as decisões tomadas e os algoritmos e operações implementados. Os resultados experimentais de ambas as integrações são avaliadas no capítulo 5 e, por fim, no capítulo 6 apresentamos as conclusões e discutimos os trabalhos futuros.

Capítulo 2

Referencial Teórico

Para melhor entender este trabalho, precisamos primeiramente apresentar alguns conceitos relacionados. Nas seções seguintes apresentaremos: o COMPSs e o seu modelo de programação; o HDFS, junto com o seu sistema de distribuição de arquivos; o Lemonade e os seus componentes; e por fim, os trabalhos relacionados.

2.1 COMPSs

O COMPSs, desenvolvido pelo Centro de Supercomputação de Barcelona (BSC), é um modelo de programação que está em constante desenvolvimento com versões atualizadas a cada seis meses, trazendo novas funcionalidades e/ou suportes [Lezzi et al., 2011; Tejedor et al., 2017]. Atualmente na versão 2.2, suporta nativamente aplicações Java, que chamaremos de Java COMPSs, mas também provê *bindings* para Python (PyCOMPSs) e C/C++. As aplicações em COMPSs são escritas de acordo com o paradigma sequencial, com a adição de algumas anotações no código que são usadas para informar que um dado método no código pode ser executado em paralelo. Por convenção, os métodos que possuem tais anotações são chamados de tarefas, para não serem confundidos com métodos da aplicação que não são paralelizáveis. No caso de Java e C++, essas anotações são semelhantes a uma interface de método, sendo necessário informar, por exemplo, os tipos de dado dos parâmetros de entrada e de saída. Já no caso do Python, essa anotação nada mais é que um *decorator* iniciado com "*@task*"em cima de um método a ser paralelizado. A partir dessas anotações, o COMPSs é capaz de identificar as dependências entre tarefas e assim, explorar o paralelismo inerente entre tarefas em tempo de execução, respeitando o comportamento original determinado pela aplicação.

A figura 2.1 contém um exemplo de uma implementação de um algoritmo para

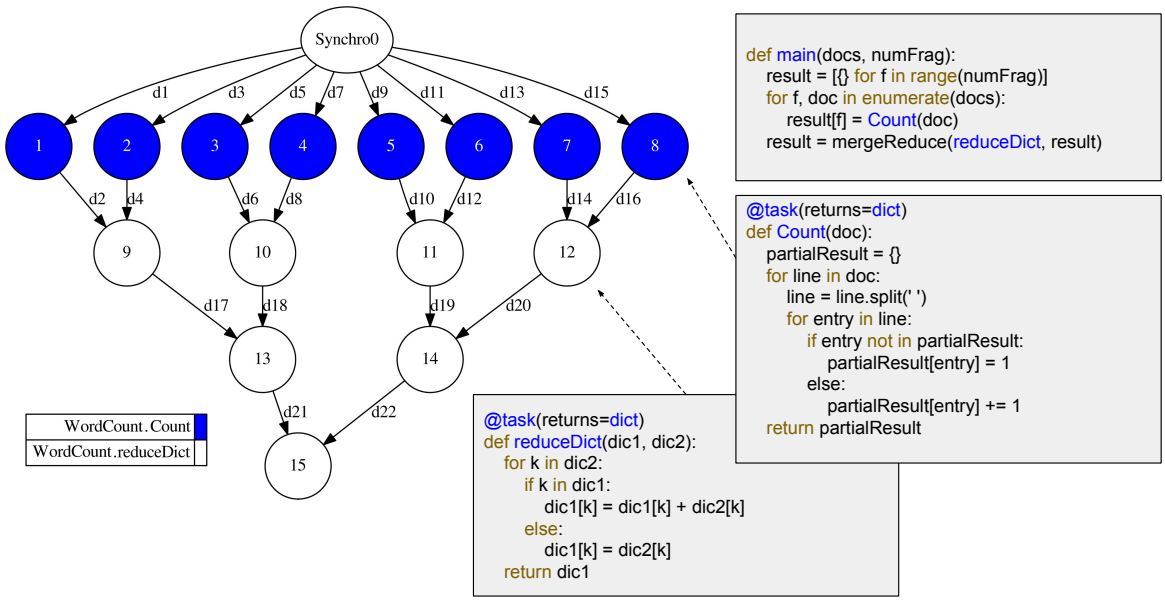


Figura 2.1: Exemplo de um contador de palavras em PyCOMPSs

contagem de palavras (WordCount) em COMPSs na linguagem Python. Pode ser visto que o código da aplicação não envolve nenhuma nova sintaxe, diferentemente de outros modelos de programação distribuído como Spark. A ideia da aplicação é realizar a contagem das palavras em cada fragmento do arquivo paralelamente para posteriormente realizar uma etapa de redução, agregando os resultados parciais. Nesse exemplo, as anotações das duas tarefas são iguais e informam que tais métodos são paralelizáveis, que o resultado de ambas são dicionários Python e que seus parâmetros de entrada são apenas de leitura (configuração padrão). Analisando o código da figura, podemos realizar uma divisão estrutural dessa aplicação em três partes:

- O código principal (main) - executado sequencialmente e que contém as chamadas para as tarefas que serão executados pelo COMPS como paralelas e assíncronas;
- O código dos métodos remotos (Count e reduceDict) - contêm as implementações das tarefas;
- As anotações (interface ou *decorator*) - declara os métodos a serem executados como tarefas, juntamente com as informações dos metadados necessários para a orquestração das tarefas corretamente.

Para fins de explicação, uma outra forma de implementação dessa aplicação de conta palavras seria alterar a anotação da tarefa *reduceDict* para "*@task(dic1=INOUT)*". Dessa forma, estaríamos informando ao COMPSs que *dic1* é

tanto um parâmetro de entrada como de saída, isto é, tal tarefa produzirá um resultado na variável *dic1*. Uma desvantagem dessa abordagem é que, para essa aplicação, a execução da tarefa *reduceDict* não seria paralelizada pois haveria uma dependência entre todos as tarefas que tivessem como parâmetro de entrada a variável *dic1*. Uma anotação pode ter múltiplos argumentos; a tabela 2.1 contém uma amostra, como exemplo, de alguns argumentos suportados pelo PyCOMPSs na definição de tarefas. Além dessas, é possível criar restrições sobre tarefas, por exemplo, número de cores disponível, quantidade de memória requerida, ou a exigência de usar um determinado computador com uma aplicação/biblioteca instalada [COMP Superscalar, 2017].

Tabela 2.1: Argumentos suportados pela anotação PyCOMPSs

Argumento	Valor
nome do parâmetro	IN: parâmetro somente de leitura (padrão) INOUT: parâmetro de leitura e escrita OUT: parâmetro somente de escrita FILE: o arquivo será apenas lido (padrão) FILE_INOUT: o arquivo será lido e escrito FILE_OUT: o arquivo será apenas escrito.
returns	int (para tipos integer e boolean), long, float, str, dict, list, tuple e instância de uma classe
priority	True ou False (padrão)

A arquitetura do COMPSs é baseada em um componente central (*master*), que executa o código principal da aplicação, e um conjunto dos nós (*worker*) que executam as tarefas. Esses nós podem ser executados em infraestruturas distribuídas como, em *clusters* físicos, máquinas virtuais dinamicamente instanciadas ou em *containers*. Para isso, é necessário definir um arquivo de configuração COMPSs contendo a especificação da infraestrutura escolhida. No caso de execuções em *cluster*, é necessário informar, por exemplo, o conjunto de máquinas pertencentes à execução, já no caso de execuções em nuvem, é necessário as especificações das *Virtual-Machines* (VMs) ou dos *containers*. Um nó especificado como central é responsável pelo gerenciamento da aplicação, pela transferências de dados, agendamento das tarefas e pela gestão da infra-estrutura [Conejero et al., 2017]. Todos os outros nós atuam como trabalhadores e são responsáveis por realizar qualquer ação solicitada pelo *master*. Uma característica importante do COMPSs, quando executado em infraestruturas de nuvem controladas por orquestrados de recursos como Singularity ou Mesos, é a sua capacidade de adaptar elasticamente, em tempo de execução, a quantidade de recursos à carga de trabalho

atual [Ramon-Cortes et al., 2018]. Nesses casos, quando o carregamento agregado das tarefas é excessivo para a configuração atual, o COMPSs é capaz de adicionar mais recursos durante a execução, selecionando o provedor de computação que melhor se adapta aos requisitos da aplicação e cria uma instância virtual onde executará as tarefas. Quando a carga diminui, o COMPSs é capaz de destruir o excesso de instâncias.

O modelo de programação do COMPS é suportado por um sistema de execução que gerencia vários aspectos para o programador, como a análise de dependências, o escalonamento das tarefas e a sincronização dos dados produzidos. Quando uma aplicação se inicia, o COMPSs analisa os métodos informados pelo usuário como paralelizáveis e cria durante essa análise um Grafo Direcionado Acíclico (DAG) que relaciona as dependências entre esses métodos. Esse DAG, semelhante ao mostrado na figura 2.1, pode ser visualizado posteriormente pelo usuário e é útil para entender o comportamento da execução, por exemplo, mostrando quais métodos estão sendo realmente paralelizados.

Durante a execução, quando as tarefas estão livres de dependências, elas são escalonadas nos recursos distribuídos disponíveis. E quando um método paralelizável (tarefa) termina a sua execução, o resultado produzido é gravado em um arquivo binário no disco, de forma transparente para o usuário, e informa ao computador mestre a sua finalização. O computador mestre, por sua vez, verifica (a partir da DAG), qual será o próximo método que irá precisar desse resultado parcial gerado. O escalonador poderá atribuir essa nova tarefa ao mesmo *worker* que gerou tal dado ou pode atribuir a um novo nó. Nesse último caso, será responsabilidade do *worker* produtor enviar o dado. Esse é um aspecto importante da abordagem que o COMPSs utiliza, uma vez que, caso existam dependências durante uma execução, os dados parciais (que estão em espera) não ocuparão espaço em memória, já que estão salvos em disco. O escalonador, além de outros fatores, leva em consideração a localidade dos dados para atribuição de novas tarefas. COMPSs utilizam semáforos para a sincronização dos dados quando o computador principal precisa acessar o resultado produzido por uma das tarefas. No momento que for necessário acessar um resultado, esses semáforos cuidarão, de forma transparente ao usuário, para que não seja iniciado nenhuma nova tarefa além das respectivas pela produção do dado em questão, além disso, são responsáveis pela transferência do dado para a execução do código principal.

Geralmente, uma etapa comum em um programa COMPSs é a divisão do conjunto de dados de entrada em n partes, onde n é geralmente igual ao número de *cores* disponíveis em um *cluster*. Dessa forma, o processamento de uma operação pode ser dividido entre os recursos disponíveis. Essa divisão dos dados pode ser feita de várias formas, sendo elas:

- A forma mais natural de se pensar é que, a princípio, o computador central lê a base de dados de entrada e, após isso, divide-os no número de fragmentos desejado. Embora seja uma solução simples, tem-se como desvantagem que a leitura dos dados seria realizada serialmente, desperdiçando recursos, além de que, nesse caso, é necessário que o dado caiba todo em memória para assim dividi-lo em n partes.

- Outra alternativa é dividir o arquivo no número de fragmentos desejados e direcionar cada um desses novos arquivos criados para cada tarefa. Desta forma, o computador central não precisará ler os dados, apenas os *workers* responsáveis pelas tarefas. Além disso, só será enviado pela rede a cada *worker* sua parte respectiva do arquivo; no entanto, a divisão e o armazenamento é uma responsabilidade do programador.

- Caso se utilize discos compartilhados, o arquivo pode ser mantido nesse disco para o acesso de cada tarefa. Nesse caso, o programador delimitaria as partes respectivas do arquivo a cada tarefa (por exemplo, pela divisão de linhas). Desta forma, a leitura seria paralelizada entre as *Threads* (desconsiderando as limitações de acesso do HD). Entretanto, o programador teria que se preocupar com aspectos da divisão dos dados entre as tarefas.

Existem também duas abordagens de programação caso o resultado da aplicação seja a criação de um arquivo de saída. A primeira exige uma sincronização da escrita dos dados, isto é, o computador central recebe os dados produzidos pelas tarefas e os escreve à medida que for recebendo-os. O problema dessa abordagem é que, mesmo que o programador tente remover a referência desses resultados parciais à medida que forem sendo escritos em disco, em linguagens como Java e Python, que adotam o conceito de *garbage collector*, não há garantias de liberação do espaço ocupado em memória. Logo, essa abordagem pode não ser viável em aplicações de *big data* em que o dado final produzido seja maior que o espaço de memória disponível no computador central. A segunda abordagem é a criação de múltiplos arquivos contendo os resultados parciais. Dessa forma, cada tarefa COMPSs fica responsável por criar o seu arquivo parcial de forma assíncrona, que posteriormente será transferido para o computador central ao final de cada execução. Caso seja necessário, é possível agrupar tais arquivos em um único após a execução.

2.2 HDFS

O Hadoop Distributed File System (HDFS) é um projeto de código aberto da Apache Software Foundation de um sistema de arquivos distribuídos inspirado no Google File System [Ghemawat et al., 2003] e implementado em Java, integrado hoje com o Apache Hadoop [White, 2015]. O sistema de arquivos HDFS foi desenvolvido exatamente para gerenciar a divisão e distribuição das partes de arquivos massivos. Nesse sistema, cada arquivo é dividido em blocos (128 MB é um tamanho usual), que são distribuídos entre os nós de armazenamento. Para fins de tolerância a falha e para aumentar a disponibilidade dos dados, cada bloco pode ser replicado em mais de um nó. Esses recursos permitem que, se qualquer conjunto de máquinas vier a falhar, a aplicação como um todo não será interrompida porque o processo que estava sendo realizado naquela máquina pode ser reiniciado ou, transferido para outra máquina disponível, tudo isso de forma transparente para o usuário. Dessa forma, o HDFS suporta o armazenamento e processamento de grandes volumes de dados em um *cluster* heterogêneo de computadores de baixo custo. Essa quantidade de dados pode chegar à ordem *petabytes*, quantidade que não seria armazenada em um sistema de arquivos tradicional. Apesar da sua implementação ser em Java, o HDFS pode ser acessado de muitas maneiras diferentes, tais como utilizando a sua API em Java, em HTTP REST (também chamada de webHDFS) e por meio de um *wrapper* de linguagem C, a libHDFS, para a comunicação com a API em Java.

Assim como o COMPSs, o HDFS também é baseado no paradigma *master-worker*. O mestre é representado por uma instância chamada Namenode e a instância Datanode representa os escravos (*workers*). O Namenode é um serviço responsável pelo espaço de nomes dos arquivos que constitui de uma árvore do sistema de arquivos contendo metadados de todos os arquivos e diretórios. O Namenode é o componente central do HDFS, portanto, é recomendável ser implantado em um nó exclusivo e, de preferência, na máquina com o melhor desempenho do *cluster*. Também por motivos de desempenho, o Namenode mantém todas as suas informações em memória para facilitar o acesso, de modo que, se um bloco requerer replicação ou um cliente precisar da localização de um bloco, essas operações são realizadas rapidamente sem ter que acessar disco para recuperar tais informações. Por fim, os Datanodes, armazenam e recuperam blocos quando eles são informados (pelos clientes ou pelo Namenode) e relatam periodicamente ao Namenode uma listas de blocos que estão armazenando.

Quando um cliente inicia uma operação de leitura de arquivos, é enviada uma solicitação ao Namenode para obter a localização de cada bloco do arquivo a ser lido. Por sua vez, o Namenode retorna, para cada bloco, uma lista ordenada de endereços

dos Datanodes que contêm uma cópia desse bloco. A ordenação dessa lista é feita com base na proximidade do Datanode em relação ao cliente. Depois disso, o cliente se conectará aos Datanodes disponíveis mais próximos que tenham os blocos desejados. Em casos onde o melhor Datanode para um determinado bloco é o próprio cliente, o HDFS fornece um mecanismo otimização chamado *Short-Circuit*, que permite o acesso de tais dados diretamente no sistema de arquivos [White, 2015, p. 308] minimizando o *overhead* das transmissões pela rede.

Para que o HDFS possa lidar com tamanhos de arquivos grandes de uma forma mais eficiente e transparente, ele adota uma estratégia de divisão em uma sequência de blocos de tamanho fixo (geralmente 128 MB, mas pode ser alterado se necessário, dependendo da magnitude dos arquivos que irão compor o *cluster*), que são armazenados como unidades independentes. No caso de dados que não ocupem todo o tamanho de bloco alocado, o espaço restante não é desperdiçado e pode ser usado por outros dados.

Quando um arquivo é dividido em blocos, o HDFS não tem conhecimento do tipo de dados que está sendo armazenado. Assim, pode haver casos em que, durante uma divisão, uma palavra ou uma linha, por exemplo, seja dividida entre dois blocos. Por isso, uma aplicação que utiliza o HDFS, por exemplo uma aplicação Hadoop, não trabalha com Blocos e sim com Blocos lógicos (também chamados de Splits) que nada mais são do que um subconjunto de dados de um ou mais blocos, com processamento de sentido lógico. Por convenção, os blocos que o HDFS cria durante a divisão e o armazenamento dos arquivos são chamados de blocos físicos. Para a delimitação de um Split, a aplicação deve informar ao HDFS como é feita a divisão de cada unidade do dado (*record*). Por exemplo, se for um texto, cada *record* equivale a uma linha. A figura 2.2 mostra um exemplo da diferença entre blocos HDFS e blocos lógicos (Splits). Nesse exemplo, embora o arquivo seja composto de apenas três blocos físicos, existem seis blocos lógicos, um para cada linha do arquivo.

Além de dividir os arquivos em blocos, o HDFS ainda replica esses blocos entre os vários Datanodes na tentativa de aumentar a confiabilidade. Por padrão, um bloco HDFS possui um fator de replicação igual a três (esse valor pode ser configurado), ou seja, cada bloco está presente em três nós diferentes.

O maior benefício da replicação é alcançar maior tolerância a falhas e confiabilidade dos dados. No caso de um nó escravo falhar, o processamento pode ser feito em outra máquina que contenha uma réplica daquele bloco, sem a necessidade de transferência de dados e interrupção da execução do aplicativo. No contexto de uma falha, ocorre uma diminuição no número de réplicas de um bloco. Periodicamente o Namenode consulta os metadados sobre os Datanodes com falha e reinicia o processo

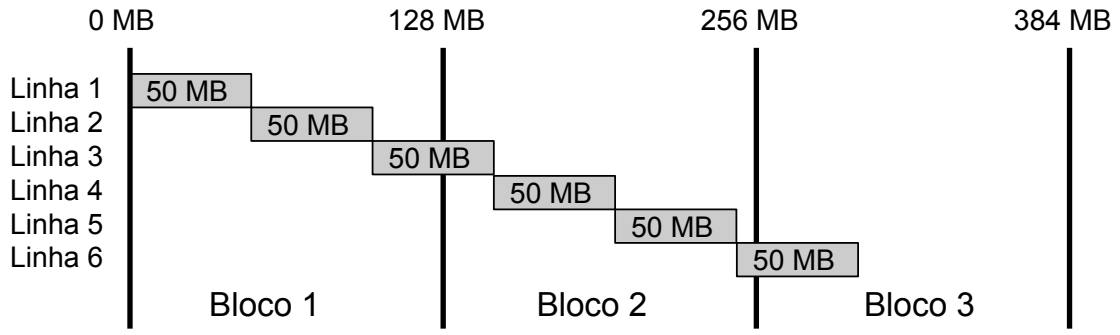


Figura 2.2: Esquema de diferença entre blocos físicos e lógicos

de replicação em outros Datanodes para garantir seu fator mínimo.

Um *cluster* HDFS pode ter centenas ou milhares de *workers* e, portanto, precisam ser organizados em diferentes *racks*. O HDFS organiza o armazenamento de blocos de arquivos e suas réplicas em diferentes máquinas e *racks*. Assim, mesmo que ocorra uma falha em um gabinete inteiro, os dados podem ser recuperados e o aplicativo não será interrompido. Ao garantir a durabilidade dos dados, esta estratégia tem a vantagem adicional de multiplicar a largura de banda de transferência de dados e há mais oportunidades para localizar a computação perto dos dados necessários.

Quando o cliente precisa escrever um arquivo no HDFS, ele envia uma solicitação para criar um arquivo usando a API FileSystem. Primeiro, um *buffer* temporário interno é criado e os dados do arquivo são enviados pelo cliente. Quando esse *buffer* possuir dados suficientes para preencher um bloco, uma solicitação é encaminhada para o Namenode, que atribui blocos e Datanodes de acordo com o fator de replicação e encaminha tais atribuições ao cliente. A lista de Datanodes é criada de acordo com a proximidade do cliente, minimizando a distância da rede do usuário para o último Datanode. O cliente por sua vez, envia dados de arquivos para o primeiro Datanode na lista (que pode ser a própria máquina). Depois disso, o Datanode, seguindo a lista ordenada, iniciará o processo de replicação dos blocos mais *n-1* vezes.

A fim de garantir a coerência dos dados e uma alta taxa de leitura, o HDFS utiliza um modelo *write-once-read-many*. Uma vez criado um arquivo, ele não pode ser alterado, exceto para anexar dados ao final. A opção de anexo, não é habilitada por padrão pois existem algumas limitações, além disso, o conteúdo só pode ser anexado ao final dos arquivos e para efetuar tal anexo, é necessário que nenhum cliente esteja lendo tal arquivo (são utilizado semáforos para bloquear o acesso).

2.3 Lemonade

Lemonade é uma ferramenta visual para o cientista de dados e visa usuários que não conhecem uma linguagem de programação ou que precisem desenvolver fluxos de trabalho usando o conjunto de ferramentas existente [Santos et al., 2017]. A plataforma é voltada para a criação de fluxos de análise e mineração de dados em nuvem, com garantias de autenticação, autorização e contabilização de acesso. Em sua versão atual, o Lemonade possui diretrizes capazes de gerar código Spark 2.0.2 em linguagem Python (PySpark) a partir de um fluxo de operações definidas visualmente pelo usuário.

Usando uma interface para a construção visual de fluxos, o Lemonade permite a criação e execução de fluxos, encapsulando os detalhes de armazenamento, codificação, segurança e o processamento distribuído, permitindo que sejam usados em ambientes em nuvem por especialistas no domínio dos dados. Para isso, ele oferece ao usuário um conjunto de caixas de operações que representam funções e algoritmos já previamente implementados em uma linguagem de programação, e partir disso, o usuário é capaz de criar um fluxo lógico a partir das ligações entre as caixas por meio pontos de comunicação direcionais (entrada ou saída). Cada caixa possui um número determinado de entradas e saídas, o tipo de resultado que será produzido, e o conjunto de parâmetros que o usuário poderá especificar durante uma execução. Existem caixas no Lemonade que a especificação de um determinado parâmetro é obrigatória, como por exemplo, o nome do arquivo de entrada a ser lido, e existem parâmetros que podem ser opcionais, como o número máximo de iterações de um algoritmo. Nesse último caso, é oferecido valores padrão caso o usuário não queira especificar. A figura 2.3 é um exemplo do tipo de aplicação, que chamaremos de Workflow, que o Lemonade é capaz de criar. Tal aplicação consiste em: uma etapa de pré-processamento dos dados representada pelas caixas de leitura, criação de um vetor de atributos e remoção de algumas colunas; uma outra etapa para amostragem dos dados, onde uma parte é utilizada na criação de um modelo de classificador; e uma última etapa para o teste do modelo criado.

A arquitetura do Lemonade define sete componentes individuais que funcionam como micro serviços especializados nas áreas de algoritmos (Tahiti), interface (Citron), criação e monitoração da execução (Juicer), execução (Stand), base de dados (Limonero), segurança/privacidade (Thorn) e visualização (Caipirinha). Sua estruturação possibilita escalabilidade dos componentes de forma independente, que podem ser encapsulados por *container* Docker[1] ou aplicações individuais em máquinas físicas.

O componente Tahiti é responsável por manter metadados referentes às operações

[1] https://www.docker.com/

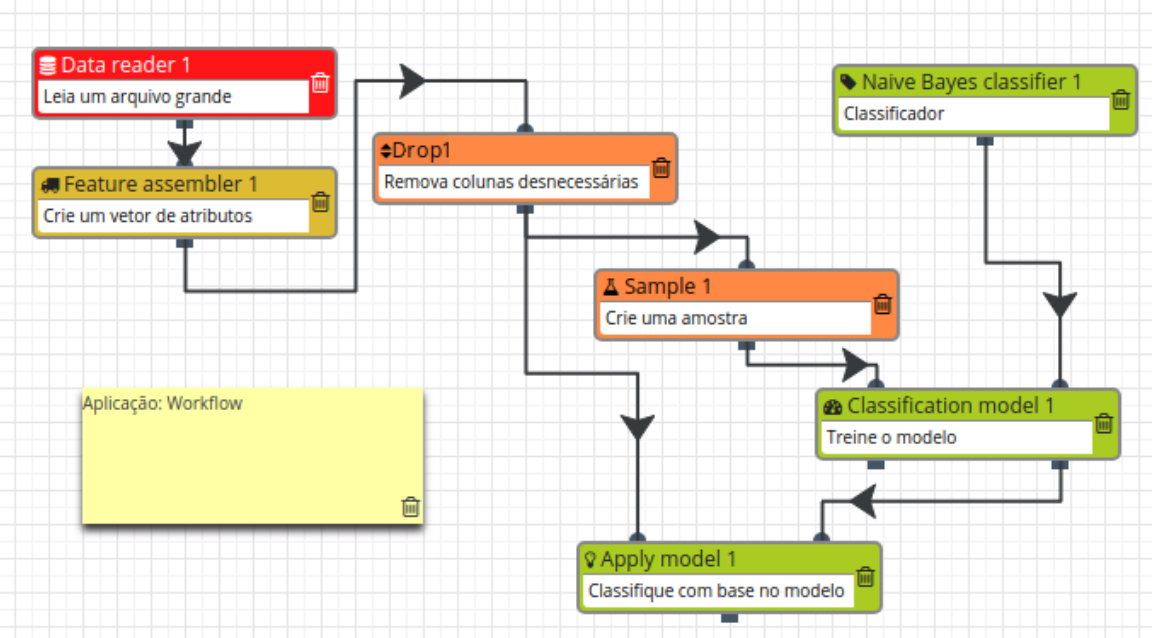


Figura 2.3: Esquema da aplicação Workflow criada a partir do Lemonade

e aos fluxos de dados criados por usuários e responder requisições sobre os mesmos. Atualmente são suportadas operações de leitura/escrita, ETL (Extração, Transformação e Carga), algoritmos de aprendizado de máquina (e avaliação de resultados) e operações sobre dados georreferenciados e sobre dados textuais. Após a implementação de uma operação ou algoritmo, é necessário fazer o cadastro de seus metadados para integrá-los ao Tahiti, como por exemplo, informar seu nome, sua descrição, suas portas e os parâmetros. Os parâmetros das operações podem ser divididos em cinco categorias: execução, privacidade/segurança, monitoramento, aparência e qualidade de serviço (QoS). Os parâmetros de execução definem o comportamento da operação, tal como o caractere separador em uma operação de leitura de arquivo tabular ou o número de iterações em uma operação iterativa. Os parâmetros de privacidade e segurança definem a visibilidade dos resultados da operação. Os parâmetros de monitoramento definem a granularidade do registro de eventos durante a execução. Os parâmetros de aparência definem as cores e a posição das caixas referentes às operações no fluxo visual de análise. Por fim, os parâmetros de qualidade de serviço estabelecem limites para tempo de execução e para uso de memória e de processamento.

O componente Citron é o responsável por prover a interface *web* para que o usuário crie, execute e monitore o seu fluxo de análise de dados. O usuário da plataforma pode escolher, arrastar e ligar operações pré-definidas (através de suas portas) para compor um fluxo de análise de dados. Cada operação pode ser inspecionada para identificar seus

parâmetros associados, que podem ser especificados pelo usuário. O Citron também mostra o status das operações submetidas para execução (pendente, em andamento ou finalizada), permitindo que o usuário acompanhe o andamento do processo. Os fluxos criados pelos usuários são armazenados em formato *JSON*; seus principais campos são: identificador, nome, descrição, usuário, plataforma, data de criação e de atualização dos fluxos de operação, além desses, existe um campo *"task"* que contém uma lista de todas as tarefas envolvidas na aplicação e seus respectivos parâmetros (dentro de um subcampo de formulários), e um campo *"flow"* que relaciona as ligações entre as tarefas do fluxo. O código 2.1 representa o *JSON* criado para a aplicação Workflow mostrada anteriormente. Podemos ver, por exemplo, que a tarefa Drop irá remover as colunas "coluna1" e "coluna2".

```
{
 "id": 1234, "name": "Workflow", "description": "Aplicação de exemplo",
 "enabled": true,
 "created": "2017-11-06T00:12:47+00:00", "updated": "2017-12-18T16:46:44+00:00",
 ...
 "tasks": [
  ...
  {
   "id": "c4d56463-91b2-4d8d-91ad-c0dbfb04bb66", "left": 35, "top": 229,
   "z_index": 11, "version": 2,
   "forms": { "attributes":{
                            "category": "execution",
                            "value": ["coluna1", "coluna2"]
                            }
   },
   "operation": {"id": 6, "name": "Drop columns",  "slug": "drop"}
  }],
  "flows": [
    ...
    {
      "source_port": 4, "target_port": 23,
      "source_port_name": "output data", "target_port_name": "input data",
      "source_id":"c4d56463-91b2-4d8d-91ad-c0dbfb04bb66",
      "target_id":"9cdff7dc-a8b0-41b4-a831-3552ea9dd159"
    }],
 ...
}
```

Código 2.1: *JSON* da aplicação *workflow*

O Juicer é o componente responsável pela execução das aplicações. Ao receber do Citron um fluxo no formato *JSON*, o Juicer gera o código-fonte executável necessário, atuando como um *transpiler* (compilador de código-fonte para código-fonte, *source-to-source*). Para isso, o tradutor interpreta o campo *"flow"* desse fluxo como um grafo dirigido, onde cada vértice é uma operação Lemonade e cada aresta direcionada é uma relação entre as caixas, de entra e saída de dados, e realiza uma ordenação

topológica para analisar a precedência entre as operações. Cada caixa é então traduzida em código-fonte seguindo a precedência estabelecida pela ordenação. Esse código contêm, além das chamadas de funções e algoritmos, pontos de geração de *logs* que serão lidas pelo componente Stand, durante a execução, e posteriormente encaminhada para outros componentes como, o Citron (para exibição dos *status* da execução) e o Caipirinha (para a visualização de dados). Após essa conversão, o código é executado, passando os parâmetros de Qualidade de Serviço (QoS) definidos pelo usuário para o ambiente de execução em nuvem, para garantir que as operações sejam executadas com recursos suficientes para atender as demandas indicadas. Durante a execução, o Juicer é responsável informar ao Limonero quaisquer novas bases de dados geradas como saída e reportar o status e eventos da execução.

O componente Stand coordena a comunicação entre o Citron e o Juicer, garantindo independência entre os dois componentes. Quando uma requisição para executar um fluxo é gerada no Citron, ela é armazenada em uma fila pelo Stand para ser consumida pelo Juicer. No sentido oposto, são armazenadas informações sobre o status de execução e eventos produzidos no Juicer. A versão atual do Stand implementa uma comunicação assíncrona através de filas produtor-consumidor.

O Limonero tem como função armazenar metadados referentes às fontes de dados e responder requisições sobre as mesmas. Para cada fonte de dados disponível na plataforma são associadas informações sobre a sua localização, permissão de acesso dos usuários, informações gerais do arquivo no sistema (nome, tipo de dados, precisão, tamanho, formato dos dados) e sobre os atributos dos registros que o compõem, tais como distribuição, valores faltantes, valor da média, valor máximo e outros.

O Thorn é o componente responsável pela segurança, privacidade e acessos (do inglês, AAA - *Authentication, Authorization and Accounting*). No Lemonade, uma base de dados pode ter atributos sensíveis, logo, o ideal é que cada usuário tenha a sua permissão diferente a essa base, além disso, quando uma operação utilizar múltiplas fontes de dados, com diferentes níveis de permissão, a base resultante será uma composição das múltiplas permissões.

O componente Caipirinha tem como função a elaboração de visualizações de dados, através de diferentes metáforas visuais. Entre estas destacam-se inicialmente visualizações estáticas de dados em formatos bem definidos, como exibição de amostras dos registros de uma base de dados, gráficos de séries temporais e histogramas, e visualizações exploratórias, onde o usuário pode ir refinando a exibição de partes do conjunto de dados, conforme seus interesses, como uma operação de zoom em um grafo complexo. Além de visualizações exploratórias e estáticas, o Caipirinha também envolve modelagens não-triviais, como por exemplo o grande volume de dados que

pode ser usado em determinadas visualizações dinâmicas e a possibilidade de se usar a visualização para selecionar novos dados de entrada para o fluxo de dados.

2.4 Trabalhos Relacionados

Diversos projetos estão sendo feitos para tornar o COMPSs ainda mais adaptado a cenários de *big data*. Martí [2017] apresenta uma integração com o DataClay, um projeto atual do BSC, capaz de armazenar dados do objeto, métodos (código a ser usado em um objeto) e políticas que definem questões como segurança, privacidade, integridade, ciclo de vida entre outros. Desta forma, facilita a tarefa de implementar programas centrados em dados, aproveitando ao mesmo tempo o paralelismo disponível e possibilitando ao programador a escolha do local de computação de um determinado dado.

Tejedor et al. [2017] apresenta uma integração com o Hecuba, também desenvolvido pelo BSC, um *middleware* em Python que visa facilitar interações com bancos de dados não-relacionais. O *backend* de armazenamento usado em Hecuba é o Cassandra, um banco de dados distribuído e altamente escalável para dados no formato de chave-valores, que implementa uma arquitetura não centralizada, com base em comunicações *peer-to-peer* em que todos os nós do *cluster* são capazes de receber e atender consultas. Nesse mesmo trabalho, é apresentado a Storage API, uma interface de software que permite a aplicações e ao COMPSs de trabalhar com objetos persistentes (*Persistent Objects*, POs). A Storage API pode ser implementada em múltiplas *back-ends* e permite criar, excluir, inserir, recuperar, interagir sobre dados persistentes e principalmente para obter informações de localidade sobre esses dados.

Já Lezzi et al. [2011] criaram uma plataforma de serviço de nuvem orientado baseado na virtualização de tecnologias. Para tal, foi realizado uma integração do COMPSs no ambiente VENUS-C, uma ferramenta que tem como objetivo permitir a um pesquisador executar aplicações científicas na nuvem de forma transparente. Com tal recurso, o pesquisador é capaz de executar um processo na plataforma e, dependendo do tipo/modelo da tarefa (p.ex., MapReduce, Batch ou HPC), o VENUS-C a executará de modo transparente em um dos dois modelos de programação suportados pela plataforma, o COMPSs ou EMIC Generic Worker.

Relacionado à criação e execução de aplicações baseadas em fluxos de dados, várias ferramentas como RapidMiner [Mierswa et al., 2006], Orange [Demšar et al., 2013] e KNIME [Berthold et al., 2009] foram propostas. Entretanto, poucas delas suportam a execução distribuída em *clusters* ou em nuvem. As soluções que suportam

o processamento distribuído podem ser divididos em duas categorias: as que fornecem o paralelismo de fluxo de trabalho, que se referem à habilidade de executar múltiplos fluxos de trabalho concorrentemente, e as que fornecem o paralelismo de tarefas, que é a capacidade de execução de múltiplas tarefas da mesma aplicação [Belcastro et al., 2015].

O ClowdFlows [Kranjc et al., 2012], uma das plataformas *open-source* mais conhecidas de criação e execução de *workflows* visuais em nuvem, tem foco no paralelismo de fluxo de trabalho, mas apesar de utilizar bibliotecas de outros ambientes, como WEKA [Hall et al., 2009] e Orange, suporta alguns algoritmos implementados no modelo de programação MapReduce. Existem outras soluções como o Orange4WS [Podpečan et al., 2011], que embora forneça ambos os níveis de paralelismo, as implementações das funções suportadas são seriais. Desse modo, só há um ganho de desempenho significativo em aplicações com tarefas independentes.

As ferramentas mais próximas da proposta do Lemonade são o Microsoft Azure Machine Learning (ML) Studio [Mund, 2015] e o Radoop [Prekopcsak et al., 2011]. O Azure ML é uma plataforma comercial que tem como objetivo prover a criação e a execução de modelos de análises de dados a partir de fluxos e interações entre diversos componentes. Possui compatibilidades com outros serviços da Microsoft como HDInsight e Azure SQL Database, além de suportar execuções de aplicações externas como as implementadas em Spark e MapReduce. Já o Radoop é uma integração entre o RapidMiner e o Hadoop que permite a execução de fluxos de operações criados visualmente como aplicações Hadoop. No entanto, além de tal solução não ser gratuita, existe uma quantidade limitada de operações suportadas.

Diferentemente das soluções apresentadas, o Lemonade é uma plataforma de criação de fluxos de operações a partir de uma interface visual e de execução em nuvem que fornece ambos os níveis de paralelismo, o de execução de múltiplas aplicações e de tarefas, simultaneamente. Atualmente, com a contribuição deste trabalho, o Lemonade é capaz de criar e executar operações inteiramente em Spark ou em COMPSs, plataformas que fornecem, de fato, modelos de programação paralela e distribuída.

Capítulo 3

Integração entre o HDFS e o COMPSs

Como já mencionado, COMPSs é baseado em tarefas, logo, para explorar um maior nível de paralelismo em processos com grande volume de dados é desejável trabalhar sobre fragmentos desse conjunto. No entanto, o usuário tem a responsabilidade de criar esses fragmentos, seja pela divisão do arquivo de entrada ou dos dados já em memória. Além disso, em execuções COMPSs sem o uso de discos compartilhados, esses arquivos precisam ser armazenados no computador principal para poderem ser transferidos pela rede para os *workers* em tempo de execução.

O principal benefício apresentado na integração entre o HDFS e o COMPSs é a delegação de algumas responsabilidades ao HDFS, tais como a divisão dos arquivos de entrada em blocos e a transferência desses blocos para cada tarefa COMPSs. Além disso, sem a necessidade da transferência dos arquivos por parte do computador central, o envio das tarefas passa a ser mais rápido, o que permite mais *workers* iniciarem suas execuções mais cedo. Além disso, todas as outras vantagens que o HDFS busca garantir, continuam sendo válidas para essa integração, como por exemplo, portabilidade e tolerância a falhas de *hardware*.

Para essa integração, Java e Python foram as linguagens alvo. A escolha do desenvolvimento em linguagem Java se deve ao fato de ser a linguagem nativa tanto do COMPSs quanto do HDFS. Já o desenvolvimento em Python foi devido a relação com o ambiente Lemonade, que será descrito no próximo capítulo. Na integração dos sistemas HDFS e COMPSs, o primeiro passo é decidir como as abstrações do HDFS se tornam disponíveis para o programador COMPSs.

3.1 Abstração de dados

A API de integração provê ao programador COMPSs dois tipos de entidades com funções bem definidas. A primeira entidade, representada pela classe HDFS, é executada na aplicação diretamente ou indiretamente durante todas as interações com o HDFS. Já a segunda, representada pela classe Block, é responsável pela abstração e pelo acesso ao arquivos divididos em fragmentos no HDFS. Com essas duas entidades o programador é capaz de realizar diversas operações no HDFS sem alterar o modelo de programação em COMPSs. Ela permite por exemplo, caso o usuário deseje, ler um arquivo completo de uma única vez do HDFS ou ler por fragmentos.

3.1.1 Leitura dos dados

Em aplicações COMPSs que precisam de ler múltiplos arquivos de entrada utilizando um sistema de arquivos convencional, é necessário primeiro obter a relação desses arquivos de entrada, que podem ser informados manualmente por parâmetro ou por meio de um comando de listagem de arquivos do sistema operacional. Já utilizando a API de integração do HDFS, essa etapa se resume na execução de um método, fornecido pela abstração, para a definição da lista de fragmentação dos arquivos. Existem dois métodos que podem ser utilizados para obter a lista de fragmentação: o método *findBlocks*, que define uma lista com um número de blocos lógicos iguais ao número de blocos físicos, tal como o HDFS os armazenou; e o método *findNBlocks*, dessa forma o usuário é capaz de passar por parâmetro o número de fragmentos desejados. No momento da leitura, dentro de uma tarefa COMPSs, a entidade Block, que delimita o espaço de um fragmento, será responsável por se comunicar com a classe HDFS para escolher o melhor fornecedor (*Datanode*) e pela transferência em si. A ideia por trás disso é que as tarefas funcionam em blocos e cada bloco será executado em uma tarefa. O algoritmo 1 exemplifica um trecho de execução COMPSs utilizando a API.

Algoritmo 1: Exemplo de uso da integração do HDFS ao COMPSs

```
Dados: N e X
início
    BLOCKSLIST = recupera a lista com N fragmentos de um arquivo X
     no HDFS;
    para block ∈ BLOCKSLIST faça
        task1(block);
    fim
fim
```

Nesse exemplo, cada um dos N fragmentos de um arquivo X armazenado no HDFS será lido paralelamente dentro da sua instância da tarefa (*task1*). A partir daí, os próximos passos são similares às soluções já existentes na programação COMPSs, isto é, cada resultado parcial pode ser salvo em um arquivo distinto ou pode servir de entrada para uma nova tarefa. A figura 3.1 contém a adaptação do WordCount (mostrado inicialmente na figura 2.1) para utilizar o HDFS. As únicas alterações necessárias foram as linhas destacadas por chaves logo no início das definições de *main* e *Count*. Em *main*, elas contêm basicamente a definição e delegação de cada fragmento a uma tarefa. Já em *Count*, é necessário a utilização do método *readBlock* para ler o fragmento do arquivo. Esse método retorna um objeto StringIO [1] que é capaz ler o conteúdo como um *buffer*, de forma análoga à leitura de um arquivo comum.

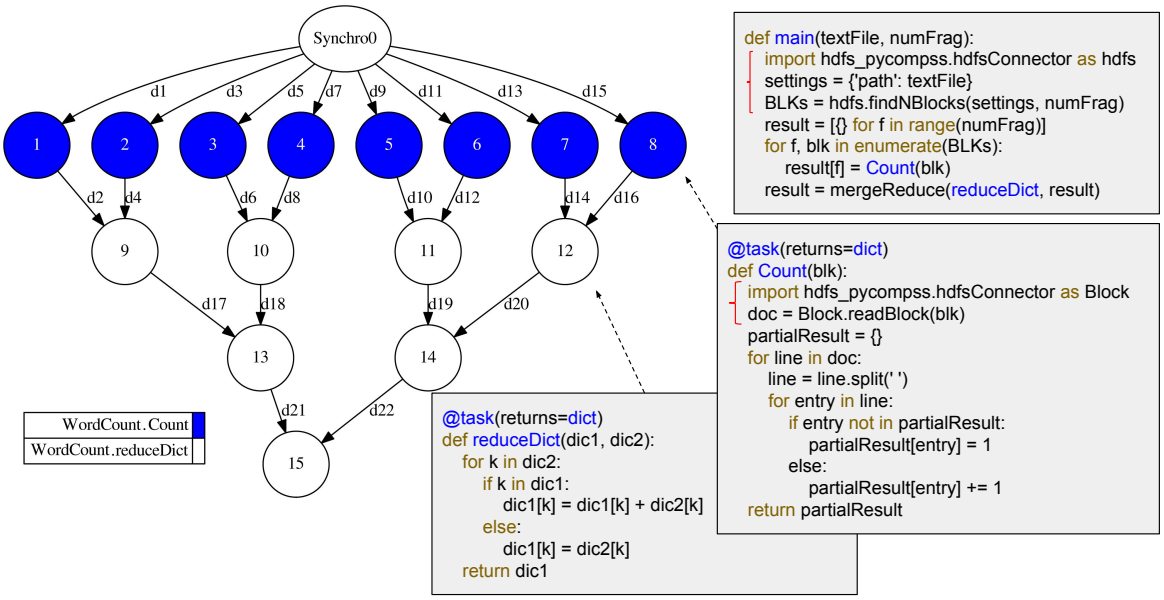


Figura 3.1: Exemplo de um contador de palavras em PyCOMPSs utilizando o HDFS

3.1.2 Escrita dos dados

A API do HDFS não permite a criação de blocos sob controle de uma aplicação. Utilizando essa API, só é possível criar e escrever arquivos, sendo responsabilidade do HDFS dividi-los em blocos e replicar esses blocos. Isso se deve ao fato dos arquivos serem escritos no HDFS sequencialmente ou adicionados ao final de um arquivo existente através da operação de anexar. Esta seria a primeira limitação do HDFS para as tarefas COMPS: a impossibilidade de escrita concorrente dos resultados obtidos pelas

[1] https://docs.python.org/2/library/stringio.html

tarefas. Isso significa que cada Datanote irá escrever em um arquivo sem qualquer competição, caso contrário seria gerada inconsistência de dados entre Datanodes. A segunda limitação do HDFS é que os arquivos não podem ser alterados depois de escritos. Diante dessas limitações, a API fornece dois meios distintos para a escrita de dados no HDFS que também são semelhantes às fornecidas pelo COMPSs quando se utiliza um sistema de arquivos convencional. A seguir será discutido cada uma delas, comentando as suas vantagens e limitações.

A primeira forma consiste na agregação de todo o resultado produzido pelo nó central, que o escreve em um único arquivo. Embora essa solução seja simples, em aplicações de *big data*, manter esse dado final na memória RAM do computador central pode não ser possível.

A segunda abordagem de escrita é a criação de arquivos parciais, um para cada resultado parcial produzido. Dessa forma, não existem atrasos provocados pela sincronização dos semáforos e transferência dos dados ao computador central. No entanto, a criação de vários arquivos pode não ser o ideal para uma dada aplicação, nesse caso, assim como quando utilizado um sistema de arquivos convencional, é possível agrupá-los em um único arquivo por meio de um comando *shellscript*[2]. Caso o usuário queira realizar tal procedimento a nível de aplicação, a API fornece o comando *mergeFiles* para tal função.

3.2 Abstração com localidade dos dados

Além da abstração descrita acima, é disponibilizado uma outra capaz de explorar a localidade dos dados. O diferencial desta é que ela foi implementada a partir da extensão da Storage API desenvolvida pelo BSC, o que permite dentre outras funções, um melhor manuseio de dados persistentes. Para o contexto dessa integração, ela é utilizada para informar ao COMPSs, em tempo de execução, a localidade dos blocos do HDFS e assim, sempre que possível, direcionar a leitura de um bloco à um computador que o possuir, evitando a transferência desnecessária de dados entre máquinas. No entanto, não há diferenças entre as abstrações para a etapa de escrita dos dados uma vez que é de responsabilidade do HDFS escolher os nós que serão utilizados no armazenamento dos blocos e de suas replicações.

Para o programador, existem três diferenças de utilização durante a leitura dos dados. A primeira é que se deve utilizar a classe BlockLocality ao invés da classe Block. Apesar da BlockLocality de conter todos os métodos oferecidos pela anterior,

[2]hadoop fs -getmerge /path/

essa também contém as implementações dos métodos internos necessários da Storage API. A segunda é que o usuário deve passar um arquivo de texto, como parâmetro de execução COMPSs, contendo a relação dos arquivos armazenados no HDFS que a aplicação desejará ler. Isso se faz necessário por causa da Storage API, que para poder escalonar as tarefas explorando a localidade dos dados, ela precisa de uma etapa inicial, antes mesmo do início da aplicação para definir os blocos e conhecer suas localidades. A terceira diferença é que ao utilizar tal abstração, será utilizado internamente o método *findBlocks* da classe HDFS para obter a lista de blocos que será a mesma que o HDFS usou para dividir o arquivo durante sua criação. Essa limitação vem da necessidade de definir a localidade dos blocos a serem trabalhados. Caso fosse utilizado o método *findNBlocks*, a API teria que ser capaz de atribuir a localidade de um dado baseado em heurísticas uma vez que estaríamos trabalhando com blocos lógicos e em determinados casos, um bloco lógico poderia estar armazenado em uma ou mais máquinas. No entanto, heurísticas que levam em consideração a escolha do nó com maior quantidade dos bytes do bloco não será sempre a melhor decisão, isso poderá levar a casos desbalanceados desnecessariamente ou até mesmo deixar um nó ocioso durante toda a execução.

A utilização desta abstração não é recomendada para aplicações com processamento intensivo (nesse contexto, onde o tempo de processamento é significativamente maior que o de transferência) que trabalham com dados onde o número de blocos do HDFS é menor do que o número de *cores* disponíveis. Pois nesse caso, também provocariam *cores* ociosos durante toda a execução.

No entanto, essa segunda abstração só está disponível para a linguagem Java, pois até o desenvolvimento dessa integração, a Storage API só havia sido liberada para essa linguagem. Atualmente, a Storage API foi disponibilizada também para Python, sendo possível a implementação dessa segunda abstração para trabalhos futuros.

3.3 Comunicação com o HDFS

Enquanto que o HDFS fornece uma API completa para programadores em linguagem Java, o mesmo não acontecesse para a linguagem Python. Atualmente o Hadoop/HDFS não suporta nativamente Python, o que nos levou a considerar diferentes alternativas de integração. A seguir listamos três abordagens: a utilização do webHDFS, uma API nativa do HDFS para a comunicação utilizando protocolo HTTP REST [Gonzales, 2016]; a criação de um serviço de comunicação entre a aplicação Python e um processo Java, semelhante ao que é realizado pelo PySpark utilizando o módulo Py4J [Rosen,

2016]; e por fim, a utilização do *wrapper* C da Java API (libHDFS), também utilizada em outros trabalhos [Leo & Zanetti, 2010].

Apesar do webHDFS fornecer os mesmos métodos disponíveis na API em Java, a utilização do webHDFS é significativamente mais lenta. Essa lentidão é tanto devida ao tempo de requisição quanto ao *overhead* da utilização do protocolo HTTP. A segunda abordagem consistiria na criação de um serviço de comunicação entre a aplicação Python e um processo Java, semelhante ao que é realizado pelo PySpark utilizando o módulo Py4J [Rosen, 2016]. O PySpark, que possui uma estrutura híbrida de Python e JVM, utiliza o Py4J já internamente apenas como um *driver* de requisição. Já no caso do COMPSs, sem uma mudança no seu código-fonte, não seria possível manter tal nível de integração. Caso contrário, cada *thread* criada em COMPSs precisaria se comunicar com um intermediador em Java (que se conectaria ao HDFS) externamente ao COMPSs para solicitar e receber dados. Além de consumir memória tais processos externos, a transferências desses dados teria que ser por meio do próprio intermediador, diferente do que ocorre no PySpark.

A abordagem escolhida foi a libHDFS. Esse *wrapper* se comunica diretamente com Java utilizando a Java Native Interface (JNI), uma tecnologia para comunicação de aplicações diretamente na JVM do Java. Por sua vez, Python possui, por padrão, módulos capazes de interpretar e converter tipos de dados da linguagem C. Com base nisso, foi possível utilizar a libHDFS a partir de um mapeamento das funções e dos tipos de dados a serem utilizados pela integração do HDFS. Dessa forma, os métodos invocados em Python são executados internamente em linguagem C a partir desse *wrapper*. A libHDFS utilizada nessa integração contém algumas mudanças em relação à aquela disponibilizada pelo HDFS no seu binário de instalação, como a remoção de algumas variáveis de ambiente, uma vez que até a presente versão do COMPSs (v2.2), as variáveis de ambiente definidas no *master* não eram carregadas para as tarefas paralelizadas. A vantagem de se utilizar essa última abordagem é que como essas operações são executadas internamente em C ela é mais rápida e eficiente que usar a webHDFS ou usar intermediadores de Java para Python como o Py4j.

Capítulo 4

Suporte do Lemonade ao COMPSs

Como discutido na seção 2.3, a ideia de funcionamento do Lemonade é a criação do código-fonte a partir do fluxo descrito pelo usuário por meio da ligação de caixas de operações. Para isso, é necessário um conjunto básico de etapas durante a integração do Lemonade com uma nova linguagem de programação ou plataforma: primeiramente é necessário escrever todas as operações e algoritmos que serão oferecidas em caixas Lemonade nessa nova linguagem; a próxima etapa é o registro dessas funções no Tahiti; e por fim, a criação de um novo *transpiler*, no Juicer, respectivo a essa nova linguagem.

As aplicações geradas pelo Lemonade podem ser divididas em duas partes, o código estático que contém as implementações das funções e o código dinâmico, que contém a chamada dessas operações. No caso de uma aplicação Spark criada pelo Lemonade, a parte estática são as funções MLlib fornecidas pelo Spark e a parte dinâmica são chamadas das funções com seus respectivos parâmetros. O suporte do Lemonade ao COMPSs utiliza a mesma ideia, no entanto, as operações e algoritmos a serem suportados tiveram que serem implementados, uma vez que o COMPSs, até a versão atual (2.2), não fornece uma biblioteca semelhante.

Entre as versões de COMPSs disponível, nosso alvo para a integração com o Lemonade foi a versão Python. Python é uma linguagem de programação de fácil codificação orientada a objetos comparável a Perl, Ruby ou Java que vem ganhando impulso nos últimos anos na computação científica, às vezes substituindo ferramentas tradicionais como Matlab [Tejedor et al., 2017]. Não só é ideal para prototipagem, mas também para projetos maiores, pois vem com bibliotecas que suportam muitas tarefas de programação comuns e pode seja facilmente estendido, adicionando outras bibliotecas escritas em idiomas como C/C++. É um software livre e pode ser executado em muitas arquiteturas de computadores e sistemas operacionais.

Neste capítulo será explicada a integração desenvolvida entre o COMPSs e o

Lemonade. Será detalhado a biblioteca de operações desenvolvidas e como ela foi adicionada ao Lemonade, no que diz respeito aos metadados das operações, a conversão do fluxo de operações em código-fonte e a execução das aplicações na plataforma COMPSs.

4.1 Algoritmos e Operações

A ideia por trás das implementações dos módulos COMPSs para o Lemonade foi de aproveitar a noção de blocos HDFS e decompor grandes arquivos de dados em partes, para que o algoritmo seja dividido em tarefas menores, cada uma processando uma parte dos dados com pouca ou nenhuma dependência com outras tarefas, para que possam ser paralelizadas. Para a leitura dos arquivos no HDFS, existirá uma caixa Lemonade chamada de *data-reader* que será encarregada de criar n tarefas COMPSs, uma para cada bloco de um arquivo HDFS. Cada tarefa produzirá um *DataFrame*, que fará parte de uma lista, que por sua vez, representará o dado completo.

Foi utilizado o *DataFrame* Pandas[1] como a estrutura de dados principal. O módulo Pandas é uma biblioteca *open-source* em Python desenvolvida em cima do módulo NumPy e que oferece uma implementação eficiente de uma estrutura tabular; além disso, possui um conjunto abrangente de funções auxiliares [McKinney, 2012].

Todas as operações e algoritmos suportados podem ser interpretados como "caixas-pretas" com entradas e saídas definidas. Os parâmetros de cada função podem ser de dois tipos, o dado de entrada e um dicionário de configuração. O dicionário de configuração é responsável por listar todos os parâmetros de uma função especificados pelo usuário durante a criação do fluxo na interface. Já o dado de entrada/saída é sempre uma lista de *DataFrames*. Internamente, uma função pode fazer agregações e pode executar outras funções internamente com outros tipos de parâmetros, mas o resultado final de uma função que representa uma caixa do Lemonade será sempre uma lista de *DataFrames* com o mesmo número de elementos da entrada. Essa padronização garante a compatibilidade entre a saída de uma função e entrada de outra

A escolha dos algoritmos a serem implementados foi baseada nos algoritmos existentes em Spark no Lemonade e também seguiu a lista dos algoritmos mais populares em mineração de dados [Wu et al., 2008]. Foram implementadas 44 funções, disponibilizadas e documentadas no GitHub[2]. A tabela 4.1 contém uma sumarização de todos os tipos de funções, divididas em oito categorias: operações de leitura/escrita (Entrada e saída); operações de auxílio (Utilitários); operações de transformação e

[1] `https://pandas.pydata.org`
[2] `https://github.com/eubr-bigsea/Compss-Python`

extração de informação (Transformação de dados); algoritmos de *Machine Learning* (Aprendizado de máquina); avaliadores de qualidade dos modelos de *Machine Learning* (Avaliadores de qualidade); operações sobre dados textuais (Processamento de texto); operações sobre dados georreferenciados (Processamento geográficos); e operações sobre grafos (Grafos). A lista detalhada das funções disponíveis é apresentada na tabela B.2 contida no apêndice B.

Tabela 4.1: Operações e algoritmos implementados em COMPSs

Categoria	Operações e/ou Algoritmos
Entrada e saída	Ler e salvar arquivo
Transformação de dados	Adição de colunas, agregação, tratamento de linhas/colunas com dados faltando, remoção de linhas duplicadas, interseção de dois conjuntos de dados, filtragem de linhas baseada em uma condição, projeção de colunas, junções (*inner, left* e *right join*), normalização, substituição de valores, ordenação, operação de transformação, união de conjuntos de dados
Utilitários	Alterar metadados de uma coluna, balancear as partições
Processamento geográficos	Ler arquivo de *Shapefile*, verificar se um ponto está dentro de uma região e ST-DBSCAN
Aprendizado de máquina	K-Means, DBSCAN, KNN, Naive Bayes, SVM, Regressão Logística, Regressão Linear e Apriori
Avaliadores de qualidade	Accuracy, Precision/Recall, F-mesure, MSE, RMSE, MAE e o R^2
Processamento de texto	Vetorização por Bag-of-Words e Tf-idf, transformação de sentenças em *tokens* e remoção de *stop-words*
Grafos	PageRank

Uma vez implementados, tais algoritmos e operações foram inseridos em uma biblioteca COMPSs disponível pelo Lemonade. Essa biblioteca será utilizada pelas aplicações COMPSs criadas pelo Lemonade como um módulo externo. Uma das vantagens dessa adoção é que não é necessário a substituição de uma aplicação em caso de uma re-estruturação de um algoritmo da biblioteca.

4.2 Integração e geração de código

Além da implementação, foi necessário a integração dos algoritmos e operações ao Lemonade. Essa etapa compreendeu o registro dos metadados dos mesmos no Tahiti para que o componente Citron (responsável pela interface *web*) exiba as caixas de operações suportadas na plataforma COMPSs e a criação de um interpretador do fluxo de operações em código-fonte COMPSs para o componente Juicer. Além disso, foi necessário a criação de um orquestrador também no componente Juicer, chamado de *minion*, para a submissão e execução das aplicações na plataforma COMPSs.

Em relação ao Tahiti, não houve grandes mudanças no que foi feito para a plataforma Spark, isto é, não precisou criar ou alterar nenhuma tabela do banco de dados responsável em armazenar os metadados. Basicamente, essa etapa consistiu em popular o banco do dados do Tahiti com as informações das operações e algoritmos desenvolvidos em COMPSs. Cada algoritmo e operação foi identificado internamente por um identificador mesmo que tal algoritmo já fosse disponível para o Spark. Tal decisão foi tomada principalmente ao constante desenvolvimento do Lemonade, dessa forma, no futuro, uma alteração feita para a plataforma Spark não impactará na plataforma COMPSs.

Já em relação ao componente Juicer, foi necessária a implementação de um novo *Transpiler* para interpretação dos fluxos, representados em *JSON*, para código-fonte PyCOMPSs. Além disso, foi alterado o método de iniciação do processo de tradução, adicionando uma etapa inicial de verificação da plataforma alvo do usuário, podendo ser Spark ou COMPSs.

O código-fonte a ser criado pelo tradutor segue um modelo *top-down* semelhante ao realizado na versão Spark. As instâncias das funções se encontram no topo do código, já as suas chamadas no final do código, além disso, foi adotado o mesmo modelo de registro de *logs* usado para a versão Spark. Isso foi necessário para garantir compatibilidade lógica com os componentes, como o Stand, Caipirinha e o Juicer. O Stand, durante a execução da aplicação, precisa consumir algumas informações geradas durante o monitoramento da execução para a atualização dos status da execução exibido na interface (Citron).

Ao iniciar um processo de tradução, uma vez identificado que se trata de um *workflow* COMPSs, o tradutor fará uma ordenação topológica no campo *Flow* do arquivo *JSON*. Como na dito seção 2.3, tal campo contém a relação entre as caixas de operações do Lemonade e pode ser interpretado como uma lista de arestas de um grafo, onde cada vértice é uma operação. A ordenação topológica a ser realizada nos dará uma ordem de dependências entre as tarefas presentes no fluxo de operações. Como

resultado principal dessa ordenação temos a atribuição de um número identificador que, concatenado à um prefixo (p.ex., "data"), garante a relação lógica e a unicidade entre os nomes das variáveis de entrada e saída de cada operação a ser executada.

Terminada a ordenação, o processo de tradução é iniciado. Existirá uma função no código-fonte para cada caixa de operação do fluxo. Cada função criada também possuirá o identificador concatenado ao nome da função para garantir que uma caixa possa ser usada mais de uma vez em um fluxo. E por fim, o resultado da tradução é salva em um arquivo executável; no caso da plataforma COMPSs, em um arquivo '.py'.

Códigos em COMPSs criados pelo Lemonade podem ter uma quantidade de tarefas maior do que em um código implementado manualmente. Algumas caixas Lemonade, em que seu DAG contenha apenas um estágio de tarefas, como a leitura de dados ou a seleção de colunas, podem ser ser agrupadas com as tarefas subsequentes, semelhante ao conceito de *stages* adotado pelo Spark. O impacto disso em tempo de execução será avaliado no próximo capítulo.

Uma vez criado o código-fonte, a aplicação pode ser submetida pelo próprio Juicer a partir da infraestruturada fornecida pelo Lemonade. Para isso, o orquestrador *Minion* cria um arquivo de configuração COMPSs contendo os detalhes das infraestrutura que a aplicação deverá ser executada. Embora a execução pudesse ser feita em diversos tipos de infraestrutura, como local ou em *cluster*, decidimos por causa da origem do Lemonade, a utilização de uma infraestrutura de nuvem, utilizando *containers* Docker com uma imagem COMPSs previamente configurada e utilizando recursos providos pelo Lemonade a partir do Mesos. O arquivo de configuração criado contém informações como: o nome da imagem Docker, o número mínimos e máximos de *containers* a serem instanciados e os requisitos de *hardware* necessário. Dessa forma, o Lemonade é capaz de lidar com a execução de várias aplicações simultaneamente, uma vez que os recursos computacionais podem ser gerenciados pelo Mesos de forma transparente.

Capítulo 5

Avaliação experimental

Neste capítulo avaliamos as integrações do COMPSs com o HDFS e com o Lemonade comparando-os com casos de uso sem tais integrações. Podemos dividir os experimentos em dois eixos, os relacionados ao HDFS e os relacionados ao Lemonade.

Avaliando a integração do HDFS, fizemos uma subdivisão para avaliar o desempenho de leitura e de escritas de dados no HDFS separadamente. As hipóteses que buscamos verificar foram as seguintes:

1*a*. As soluções em Java ou em Python utilizando o HDFS como o sistema de arquivos possuem tempos de execução significativamente diferentes das soluções existentes;

2*a*. Como o HDFS foi construído para o gerenciamento de grandes dados, à medida que se aumente a carga de trabalho, a integração desenvolvida se tornará mais eficiente;

3*a*. A localidade dos dados tem uma importância significativa quando avaliada uma aplicação COMPSs utilizando o HDFS como sistema de arquivos;

4*a*. A escrita dos dados no HDFS, em cenários semelhantes ao modo convencional, não diminui o desempenho significativamente.

Já em respeito ao Lemonade, a hipótese avaliada foi:

1*b*. O tempo de execução de uma aplicação codificada diretamente pode ser significativamente diferente do tempo de uma aplicação gerada automaticamente pelo Lemonade.

5.1 Planejamento dos experimentos

Para avaliar tais hipóteses, foram utilizados cinco aplicações: WordCount, Grep, Workflow, kNN e ReadAndSave. Esse conjunto abrange aplicações mais simples, com pouco processamento envolvido (como Grep), aplicações com várias etapas de tarefas (como Workcount) e aplicações com muito processamento envolvido (como Workflow e kNN). Tais cenários foram implementados tanto utilizando a API de integração do HDFS quanto utilizando o sistema de arquivos convencional e, em alguns casos, que serão mencionados a seguir, criados ou não pelo Lemonade.

O primeiro algoritmo, WordCount, implementado em Python e em Java, é um simples conta palavras em uma base textual. Sua implementação e seu DAG de execução em PyCOMPS foram mostradas inicialmente em figura 2.1 utilizando o sistema de arquivos convencional, e na figura 3.1, utilizando o HDFS. Esse caso de uso foi escolhido por ser uma aplicação onde o tempo de execução é basicamente relacionado ao tempo de leitura e de interação sobre os dados lidos além de envolver muitas etapas para junção dos resultados parciais.

Grep, implementado em Python e em Java, é uma aplicação para pesquisar pelas ocorrências de um determinado padrão. Embora seja uma aplicação trivial, foi escolhida porque é embaraçosamente paralela e, diferentemente do WordCount, não requer uma etapa de junção de resultados parciais. O tempo de execução dessa aplicação está relacionado principalmente com o custo de leitura do arquivo de dados.

O Workflow é um exemplo de fluxo de operações que pode ser criado pelo Lemonade, mostrado inicialmente na figura 2.1, composto por múltiplas operações e algoritmos. Tal aplicação contém o fluxo mínimo necessário para a classificação em mineração de dados e é um exemplo de execução esperada com o uso do Lemonade. Consiste em uma etapa de pré-processamento e uma de classificação de dados, utilizando nesse exemplo, um classificador Naive Bayes. Essa aplicação será útil para avaliar as capacidades do Lemonade, uma vez que tal fluxo envolve uma série de operações que poderiam ser otimizadas em uma implementação manual seguindo as boas praticas de implementação COMPSs. Como a biblioteca de algoritmos que propomos para o Lemonade é em linguagem Python, reproduzir a aplicação Workflow em Java fugiria do escopo do trabalho. Sendo assim, essa aplicação foi utilizada apenas para a linguagem Python.

Por causa disso, optamos por utilizar outra aplicação como substituição ao Workflow para a linguagem Java. No lugar, foi escolhido o *k-nearest neighbors* (kNN), um algoritmo que embora sua implementação seja simples, é computacionalmente complexo. O kNN é um algoritmo de mineração de dados usado para classificação de

elementos capaz de inferir a classe de um elemento com base na classe majoritária entre seus k vizinhos mais próximos já computados [Zaki & Meira Jr., 2014]. A aplicação exige dois arquivos como entrada, um com a base de dados já rotulada e outro com os elementos a serem classificados. A implementação em COMPSs consiste em usar cada *core* disponível no *cluster* para ler uma parte do arquivo a ser classificado. A métrica de proximidade dessa implementação é a distância euclidiana. Dentro de cada partição, k vizinhos mais próximos são computados de maneira distribuída para cada ponto da entrada de teste e, por fim, são combinados e processados, produzindo a classificação final baseado na classe majoritária.

O ReadAndSave é uma aplicação genérica de cópia de conteúdos de arquivos. Escolhemos utilizar uma aplicação desse tipo para poder analisar melhor o desempenho das aplicações durante a escrita de dados. Como a aplicação envolve pouco processamento, ela é útil para poder isolar o tempo de escrita em relação ao de leitura e processamento. Nesse exemplo, cada tarefa lê um fragmento do arquivo de entrada mas também é responsável pela escrita desse mesmo dado em um novo arquivo. Na versão em que se utiliza o sistema de arquivos convencional, o COMPSs é responsável pela transferência dos arquivos criados em cada tarefa para o computador central ao final de cada execução. Já quando se utiliza o HDFS, cada *worker* é capaz de escrever diretamente no sistema distribuído. Ambas as versões leem dados diretamente do HDFS, o que muda é apenas o modo de escrita de dados.

Nem todos as aplicações possuem tipos de entradas de carga iguais, o WordCount e Grep, por exemplo, impõe cargas de entrada textuais, já o Workflow e kNN utilizam dados numéricos. Embora o ReadAndSave aceitasse qualquer tipo de entrada pois é uma simples aplicação de cópia, decidimos padronizar a utilização de apenas cargas textuais para esses experimentos. Foi utilizado o Data Set Higgs[1] para a carga de dados numéricos, que contém, em sua base original, 11×10^6 amostras com 28 dimensões, derivados de processos de sinal simulado que produzem os Bosons de Higgs. Já a carga de texto foi criada a partir da junção de vários livros no formato *txt* disponibilizados pelo Projeto Gutenberg[2]. É interessante observar que todas as aplicações não impõem condições dependentes da base para um melhor ou pior caso de execução, isso permitiu, para fim de testes de escalabilidade, experimentos com a diminuição ou aumento por concatenação de novos elementos.

As avaliações de desempenho foram realizadas aplicando três técnicas experimentais: o teste pareado, a regressão linear e o projeto fatorial 2^kr. A primeira técnica foi utilizada para avaliar as hipóteses $1a$, $1b$ e $4a$. A regressão linear foi utilizada

[1] disponível em: `https://archive.ics.uci.edu/ml/datasets/HIGGS`

[2] `http://www.gutenberg.org`

Tabela 5.1: Planejamento dos experimentos

Cenário	Técnica aplicada	Motivo de avaliação	Algoritmos
JAVA e HDFS	Projeto $2k^r$	A importância da localidade na leitura dos dados	Grep
	Teste Pareado Z	A diferença de performance entre os sistemas durante a leitura dos dados	Grep, WordCount, kNN
	Regressão	A possível existência de um limiar onde ambos sistemas são semelhantes	Grep
Python e HDFS	Teste Pareado Z	Se os sistemas, para a leitura dos dados, são significativamente diferentes	Grep, WordCount, Workflow
	Regressão	A existência de um limiar onde ambos sistemas são semelhantes	Grep
	Teste Pareado Z	A diferença de performance entre os sistemas durante a escrita dos dados	ReadAndSave
Lemonade	Teste Pareado Z	A diferença de performance entre os sistemas	Workflow

especificamente na hipótese $2a$. Já o projeto fatorial foi utilizado em $3a$. Como a integração com o HDFS foi elaborada em duas linguagens diferentes, foram feitos, quando pertinente, testes em ambos casos. A tabela 5.1 sumariza os de experimentos realizados.

Salvo em situações específicas, identificadas posteriormente, todos os experimentos utilizaram um *cluster* COMPSs/HDFS com um nó *master* dedicado e oito nós *slaves/workers*. Utilizamos a mesma infraestrutura de *cluster* em todos os experimentos, mesmo para as aplicações criadas no Lemonade, para poder comparar os resultados entra as aplicações. Nesses casos, as aplicações eram criadas e convertidas em código-fonte '.py' pelo Lemonade mas eram executadas manualmente no *cluster* mencionado. As máquinas possuem processadores Intel E56xx de 2,5 GHz com 4 cores e 8 GB de RAM executando Ubuntu Linux 16.04 LTS. O Hadoop/HDFS, versão 2.7.4, foi configurado com um fator de replicação igual a três.

5.2 Impacto da utilização do HDFS

Nesta seção, são mostrados e discutidos os resultados do estudo do impacto da utilização do HDFS em aplicações COMPSs. Tais resultados foram divididos em três partes: avaliação do impacto das duas abstrações de leitura providas pela API de integração do HDFS, avaliação da diferença de desempenho entre execuções COMPSs que leem dados do sistema de arquivos convencional e aquelas que leem dados do HDFS, e por fim, avaliação da diferença de desempenho ao se escrever dados no HDFS. Das formas disponíveis pela API, a que mais se adapta ao contexto deste trabalho (isto é, cenários de *big data*) é a qual são escritos diversos arquivos de saída, um para cada fragmento. Sendo esta, a escolhida para análise.

5.2.1 Estudo do impacto da localidade dos dados

Quando avaliado o impacto da localidade dos dados, objeto de questionamento da hipótese $3a$, duas principais questões foram levantadas: qual é a importância da localidade dos dados, e como o desempenho se comporta em outros cenários, por exemplo, com conjuntos de dados maiores. Para tentar responder a primeira pergunta foi realizado um projeto fatorial $2^k r$ e para a segunda foi realizado uma regressão linear para avaliar o tempo de execução em função do aumento da carga de trabalho.

Para o primeiro experimento, foi executado o algoritmo Grep em Java com versões utilizando a API de integração do HDFS a abstração que explora a localidade dos dados e outra com a abstração sem localidade, dez vezes em cada configuração possível do projeto fatorial. Para esse cenário de experimento, foram utilizadas duas configurações, onde a primeira continha um *master* e quatro *slaves* e a segunda um *master* e oito *slaves*, ambas com o fator de replicação igual a um. Foi escolhido também utilizar arquivos de 4 e 8 GB, proporcionalmente ao aumento do número de nós. A tabela 5.2 sumariza as escolhas feitas.

Tabela 5.2: Configurações dos parâmetros para os projetos fatoriais $2^3 10$

Parâmetro	Limite inferior	Limite superior
Número de *workers* (A)	4	8
Tamanho do arquivo em GB (B)	4	8
Interface da API (C)	s/ localidade	c/ localidade

Projetos fatoriais assumem, dentre outras suposições, que os erros possuem uma distribuição normal, apresentando um desvio padrão constante (*homoscedascity*) e são

independentes. Com isso, apesar de um modelo aditivo de projeto fatorial poder se encaixar no projeto realizado, os melhores resultados bem como a explicação da variação e a validação das premissas necessárias para o projeto tiveram melhores resultados em um modelo multiplicativo. Isso faz sentido quando temos como base que a relação processamento e carga de trabalho não se somam, mas sim, se multiplicam. A tabela 5.3 contém o resultado dessa etapa, mostrando os efeitos, alocação das variações, intervalos de confiança, porcentagem explicada e desvio padrão dos efeitos.

Tabela 5.3: Resultado do projeto fatorial $2^3 10$

Fator	Efeito	Variação Explicada (%)	Intervalo de 90% de confiança
A	-0,0569	18,27	(-0,0699, -0,0439)
B	0,0866	42,27	(0,0736, 0,0996)
C	-0,0749	31,65	(-0,0879, -0,0619)
AB	-0,0032	0,06	(-0,0161, 0,0098) †
AC	-0,0183	1,88	(-0,0313, -0,0052)
BC	-0,0049	0,14	(-0,0179, 0,0080) †
ABC	-0,0020	0,02	(-0,0149, 0,0110) †
Efeitos (%)		94,28	

†, parâmetros não significativos

A teoria da utilização dos intervalos de confiança nos diz que um parâmetro é significativo se o seu respectivo intervalo de confiança não contiver o valor zero. Podemos ver que os três fatores A, B e C são significativos, com 90% de confiança. Se somados os três, eles são responsáveis por cerca de 92,19% da variação do tempo de execução, sendo o fator B (tamanho dos dados) o mais impactante, como era de se esperar. É interessante observar que depois dele, o fator C (versão da API) mais impactante, isto é, utilizar versão da API que explora a localidade dos dados beneficia em aproximadamente 31,65% o tempo de execução. Já o número de nós (A), é responsável por explicar 18,27% da variação. Todas as interações entre os três fatores tiveram resultados não significativos, com exceção da iteração entre A e C (número de nós e localidade dos dados). Tal interação representa uma variação de 1,88% no desempenho do sistema; uma possível interpretação é que, ao se aumentar o número de nós, mais transferências de arquivos serão necessárias durante uma execução. No entanto, ao se explorar a localidade dos dados, há uma diminuição significativa de transferências quando comparado à primeira versão sem localidade e, com isso, uma diminuição no tempo de execução.

Devido ao atendimento às premissas do projeto fatorial, mostrada no apêndice

A.1, a conclusão obtida pela análise da Tabela 5.3 pode ser levada em consideração e, de fato, os resultados parecem consistentes com as características do problema em questão. Assim, é fácil entender que o fator mais impactante no tempo de execução de um programa seja a quantidade de dados processados; entretanto, o fato de não precisar transferir partes dos arquivos entre as diversas máquinas (quando se usa localidade) é um outro fator muito importante.

Para fazer um estudo mais específico do impacto da localidade dos dados durante as execuções decidimos fazer uma regressão simples para as duas versões da API. Os experimentos consistiam na execução do algoritmo Grep, desta vez fixando o número de nós (*workers*) em oito e variando o tamanho dos arquivos de 2 GB a 12 GB, também com um fator de replicação igual a um. Cada configuração foi executada dez vezes, e calculamos a média. Calculamos o intervalo de confiança e o coeficiente de determinação das regressões para analisar a veracidade dos resultados obtidos. A tabela 5.4 sumariza os resultados e a figura 5.1 contém a representação visual dos resultados.

Tabela 5.4: Localidade dos dados: análise das regressões lineares (y = αx+β)

Informação	sem localidade	com localidade
Parâmetro α	0,0042	0,0023
Parâmetro β	6,7111	7,1444
Intervalo de 90% de Confiança de α	(3,6854, 9,7367)	(5,7584, 8,5304)
Intervalo de 90% de Confiança de β	(0,0042, 0,0042)	(0,0023, 0,0023)
R^2	0,9705	0,9787
Escala de x	Mega bytes	Mega bytes
Escala de y	segundos	segundos

Pela Tabela 5.4, podemos verificar que a API com localidade é quase duas vezes mais rápida que a primeira, sendo tal resultado corroborado com as afirmações feitas no projeto fatorial. Podemos ver que embora o parâmetro β da versão sem localidade seja um pouco menor que o da versão com localidade, como os intervalos de confiança se sobrepõem, não se pode dizer que sejam significativamente diferentes, embora tal hipótese faça sentido, pois espera-se que exista um custo inicial de orquestrar as tarefas na versão com localidade. Devido a satisfatoriedade de todos os parâmetros serem significativos, por ser uma regressão linear simples, e porquê os coeficientes de determinação são ótimos, podemos afirmar que tais regressões criadas são capazes de explicar significativamente as variações do tempo de execução sem a necessidade de um

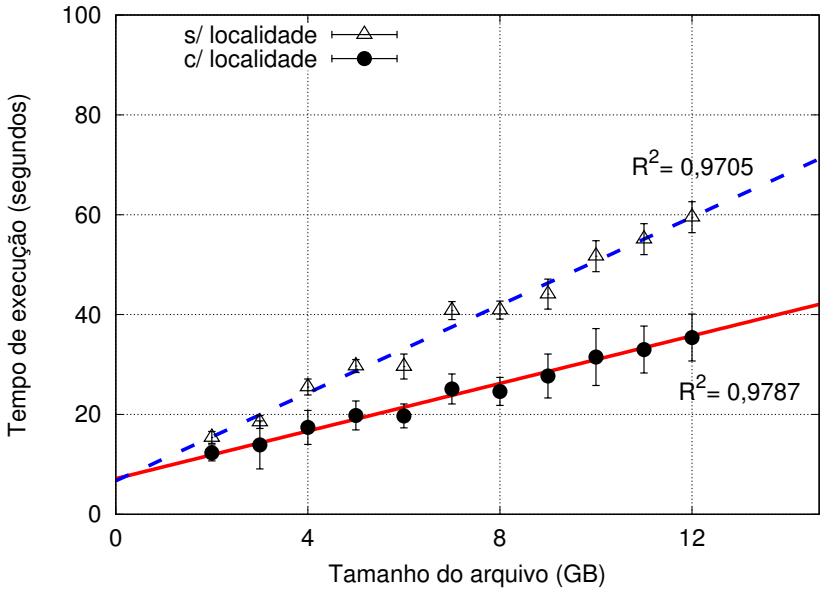


Figura 5.1: Localidade dos dados: tempo de execução do Grep em função do tamanho do arquivo de entrada

teste-F[3]. Com isso, temos que os resultados obtidos são condizentes e representam bem os dados amostrais coletados. As regressões criadas são significativamente diferentes e podem ser utilizadas para análises de desempenho.

5.2.2 Estudo do impacto do HDFS na leitura de arquivos

Quando avaliado o impacto do HDFS na leitura de dados, objeto de questionamento das hipóteses $1a$ e $2a$, duas questões foram levantadas: ler dados do HDFS muda significativamente o tempo de execução de uma aplicação? e à medida que se aumenta a quantidade de dados, essa diferença aumenta? Cada pergunta foi respondida utilizando uma técnica: um teste pareado com uma replicação de 40 vezes para cada algoritmo para a análise da primeira pergunta e aplicações de regressão linear para estimar o comportamento do tempo de execução em função da carga de trabalho para responder a segunda questão.

Para a primeira parte, foram executados três algoritmos (Grep, WordCount e kNN) em Java e outros três (Grep, WordCount e Workflow) em Python, todos com versões com e sem o HDFS. Neste estudo foi utilizado apenas a versão da API que não explora localidade dos dados pois, como já vimos na seção anterior, é a versão mais lenta e por conseguinte os resultados obtidos podem ser utilizados para indicar

[3]Utilizado para avalair a existência de uma relação entre a variável dependente e a independente.

que a versão mais lenta ainda será superior quando comparada à versão convencional. A quantidade de repetições utilizada em cada algoritmo permitiu a realização de um teste pareado utilizando o coeficiente Z (teste pareado Z). Todas as execuções a seguir foram executadas com um nó *master* e oito nós *workers* e, dessa vez, uma configuração padrão do HDFS, isto é, com fator de replicação igual à três. A tabela 5.5 contém os parâmetros de entrada de cada conjunto de execuções, já as tabelas 5.6 e 5.7 contêm os resultados do cálculo do intervalo de confiança de 90% de um-lado do teste pareado na escala de segundos.

Tabela 5.5: Leitura de arquivos no HDFS: relação das configuração dos algoritmos

Algoritmo	Tipo de Arquivo	Número de linhas	Tamanho do(s) arquivo(s)
Grep	Texto	344 mil	8 GB
WordCount	Texto	344 mil	8 GB
K-NN	Numérico (28-dim)	10 mil (treino) e 18,5 milhões (teste)	250 KB e 8 GB
Workflow	Numérico (28-dim)	18,5 milhões	8 GB

Tabela 5.6: Leitura de arquivos no HDFS: teste pareado Z entre versões em Java (tempo em segundos)

Informação	Grep	WordCount	kNN
Tamanho da amostra	40	40	40
Média das diferenças	-69,9	-71,9	-119,3
Desvio padrão das diferenças	11,9	11,5	43,6
Intervalo de 90% de confiança de um-lado	$(-\infty, -67,5)$	$(-\infty, -69,6)$	$(-\infty, -110,4)$
Tempo de execução médio sem HDFS	100	125	228
Tempo de execução médio com HDFS	30	55	158
Speedup	3,33	2,27	1,45

De modo geral, foram observados melhores desempenhos para os três algoritmos analisados em linguagem Java. Para o Grep, o intervalo de confiança de um-lado

$(\text{-}\infty, -67,47)$ nos diz que uma aplicação utilizando o HDFS pode ser por volta de 67 segundos mais rápida que a convencional. O WordCount obteve resultados similares mas com um *speedup*[4] menor que o observado no algoritmo Grep. Por fim, o kNN obteve um desvio padrão maior devido ao tempo de cada execução ser em média sete vezes maior que os outros dois algoritmos, justificado pela sua maior ordem de complexidade. No entanto, apesar de grande parte do tempo ter sido devido ao processamento, a leitura e a transferência dos arquivos impactam no resultado, deixando a versão com o HDFS mais rápida.

Tabela 5.7: Leitura de arquivos no HDFS: teste pareado Z entre versões em Python (tempo em segundos)

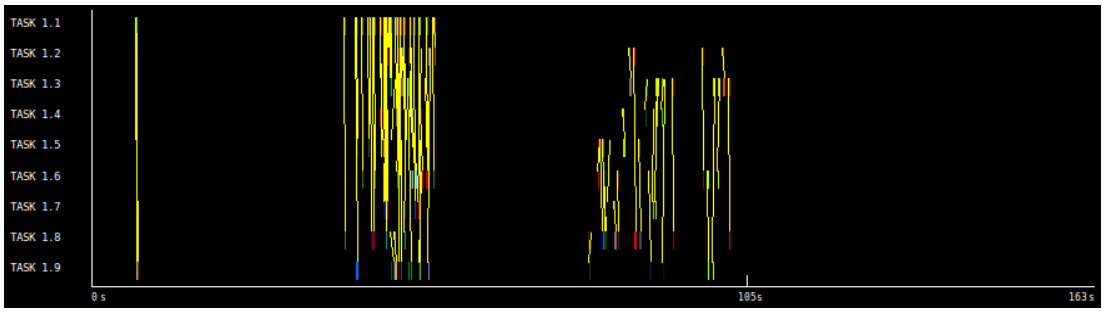
Informação	Grep	WordCount	Workflow
Número de amostras	40	40	40
Média das diferenças	-145,2	-120,5	-173,8
Desvio padrão das diferenças	66,9	54,2	32,1
Intervalo de 90% de confiança de um-lado	$(\text{-}\infty, -131,6)$	$(\text{-}\infty, -109,5)$	$(\text{-}\infty, -167,3)$
Tempo de execução médio com HDFS	36	99	317
Tempo de execução médio sem HDFS	172	211	497
Speedup	4,78	2,12	1,56

De acordo com os resultados da tabela 5.7, os algoritmos implementados em Python utilizando o HDFS mostraram também um desempenho superior às respectivas versões usando arquivos tradicionais. A média das diferenças do tempo de execução de ambas versões do Grep foi de 145 segundos, entretanto, considerando os tempos médios de execução, o Grep em Python utilizando o HDFS obteve um *speedup* de 4,78. Assim como o observado na versão em Java, o WordCount também obteve um *speedup* menor em relação ao algoritmo Grep, isso é devido ao fato que o Grep é um algoritmo de apenas um estágio, enquanto que o WordCount possui etapas de agregação dos dados, amortizando assim o ganho de desempenho. O intervalo de confiança de um-lado indica que a versão com HDFS pode ser mais de 109 segundos mais rápido do que a versão sem utilizar o HDFS. É interessante observar que esse ganho ao se utilizar o HDFS na aplicação Workflow foi menor que o obtido para o WordCount; entretanto, isso pode ser

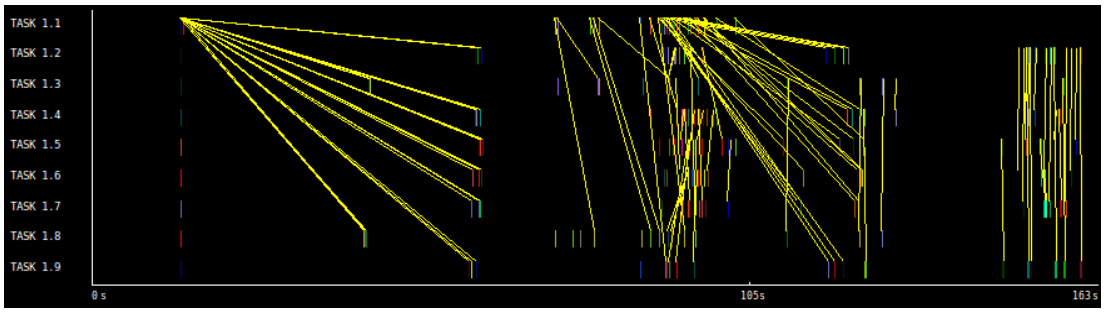
[4]Métrica derivada dos tempos médios que representa quantas vezes uma execução é mais rápida em comparação com outra.

explicado considerando-se que tal aplicação possui uma carga maior de processamento envolvido, assim como o kNN da versão Java.

As figuras 5.2a e 5.2b mostram *traces* de mapeamento das transferências de dados entre os nós do *cluster* gerados pelo COMPSs durante a execução do WordCount em Python com e sem a integração com HDFS. As linhas representam a transferência de arquivos durante a execução das aplicações, cada tarefa (*task*) presente no eixo vertical dos *traces* representa um computador. A tarefa 1.1, no topo do eixo vertical, representa o computador central (o que orquestra e envia os dados) e as demais representam os oito *workers* utilizados na execução. Tais *traces* nos ajudam a entender um dos motivos para o COMPSs com HDFS se mostrar mais rápido que a versão utilizando o sistema de arquivos convencional.



(a) WordCount com HDFS: aplicação termina com menos de 105 s



(b) WordCount sem HDFS: aplicação termina por volta de 163 s

Figura 5.2: Leitura de arquivos no HDFS: *traces* da transferência de dados durante a execução do WordCount

Na figura 5.2b, o *master* é o responsável pela transferência de todos os arquivos de dados (linhas inclinadas iniciando na tarefa 1.1), assim, parte do tempo de execução do algoritmo é devido a espera pela transferência desses arquivos. As transferências vistas aproximadamente na metade da execução são os resultados parciais que foram gerados e que precisam ser agrupados; isso não é realizado exclusivamente pelo *master*, mas sim pelos nós que produziram tais resultados. Esse comportamento é o esperado nas versões utilizando o HDFS, pois nesse cenário o COMPSs transfere apenas uma

lista contendo informações sobre os fragmentos dos arquivos que cada nó deverá acessar e o HDFS é quem vai transferir os dados dos diversos blocos para os nós.

A respeito da aplicação da regressão linear, na tentativa de responder a hipótese $2a$, objetiva-se verificar se existe algum limiar onde usar HDFS passa a ser melhor ou pior que não usá-lo. Para isso, executamos o Grep em Python com diferentes tamanhos de arquivos e realizamos duas regressões lineares, uma para mapear o tempo de execução de uma aplicação utilizando o HDFS e outra sem o seu uso. Escolhemos utilizar apenas o Grep como base desse experimento, pois é o que possui o tempo de execução mais dependente do tempo de leitura. O resultado obtido nesse cenário pode representar um indicativo para as outras classes de algoritmos. As regressões foram obtidas coletando o tempo de execução de cada aplicação para diversos tamanhos de arquivo, de 2 GB a 16 GB, aumentando 1 GB a cada cenário e repetindo quinze vezes.

O gráfico da figura 5.3 contém o resultado da aquisição de dados desse experimento. Pode ser observado que ambas as versões obtiveram um tempo de execução semelhante para cargas de trabalho menores de 6 GB. No entanto, há um aumento expressivo do tempo de execução para a aplicação sem a utilização do HDFS no intervalo de arquivos de 6 GB a 8 GB. O desvio padrão calculado para esses pontos nos mostra que, nesse intervalo, as execuções variaram bastante. Analisando os experimentos, percebemos que essa variação está relacionada ao tempo de transferências dos arquivos para cada *worker*. Essa afirmação é verdadeira pois não foi encontrado nenhuma relação dessa variação observada quando confrontado com o tempo de execução interno de cada tarefa. Para os experimentos realizados, quanto maior o arquivo de entrada, maior a quantidade de fragmentos a serem transferidos, por consequência, maior o tempo médio para a transferência. Acreditamos que a região de 6 GB a 8 GB, é uma área de mudança de comportamento do sistema, em determinadas execuções, há casos onde o COMPSs consegue escalonar a transferência dos arquivos de forma ótima, no entanto, há outros casos onde o tempo de médio de transferência fica alto devido ao *overhead* das múltiplas conexões. Por exemplo, no *trace* da figura 5.2b mostrada anteriormente, vemos que naquela execução, o computador central conseguiu transferir alguns arquivos muito antes dos demais, já em outras execuções, o *trace* poderia ser outro, influenciando no tempo da execução. É interessante observar que após esse intervalo, as execuções nesse mesmo cenário passam a ter um comportamento similar, com um desvio padrão menor e com um aumento de tempo mais próximo do linear. Com isso em mente, foi decidido a definição das regressões com base nos dados coletados com carga de trabalho acima de 8 GB, sendo essa análise válida, uma vez que objetivamos cenários com cargas maiores. A figura 5.3 ilustra as médias calculadas com seus desvios padrão para cada cenário e as regressões obtidas. Considerando o

intervalo utilizado para a criação da regressão, fica claro o melhor desempenho do HDFS, indicação de que, em configurações sem o uso de disco compartilhado, o custo de envio de múltiplos arquivos e/ou arquivos pesados é significativo. Com a utilização do HDFS, o custo de transferência dos arquivos é amortizado entre as máquinas do *cluster* HDFS.

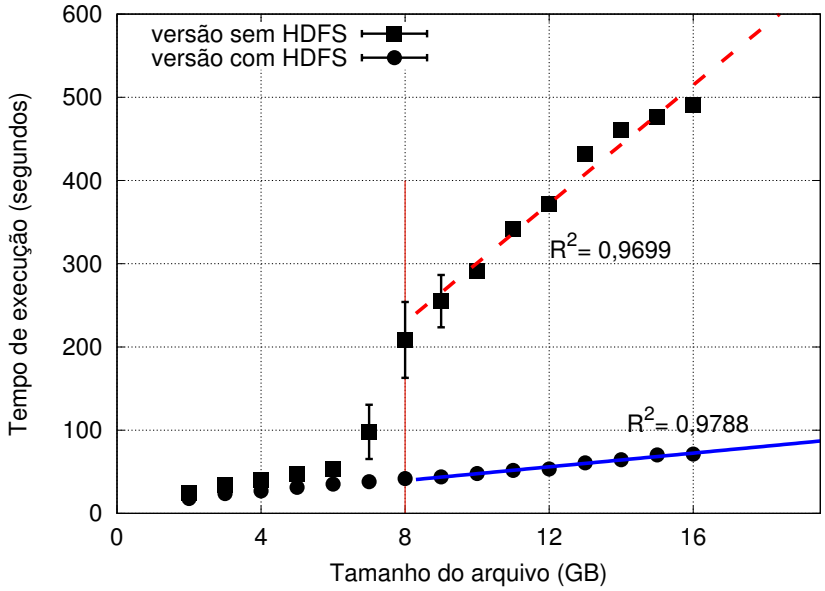


Figura 5.3: Leitura de arquivos no HDFS: tempo de execução do Grep (PyCOMPSs) em função do tamanho do arquivo

A tabela 5.8 contém uma sumarização dos resultados obtidos onde, para cada uma das regressões, calculamos seus parâmetros e os respectivos coeficientes de determinação. Além disso, calculamos o intervalo de confiança de cada um dos parâmetros para verificar o quão significativos eles são para as regressões. O intervalo de confiança apresentado nessa tabela nos diz que o parâmetro β de ambas regressões não são significativos, isto é, eles não representam bem o coeficiente linear da reta de ambas regressões. No entanto, os coeficientes angulares (parâmetro α) são significativos, além disso, as duas regressões satisfazem as premissas de normalidade e de independência dos erros que são mostradas no apêndice A.2, o que torna válidos os resultados discutidos nesse estudo. Podemos ver ainda pela tabela 5.8 que os altos coeficientes de determinação de ambas as regressões indicam que o trecho avaliado foi bem representado pelas regressões. Além disso, o coeficiente angular da regressão "com HDFS" foi aproximadamente dez vezes menor que o da regressão "sem HDFS", um indicativo que utilizar o HDFS torna o sistema mais estável ao aumento da quantidade de carga de trabalho.

Tabela 5.8: Leitura de arquivos no HDFS: análise das regressões lineares (y = αx+β)

Informação	com HDFS	sem HDFS
Parâmetro α	0,0040	0,0347
Parâmetro β	16,4363	-55,5119
Intervalo de 90% de Confiança de α	(0,0040, 0,0040)	(0,0347, 0,0347)
Intervalo de 90% de Confiança de β	(-4,7818, 17,8056)†	(-172,4359, 61,4121)†
R^2	0,9788	0,9699

†, parâmetros não significativos

5.2.3 Estudo do impacto da escrita de arquivos

Com o objetivo de avaliar o impacto do uso do HDFS para a escrita de dados foi realizado um estudo comparativo para verificar se a adição do HDFS torna a execução, em termos de tempo de execução, significativamente diferente das execuções utilizando o sistema de arquivos convencional. Tal estudo envolveu a aplicação do teste pareado Z utilizando o caso de uso ReadAndSave em Python, caracterizado como uma simples aplicação de cópia de arquivos. Tal resultado pode ser projetado para aplicações também em Java, pois desconsiderando diferença entre as linguagens, o HDFS irá operar da mesma forma. Esse teste consistiu na realização de quarenta execuções, para cada versão, da aplicação ReadAndSave com um arquivo de entrada (e por sua vez, de saída) de 8 GB. A tabela 5.9 contém os resultados desse experimento, bem como o intervalo de confiança de um-lado e os tempos de execuções média de ambas versões.

Tabela 5.9: Escrita de dados no HDFS: teste pareado Z entre versões em Python (tempos em segundos)

Informação	ReadAndSave
Número de amostras	40
Média das diferenças	-16,1
Desvio padrão das diferenças	6,9
Intervalo de 90% de Confiança de um-lado	(-∞, -14,7)
Tempo de execução média (com HDFS)	91
Tempo de execução média (sem HDFS)	107
Speedup	1,17

A tabela 5.9 nos diz que não só a versão utilizando o HDFS é significativamente

diferente, com 90% de confiança, da versão convencional, mas também é mais rápida. Mesmo que tal diferença de tempo em termos relativos seja pouco (isto é, 1,17 mais rápido) o ganho apresentado em cenários maiores e aplicações mais custosas pode vir a ser importante. Além disso, desconsiderando o desempenho em tempo de execução, a utilização do HDFS tem vantagens superiores ao sistema convencional, como a distribuição do armazenamento entre as máquinas do *cluster* e a replicação para garantir a disponibilidade dos dados.

5.3 Impacto da integração com o Lemonade

Para o questionamento da hipótese 1*b*, analisamos a diferença de desempenho entre aplicações criadas pelo Lemonade com aplicações escritas diretamente pelo programador e cuidadosamente otimizadas. Como já citado, as aplicações COMPSs criadas a partir do Lemonade utilizam funções fornecidas em uma biblioteca estática. Dessa forma, em casos específicos, uma aplicação codificada diretamente, sem utilizar o Lemonade, pode ser criada usando menos sequências de tarefas COMPSs (isto é, com maior otimização). Para o COMPSs, essa otimização significaria uma possível redução do escalonamento de novas operações e uma redução no número de vezes que um arquivo serializado pelo COMPSs terá que ser lido e escrito. Para avaliar esse impacto, realizamos novamente um teste pareado Z, com duas aplicações equivalentes, sendo uma delas construída utilizando o Lemonade e a outra implementada manualmente, de forma mais otimizada. Ambas utilizaram o HDFS e cada uma foi executada 40 vezes. A aplicação escolhida para esse experimento foi o Workflow que, como já visto, possui uma série de tarefas em sequência, sendo assim, um bom exemplo de situação em que pode haver alguma otimização não identificada pelo Lemonade.

Tabela 5.10: Impacto do Lemonade: teste pareado Z (tempo em segundos)

Informação	Workflow
Número de amostras	40
Média das diferenças	-23,2
Desvio padrão das diferenças	15,9
Intervalo de 90% de Confiança de um-lado	$(-\infty, -20, 0)$
Tempo de execução médio (com Lemonade)	317
Tempo de execução médio (sem Lemonade)	292
Speedup	1,08

A tabela 5.10 mostra os resultados obtidos nesse experimento. Podemos ver que para essa configuração, as duas aplicações possuem tempos de execução significativamente diferentes. O intervalo de confiança de um-lado $(-\infty, -20, 0)$ nos diz que uma aplicação codificada diretamente, com atenção para as otimizações possíveis, pode ser por volta de vinte segundos mais rápida que uma aplicação semelhante construída com o Lemonade. No entanto, em termos relativos (*speedup*), a versão otimizada vai ser executada apenas 1,08 vezes mais rápido que a do Lemonade. Logo, essa diferença não é tão impactante ao levarmos em conta os benefícios e praticidades de se utilizar o Lemonade, uma vez que o usuário não precisa mais codificar e se preocupar com a infraestrutura de uma aplicação COMPSs.

5.4 Aplicabilidade da solução

Vimos na seção 2.1 que a implementação de aplicações COMPSs em cenários de *big data*, com a infraestrutura baseada em *clusters* ou nuvem e sem a presença de discos compartilhados, envolvem alguns desafios para um melhor desempenho. Esta seção destina-se a uma avaliação qualitativa da solução tanto no que diz respeito à implementação de aplicações em COMPSs nestes cenários quanto à utilização da API de integração com o HDFS e a criação e execução das aplicações pelo Lemonade.

Quando pensamos nos meios que o COMPSs oferece para a leitura de dados massivos em uma infraestrutura de usualmente relacionada a cenários de *big data* percebemos que tais soluções não são adequadas por dois principais motivos, o armazenamento e custo de transferência pela rede. Em cenários desse tipo, em que é necessário trabalhar com fragmentos de dados para se obter um melhor nível de paralelismo, o melhor meio para a leitura distribuída desse dado é justamente tê-lo já disponibilizado em fragmentos. No entanto, isso não é prático, pois manter o dado dessa forma irá fixar o numero de fragmentos para todas as futuras execuções. Pode-se utilizar uma cópia temporária que, no qual, em uma etapa de pré-processamento, irá fazer uma cópia do arquivo original e fragmentá-lo na quantidade de partes desejadas. No entanto, essa não é a melhor alternativa também pois o custo desse pré-processamento talvez seja muito elevado, seja por manter uma cópia de um arquivo grande, como na ordem de giga ou tera, em um único computador e o próprio custo da divisão dos arquivos em cada inicio de aplicação. Além disso, quanto maior o arquivo de entrada, maior será a quantidade de fragmentos e/ou tamanho de cada fragmento. Esses aumentos podem provocar uma sobrecarga de transferências, do computador central para os nós, gerando assim serializações das tarefas ou congestionamentos das transmissões.

Nesse contexto, a utilização da API de integração do HDFS se mostra necessária. A divisão de dados fornecidos pela abstração de dados por esta API, nos dá a praticidade de ler dados distribuídos pelo HDFS. Podemos escolher o número fragmentos que desejamos a cada novo inicio de execução, sem termos que aplicar nenhuma alteração sobre esse arquivo no sistema, ou nos preocupar com os aspectos operacionais da divisão dos arquivos. Foi visto também nas seções anteriores, que utilizar o HDFS deixa as execuções COMPSs (que precisam ler dados grandes) mais rápidas, isso faz sentido pois o custo de transferência dos fragmentos é dividido entre todas as máquinas que possuem tais dados. Além disso, o HDFS oferece outras vantagens como a tolerância a falhas pela adoção de replicação dos dados.

Embora o COMPSs forneça um modelo de programação que não envolve nenhuma nova sintaxe, ainda é necessário que o programador saiba identificar quais métodos podem ser paralelizados e onde são os pontos necessários para sincronização dos resultados parciais. Além disso, implementar uma aplicação adotando o conceito de blocos, isto é, trabalhar com fragmentos de dados para melhor ajuste de tarefas de *big data*, nem sempre é uma tarefa trivial. Mesmo utilizando a recém criada API de integração do HDFS, a leitura de dados é apenas uma parte de uma aplicação. Acreditamos que a abstração de criar aplicações por meio de operações de arrastar e soltar (*drag-and-drop*) em uma interface visual, como a fornecida pelo Lemonade, facilita muito o desenvolvimento. Dessa forma, não é necessário escrever nenhuma linha de código, basta que o usuário saiba expressar suas aplicações na forma de fluxos para podê-lo criar. Em contrapartida, em uma aplicação como a Workflow, mostrada anteriormente na figura 2.3, foi necessário codificar 420 linhas de código seguindo as boas práticas de implementação COMPSs em Python. Para aplicações maiores, com mais etapas, tanto a quantidade de linhas poderá facilmente ser maior quanto a complexidade das operações. Embora esse aumento de abstração promova uma diferença de performance em relação a uma aplicação manualmente implementada, essa diferença é pouca. Usuários sem conhecimentos de linguagem de programação podem utilizar o Lemonade para a criar códigos COMPSs com desempenhos próximos a aqueles com conhecimentos avançados nessa linguagem.

Capítulo 6

Conclusão

Os avanços nas áreas de HPC e *big data* têm levado à aproximação das ferramentas e técnicas usadas em ambas. No entanto, ainda há grandes oportunidades e desafios para essa convergência, será necessário pesquisas e padronizações sobre as tecnologias de ambos eixos, tanto por parte do setor privado e quanto do acadêmico para criar um caminho mais claro sobre o futuro desse fenômeno. Fica claro no entanto, a importância da interoperabilidade dessas ferramentas, pois nessa era de avanços científicos cada vez mais frequentes, é cada vez mais complexo que uma única ferramenta se especialize em várias técnicas.

Neste trabalho propomos duas novas extensões ao ambiente COMPSs, desenvolvido originalmente para a área de HPC, para aumentar seu desempenho e facilitar sua utilização em aplicações *big data*. Essas extensões permitem sua integração com o sistema de arquivos distribuídos HDFS e com uma ferramenta visual para *Data Analytics*, ambas de código aberto. Acreditamos que tais contribuições ajudarão a popularizar o COMPSs e contribuirão na sua convergência com o mundo *big data*.

Em uma execução COMPSs com um grande volume de dados e sem o uso de discos compartilhados, a leitura e a fragmentação desses dados se tornam um gargalo para a execução. Armazenar tais arquivos já divididos em fragmentos no disco nem sempre é a melhor opção, uma vez que isso fixa o nível de paralelização. Os resultados obtidos claramente demonstram os benefícios introduzidos pela utilização do HDFS. Ela não só é recomendada pela abstração dos dados, que faz com que o usuário não precise se preocupar com tal divisão, como também pelo ganho de desempenho promovido. Por sua vez, apesar de gerar código menos eficiente que um programa cuidadosamente otimizado, o Lemonade possui aspectos como facilidade de utilização, interface de usuário para criação e execução de fluxos utilizando a funcionalidade de arrastar e soltar elementos em uma interface visual que justificam seu uso. Além disso,

programadores iniciantes ou avançados em COMPSs podem utilizar as operações e algoritmos implementados para o Lemonade como uma biblioteca externa para suas aplicações, caso não desejem utilizar o Lemonade diretamente.

6.1 Trabalhos futuros

Como trabalhos futuros relacionados a API de integração do HDFS, mencionamos a avaliação experimental API com a abstração que estendida da Storage API, capaz de explorar a melhor a localidade dos dados, já em desenvolvimento para a linguagem Python. É esperado que as avaliações feitas para a linguagem Java, apresentada neste trabalho, seja semelhante aos que serão obtidos para a API da versão em Python.

No que diz respeito ao Lemonade, consideramos quatro linhas principais para contribuições futuras: o aperfeiçoamento da geração de códigos COMPSs a partir do Lemonade, o suporte de novas operações e algoritmos ao Lemonade, a integração com os demais componentes do Lemonade nao abordados nesse trabalho e a integração do COMPSs com outras ferramentas de *big data*.

Como vimos na seção 4.2, códigos criados pelo Lemonade podem ter uma quantidade de tarefas maior do que um código manualmente otimizado. Em geral, algumas funções em que seu DAG contém apenas um estágio de tarefas e que seus resultados produzidos são entrada para apenas um novo tipo de tarefa, são cabíveis de otimizações agrupando as operações desta tarefa com a da caixa anterior. Para o COMPSs, essa otimização significa reduzir o escalonamento de novas operações e reduzir etapas de leitura/escrita dos resultados parciais de tarefa em arquivos binários. Com a geração de um código COMPSs mais dinâmico por parte do Lemonade, a diferença de desempenho medida na seção 2.3 poderá diminuir ou até mesmo anular.

Neste presente trabalho, implementamos um conjunto de operações e algoritmos de mineração de dados para o suporte do Lemonade e, embora esse conjunto inicial satisfaça muitas aplicações, é necessário expandir tal conjunto para uma melhor experiencia ao usuário. Além disso, é interessante a integração aos demais componentes Lemonade que não foram abordados neste trabalho, como por exemplo, no suporte ao módulo Caipirinha, responsável pela visualização de dados durante a execução, e ao Limonero, responsável por gerenciar as bases de dados disponíveis para as aplicações.

Uma outra linha de contribuição seria a integração com novas ferramentas de *big data*, por exemplo, com a interoperabilidade com novas ferramentas do Ecossistema Hadoop ou até mesmo com a capacidade de execução de códigos criados em outros *frameworks* (p.ex., MapReduce ou Spark) em COMPSs. Esse último envolve muitos

desafios, como por exemplo a criação de um interpretador (*transpiler*) capaz de converter a linguagem de um modelo para o outro. Tal interpretador necessitaria de um *parser* capaz de identificar e de mapear todas as operações envolvidas em uma execução, além disso, devido ao fato de o arquivo de entrada ser um código-fonte de um programador, cada código teria suas singularidades devido ao modo de implementação de cada usuário; com isso, seria necessário o desenvolvimento de uma solução para geração de um código-fonte mais dinâmico, diferentemente do implementado no suporte ao Lemonade neste trabalho.

Referências Bibliográficas

Agneeswaran, V. S.; Tonpay, P. & Tiwary, J. (2013). Paradigms for realizing machine learning algorithms. *Big Data*, 1(4):207–214.

Belcastro, L.; Marozzo, F.; Talia, D. & Trunfio, P. (2015). Programming visual and script-based big data analytics workflows on clouds. *Big Data and High Performance Computing*, 26:18.

Berthold, M. R.; Cebron, N.; Dill, F.; Gabriel, T. R.; Kötter, T.; Meinl, T.; Ohl, P.; Thiel, K. & Wiswedel, B. (2009). KNIME - the Konstanz Information Miner: Version 2.0 and Beyond. *ACM SIGKDD Explorations Newsletter*, 11(1):26–31.

COMP Superscalar (2017). Application development guide. User Manual v2.2, Barcelona Supercomputer Center. Disponível em: `http://compss.bsc.es`. Acessado em 20/06/2018.

Conejero, J.; Corella, S.; Badia, R. M. & Labarta, J. (2017). Task-based programming in COMPSs to converge from HPC to Big Data. *The International Journal of High Performance Computing Applications*, 32:45–60.

Demšar, J.; Curk, T.; Erjavec, A.; Črt Gorup; Hočevar, T.; Milutinovič, M.; Možina, M.; Polajnar, M.; Toplak, M.; Starič, A.; Štajdohar, M.; Umek, L.; Žagar, L.; Žbontar, J.; Žitnik, M. & Zupan, B. (2013). Orange: Data mining toolbox in python. *Journal of Machine Learning Research*, 14:2349–2353.

Fox, G.; Qiu, J.; Jha, S.; Ekanayake, S. & Kamburugamuve, S. (2016). Big data, simulations and HPC convergence. Em *Proceedings of the 6th International Workshop on Big Data Benchmarking (WBDB)*, pp. 3–17, Cham. Springer.

Ghemawat, S.; Gobioff, H. & Leung, S.-T. (2003). The Google File System. Em *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 29–43, New York, NY, USA. ACM.

Gonzales, S. D. (2016). PyWebHDFS: a Python wrapper for the Hadoop WebHDFS REST API. Disponível em: `https://pypi.python.org/pypi/pywebhdfs`. Acessado em 14/12/2017.

Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P. & Witten, I. H. (2009). The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18.

Hall, M. W.; Kirby, R. M.; Li, F.; Meyer, M. D.; Pascucci, V.; Phillips, J. M.; Ricci, R.; van der Merwe, J. E. & Venkatasubramanian, S. (2013). Rethinking abstractions for big data: Why, where, how, and what. *Computing Research Repository (CoRR)*, abs/1306.3295.

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley professional computing. Wiley.

Kamburugamuve, S.; Govindarajan, K.; Wickramasinghe, P.; Abeykoon, V. & Fox, G. (2017). Twister2: Design of a big data toolkit. Em *Proceedings of the 5th Workshop on Exascale MPI (EXAMPI)*, Denver, CO, USA.

Kranjc, J.; Podpečan, V. & Lavrač, N. (2012). Clowdflows: A cloud based scientific workflow platform. Em *Proceedings of the 2012 European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, pp. 816–819, Berlin, Heidelberg. Springer Berlin Heidelberg.

Leo, S. & Zanetti, G. (2010). Pydoop: a Python MapReduce and HDFS API for Hadoop. Em *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 819–825. ACM.

Lezzi, D.; Rafanell, R.; Lordan, F.; Tejedor, E. & Badia, R. M. (2011). COMPs in the VENUS-C Platform: enabling e-Science applications on the cloud. Em *Proceedings of the 4th Iberian Grid Infrastructure Conference*, volume 1, Braga, Portugal. Universidade do Minho.

Martí, J. (2017). *dataClay: next generation object storage*. Tese de doutorado, Universitat Politècnica de Catalunya Barcelona, Barcelona, Espanha.

McKinney, W. (2012). *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*, capítulo Essential Python Libraries, pp. 1–14. O'Reilly Media, Inc., Sebastopol, CA, USA, 1ª edição.

Mierswa, I.; Wurst, M.; Klinkenberg, R.; Scholz, M. & Euler, T. (2006). YALE: Rapid Prototyping for Complex Data Mining Tasks. Em *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pp. 935–940, New York, NY, USA. ACM.

Mund, S. (2015). *Microsoft Azure Machine Learning*, volume 1. Packt Publishing, 1ª edição.

Podpečan, V.; Zemenova, M. & Lavrač, N. (2011). Orange4ws environment for service-oriented data mining. *The Computer Journal*, 55(1):82–98.

Prekopcsak, Z.; Makrai, G.; Henk, T. & Gaspar-Papanek, C. (2011). Radoop: Analyzing big data with rapidminer and hadoop. Em *Proceedings of the 2nd RapidMiner Community Meeting and Conference (RCOMM)*, pp. 1–12.

Ramon-Cortes, C.; Serven, A.; Ejarque, J.; Lezzi, D. & Badia, R. M. (2018). Transparent orchestration of task-based parallel applications in containers platforms. Em *Journal of Grid Computing*, pp. 137–160. Springer Netherlands.

Reed, D. A. & Dongarra, J. (2015). Exascale computing and big data. *Communications of the ACM*, 58(7):56–68.

Rocha, R. C.; Hott, B.; dos Santos Dias, V. V.; Ferreira, R.; Meira Jr., W. & Guedes, D. (2016). Watershed-ng: an extensible distributed stream processing framework. *Concurrency and Computation: Practice and Experience*, 28(8):2487–2502.

Rosen, J. (2016). Pyspark internals. Disponível em: `https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals`. Acessado em 14/12/2017.

Santos, W.; Carvalho, L. F. M.; d. P. Avelar, G.; Silva, A.; Ponce, L. M.; Guedes, D. & Meira Jr., W. (2017). Lemonade: A scalable and efficient spark-based platform for data analytics. Em *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 745–748.

Tejedor, E. & Badia, R. M. (2008). COMP superscalar: Bringing grid superscalar and gcm together. Em *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pp. 185–193. IEEE.

Tejedor, E.; Becerra, Y.; Alomar, G.; Queralt, A.; Badia, R. M.; Torres, J.; Cortes, T. & Labarta, J. (2017). PyCOMPSs: Parallel computational workflows in Python. *The International Journal of High Performance Computing Applications*, 31(1):66–82.

White, T. (2015). *Hadoop: The Definitive Guide*, capítulo The Hadoop Distributed Filesystem, pp. 43–78. O'Reilly Media, Inc., Sebastopol, CA, USA, 4ª edição.

Wu, X. et al. (2008). Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37.

Zaharia, M. et al. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Em *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pp. 15–28, San Jose, CA, USA. USENIX.

Zaki, M. J. & Meira Jr., W. (2014). *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, New York, NY, USA, 1ª edição.

Glossário

Citron Componente do Lemonade responsável por prover a interface *web* para que o usuário crie, execute e monitore o seu fluxo de análise de dados.
COMPSs COMP Superscalar, *framework* de processamento paralelo e distribuído, originado para área de computação de alto desempenho.

DAG Grafo Direcionado Acíclico.
Data Analytics Processo de análise de dados que gera informação para orientar decisões de negócios e testar teorias científicas.

HDFS Hadoop Distributed File System (HDFS) é um projeto de código aberto da Apache Software Foundation de um sistema de arquivos distribuídos.

Juicer Componente do Lemonade responsável pela execução das aplicações.

Lemonade Lemonade (do ingês, Live Exploration and Mining Of a Non-trivial Amount of Data from Everywhere), é uma plataforma *web* para criação e execução de *workflows* descritos visualmente.
libHDFS *Wrapper* em C para a comunicação com o API em Java do HDFS.

Minion Componente interno do Juicer para a submissão e execução das aplicações.

PyCOMPSs Nome dado para aplicações COMPSs criadas em linguagem Python.

Spark Apache Spark, *framework* de processamento paralelo e distribuído, originado para área de *big data*.
Storage API API desenvolvida pelo Centro de Supercomputação de Barcelona, o que permite dentre outras funções, um melhor manuseio de dados persistentes.

Tahiti Componente do Lemonade responsável pelo armazenamento dos metadados dos algoritmos e operações.
Transpiler Conversor de código-fonte escrito em uma linguagem para código-fonte em outra. No Juicer, o *transpiler* é responsável em converter *JSON* em código COMPSs ou Spark.

Apêndice A

Premissas do projeto fatorial e das regressões

No capítulo 5, aplicamos a metodologia do projeto fatorial 2^kr para estimar o impacto de explorar a localidade dos dados em execuções COMPSs utilizando o HDFS. Os resultados obtidos são apresentados e discutidos na seção 5.2.1. Além do projeto fatorial, aplicamos a metodologia da regressão linear, na seção 5.2.1 e 5.2.2. No primeiro caso, buscamos um estudo mais específico do impacto da localidade dos dados em execuções COMPSs utilizando o HDFS em função do tamanho do arquivo de entrada e para avaliar. Já no segundo, buscamos avaliar o impacto de se utilizar o HDFS em contraste do sistema de arquivos convencional. O objetivo desse Apêndice é discutir a qualidade de tais modelos.

A técnica de Projeto Fatorial 2^kr e de Regressão modelam bem um sistema quando, dentre outras premissas de linearidade, as medições obtidas satisfazem seguintes premissas [Jain, 1991]:

1. Erros são normalmente distribuídos;
2. Erros são independentes dos níveis do fator;
3. Erros possuem desvio padrão constante (*homoscedascity*).

Se algum dos pressupostos for violado, as conclusões baseadas no modelo de regressão poderiam serem equivocadas. A verificação dessas premissas consistem em três análises visuais: A primeira consiste de um gráfico Quantil-Quantil dos erros (QQ-Plot) que indicam os desvios das replicações sobre a média em cada configuração do projeto. Nesse tipo de gráfico, os quantis de uma distribuição (erros medidos) são relacionados aos quantis de outra distribuição (nesse caso, uma normal). Uma

reta nesse gráfico, indica a proximidade dos erros com uma distribuição normal. A segunda consiste de um gráfico de desvios sobre a média (ou seja, os erros) em função das respostas observadas (média de cada configuração). Um bom resultado nesse tipo de gráfico é identificado por uma nuvem de pontos, sem tendências, uniformemente espalhados sobre o eixo y. A terceira consiste em um gráfico de dispersão que relaciona os resíduos e a resposta prevista para as configurações do projeto. Para observar se a condição de *homoscedascity* (desvio padrão constante dos erros) é satisfeita, os pontos nesse gráfico não deverão apresentar nenhum padrão visível nos erros em função das alterações dos fatores, ou seja, os erros seguem um desvio constante em cada um dos fatores. A segunda e terceira premissas são relacionadas, e com o mesmo teste, pode-se avaliar ambas [Jain, 1991], ou seja, com a verificação que os erros dos modelos sejam independentes e normalmente distribuídas com constantes variância.

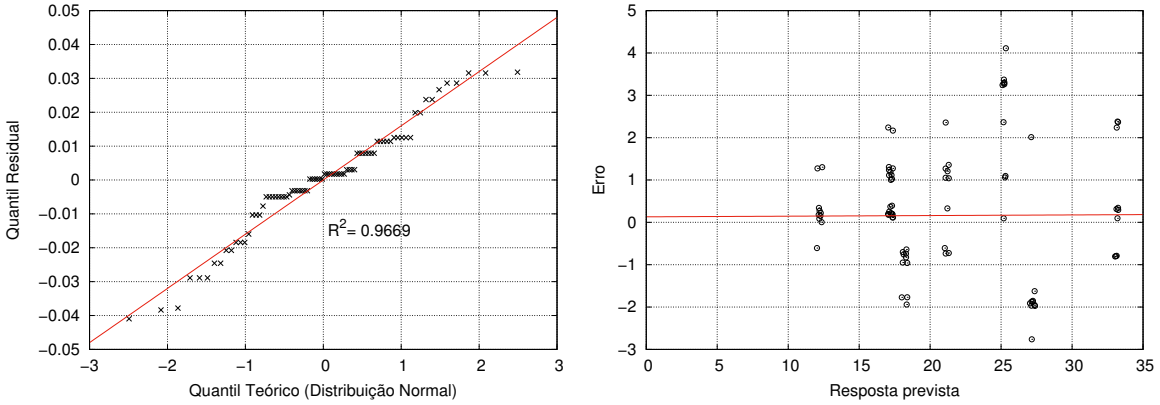
A.1 Projeto fatorial 2^kr

Dado a característica dos fatores, é sabido que relação entre poder de processamento e carga de trabalho é mais próximo de modelo multiplicativo [Jain, 1991], ao contrário de uma simples adição. Logo, se fez necessária uma transformação sobre as medições de tempo de execução, com o objetivo de eliminar as tendências e aproximar o modelo quanto a satisfatoriedade das premissas.

Na Figura A.1a, onde o gráfico Quantil-Quantil do projeto fatorial é apresentado, podemos ver que o erros tem relação próxima do linear de uma distribuição normal, é condizente então a premissa que os erros do projeto fatorial realizado são normalmente distribuídos. Além disso, nas Figuras A.1b e A.1c são mostrados os testes visuais para testar a premissa dois e três, ambas relacionadas. A partir de tais gráficos mostramos que o modelo criado não possui tendências claras quando aos erros, tais são independentes, constantes em relação aos experimentos realizados e são menores que as respostas previstas. As linhas de tendência vermelhas mostradas em ambos gráficos contribuem com a confirmação que não existe uma tendência clara nos erros gerados pelo projeto fatorial. A satisfatoriedade de tais premissas tornam condizentes a utilização de tal modelo criado para o objetivo proposto.

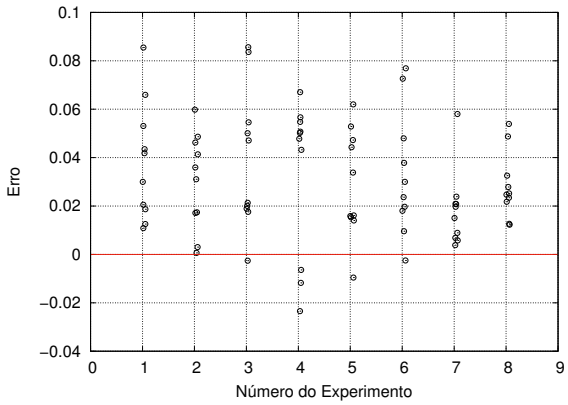
A.2 Regressão

Os gráficos da Figura A.2 e A.3 contêm os testes visuais para verificar as premissas da utilização das regressões lineares utilizadas em dois momentos desse presente trabalho.



(a) QQ-Plot para o projeto fatorial $2^3 10$

(b) Independência dos erros (IID)



(c) Resíduos por resposta prevista

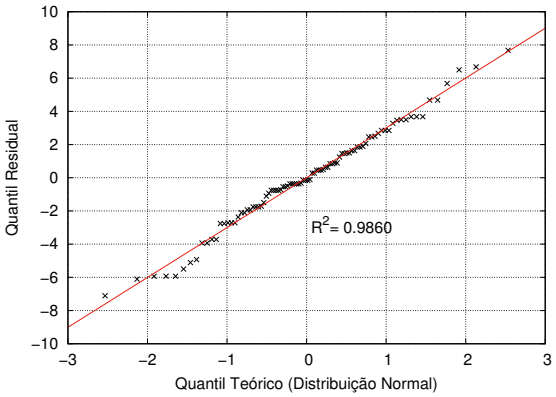
Figura A.1: Testes visuais para verificar as premissas do projeto fatorial

Em primeiro momento para avaliar o impacto da localidade dos dados em execuções COMPSs utilizando o HDFS (seção 5.2.1) e, logo após, para avaliar a diferença de desempenho ao se utilizar o HDFS como o sistema para a leitura dos dados em contraste do convencional (seção 5.2.2). Os testes para verificar a normalidade dos erros são mostrados nos gráficos Quantil-Quantil (QQ-Plot) na Figura A.2 enquanto que os testes para verificar a independências dos erros (Resíduos vs. Previsão) são mostrados na Figura A.3. Foram criados oito gráficos, um para cara par de teste/regressão, com a seguinte nomenclatura utilizada para a identificação: `<teste_visual>:<descrição>`.

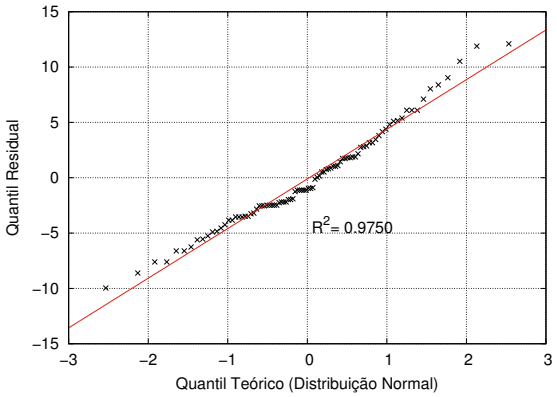
Os gráficos com descrição "HDFS sem localidade" e "HDFS com localidade" se referem as regressões criadas para avaliar o impacto da localidade dos dados, no primeiro caso, os experimentos realizados utilizaram a abstração do HDFS que não explora a localidade dos dados, já a segunda descrição é respectivo a regressão onde os experimentos utilizaram a abstração que explora, quando possível, a localidade dos dados. Já os gráficos com descrição "sem HDFS" e "com HDFS" contém os testes visuais

das regressões para o estudo onde se avaliou o custo de tempo de uma aplicação ler dados do sistema de arquivos convencional e do HDFS, respectivamente.

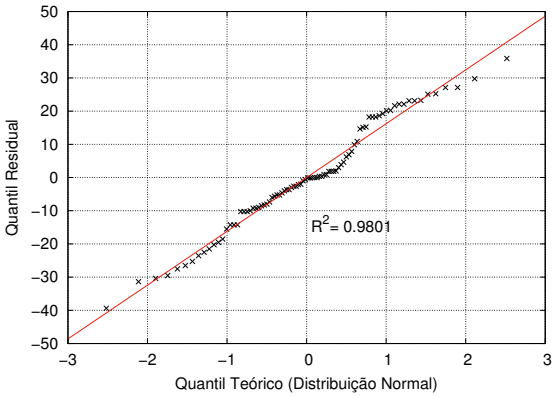
Em respeito aos testes de normalidade dos erros, assim como no projeto fatorial, quanto mais próximos de uma reta forem os erros medidos da regressão em função de um Quantil de distribuição normal, mais forte é a indicação que tais erros possuem uma distribuição normal. Todos os gráficos mostrados na A.2 satisfizeram tal condição, a reta vermelha em cada gráfico é a linha de tendencia dos pontos, juntamente com os seus respectivos coeficientes de determinação.



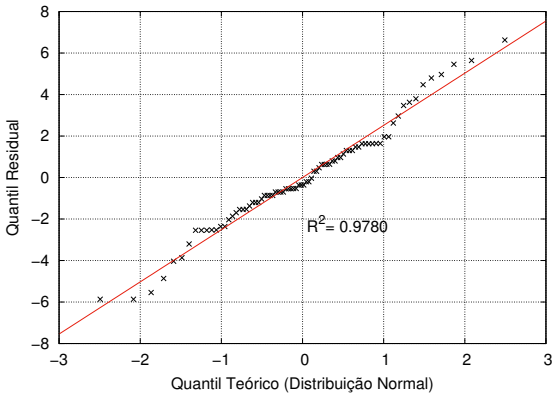
(a) QQ-Plot: HDFS sem localidade



(b) QQ-Plot: HDFS com localidade



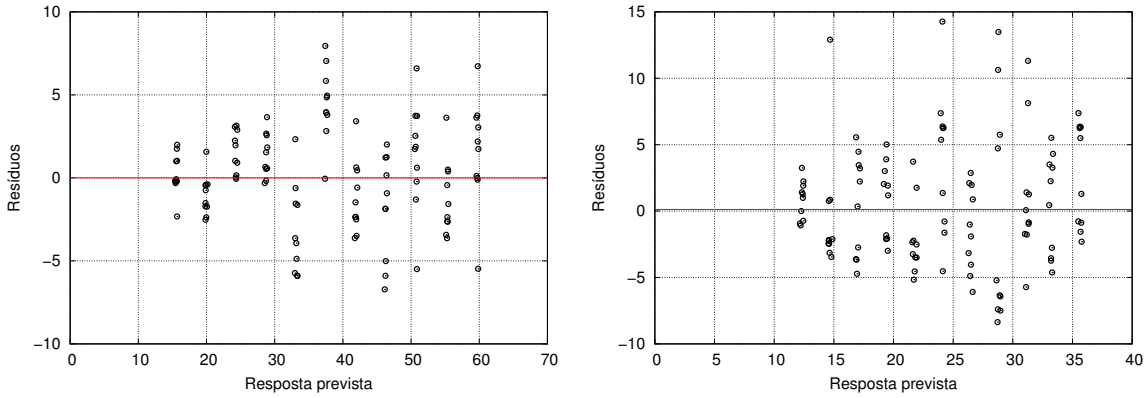
(c) QQ-Plot: sem HDFS



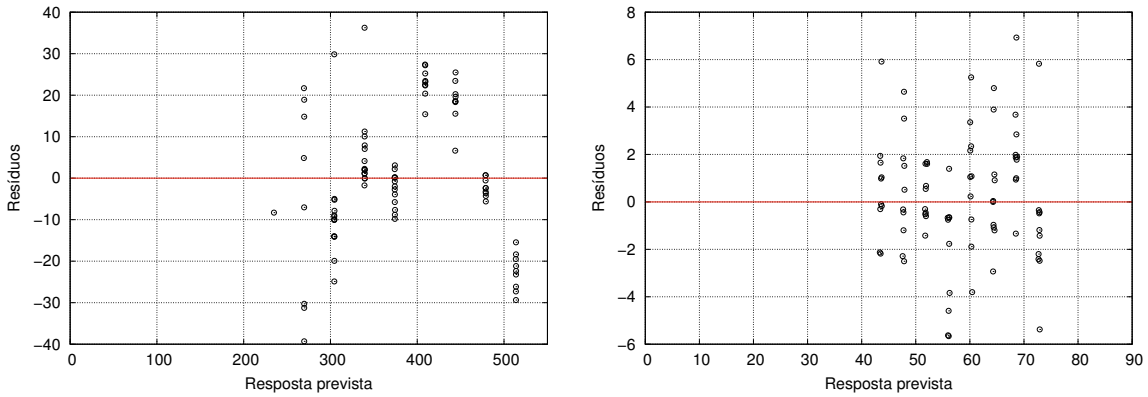
(d) QQ-Plot: com HDFS

Figura A.2: Testes visuais das regressões: distribuição normal dos erros

Por fim, em repeito aos testes para verificar se os erros são independentes dos níveis do fator, todos os gráficos, mostrados em Figura A.3, atenderam a tal condição, isto é, não é possível ver nenhuma tendência no gráfico de dispersão, isto pode ser corroborado com a linha de tendência mostrada nos mesmos gráficos que não mostram tendencias nas dispersões dos pontos.



(a) Resíduos vs previsão: HDFS s/ localidade (b) Resíduos vs previsão: HDFS c/ localidade



(c) Resíduos vs previsão: sem HDFS (d) Resíduos vs Previsão: com HDFS

Figura A.3: Testes visuais das regressões: independências dos erros

Apêndice B

Funções implementadas

A seguir apresentamos os métodos e funções disponíveis da API de integração com o HDFS tanto em linguagem Java quanto na linguagem Python, em seguida, os apresentaremos a lista com os algoritmos e operações COMPSs suportados no Lemonade.

B.1 API de integração com o HDFS

A tabela B.1 contêm os métodos disponíveis na API de integração do HDFS em Java e em Python.

B.2 Algoritmos e operações suportadas no Lemonade

A Tabela B.2 contêm a lista de operações e algoritmos PyCOMPS suportados atualmente pelo Lemonade, dividido atualmente em oito categorias: Entrada e saída (*Read/Save*), operações de leitura/escrita; Utilitários (*Utilities*), operações de auxílio; Transformação de dados (ETL), operações de transformação e extração de informação; Aprendizado de máquina (ML), algoritmos de *Machine Learning*; Avaliadores de qualidade (*Metrics*), avaliadores de qualidade dos modelos de *Machine Learning*; Processamento de texto (*Text*), operações sobre dados textuais; Processamento geográficos (*Geografic*), operações sobre dados georreferenciados; e Grafos (*Graph*), operações sobre grafos.

Tabela B.1: Métodos da API de integração HDFS

Classe	Tipo do retorno	Método	Descrição
Block (Java)	*Void*	close	Finaliza o ponteiro de acesso
	Byte[]	getChunck	Retorna um vetor de n bytes
	String	getRecord	Retorna um registro por '\n' ou '\r\n'
	String[]	getRecords	Retorna uma lista de registros
	Boolean	HasRecords	Verifica se o bloco possui registros não lidos
	Void	restart	Reinicia o ponteiro do bloco
	String	toString	Retorna uma informação do bloco
HDFS (Java)	*Void*	close	Finaliza conexão com HDFS
	Void	createFile	Cria um arquivo vazio
	ArrayList	findBlocks	Retorna a lista de blocos de um arquivo
	ArrayList	findNBlocks	Retorna uma lista de n blocos de um arquivo
	String	ls	Retorna a lista dos arquivos em uma pasta
	Void	mergeFiles	Concatena os arquivos em uma pasta
	Void	mkdir	Cria uma nova pasta
	Void	setUserHDFS	Especifica o nome do usuário
	Boolean	writeFile	Escreve um arquivo no HDFS
Block (Python)	*Void*	close	Finaliza o ponteiro de acesso
	DataFrame	readDataFrame	Retorna o bloco como um *DataFrame*
	Byte[]	readBinary	Retorna um vetor *bytes*
	Void	restart	Reinicia o ponteiro do bloco
	String	toString	Retorna informação do bloco
HDFS (Python)	*Void*	close	Finaliza conexão com HDFS
	Void	createFile	Cria um arquivo vazio
	List	findBlocks	Retorna a lista de blocos de um arquivo
	List	findNBlocks	Retorna uma lista de n blocos de um arquivo
	String	ls	Retorna a lista dos arquivos em uma pasta
	Void	mergeFiles	Concatena os arquivos em uma pasta
	Void	mkdir	Cria uma nova pasta
	Void	setUserHDFS	Especifica o nome do usuário
	Boolean	ExistFile	Verifica se um arquivo já existe
	Boolean	writeBlock	Escreve um arquivo no HDFS
	Boolean	writeDataFrame	Salva *DataFrame* em arquivo HDFS

Tabela B.2: Algoritmos/operações COMPS suportados pelo Lemonade

Categoria	Caixa Lemonade	Descrição
Read / Save	Data Reader	Lê um arquivo em *CSV* ou *JSON*
	Data Writer	Escreve um arquivo *CSV* ou *JSON*
Utilities	Attributes Changer	Renomeia ou altera o tipo de dados de uma coluna
	Workload Balancer	Rebalancea todos os fragmentos em partes iguais
	Comment	Cria um bloco de notas visual para anotação
ETL	Add Columns	Junta lado-a-lado dois *DataFrames*
	Aggregation	Agrega algumas colunas
	Clean Missing	Limpa os campos ausentes do conjunto de dados
	Difference	Remove linhas do *DataFrame* A existentes em um B
	Distinct	Remove linhas duplicadas
	Drop	Remove uma coluna
	Intersection	Realiza a interseção de dois *DataFrames*
	Join	Realiza uma operação de Junção
	Normalize	Normaliza uma coluna
	Replace Values	Substitui determinados valores de uma coluna
	Sample	Cria uma amostra de um *DataFrame*
	Select	Realiza uma projeção
	Sort	Ordena um *DataFrame*
	Split	Divide o *DataFrame* em duas partições
	Transform	Aplica uma operação de transformação do dado
	Union	Concatena dois *DataFrames*
ML	K-Means Clustering	Executa o algoritmo K-Means
	DBSCAN Clustering	Executa o algoritmo DBSCAN
	kNN Classifier	Executa o algoritmo kNN
	Naive Bayes Classifier	Executa o algoritmo Naive Bayes
	SVM Classifier	Executa o algoritmo SVM
	Logistic Regression	Executa o algoritmo Logistic Regression
	Linear Regression	Executa o algoritmo Linear Regression
	Apriori	Executa o algoritmo Apriori
	Save Model	Salva um modelo treinado
	Load Model	Lê um modelo treinado
	PCA	Executa o algoritmo PCA
	Feature Assembler	Cria um vetor a partir da projeção das colunas
Text	Tokenizer	Converte palavras em *tokens*
	ConvertWordstoVector	Converte *tokens* em vetor (BoW ou TF-IDF)
	Remove Stopwords	Remove palavras frequentes de uma sentença
	StringIndexer	Converte uma string para inteiro
Metrics	Classification Model Evaluation	Exibe a Accuracy, Precision/Recall e a F-mesure
	Regression Model Evaluation	Exibe a MSE, RMSE, MAE e o R^2
Geografic	Read Shapefile	Lê um arquivo *shapefile*
	Geo Within	Seleciona pontos de um dado presentes em uma área
	ST-DBSCAN	Executa o algoritmo ST-DBSCAN
Graph	PageRank	Executa o algoritmo PageRank