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

Lucas M. Ponce[1], **Walter dos Santos**[1],
Wagner Meira Jr.[1], **Dorgival Guedes**[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brazil

{lucasmsp,walter,meira,dorgival}@dcc.ufmg.br

Abstract. *High performance computing (HPC) and massive data processing (Big Data) are two trends in computing systems that are beginning to converge. This paper presents our experience on this path of convergence, extending COMP Superscalar (COMPSs), a parallel and distributed programming model already known in the world of HPC, for the processing of massive data. For this, it has been integrated to HDFS, the most widely used distributed file system for Big Data, and to Lemonade, a data mining and analysis tool developed at UFMG. The results show that the integration with HDFS benefits the COMPSs by the data abstraction provided and the integration with Lemonade facilitates its use and popularization in the area of Data Science.*

Resumo. *A computação de alto desempenho (HPC) e o processamento de dados massivos (Big Data) são duas tendências em sistemas de computação que estão começando a convergir. Este trabalho apresenta nossa experiência nesse caminho de convergência, estendendo o COMP Superscalar (COMPSs), um modelo de programação paralela e distribuída já conhecido no mundo de HPC, para o processamento de dados massivos. Para isso, ele foi integrado ao HDFS, sistema de arquivos distribuído mais usado para Big Data, e ao Lemonade, uma ferramenta de análise e mineração de dados desenvolvida na UFMG. Os resultados mostram que a integração com o HDFS beneficia o COMPSs pela abstração de dados fornecida e a integração com o Lemonade facilita sua utilização e popularização na área de Ciência dos Dados.*

1. Introdução

A computação paralela e distribuída tem se mostrado essencial para lidar com processamentos que exigem alto desempenho em computação, normalmente associado ao processamento de grandes volumes de dados. As aplicações de alto desempenho em computação (HPC) 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. Já em 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 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. 2015, Reed and 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 modelos, 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. Neste trabalho trabalhamos buscando uma integração entre um ambiente comumente adotado em cenários de HPC, o COMPSs, e recursos usualmente associados com ambientes de *Big Data*, como o sistema de arquivos HDFS e um ambiente de desenvolvimento visual de aplicações de mineração de dados.

Desenvolvido pelo Centro de Supercomputação de Barcelona, o COMPSs implementa um modelo de programação baseado em tarefas, modelo que vem se mostrando adequado para aplicações HPC [Lezzi et al. 2011, Lordan et al. 2016]. Um trabalho recente [Conejero et al. 2017] comparou seu desempenho ao Apache Spark, um dos *frameworks* de *Big Data* mais utilizados atualmente [Zaharia et al. 2012]. Apesar do trabalho indicar o melhor desempenho do COMPSs nos cenários testados, uma das principais vantagens do Spark é a ampla gama das bibliotecas que estão disponíveis em torno desse ambiente devido à sua popularidade (p.ex., MLlib, GrapX, Streaming e SparkSQL) [Conejero et al. 2017]. Além disso, Spark conta com uma maior interoperabilidade com outras ferramentas de *Big Data*.

Nesse contexto, propomos uma 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 usando o 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].

Com esse objetivo em mente, a seção 2 descreve o COMPSs e as seções 3 e 4 apresentam a sua integração com o HDFS e com o Lemonade, respectivamente. A avaliação de desempenho das integrações é discutida na seção 5 e finalmente a seção 6 apresenta nossas conclusões e discute trabalhos futuros.

2. COMPSs

O COMPSs é um modelo de programação que está em constante desenvolvimento, um objetivo do Centro de Supercomputação de Barcelona é o lançamento uma nova versão a cada seis meses, seja com novas funcionalidades e/ou suportes [Lezzi et al. 2011, Tejedor et al. 2017]. Atualmente na versão 2.2, esse modelo suporta linguagens Java,

Python 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. No caso de Java e C++, essas anotações são semelhantes a uma interface de um 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.

Sua estrutura é baseada no paradigma *master-worker*. Em execuções em *cluster* ou em nuvem é necessário definir, por exemplo, o conjunto de máquinas pertencentes à execução (no caso de *cluster*) e as especificações das VMs (no caso de nuvem). Um nó especificado como central (*master*) é responsável pelo gerenciamento da aplicação, pela transferências de dados, agendamento e pela gestão da infra-estrutura. Todos os outros nós atuam como trabalhadores (*workers*) e são responsáveis por realizar qualquer ação que o *master* solicite [Conejero et al. 2017].

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 a dependência entre esses métodos. Esse DAG pode ser visualizado posteriormente pelo usuário e é útil para entender o comportamento da execução em questão, por exemplo, mostrando quais métodos estão sendo realmente paralelizados. (Um exemplo desse DAG pode ser visto na fig. 2.) Durante a execução, o ambiente é capaz de submeter paralelamente todas as tarefas que não possuem dependências entre si, caso haja recurso computacional disponível; caso contrário, os métodos que não têm recursos alocados são escalonados à medida que suas dependências são satisfeitas.

Quando um *worker* finaliza a execução de uma de suas tarefas, o mesmo é responsável por salvar o resultado produzido em um arquivo binário (serializável) e por informar ao computador central sobre a sua finalização. O computador mestre, por sua vez, verifica (a partir da DAG gerada anteriormente), 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 caso, será de responsabilidade do *worker* produtor o envio de tal dado. Esse é um aspecto importante da abordagem que o COMPSs utiliza, isto é, caso existam dependências durante uma execução, os dados parciais (que estão em espera) não ocuparão espaço em memória uma vez que estão salvos em disco. Além disso, o seu escalonador, além de outros fatores, leva em consideração a localidade dos dados para atribuição de novas tarefas.

3. Integração entre o HDFS e o COMPSs

Como já mencionado, COMPSs é baseado em tarefas, logo, para explorar um maior nível de paralelismo é necessário trabalhar sobre fragmentos de um dado. Em uma execução COMPSs sem o uso de discos compartilhados, quando tarefas paralelizáveis precisam ler arquivos armazenados no sistema de arquivos convencional, esses arquivos precisam estar no computador central. A partir daí, o COMPSs os transferem pela rede para cada nó solicitante, o que serializa o acesso aos dados. Caso se deseje permitir que diversas tarefas leiam partes distintas do mesmo conjunto de dados, uma abordagem seria enviar o arquivo completo para cada tarefa, juntamente com um mapeamento que delimite a parte atribuível a cada tarefa. Embora seja mais fácil de implementar, esta alternativa

desperdiça recursos de rede e de armazenamento. Outra alternativa é dividir o arquivo no número de fragmentos desejados e direcionar cada um desses novos arquivos criados para cada tarefa. Desta forma, apenas a parte respectiva do arquivo será enviado pela rede, no entanto, a divisão é uma responsabilidade do programador. Ambas as soluções exigem a intervenção do usuário para fazer a distribuição dos dados e a decisão sobre como os mesmos podem ser distribuídos entre as tarefas.

O sistema de arquivos HDFS foi desenvolvido exatamente para gerenciar a divisão e distribuição das partes de arquivos massivos. No HDFS, 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. Quando uma aplicação distribuída deseja acessar o arquivo, ela recebe uma lista identificando todos os blocos que compõem um arquivo, com informações sobre a localização de cada bloco (e suas possíveis réplicas). A aplicação, de posse dessa informação, pode decidir como pretende distribuir as partes do arquivo entre seus nós de processamento. Cada nó recebe alguns blocos para processar e pode acessar o HDFS diretamente para obter o conteúdo de tais blocos, paralelamente a outros nós de processamento.

O principal conceito apresentado na integração entre o HDFS e 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. Na integração dos dois sistema, 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 consiste em prover ao programador COMPSs dois tipos de entidades com funções bem definidas. A primeira é responsável por lidar com o HDFS diretamente, como por exemplo, criar pastas ou obter informações sobre um arquivo. Já a outra é responsável pela abstração de arquivos divididos em fragmentos. A primeira entidade é representada pela classe HDFS, enquanto a segunda é representada pela classe Block, que possui métodos como *readBlock* (lê um fragmento de arquivo e o armazena como uma String) e *readDataframe* (lê um fragmento de arquivo e armazena como um *DataFrame*).

Para a etapa de leitura, o usuário solicita à API que o HDFS lhe informe uma lista de *n* fragmentos de um dado arquivo, onde a quantidade é definida pelo usuário, geralmente relacionada ao número de *cores* disponíveis em seu *cluster*. No momento da leitura, dentro de uma tarefa COMPSs, a entidade HDFS, com base nas informações de cada fragmento, será responsável por escolher o melhor fornecedor (*datanode*) desse dado parcial 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 escopo de execução COMPSs utilizando a API.

Nesse exemplo, cada um dos *N* fragmentos de um arquivo *X* armazenado no HDFS será lido paralelamente dentro da sua 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.

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
```

3.2. Comunicação com o HDFS

Entre as versões de COMPSs disponível, nosso alvo foi a versão em Python. Atualmente o Hadoop/HDFS não suporta nativamente essa linguagem, o que nos levou a considerar diferentes alternativas de integração.

Uma primeira solução seria utilizar o webHDFS, uma API nativa do HDFS para a comunicação utilizando protocolo HTTP REST [Gonzales 2016]. Apesar de fornecer os mesmos métodos disponíveis na API 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 para o processo Spark na JVM. 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. Dessa forma, além de tais processos externos aumentarem o consumo de memória, a transferências desses dados teria que ser por meio do próprio intermediador, diferente do que ocorre no PySpark.

A abordagem escolhida foi a utilização do *wrapper* C da Java API (libHDFS), também utilizada em outros trabalhos [Leo and Zanetti 2010, Rocha et al. 2016]. 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.

4. Suporte do Lemonade ao COMPSs

Na área de *Big Data*, o usuário que controla o processamento frequentemente carece de uma formação em Computação, especialmente em programação paralela/distribuída. Em geral esse profissional é oriundo da área onde os dados foram obtidos, sendo assim a pessoa mais adequada para interpretá-los. Com isso, um desafio reconhecido pelos pesquisadores na área é a dificuldade de se levar ao especialista no domínio dos dados uma ferramenta de processamento com que ele possa expressar suas consultas. COMPSs, apesar de reduzir a demanda sobre o programador em termos de programação paralela, ainda exige que o desenvolvedor identifique as tarefas que podem ser paralelizadas e as programe em uma linguagem como Java ou Python. Para reduzir essa barreira para o uso do ambiente, decidimos realizar a integração do ambiente de execução COMPSs com o Lemonade, um ambiente para programação visual de tarefas *Big Data*.

4.1. O ambiente Lemonade

Lemonade[1] é uma ferramenta visual para o cientista de dados e visa usuários que não conhecem a linguagem de programação ou que precisem desenvolver fluxos de trabalho usando o conjunto de ferramentas existente. 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. A figura 1 é um exemplo do tipo de aplicação que o Lemonade é capaz de criar. Cada caixa representa uma operação ou algoritmo, por exemplo, ler um arquivo ou treinar um classificador. Para cada caixa, o usuário é capaz de configurar os parâmetros da respectiva operação, por exemplo, qual o nome do arquivo a ser lido ou o número máximo de iterações de um algoritmo. 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. Com a integração com o COMPSs, programas e bibliotecas de algoritmos escritos em COMPSs podem se tornar rapidamente disponíveis para os usuários do Lemonade.

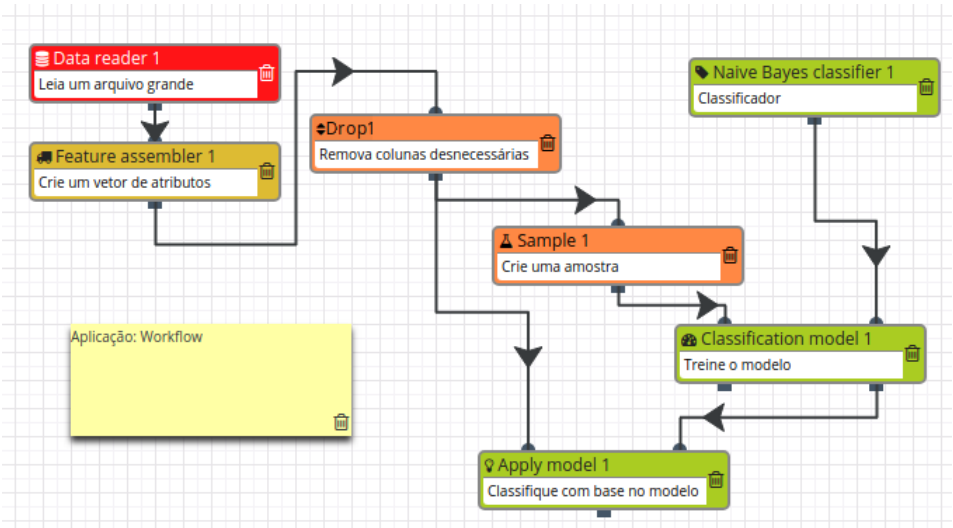


Figura 1. Esquema da aplicação Workflow criada a partir do Lemonade.

[1] https://github.com/W3CBrasil/AI-Social/tree/lemonade

A arquitetura do Lemonade define sete componentes individuais que funcionam como micro serviços especializados nas áreas de interface (*Citron*), segurança/privacidade (*Thorn*), execução (*Stand*), monitoração de execução (*Juicer*), base de dados (*Limonero*), algoritmos (*Tahiti*) e visualização (*Caipirinha*). A presente extensão do Lemonade para o Suporte do COMPSs envolveu dois desses componentes, *Tahiti* e *Juicer*.

Para o componente *Tahiti*, responsável por manter metadados referentes às operações e aos fluxos de dados criados por usuários, foi necessário o cadastro de todas as operações e algoritmos que foram implementadas em COMPSs. Tal cadastro envolve por exemplo, a categoria da operação (p.ex., operação sobre textos ou de algoritmos de aprendizado de máquina) e os parâmetros da cada operação (p.ex., nome das colunas ou valor máximo de iteração de um dado algoritmo).

O componente *Juicer* é responsável por traduzir o fluxo de operação visual criado pelo usuário (representado em um arquivo *JSON*) em código-fonte. Também é responsável por submeter a aplicação resultante para execução em um *cluster*. Para esse componente, foi necessária a criação de um novo compilador de código fonte (*source-to-source compiler*, ou *transpiler*) capaz de gerar código COMPSs. Uma vez que o usuário dispare um workflow criado, o *transpiler* lê o arquivo *JSON* produzido e identifica se é uma execução em COMPSs. Em seguida, cria um grafo onde os nós são as caixas de operações e executa uma ordenação topológica para criar uma sequência lógica de escrita do código (essa ordem é usada apenas para determinar a estrutura do código, não tendo implicações sobre a paralelização realizada pelo COMPSs durante a execução).

O código gerado pelo *transpiler* inclui a importação dos métodos presentes na biblioteca de métodos COMPS para o Lemonade e as instanciações desses métodos com a correta configuração dos parâmetros. Em outras palavras, a aplicação final é composta por duas partes, uma que é gerada dinamicamente pelo *transpiler*, que é basicamente um método principal com todas as chamadas dos métodos utilizados na aplicação, e outra que é uma biblioteca com implementações de operações e algoritmos.

4.2. Algoritmos e Operações

Diferente do Spark, que explora o paralelismo a partir de operadores sobre dados, em COMPSs o paralelismo é definido a partir da ausência de dependências entre tarefas. Em versões sequenciais usuais, um algoritmo pode ser visto como uma grande tarefa que deve ser executada sequencialmente sobre um único (grande) bloco de dados. Assim, a ideia por trás das implementações realizadas em COMPSs para o Lemonade foi de aproveitar a noção de blocos do HDFS e carregar um grande arquivo de dados dividido em diversos fragmentos. Dessa forma, em casos gerais, uma tarefa que seria executada sequencialmente em apenas um único conjunto de dados agora poderá ser executada paralelamente em conjuntos menores.

Em um caso de exemplo de um algoritmo simples de contagem de palavras (WordCount), um tipo de tarefa necessária, e que seria paralelizada pelo COMPSs, seria a contagem de cada palavra presente em cada segmento. Após isso, seria necessário realizar uma redução dessas contagens parciais. Além de paralelizar a execução, uma outra vantagem da divisão de dados em segmentos é que dessa forma, cada bloco caberá completamente em memória. O número de segmentos é relacionado com o número de *cores* disponíveis para uma dada execução. Após essa divisão do dado, caso um fragmento ainda seja maior

que a memória disponível em uma máquina, basta aumentar o número de fragmentos para um valor proporcional ao número de *cores* pois, nesse cenário, mesmo que um conjunto desses dados fiquem ociosos, eles não ocuparão espaço em memória pois estarão salvos temporariamente pelo COMPSs como arquivos binários em disco.

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 armazenar todos campos de configuração especificados pelo usuário pela interface. Já o dado de entrada/saída será sempre uma lista de *DataFrames*, essa padronização garante a compatibilidade entre a saída de uma função e entrada de outra. Internamente, uma função pode fazer agregações e pode executar outras funções 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.

A figura 2 é um exemplo de uma aplicação gerada pelo Lemonade como parte da aplicação mostrada na fig. 1, com o grafo de dependências gerado pelo COMPSs. A caixa 1, mostrada a direita, representa o que é gerado dinamicamente para cada aplicação, contém um método *main* com todas as definições de execução dos métodos (p.ex., settings_0), e com todas as chamadas dos métodos da aplicação (que estão disponíveis na biblioteca do Lemonade). Já a caixa 2 é um exemplo de uma dessas funções que podem ser utilizadas. A função em questão é utilizada para remover colunas específicas do dado de entrada em todas os fragmentos de dataFrame paralelamente. Como já mencionado, para o COMPSs saber que tal método pode ser executado paralelamente é necessário adicionar uma anotação sobre o método. Nesse exemplo, essa anotação, é representado por "*@task(returns=list)*"e informa ao COMPSs que além desse método ser paralelizável a sua saída é um tipo de lista.

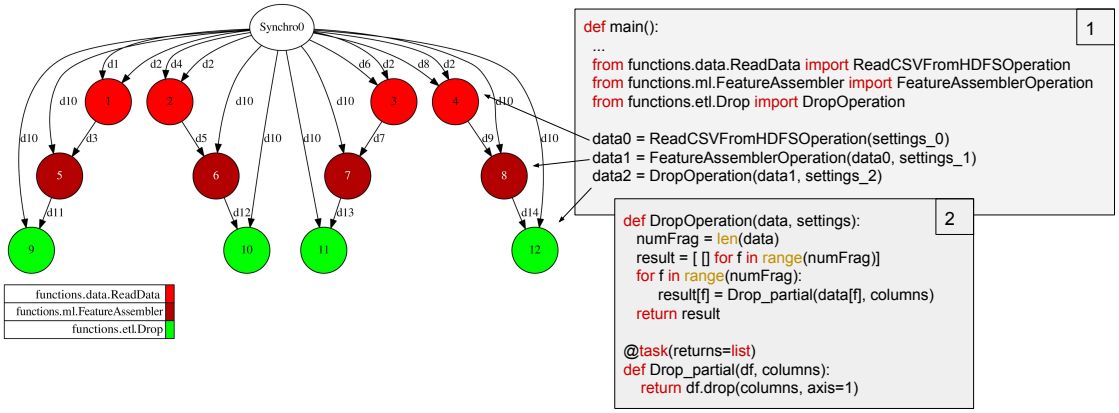


Figura 2. Exemplo de código gerado pelo Lemonade.

Já o grafo na figura 2 é um Grafo Direcionado Acíclico (DAG) gerado pelo COMPSs durante a execução. Nesse exemplo, podemos ver por exemplo que a execução continha três métodos diferentes, a saber, um para ler o dado em quatro partições, outro para criar um vetor de atributos e por fim, remover algumas colunas. Nesse caso, foram utilizados apenas quatro *cores* para a execução. 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].

5. Avaliação de desempenho

Na presente seção, avaliamos as integrações do COMPSs com o HDFS e com o Lemonade, comparando seu desempenho com dois casos de uso sem tais integrações. Especificamente, buscamos responder as seguintes perguntas: (1) As soluções utilizando o HDFS como o sistema de arquivos possuem tempos de execução significativamente diferentes das soluções existentes? (2) Caso exista diferença, existe algum limiar onde uma solução passa a ser melhor que a outra? (3) O tempo de execução de uma aplicação codificada diretamente pode ser significativamente diferente em do tempo de uma aplicação gerada automaticamente pelo Lemonade?

5.1. Metodologia e Base de dados

As avaliações de desempenho foram realizadas aplicando duas técnicas experimentais, o teste pareado e a regressão linear, sobre dois casos de uso, denominados WordCount e Workflow. A primeira técnica foi utilizada para responder as questões 1 e 2, já a regressão linear foi utilizada especificamente para responder a questão 3. O primeiro caso de uso é um simples algoritmo de WordCount, escolhido por ser uma aplicação onde o tempo de execução é basicamente relacionado ao tempo de leitura e de iteração sobre os dados lidos. O segundo caso de uso, o fluxo de operação mostrado inicialmente na figura 1, é composto por múltiplas operações e algoritmos, sendo um exemplo típico de execução esperada com o uso do Lemonade, consistindo em uma etapa de preprocessamento e uma de classificação de dados.

Os dois casos de uso contêm tipos de dados de entrada diferentes: no caso do WordCount, sua entrada é um arquivo de texto; já o Workflow recebe um arquivo tabular (csv) com atributos numéricos. Ambas as aplicações não impõem condições dependentes da base para um melhor ou pior caso de execução. Tento isso em vista, a carga de texto foi criada a partir da junção de vários livros no formato *txt* disponibilizados pelo Projeto Gutenberg[3] e depois concatenada repetidas vezes. Já para a carga numérica, foi utilizado o Data Set Higgs[4] que contém inicialmente 11×10^6 amostras com 28 dimensões de processos de sinal simulado que produzem os Bosons de Higgs.

Todos os experimentos utilizaram um *cluster* COMPSs/HDFS com um nó *master* dedicado e oito nós *slaves/workers*. As máquinas possuiam processadores Intel E56xx de 2,5 GHz com 4 cores, 8 GB de RAM, executando Ubuntu Linux 16.04 LTS. O HDFS foi configurado com um fator de replicação igual a três.

5.2. Impacto da utilização do HDFS

Para avaliar o impacto do uso do HDFS sobre execuções COMPSs foi realizado um teste pareado comparando os dois casos de uso adotados neste trabalho com suas respectivas versões com ou sem o HDFS. Para cada algoritmo, realizamos 40 execuções. Essa quantidade de repetições permite realizar um teste pareado *Z*, ou seja, utilizando o coeficiente *Z*. Após as execuções foi calculado o intervalo de confiança de um lado com 95% de confiança. A tabela 1 contém o resultado desses cálculos na escala de segundos.

[2] `https://github.com/eubr-bigsea/Compss-Python`

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

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

Tabela 1. Impacto do HDFS: teste pareado Z (tempos em segundos).

Informação	WordCount	Workflow
Número de amostras	40	40
Média das diferenças	-120,5	-173,8
Desvio padrão das diferenças	54,2	32,1
Intervalo de Confiança de um-lado (95%)	$(\text{-}\infty, -106, 4)$	$(\text{-}\infty, -165, 5)$
Tempo de execução média (com HDFS)	99	317
Tempo de execução média (sem HDFS)	211	497

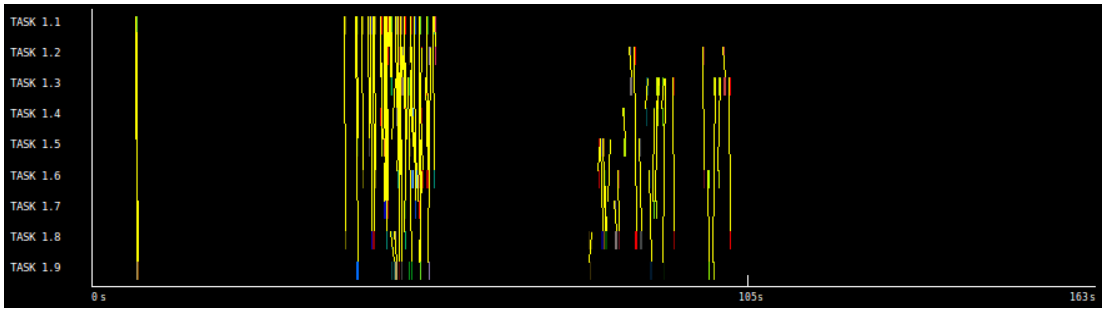
Para o primeiro algoritmo, o WordCount, o tempo de execução utilizando o HDFS foi em média 120 segundos mais rápido que o convencional. O intervalo de confiança de um lado indica que com 95% de confiança, a versão com HDFS é no mínimo 106 segundos mais rápido, que em termos percentuais equivale a aproximadamente 50% do tempo de execução do WordCount sem utilizar o HDFS.

O resultado foi similar para o segundo algoritmo. A versão com HDFS foi no mínimo 166 segundos mais rápida que o mesmo algoritmo utilizando o sistema de arquivos convencional com 95% de confiança. Em termos percentuais, equivale a aproximadamente 33% do tempo de execução da aplicação Workflow sem utilizar o HDFS. É interessante observar que esse ganho percentual é menor que o obtido para o WordCount, entretanto, isso pode ser explicado considerando-se que o segundo caso de uso possui uma carga maior de processamento envolvido.

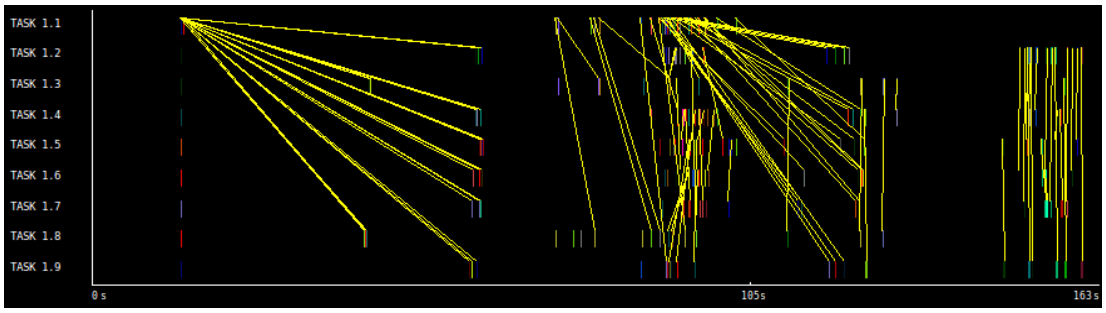
As figuras 3(a) e 3(b) mostram *traces* de mapeamento das transferências de dados entre os nós do *cluster* gerados pelo COMPSs durante a execução do WordCount com e sem a integração com HDFS. As linhas amarelas representam a transferência de arquivos durante a execução das aplicações, cada *Task* presente no eixo vertical dos *traces* representam um computador. A *Task* 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. As figuras 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.

Comparando as duas imagens podemos observar que na figura 3(b), o *master* é quem transfere todos os arquivos de dados e que parte do tempo de execução do algoritmo é devido à espera pela transferência desses arquivos. As transferências vistas aproximadamente na metade das execuções são os resultados parciais que foram gerados e que precisam ser agrupados; esses não são mais realizados exclusivamente pelo *master* e sim pelos nós que produziram tais resultados. Esse comportamento é o que de fato era 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.

O segundo tipo de experimento verifica se existe algum limiar onde usar HDFS passa a ser melhor ou pior que não usá-lo. Para isso, executamos o Wordcount 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. Utilizamos o WordCount como base desse experimento, pois o comportamento dos dois casos de uso foram semelhantes. Entretanto, o WordCount é o que possui o tempo de execução mais



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



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

Figura 3. *Traces* da transferência de dados durante a execução do WordCount.

dependente do tempo de leitura. O resultado obtido nesse cenário pode representar um indicativo para as outras classes de algoritmos. As regressões foram feitas coletando o tempo de execução de cada aplicação para diversos tamanhos de arquivo, de 2GB a 12GB, aumentando 1GB a cada cenário. Para cada tamanho de arquivo, executamos a aplicação vinte vezes, e computamos o tempo médio de execução para cada cenário.

A figura 4 ilustra as médias calculadas para cada cenário e as regressões obtidas. Observamos que a regressão da aplicação utilizando o HDFS possui um comportamento linear para todos os cenários, com pouca variância entre as execuções. Por outro lado, o mesmo não aconteceu com a aplicação utilizando o sistema de arquivos tradicional que apresentou um comportamento menos regular. Mesmo assim, as duas regressões satisfizeram as premissas de normalidade e de independência dos erros.

A Tabela 2 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 significativas são essas regressões.

Tabela 2. Resultado das regressões lineares.

Informação	com HDFS	sem HDFS
Parâmetro a	0,0113	0,0287
Parâmetro b	16,4363	-40,1454
Intervalo de Confiança de a (95%)	(13,5497, 19,3229)	(-67,4188, -12,8720)
Intervalo de Confiança de b (95%)	(0,0113, 0,0113)	(0,0287, 0,0287)
R^2	0,9939	0,9545

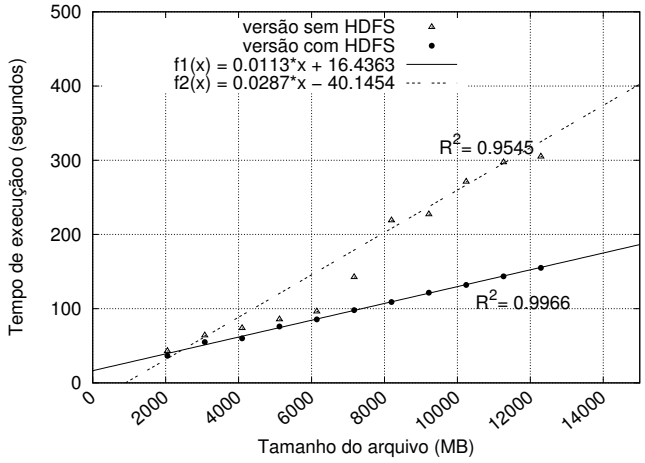


Figura 4. Tempo de execução do WordCount em função do tamanho do arquivo.

Assim, para a aplicação WordCount, a API de integração com o HDFS não só se mostrou mais rápida que o sistema tradicional, como também mais estável em relação ao aumento dos dados. Podemos ver pela Tabela 2 que o seu coeficiente de determinação é 0,9939 e seus dois parâmetros são significativos. Tais resultados indicam que a regressão representa bem o cenário tratado. Seu parâmetro *a* (coeficiente angular) é o menor das duas regressões criadas, combinando com as outras observações de que o HDFS possui melhor desempenho. Com base apenas nas regressões geradas, podemos imaginar que para dados pequenos, utilizar ou não o HDFS não tem tanto impacto ou talvez seja até pior; entretanto, ainda assim, a diferença de tempo é pequena. Por outro lado, o HDFS fornece ganhos significativos nos cenários com dados massivos.

5.3. Impacto da integracão com o Lemonade

Por fim, o último experimento analisa o impacto do Lemonade sobre o COMPSs. 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 (i.e., com maior otimização). Para avaliar esse impacto, realizamos novamente um teste pareado Z, com duas aplicações equivalentes, só que uma delas construída utilizando o Lemonade e a outra implementada manualmente, de uma 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.

A Tabela 3 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 $(\text{-}\infty, -19,1)$ 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 de percentuais, esse tempo corresponde a apenas 6% de uma execução média (considerando uma execução sem o Lemonade). Logo, essa diferença

não é tão impactante ao levarmos em conta os benefícios e praticidades de se utilizar o Lemonade, que libera o usuário da tarefa de desenvolver o programa COMPSs otimizado.

Tabela 3. Impacto do Lemonade: teste pareado Z (tempos em segundos).

Informação	Workflow
Número de amostras	40
Média das diferenças	-23,2
Desvio padrão das diferenças	15,9
Intervalo de Confiança de um-lado (95%)	$(\text{-}\infty, -19,1)$
Tempo de execução médio (com Lemonade)	317
Tempo de execução médio (sem Lemonade)	292

6. Conclusão e Trabalhos Futuros

Os avanços nas áreas de HPC e *Big Data* têm levado à aproximação das ferramentas e técnicas usadas em ambas. Neste trabalho propomos duas novas extensões ao ambiente COMPSs, desenvolvido originalmente para a área de HPC, para aumentar seu desempenho e facilitar sua utilização e aplicações *Big Data*. Essa extensões permitem sua integração com o sistema de arquivos distribuídos HDFS e com uma ferramenta visual para *Data Analytics*, ambas *open-source*. Acreditamos que tais contribuições ajudarão a popularizar o COMPSs e contribuirão na sua convergência com o mundo *Big Data*.

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.

Trabalhos futuros incluem extensões para novos conjuntos de dados e outros cenários com aplicações reais. Além disso, com relação ao HDFS, pretendemos estender o estudo feito tanto para operações com escrita de dados sobre o HDFS quanto para uma nova integração, já em andamento, usando a linguagem Java. Sobre o Lemonade, pretendemos estudar a possibilidade de geração de código mais dinâmico, com mais recursos de otimização, de forma que conjuntos de tarefas possam ser agrupados em uma única tarefa paralelizada e, com isso, seja possível gerar codigo mais otimizado automaticamente.

Agradecimentos

Este trabalho foi parcialmente financiado por Fapemig, CAPES, CNPq, e pelos projetos MCT/CNPq-InWeb (573871/2008-6), FAPEMIG-PRONEX-MASWeb (APQ-01400-14), H2020-EUBR-2015 EUBra-BIGSEA e H2020-EUBR-2017 Atmosphere.

Referências

Conejero, J., Corella, S., Badia, R. M., and Labarta, J. (2017). Task-based programming in COMPSs to converge from HPC to Big Data. *The International Journal of High Performance Computing Applications*, 17.

Fox, G. et al. (2015). Big data, simulations and HPC convergence. In *Workshop on Big Data Benchmarks*, pages 3–17. Springer.

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.

Kamburugamuve, S., Govindarajan, K., Wickramasinghe, P., Abeykoon, V., and Fox, G. (2017). Twister2: Design of a big data toolkit. In *EXAMPI 2017 workshop SC17 Conference*, Denver CO.

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

Lezzi, D., Rafanell, R., Lordan, F., Tejedor, E., and Badia, R. M. (2011). COMPSs in the VENUS-C platform: enabling e-science applications on the cloud. In *4th Iberian Grid Infrastructure Conference*, volume 1, Braga, Portugal. Universidade do Minho.

Lordan, F., Ejarque, J., Sirvent, R., and Badia, R. M. (2016). Energy-aware programming model for distributed infrastructures. In *24th Euromicro Int'l Conf. on Parallel, Distributed, and Network-Based Processing (PDP)*, volume 24, pages 413–417.

Reed, D. A. and 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., Jr., W. M., and 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., and Meira, W. (2017). Lemonade: A scalable and efficient spark-based platform for data analytics. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 745–748.

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

Wu, X., Kumar, V., Quinlan, J. R., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G. J., Ng, A., Liu, B., Philip, S. Y., 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. In *Proc. of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA. USENIX.