**Uma análise comparativa de ambientes para *Big Data*:
Apache Spark e HPAT**

Rafael Aquino de Carvalho

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO
DE
MESTRE EM CIÊNCIAS

Programa: Ciência da Computação
Orientador: Prof. Dr. Alfredo Goldman

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro
da CAPES e da empresa HPE

São Paulo, Fevereiro de 2018

Uma análise comparativa de ambientes para *Big Data*: Apache Spark e HPAT

Esta é a versão original da dissertação elaborada pelo
candidato (Rafael Aquino de Carvalho), tal como
submetida à Comissão Julgadora.

Uma análise comparativa de ambientes para *Big Data*: Apache Spark e HPAT

Esta versão da dissertação contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa da versão original do trabalho, realizada em 16/04/2018. Uma cópia da versão original está disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Alfredo Goldman (orientador) - IME-USP
- Prof. Dr. Edmundo Roberto Mauro Madeira - UNICAMP
- Prof. Dr. Emílio de Camargo Franceschini - UFABC

Agradecimentos

Agradeço ao meu orientador Alfredo Goldman, a todos os participantes do grupo de Sistemas do IME-USP por toda a ajuda que me deram durante o meu mestrado. Agradeço também a minha família por todo o suporte dado durante a minha estadia em São Paulo. Agradeço a Mariah por todo o apoio durante estes anos em que precisei de apoio e que estávamos morando em uma outra cidade. Obrigado a Capes e a empresa HPE por ter apoiado financeiramente a minha pesquisa de mestrado.

Resumo

AQUINO DE CARVALHO, R. **Uma análise comparativa de ambientes para Big Data: Apache Spark e HPAT**. 2018. 36 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

Este trabalho compara o desempenho e a estabilidade de dois arcabouços para o processamento de *Big Data*: Apache Spark e *High Performance Analytics Toolkit* (HPAT). A comparação foi realizada usando duas aplicações: soma dos elementos de um vetor unidimensional e o algoritmo de clusterização *K-means*. Os experimentos foram realizados em ambiente distribuído e com memória compartilhada com diferentes quantidades e configurações de máquinas virtuais. Analisando os resultados foi possível concluir que o HPAT tem um melhor desempenho em relação ao Apache Spark nos nossos casos de estudo. Também realizamos uma análise dos dois arcabouços com a presença de falhas.

Palavras-chave: Comparação de desempenho, Arcabouços de Big Data, HPAT, Apache Spark

Abstract

AQUINO DE CARVALHO, R. **A comparative analysis for Big Data environments: Apache Spark and HPAT**. 2018. 36 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

This work compares the performance and stability of two Big Data processing tools: Apache Spark and High Performance Analytics Toolkit (HPAT). The comparison was performed using two applications: a unidimensional vector sum and the *K-means* clustering algorithm. The experiments were performed in distributed and shared memory environments with different numbers and configurations of virtual machines. By analyzing the results we are able to conclude that HPAT has performance improvements in relation to Apache Spark in our case studies. We also provide an analysis of both frameworks in the presence of failures.

Keywords: Performance comparison, big data frameworks, HPAT, Apache Spark.

Sumário

Lista de Abreviaturas	ix
Lista de Figuras	xi
Lista de Tabelas	xiii
1 Introdução	1
1.1 Considerações Preliminares	2
1.2 Objetivos	2
1.3 Organização do Trabalho	3
2 Conceitos	5
2.1 <i>Big Data</i>	5
2.2 <i>MapReduce</i>	6
2.3 Hadoop	7
2.4 <i>Apache Spark</i>	9
2.5 RDD - <i>Resilient Distributed Dataset</i>	10
2.6 Sistema de tolerância a falhas	11
2.7 HPAT	11
2.8 HDF5 - <i>Hierarchical Data Format 5</i>	13
3 Trabalhos Relacionados	17
3.1 Comparações de ferramentas MapReduce	17
3.2 Reprodutividade	18
3.3 Linguagem Julia	19
4 Experimentos	21
4.1 Experimento Preliminar	21
4.2 Experimentos	22
4.2.1 Experimento 1: Soma dos Elementos de um Vetor	23
4.2.2 Experimento 2: <i>K-means</i>	23
4.2.3 Simulação de Ambiente com Falhas	23
4.3 Resultados	24
4.3.1 Experimento 1	24
4.3.2 Experimento 2	26
4.3.3 Resumo dos Resultados	27

4.3.4	Simulação de Ambiente com Falhas	27
5	Conclusões	31
5.1	Trabalhos Futuros	32
A	Dificuldades Encontradas	33
	Referências Bibliográficas	35

Lista de Abreviaturas

CSV	<i>Comma Separated Values</i>
DSM	<i>Distributed Shared Memory</i>
HDF5	<i>Hierarchical Data Format 5</i>
HDFS	<i>Hadoop Distributed File System</i>
HPAT	<i>High Performance Analytics Toolkit</i>
HPC	<i>High Performance Computing</i>
RDD	<i>Resilient Distributed Dataset</i>
MPI	<i>Message Passing Interface</i>
MV	Máquina Virtual
MVs	Máquinas Virtuais
vCPU	<i>Virtual Computing Process Unit</i>
YARN	<i>Yet Another Resource Negotiator</i>

Lista de Figuras

2.1	Diagrama apresentando o fluxo de uma execução MapReduce utilizando o algoritmo <i>1D Sum</i> como exemplo.	6
2.2	Diagrama apresentando a comunicação entre <i>Master</i> e <i>Workers</i>	7
2.3	Arquitetura do Hadoop 1.0. Imagem retirada do livro [Whi12]	8
2.4	Arquitetura do YARN, ou Hadoop 2.0. Imagem retirada do artigo [VMD ⁺ 13]	9
2.5	Diagrama apresentando a comunicação entre os componentes do Spark. Figura obtida do artigo [ZCD ⁺ 12]	10
2.6	Diagrama representando o fluxo de execução da ferramenta HPAT. Figura obtida do artigo [TAS17]	12
2.7	Imagem representando a geração de código em cada componente da ferramenta HPAT. Imagem obtida em [TAS17]	13
2.8	Imagem representando de forma visual os grupos de um arquivo HDF5. Imagem obtida em [FHK ⁺ 11]	14
4.1	Gráfico com medições da transferência de arquivo e do cálculo do Pi no cenário 1.	22
4.2	Gráficos com as medições de tempo de execução no Experimento 1 Cenário 1. Os gráficos superiores apresentam <i>boxplots</i> para os dois tipos de execuções. Os Gráficos inferiores apresentam a duração de cada iteração para os dois tipos de execuções.	24
4.3	Gráficos com as medições de tempo de execução no Experimento 1 Cenário 2. Os gráficos superiores apresentam <i>boxplots</i> para os dois tipos de execuções. Os Gráficos inferiores apresentam a duração de cada iteração para os dois tipos de execuções.	25
4.4	Gráficos com as medições de tempo de execução no Experimento 2 Cenário 1. Os gráficos superiores apresentam <i>boxplots</i> para os dois tipos de execuções. Os Gráficos inferiores apresentam a duração de cada iteração para os dois tipos de execuções.	26
4.5	Gráficos com as medições de tempo de execução no Experimento 2 Cenário 3. Os gráficos superiores apresentam <i>boxplots</i> para os dois tipos de execuções. Os Gráficos inferiores apresentam a duração de cada iteração para os dois tipos de execuções.	27
4.6	Gráfico da simulação do Experimento 1 Cenário 1 e Experimento 2 Cenário 2. Cada ponto representa 100 execuções e possui uma barra representando o erro padrão.	29

Lista de Tabelas

4.1	Tipos de máquinas virtuais	21
4.2	Configuração dos cenários de execução	23
4.3	Resumo dos resultados obtidos nos experimentos	28

Capítulo 1

Introdução

Com o crescente número de dados sendo gerados diariamente, surgiu a necessidade de realizar processamento e análises deste grande volume de dados, também chamado de *Big Data* [MCB⁺11]. Como este volume de dados era muito maior que o convencional, a forma tradicional de analisar dados não conseguia realizar o processamento em um tempo de execução aceitável. Então, para solucionar este problema foram criados algoritmos e ferramentas para solucionar este problema e realizar o processamento destes dados de uma forma eficiente, gerando o resultado em um tempo de execução aceitável.

Uma das definições de *Big Data*, que foi criada pela *Meta Group*, é a dos três Vs [Lan01], que são volume, velocidade e variedade. Após esta definição várias outras definições foram sugeridas, aumentando a quantidade de Vs [Tro12, DeV16, Fir17], mas mantendo os três Vs definidos anteriormente. Estas definições foram importantes para não só definir o que é *Big Data*, como também para categorizar os problemas que podem ser enfrentados nesta área.

O modelo de programação mais comum para manipulação dos dados é baseado em uma estratégia de redução mostrada no trabalho *MapReduce* [DG08]. Esta estratégia usa as funções de *map* e *reduce* das linguagens de programação funcional em um ambiente de *big data*. *MapReduce* provê uma abstração para processar grandes conjuntos de dados executando as funções *map* e *reduce*. A função *map* é aplicada sobre todo o conjunto de dados, extraíndo todos os dados que combinam com uma certa propriedade. Um conjunto de dados é gerado como saída da função. Esta base de dados contém um conjunto de pares (chave,valor) representando os dados extraídos e o resultado da computação realizada sobre estes dados, respectivamente. A função *reduce* é aplicada sobre todos os pares criados para produzir uma informação combinando todos estes valores.

Baseado neste modelo introduzido pela *Google*, foi criada a ferramenta Hadoop [Whi12]. Este arcabouço cria uma implementação direta do modelo MapReduce. Por ser um *software* livre, esta implementação fez com que todos tivessem um acesso mais fácil a este modelo e pudessem realizar processamentos de dados massivos. Em conjunto com o Hadoop, foi também criado o *Hadoop Distributed File System* [SKRC10], um sistema de arquivos distribuído que é bastante utilizado para o armazenamento de dados massivos. O HDFS foi baseado em um outro trabalho da *Google* [GGL03], onde é apresentada uma maneira de armazenar arquivos grandes de forma distribuída.

Também com o intuito de realizar processamentos em grandes volumes de dados, o arcabouço *Apache Spark* [ZCF⁺10] é apresentado com uma solução aprimorada do Hadoop. O diferencial desta solução está no uso da memória principal para o armazenamento dos dados utilizados durante a execução, criando um conjunto de dados distribuído chamado *Resilient Distributed Dataset*, ou RDD [ZCD⁺12]. Nos resultados mostrados em seu artigo de apresentação [ZCF⁺10], o *Spark* possui um tempo de execução inferior ao obtido pelo Hadoop, mostrando que com o uso da memória principal este arcabouço consegue tempos de execução melhores.

O HPAT aparece com uma alternativa para a *Spark* sendo que seus resultados já foram apresentados de forma informal em diversas apresentações. Mas, não conhecemos nenhuma validação mais formal de suas vantagens. Um vídeo disponível sobre uma apresentação do HPAT² deixa claras as suas vantagens, assim como a ausência de tolerância a falhas. O nosso objetivo foi prover uma

validação independente e reprodutível do potencial do HPAT, verificando o seu real potencial face ao Apache Spark.

Este trabalho realiza uma análise comparativa dos dois arcabouços para processamento de *Big Data*, o *High Performance Analytics Toolkit* (HPAT) [TAS17], e o *Apache Spark* [ZCF⁺10]. A análise busca compreender o melhor cenário para aplicação de cada ferramenta, bem como explicitar as vantagens do uso de cada uma. Foram realizados experimentos em ambientes com memória distribuída e compartilhada, medindo o desempenho, a estabilidade das execuções, e suas respectivas variações. Também foi realizada a simulação da execução das aplicações em um ambiente com falhas.

Para a comparação entre os dois arcabouços, foram escolhidos dois algoritmos diferentes, ambos explorando as possibilidades do MapReduce. Os algoritmos são a soma dos elementos de um vetor unidimensional e o *K-means*. Os dois algoritmos permitem a análise da escalabilidade dos arcabouços escolhidos. O algoritmo utilizado no primeiro experimento, soma dos números contidos em um vetor, pode ser utilizado quando há a necessidade de combinar uma grande quantidade de números, podendo ser utilizado para encontrar o resultado de fórmulas matemáticas. Já o segundo experimento é utilizado, geralmente, para identificar diferentes grupos em dados, também conhecido como clusterização, sendo utilizado em áreas como aprendizado de máquina, análise de dados e ciência dos dados.

Os algoritmos foram escolhidos pelos seguintes critérios: O algoritmo para a soma dos valores contidos em um vetor realiza muita movimentação de dados e pouco processamento. Já o algoritmo *K-Means* tem muita computação e é bastante utilizado na área de aprendizado de máquina.

1.1 Considerações Preliminares

Dentro do contexto de arcabouços para *big data* surge, como uma possível alternativa, a ferramenta HPAT. Em seu artigo introdutório [TAS17] os autores alegam que o arcabouço possui um tempo de execução menor do que o obtido pelo Spark. Resultados semelhantes ao apresentado no artigo foram apresentados pela equipe do HPAT na conferência *Julia Con* de 2016¹.

Iremos, através deste trabalho, tentar reproduzir o que foi apresentado tanto no artigo quanto na *Julia Con* de 2016 e tentar observar através do tempo de execução das aplicações executadas se o arcabouço HPAT realmente é uma nova alternativa que está surgindo para a execução de tarefas MapReduce.

Os nossos experimentos foram executados tanto em cenários com memória compartilhada quanto com memória distribuída e com diferentes números de máquinas virtuais. Além disso também foi realizada uma simulação de ambientes com probabilidade de falhas. Os resultados obtidos pelos experimentos foram publicados como artigo na conferência IEEE NCA 2017.

1.2 Objetivos

Este trabalho tem como principal objetivo realizar uma comparação de eficiência e estabilidade dos arcabouços HPAT e *Apache Spark*, levando em consideração o tempo de execução levado em cada instância dos experimentos e busca responder se o arcabouço *High Performance Analytics Toolkit* é uma alternativa, em desempenho, para o Apache Spark.

Esta comparação tem o intuito de descobrir qual destas ferramentas produz os melhores resultados, dada as variáveis, e realizar uma reprodução, com menor escala, dos resultados apresentados pela equipe de desenvolvimento da HPAT na conferência *Julia Con* do ano de 2016², na página do GitHub do projeto³ e também no trabalho [TAS17]. A resposta para a pergunta de pesquisa, se o HPAT é uma alternativa para o Spark em termos de tempo de execução, será respondida em dois

¹<http://juliacon.org/2016/abstracts.html#HPAT>. Esta apresentação pode ser assistida em <https://www.youtube.com/watch?v=Qa7nfaDacII> [Acessados em 07/05/2018]

²<http://juliacon.org/2016/abstracts.html#HPAT>. Esta apresentação pode ser assistida em <https://www.youtube.com/watch?v=Qa7nfaDacII> [Acessados em 23/03/2017]

³<https://github.com/IntelLabs/HPAT.jl> [Acessado em 18/12/2016]

ambientes distintos. Um sem a ocorrência de falhas e um outro em que se simula o ambiente com falhas.

1.3 Organização do Trabalho

Esta dissertação está organizada da seguinte forma: O Capítulo 2 apresenta os conceitos necessários para o entendimento deste trabalho e também introduz as ferramentas HPAT e *Apache Spark*. O Capítulo 3 mostra trabalhos relacionados. O Capítulo 4 mostra o experimento preliminar e explica os experimentos realizados neste trabalho e apresenta os resultados obtidos nestes experimentos. O Capítulo 5 apresenta de forma resumida os resultados obtidos e conclui o trabalho.

Capítulo 2

Conceitos

Este capítulo irá apresentar os principais conceitos necessários para o entendimento dos arcabouços, *Apache Spark* e *Intel HPAT*, utilizados nos experimentos desta dissertação. Este capítulo apresenta o modelo de programação *MapReduce* na Seção 2.2. Na Seção 2.3 é apresentado o arcabouço Hadoop. A Seção 2.4 apresenta um dos arcabouços utilizados, o Apache Spark e na Seção 2.5 é apresentada a principal abstração utilizada no arcabouço o *Resilient Distributed Dataset* ou RDD. A Seção 2.7 é apresentado o arcabouço *Intel High Performance Analytics Toolkit* ou HPAT. E por último é apresentado o formato de arquivo utilizado pelo HPAT, o *Hierarchical Data Format 5*, ou HDF5.

2.1 *Big Data*

Big Data é um dos principais conceitos por trás da criação dos arcabouços utilizados nesta pesquisa. Esta área lida, principalmente, com um grande volume de dados. Mas ela foi primeiro caracterizado não só por seu grande volume de dados, mas também tendo que lidar com velocidade e a variedade dos dados, formando assim os 3 Vs [Lan01]. Sendo velocidade o tempo de processamento dos dados, ou seja o tempo de ter uma resposta do processamento e variedade caracterizando os dados como não-estruturados na maioria dos casos.

No artigo *Big data meets big analytics* [Tro12] é apresentado um novo V, representando valor. Este novo V mostra que os dados também seu valor e a partir da necessidade da análise dos dados é mostrado que existe um verdadeiro valor nos dados. No mesmo artigo também é mostrado que existe uma complexidade na análise destes dados.

Em 2016 já nos é mostrado que Big Data pode ser caracterizado em não só 4, mas 7 Vs [DeV16], adicionando aos 4 anteriores: Veracidade, Visualização e Variabilidade. Estes novos 3 Vs vem da necessidade de termos certeza de que os dados são verdadeiros, ou seja, atestarmos a veracidade dos dados. Como os dados podem estar constantemente variando. E, por último, é a necessidade de saber como visualizar todos estes dados.

No ano de 2017 foi sugerida novas adições de Vs [Fir17]. Este artigo adiciona 3 Vs aos 7 apresentados em [DeV16], passando a ser 10. Os Vs que foram acrescentados são: Validade, Vulnerabilidade e Volatilidade. Validade é a necessidade de se validar os dados, de ter certeza que os dados estão corretos para o propósito que eles serão utilizados. Vulnerabilidade envolve a segurança destes dados, grande quantidade de dados pode gerar grandes brechas, gerando a necessidade de se ter uma maior segurança e não deixar estes dados tão vulneráveis. Por último temos volatilidade, que tenta verificar o quão velho os dados precisam ser para serem considerados irrelevantes, gerando a pergunta: Por quanto tempo estes dados precisam ser armazenados?

Além destes textos, também existem diversos outros caracterizando *Big Data* utilizando uma quantidade de Vs, podendo variar de 3 (sendo estes os textos mais antigos) até 10 Vs. Isto mostra a necessidade de incluir termos para quem for utilizar os dados terem alguns conceitos em mente, como pode ser visto nos termos que passam a ser incluídos na lista de Vs.

Todos estes Vs tem que ser considerados não só no armazenamento dos dados, mas também

na manipulação e processamento dos dados, então todas estas características que se atribui ao *big data* tem que ser também levadas em consideração aos arcabouços ou ferramentas que irão realizar o processamento destes dados.

2.2 MapReduce

MapReduce é um modelo de programação proposto pela equipe da *Google* [DG08]. Este modelo tem como objetivo facilitar tanto a criação quanto o processamento de grandes conjuntos de dados. Ele foi criado para suprir a necessidade que a empresa possuía de realizar o processamento de uma grande quantidade de dados para realizar as análises necessárias em seu sistema de busca.

Este modelo foi baseado nas funções de *Map* e *Reduce* presentes nas linguagens de programação do paradigma funcional. Neste modelo, a etapa de *Map* realiza um mapeamento dos dados a serem processados e cria sub-conjuntos intermediários. Para a etapa de *Reduce* é executada uma função que irá processar os dados intermediários gerados no *Map* e devolver um conjunto de dados possivelmente menor como resultado. Estas etapas podem ser observadas na figura

Figura 2.1: Diagrama apresentando o fluxo de uma execução MapReduce utilizando o algoritmo 1D Sum como exemplo.

A implementação deste modelo possui dois componentes principais para a execução das tarefas, o *Master* e os *Workers*, seguindo o modelo de execução de mestre e escravos.

- **Master:** Este componente é responsável por enviar as tarefas para os *workers*. Também é responsável por armazenar o status de cada tarefa (ociosa, em progresso, concluída). O *master*, após a conclusão de cada tarefa do tipo *map*, armazena a localização dos dados intermediários para informar as tarefas do tipo *reduce* onde os dados estão.
- **Workers:** Responsável pela execução das tarefas e de salvar em disco o arquivo com o resultado final da execução após as tarefas de *reduce*.

O fluxo de execução de uma aplicação *MapReduce*, levando em consideração a comunicação entre estes componentes, acontece da seguinte maneira: Após a aplicação ser inicializada pelo usuário, a aplicação é enviada para o *master*. O *master* distribui, entre os *workers* disponíveis, as respectivas tarefas do tipo *map*. Após terminar a tarefa, cada worker avisa ao *master* sobre o término e quando todas estiverem finalizadas, no final desta etapa são gerados conjuntos de dados intermediários. As próximas tarefas são as referentes ao *reduce*. Assim como foi feito na etapa de *map*, o *master* envia

a respectiva tarefa para os *workers*, acabando as tarefas os *workers* avisam para o *master* sobre o término. No final desta etapa, diferente do *map*, um ou mais arquivos são salvos em disco com o resultado final. Este fluxo de execução pode ser observado na figura.

Figura 2.2: Diagrama apresentando a comunicação entre Master e Workers.

Além de demonstrar como funciona o fluxo de execução deste modelo de programação, a implementação de um arcabouço *MapReduce* também inclui tolerância a falhas, localidade das execuções e a granularidade das tarefas. A tolerância a falhas é utilizada por, usualmente, um programa *MapReduce* utilizar várias máquinas, então é necessário a tolerância a falhas para caso alguma das máquinas falhe durante a execução. A localidade se refere aos locais onde o programa será executado, ou seja, quais *workers* serão utilizados. Isto ocorre porque a largura de banda de uma rede ainda pode ser um gargalo, então quanto mais próximo os *workers* estiverem do *master* menos transferências pela rede serão realizadas. Já a granularidade é referente ao tamanho de cada tarefa que será encaminhada para cada *worker*.

2.3 Hadoop

Apache Hadoop é uma das várias implementações do modelo de programação *MapReduce*, Google [DG08] apresentado na seção anterior. Este arcabouço foi criada com o intuito de atacar a

crescente escala necessária para realizar indexação de *web crawls* [VMD⁺13]. A estrutura utilizada pelo Hadoop 1.0 era a mesma apresentada em [DG08]. Assim como no artigo sobre o *MapReduce*, o Hadoop apresenta um modelo Mestre-Escravo. Em sua primeira versão, possuía uma estrutura onde continha os componentes *clientjob tracker* e *task tracker*. Esta é uma implementação literal do modelo apresentado em [DG08].

- **Client:** é responsável por inicializar a aplicação. Seria o local onde o cliente executou o comando para começar o programa de *MapReduce*.
- **Job Tracker:** É similar ao *Master* apresentado em [DG08]. Ele é responsável por inicializar as tarefas de *map* ou *reduce* nos respectivos *workers*. Também tem a responsabilidade de monitorar a situação de cada tarefa e também do local onde estão sendo executadas utilizando *heartbeats* para estas verificações.
- **Task Tracker:** É o equivalente ao *worker* onde serão executadas as tarefas de *map* ou *reduce*.

Esta arquitetura apresentada na primeira versão do Hadoop pode ser observada na Figura 2.3.

Figura 2.3: Arquitetura do Hadoop 1.0. Imagem retirada do livro [Whi12]

A segunda versão do Hadoop, também chamada de YARN (*Yet Another Resource Negotiator*), além de mostrar um desempenho melhor do que a sua primeira versão, também possui algumas mudanças em sua arquitetura. Uma dessas mudanças foi a criação de um gerenciador de aplicações, chamado de *application manager*, que aceita diferentes *plugins* e que possibilita a execução de diferentes tipos de aplicações, por exemplo MPI e Spark.

O YARN possui em sua arquitetura os seguintes componentes: *client*, *resource manager* e *node manager*. Esta arquitetura pode ser vista na figura 2.4.

- **Client:** Assim como na versão 1.0 do Hadoop, ele é responsável por inicializar a aplicação. Seria o local onde o cliente executou o comando para começar o programa de *MapReduce*.

- **Resource Manager:** Este componente possui duas interfaces públicas. Uma para a comunicação com o cliente e uma segunda para a comunicação com o *master* de cada aplicação (*Application Masters*). A primeira interface serve para a submissão de aplicações. Já a segunda é utilizada para cada *master* negociar recursos dinamicamente.
- **Node Manager:** É o equivalente ao *worker* onde serão executadas as tarefas de *map* ou *reduce*. Este componente também fica responsável em reportar a quantidade de recursos disponíveis para o escalonador do YARN.

Figura 2.4: Arquitetura do YARN, ou Hadoop 2.0. Imagem retirada do artigo [VMD⁺13]

Um outro componente importante dentro do YARN é o *application master*. Este componente é o que torna possível a utilização de outros arcabouços de processamento de dados massivos dentro do YARN. O *application master* utiliza *plugins* de outros arcabouços para a execução da aplicação. A informação do *plugin* é fornecida pelo cliente na criação da aplicação. Isto torna o YARN mais versátil do que a versão anterior do Hadoop.

As aplicações do Hadoop, tanto em sua versão 1.0 quanto em sua versão 2.0, possuem uma mesma característica. Em ambas as versões os dados gerados são escritos ou lidos diretamente do disco. Como iremos ver nas próximas seções deste capítulo, esta característica do Hadoop irá fazer com que este arcabouço tenha um desempenho inferior a alguns arcabouços criados posteriormente que utilizam o armazenamento dos dados em memória.

2.4 Apache Spark

O arcabouço para processamento de *Big Data Apache Spark* [ZCF⁺10] surgiu com uma ideia de criar um arcabouço que fosse mais eficiente do que a *Apache Hadoop* [Whi12], que foi criada para ser uma implementação do modelo de programação *MapReduce*. A principal diferença entre o *Spark* e o *Hadoop*, é o uso da memória para o armazenamento de dados durante a execução das aplicações. Este armazenamento de dados em memória compartilhada é chamado de *Resilient Distributed Dataset* (RDD) [ZCD⁺12]. Com este armazenamento em memória o Spark conseguiu superar o desempenho da *Hadoop*.

Em [ZCF⁺10] e [ZCD⁺12] podemos entender alguns conceitos e abstrações presentes no arcabouço *Spark*. Além da abstração criada para o armazenamento de dados utilizando memória distribuída, RDD, podemos observar também outros conceitos presentes para entender o funcionamento desta arcabouço.

O arcabouço possui três operações paralelas principais, como pode ser observado em [ZCF⁺10] que são:

- **Reduce:** É responsável por combinar os elementos de um conjunto de dados utilizando uma função associada para gerar um resultado a partir destes dados.
- **Collect:** Envia todos os elementos de um conjunto de dados para o nó mestre, que está executando o programa
- **Foreach:** Envia cada elemento do conjunto de dados para uma função fornecida pelo usuário.

Figura 2.5: Diagrama apresentando a comunicação entre os componentes do Spark. Figura obtida do artigo [ZCD⁺12]

A arquitetura de execução do Spark, como mostrado em [ZCD⁺12], é composta por dois componentes principais. O *Driver*, que é o local onde é iniciada a execução da aplicação, e os *Workers*, que são os locais em que são realizadas as computações da aplicação. Durante a execução da aplicação existe uma comunicação bilateral entre o *Driver* e os *Workers*, porém não existe comunicação entre os *Workers*. Este relacionamento pode ser observado na Figura 2.5.

O arcabouço para processamento de *Big Data Apache Spark* [ZCF⁺10] utiliza o *Resilient Distributed Dataset* (RDD) [ZCD⁺12], permitindo o armazenamento de dados em memória durante a execução das aplicações. O RDD também permite armazenar em disco os dados que não couberem na memória disponível.

O *Apache Spark* permite a criação de aplicações que não utilizem o modelo de programação *MapReduce*, e fornece extensões para a criação de aplicações como *streaming* e processamento de grafos. Além disso, o *Apache Spark* fornece suporte para vários tipos de arquivo.

Em contraste ao HPAT, o uso do *Apache Spark* já é bastante difundido na comunidade de processamento de *Big Data*. Além disso, o arcabouço possui mecanismos de tolerância a falhas ainda não presentes no HPAT. No entanto, o HPAT apresenta resultados iniciais bastante promissores.

2.5 RDD - *Resilient Distributed Dataset*

Resilient Distributed Dataset é uma das principais abstrações utilizada pelo arcabouço *Apache Spark*. Esta abstração tem como seu principal objetivo o de criar uma memória compartilhada entre os *workers* em execução de uma aplicação no Spark. O RDD também faz com que o reuso de dados seja possível, sendo também tolerante a falhas e uma estrutura de dados onde o usuário pode,

explicitamente, realizar a persistência de resultados na memória, controlar o seu particionamento, além de fornecer uma grande quantidade de operações [ZCD⁺12].

Um RDD só pode ser criado de duas formas: A partir dos dados de algum armazenamento estável ou realizando operações em algum RDD já existente, essas operações também podem ser chamadas de transformações.

Como o RDD é uma abstração de memória distribuída, foi feita uma comparação com o *Distributed Shared Memory*, ou DSM, para apontar as vantagens do RDD com relação ao DSM [ZCD⁺12]. A principal diferença é que a escrita em um RDD, como é somente leitura, se dá através das transformações de grão-grosso e a leitura pode ser realizada de forma grão-fino. Enquanto o DSM permite a escrita e leitura de dados em grão-fino.

Outra vantagem que o RDD possui é a facilidade de se executar tarefas de backup para caso um nó esteja muito lento. Em uma aplicação que utilize RDD, caso exista uma tarefa que esteja lenta, uma cópia desta tarefa é executada e nenhuma das duas execuções irá causar interferência durante as atualizações dos dados. A mesma coisa feita em cima de uma DSM seria bem difícil de se implementar, pois as tarefas principal e de backup estariam acessando o mesmo endereço de memória e uma poderia afetar a atualização realizada pela outra.

O RDD foi criado com foco em execuções *batch* que utiliza a mesma operação em todos dados presentes no conjunto de dados. Então, existem alguns tipos de aplicações que não se encaixam na proposta do RDD. Estas aplicações são assíncronas e utilizam atualizações de grão-fino.

2.6 Sistema de tolerância a falhas

Spark utiliza um sistema de tolerância a falhas a nível de tarefas. Fazendo ser possível reexecutar a tarefa a partir do ponto em que sofreu a falha [ZCD⁺12]. Este sistema de tolerância a falhas utiliza o método de linhagem de modificações do RDD para identificar o ponto em que a tarefa parou.

Este método de linhagem consegue tanto identificar em que ponto a tarefa parou, como consegue recriar o RDD, caso tenha ocorrido perda dos dados no momento da falha [ZCD⁺12].

Se uma tarefa falhar, esta tarefa é re-executada em outro nó desde que pai do estágio atual ainda esteja disponível. Se algum estágio se tornou indisponível (por exemplo porque uma saída de alguma operação do *map* foi perdida), esta tarefa será re-submetida para que seja recalculada toda as partições perdidas, em paralelo.

Além do método de tolerância a falhas por linhagem, Spark também possui um sistema de *checkpointing*. O *checkpointing* garante uma recuperação mais rápida quando o grafo de linhagem, representado por um grafo acíclico dirigido, for muito grande, guardando informações necessárias, de tempos em tempos, do estado em que se encontra a execução da tarefa.

2.7 HPAT

O *High Performance Analytics Toolkit* (HPAT) é um arcabouço desenvolvido na linguagem Julia pela *Intel Labs*¹ para o processamento de grandes conjuntos de dados. Este arcabouço está focado no uso do modelo de programação MapReduce para o desenvolvimento de aplicações a serem executadas.

Diferente de muitos arcabouços que são baseadas em bibliotecas, o HPAT é o primeiro arcabouço baseada em compilação que paraleliza automaticamente programas de análise de dados [TAS17]. Segundo a equipe de desenvolvimento do HPAT, esta paralelização automática em tempo de compilação é possível por alguns fatores como, por exemplo *checkpoint* automático e particionamento e paralelização específica de domínio.

As principais características do HPAT são um melhor uso da memória *cache* e a ausência de mecanismos de tolerância a falhas.

¹<https://github.com/IntelLabs/HPAT.jl> [Acessado em 18/12/2016]

Como mostra [TAS17], este arcabouço é composto pelos seguintes componentes: *Macro-Pass*, *Domain-Pass*, *Distributed-pass* e *HPAT Code Generation (MPI)*. Estes componentes tem as seguintes funções:

- **Macro-Pass:** É utilizado para transformar as extensões utilizadas pelo HPAT em chamadas de funções e anotações de tipo para poder ser possível a compilação em Julia. Então após esta transformação o código é compilado pelo compilador de Julia e então o código é enviado para o *Domain-Pass*.
- **Domain-Pass:** É responsável para adequar o código para as próximas etapas de processamento que vão ser realizadas pelo *Domain-IR* e *Parallel-IR*. Para isto ser possível o *Domain-Pass* detecta as extensões do HPAT e então transforma o código para uma forma mais adequada para a otimização a ser realizada nas próximas etapas. *Domain-IR* e *Parallel-IR* são componentes do *Parallel Accelerator*, outro projeto criado pela *Intel Labs*.
- **Distributed-Pass:** Tem a responsabilidade de traduzir algumas funções especiais, além de adicionar alguns parâmetros, de funções utilizadas pelo HPAT, necessários para o funcionamento adequado do programa. Este componente também realiza a identificação se determinada parte do código deve ser executada de forma particionada ou sequencial. Parte desta detecção foi realizada em estágios anteriores.
- **HPAT Code Generation:** Este componente é responsável em gerar o código *MPI* para executar a aplicação de forma adequada no arcabouço HPAT.

A comunicação entre os componentes pode ser observada na Figura 2.6 retirada de [TAS17].

Figura 2.6: Diagrama representando o fluxo de execução da ferramenta HPAT. Figura obtida do artigo [TAS17]

Na imagem 2.7 pode ser observada como é realizada a geração de código em cada etapa do fluxo de execução do HPAT. Este exemplo mostra a leitura de um conjunto de dados que tem armazenado alguns pontos. Na primeira linha da imagem podemos observar como escrevemos o código na linguagem Julia para a execução do HPAT. Este código sofre alterações enquanto passa nos componentes do arcabouço, até a última linha, que é o código para a execução em *C++/MPI*, gerado pelo último componente.

Além disso, o HPAT gera código em *C++/MPI* a partir de código Julia. Uma vez que a linguagem Julia permite a implementação de código em alto-nível de abstração, o HPAT favorece a escrita de código rápido sem abrir mão do desempenho do *MPI C++*.

Figura 2.7: Imagem representando a geração de código em cada componente da ferramenta HPAT. Imagem obtida em [TAS17]

Assim como o *Apache Spark*, o HPAT armazena dados em memória, garantindo um acesso rápido. No momento da escrita deste trabalho o HPAT não oferecia suporte para aplicações que não fossem do tipo *MapReduce* e também não permitia a execução de dados que excedessem a capacidade de memória da máquina que executa a aplicação. Além disso, a versão do HPAT utilizada tem suporte apenas para o formato de arquivo HDF5 [FHK⁺11], um formato de compressão hierárquico.

Durante o JuliaCon de 2016², foi mostrado que o HPAT atinge um desempenho muito superior ao Spark. Na apresentação foi mostrado, por exemplo, que atinge uma diferença de desempenho de 30x para o algoritmo de soma dos elementos de um vetor (*1D Sum*), e 23x com o algoritmo *K-Means*, que são os algoritmos que utilizamos nos nossos experimentos. A maior diferença de desempenho, mostrada na apresentação do arcabouço foi com o Monte Carlo Pi, que apresentou uma diferença de desempenho de 1680x. Dado que até o momento não foi possível encontrar dados mais concretos sobre esses ganhos de desempenho, além do que foi mostrado na conferência, propomos a nossa análise comparativa entre o arcabouço e o *Spark*.

2.8 HDF5 - Hierarchical Data Format 5

HDF5 é um conjunto de tecnologias composto por uma biblioteca, um modelo de dado e um formato de arquivo [FHK⁺11]. Este formato de arquivo aceita diferentes tipos de conjuntos de dados, possui um sistema de entrada e saída de dados flexível e eficiente e também foi projetado para grandes volumes de dados e complexos.

O modelo de dados é composto pelo conjunto de dados (*HDF5 datasets*), grupos (*HDF5 groups*), tipos de dados (*HDF5 datatypes*), links (*HDF5 links*) e atributos (*HDF5 attributes*).

- **HDF5 datasets:** São representados por vetor de variáveis que os elementos dos dados estão estabelecidos como um vetor multi-dimensional. Um *HDF5 dataset* pode ser utilizado com algumas estratégias diferentes, estas estratégias são escolhidas dependendo do tipo de armazenamento. Até o momento da publicação do artigo [FHK⁺11] as opções disponíveis eram: *contiguous*, *chunked* e *compact*.

A estratégia *contiguous* armazena o vetor de elementos como uma sequência única dentro da base de dados do HDF5. Já a estratégia *chunked* os dados são guardados como coleções de

²<http://juliacon.org/2016/abstracts.html#HPAT>. Esta apresentação pode ser assistida em <https://www.youtube.com/watch?v=Qa7nfaDacII> [Acessados em 23/03/2017]

Até o momento da escrita do artigo [FHK⁺11] os links eram apenas com comunicação unilateral, da fonte para o destino.

- **HDF5 attributes:** Os atributos são mecanismos para anotação dos *HDF5 datasets*, *HDF5 groups* e *HDF5 datatype*. Estes atributos precisam possuir nomes únicos. *HDF5 attributes* são similares aos datasets, pois necessitam de *dataspace* e *datatype* para a sua definição, porém possui a diferença de utilizar a função de anotação no conjunto de informações.

Mesmo o HDF5 sendo um formato de arquivo hierárquico eficiente, ele não é um arquivo simples de ser criado. Enquanto o Spark aceita formas mais simples de arquivo, ser limitado a um formato de arquivo mais complexo como o HDF5 pode trazer uma eficiência no processamento dos dados, mas faz com que isso seja uma desvantagem para o HPAT por causa de se limitar a aceitar somente este tipo de arquivo, enquanto os outros arcabouços tratam uma variedade maior de tipos de arquivo e dados.w

Neste capítulo foi visto os principais conceitos que envolvem este trabalho como: *Big data* e *MapReduce*. Alguns detalhes sobre os arcabouços HPAT e Spark e também de algumas tecnologias que as envolvem como RDD e HDF5. No próximo capítulo iremos mostrar alguns trabalhos que se relacionam com estes conceitos e também com os experimentos que envolvem este trabalho.

Capítulo 3

Trabalhos Relacionados

Neste capítulo serão apresentados alguns trabalhos relacionados. Estes trabalhos serão categorizados da seguinte maneira: na seção 3.1 serão apresentados alguns estudos que realizaram comparações entre ferramentas de *big data*, e principalmente que utilizam como principal modelo de programação o MapReduce. Na seção 3.2 será apresentado um trabalho que mostra a importância da reprodutibilidade dos experimentos focando na área de *big data*. Na seção 3.3 serão introduzidos estudos envolvendo a linguagem de programação Julia, esta linguagem é importante pois é uma linguagem nova e é utilizada pela ferramenta HPAT.

3.1 Comparações de ferramentas MapReduce

O artigo *Big Data Frameworks: A Comparative Study* [IAMJ16] apresenta uma comparação entre os frameworks Hadoop, Spark e Flink para a execução em *batch*, e uma comparação entre *Spark, Flink e Storm* [TTS⁺14] para a execução em *Stream*. A comparação entre estas ferramentas é realizada em termos de tamanho dos dados, na quantidade de máquinas que estão sendo utilizadas e o consumo de CPU, memória principal (RAM), disco e largura de banda em um ambiente com duas a dez máquinas. Para os experimentos é utilizado o algoritmo *Word Count*, que consiste na contagem de palavras presentes em um texto. Os dados utilizados foram *tweets* que foram coletados pelo *Apache Plume* e armazenados utilizando o *Hadoop File System* (HDFS) [SKRC10]. O resultado dos experimentos mostram que para as execuções tanto em *batch* como em *stream*, Spark lida melhor com base de dados grandes e complexas do que as outras ferramentas.

O artigo *Comparative Performance Analysis of a Big Data NORA Problem on a Variety of Architectures* [KB13] compara diferentes arquiteturas de computador utilizando o mesmo problema e a mesma ferramenta para esta comparação. Diferente do artigo de estudo comparativo [IAMJ16] mostrado anteriormente, que utilizava diferentes ferramentas na comparação, este utiliza somente a linguagem ECL, que tem um desempenho melhor do que a ferramenta Hadoop, e ainda tem uma cobertura maior de problemas que podem ser implementados. Foram comparados o uso de disco, memória, CPU e rede em cada uma das arquiteturas.

O artigo que introduz a nova versão da ferramenta Hadoop, a YARN [VMD⁺13], realiza uma comparação entre as duas versões do Hadoop. A comparação é feita com base no tempo de execução e no *throughput* da versão Hadoop 1.2.1 e a versão 2.1.0 (YARN). Para a comparação foram executadas diferentes aplicações em cada uma destas versões da Hadoop ex: *RandomWriter, Terasort, Shuffle*. Os resultados mostraram um ótimo ganho de desempenho para a versão YARN da ferramenta Hadoop.

No Artigo *Spark: Cluster Computing with Working Sets* [ZCF⁺10], é apresentada a ferramenta Spark e também é realizada uma comparação entre Spark e Hadoop. Para esta comparação foi utilizado o algoritmo de regressão logística e de alternar os quadrados mínimos. Foi utilizado o tempo de execução como parâmetro de comparação. O resultado obtido mostra uma diferença de tempo de execução de até 10x para o regressão logística e de aproximadamente 2.8x para o algoritmo de alternar os quadrados mínimos.

O artigo [TAS17] é utilizado para apresentar o arcabouço e o que ela tem de diferente de outros *frameworks* de Big Data. Uma dessas diferenças é que HPAT é baseada no compilador e não em biblioteca, como é o caso do Spark. Também são apresentados alguns resultados de comparação entre os arcabouços HPAT e Spark e também com a versão MPI/C++ do código gerado pela compilação da aplicação executada no HPAT. Os resultados desta comparação mostram uma diferença de desempenho de 14x, na aplicação *1D Sum Filter*, e 400x da aplicação Monte Carlo Pi. Os experimentos foram executados no super computador Cori na LBL/NERSC e também utilizando instâncias c4.8xlarge com 36 vCPUs na nuvem AWS.

Com uma proposta similar ao apresentado pela ferramenta HPAT, de utilizar o MPI para ter um desempenho maior do que o Spark. O trabalho [ASS⁺17] apresenta uma ferramenta que tem como proposta a utilização de elementos de HPC (*High Performance Computing*) em conjunto com ferramentas de *Big Data*. A ferramenta criada pela equipe insere o uso de MPI no Spark, desta forma podem ser realizados processamentos utilizando técnicas de HPC. Além de introduzir esta nova ferramenta, o artigo também realiza uma comparação entre Spark+MPI e o Spark puro e os resultados apresentados mostram que a ferramenta proposta obtém cerca de uma ordem de grandeza mais eficiência do que o Spark sem as alterações.

Utilizando a mesma ideia de mesclar elementos de HPC com ferramentas de *big data*, o artigo [LLW⁺14] apresenta a ferramenta DataMPI que realiza, assim como a apresentada por [ASS⁺17], uma mistura de MPI com algum arcabouço de *big data*. No caso desta ferramenta a ferramenta de processamento de dados massivos utilizada é uma implementação similar ao Hadoop. O artigo também realiza uma comparação entre a ferramenta proposta com outras ferramentas que realizam o processamento de *big data*. Nos experimentos foram utilizados cinco aplicações diferentes *Word-Count*, *TeraSort*, *PageRank*, *K-means*, *Top-K*, as comparações foram realizadas entre o DataMPI e Hadoop e mostram um grande ganho de desempenho do DataMPI em comparação com o Hadoop.

Outros trabalhos relacionados podem ser vistos comparando ferramentas com propósitos diferentes ao deste trabalho. Em [ZDL⁺13] são comparadas as ferramentas *Spark Streaming* com *Storm*. Para esta comparação são utilizados os algoritmos *grep*, *word count* e *top k count*. Já em [XGFS13] é realizado uma comparação entre as ferramentas *Graphx*, *Mahout* e *Power Graph* utilizando o algoritmo de *page rank* em um grafo com 4.8 milhões de vértices e 69 milhões de arestas.

Alguns destes resultados são reproduzíveis, como é o caso dos obtidos no artigo de [IAMJ16] que utiliza um ambiente de dez máquinas virtuais, alguns algoritmos simples e dados que podem ser obtidos de forma simples. O artigo [TAS17] utiliza das mesmas ferramentas que foram utilizadas nesta pesquisa, HPAT e Spark, e é uma das fontes que inspiraram a realização desta pesquisa, no entanto não há todas as informações necessárias para a reprodução dos experimentos apresentados. Já o [ZCF⁺10] utiliza um ambiente que é possível ser replicado, mas não informa a base de dados utilizada em seus experimentos. Os outros artigos possuem ambientes difíceis de serem replicados, ou não dão todas as informações necessárias para reproduzir os experimentos realizados.

Baseado nos artigos apresentados, decidimos utilizar tempo de execução como base para as comparações entre os arcabouços HPAT e Spark. As aplicações foram escolhidas por terem sido utilizadas em artigos [LLW⁺14, TAS17] e também para tentar replicar os resultados obtidos pelo HPAT em seu artigo utilizando um ambiente de execução menor do que o utilizado em seu artigo.

3.2 Reprodutividade

Diante do que é mostrado em [Bor12], é de grande importância que os dados estejam disponíveis, principalmente se tratando de um trabalho realizado em um local público, para que haja a possibilidade de reproduzir os experimentos realizados. Ter a possibilidade de realizar os experimentos demonstrados é importante tanto para a validação dos resultados apresentados como para o melhor entendimento do que está sendo proposto.

3.3 Linguagem Julia

A linguagem de programação *Julia* [BKSE12, BEKS14] tem como objetivo permitir o desenvolvimento de programas para Computação Científica, Paralela e Distribuída, e de Alto Desempenho num alto nível de abstração. Utilizando o modelo de paralelismo de *troca de mensagens*, a linguagem fornece abstrações de alto nível para a programação paralela e distribuída. Interfaces para diversas ferramentas para programação paralela e distribuída estão disponíveis ¹. Julia também implementa uma interface nativa a *threads* [Kno14].

A linguagem Julia é multi-paradigma, permitindo ao programador utilizar construções sintáticas de linguagens derivadas de linguagens funcionais, procedurais e orientadas a objetos. Uma das mais importantes características da linguagem é a capacidade de realizar *despacho múltiplo* de métodos, baseando-se nos tipos dos parâmetros. Em Julia, uma *função* pode ser considerada como uma classe abstrata. O programador pode então definir *métodos* relacionados a uma determinada função, especificando diferentes tipos para os argumentos definidos na função.

Julia tem sido usada em diferentes áreas da Computação Científica, Paralela e Distribuída, e de Alto Desempenho. Dunning et al. utilizaram Julia na implementação de uma linguagem para modelagem de problemas em Otimização Matemática [DHL15], e Lubin e Dunning usaram a linguagem para solução de problemas em Pesquisa Operacional [LD15]. A empresa Intel implementa ferramentas para melhora de desempenho da linguagem ².

¹<https://github.com/JuliaParallel> [Acessado em 23/03/2017]

²<https://github.com/IntelLabs/ParallelAccelerator.jl> [Acessado em 23/03/2017]

Capítulo 4

Experimentos

Neste Capítulo serão apresentados os experimentos realizados comparando Spark e HPAT. Primeiro será apresentado o ambiente de execução e um experimento realizado para verificar sua estabilidade em diferentes turnos do dia na Seção 4.1.

Na Seção 4.2 mostraremos os dois experimentos realizados em um ambiente sem ocorrência de falhas e um experimento realizado para simular um ambiente em que possui uma probabilidade de ocorrer falhas a cada segundo.

4.1 Experimento Preliminar

Para realizar uma melhor comparação entre os cenários de execução, foi realizado um experimento para verificar se existe diferença entre os tipos de máquina virtual, MV. Este experimento possui o intuito de verificar se pode ser realizada uma comparação entre as MVs, caso os tempos de execução sejam estáveis.

Tipos de máquina virtual	vCPUs	Memória	Disco
n1-highmem-8	8	52GB	1TB
n1-highmem-2	2	13GB	300GB
n1-highmem-1	1	6.5GB	200GB

Tabela 4.1: *Tipos de máquinas virtuais*

O ambiente utilizado para os experimentos apresentados nas Sub-seções 4.2.1 e 4.2.2 foi o *Google Compute Engine*, as máquinas virtuais utilizaram o limite permitido para uma conta gratuita, mais precisamente 8 vCPUs e 52GB de memória RAM no total. Os tipos de máquina virtual podem ser observada na Tabela 4.1. Estas máquinas estão localizadas na região us-central1-b¹, que está localizada no estado de Iowa nos Estados Unidos da América, o processador das MVs é Intel Xeon E5 v3 com frequência de 2.3GHz e arquitetura Haswell².

Além disso, os experimentos foram separados em três cenários diferentes, utilizando os diferentes tipos de MVs. O Cenário 1 é representado por uma MV com 8 vCPUs. O Cenário 2 possui 4 MVs, cada uma com 2 vCPUs. O Cenário 3 possui 8 MVs, cada uma com 1 vCPU. Mais detalhes sobre cada cenário podem ser vistos na Tabela 4.2.

Para as comparações foram utilizados a formula de Leibniz para o cálculo do valor do número Pi e a transferência de um arquivo de 115MB. O arquivo utilizado para a transferência foi criado utilizando um código em java que gera uma lista de números aleatórios. Os tipos de MVs utilizadas podem ser observado na tabela 4.2. Para o cenário 1 foi utilizada somente uma MV e nos cenários 2 e 3 foram utilizadas duas MVs. A transferência, nas MVs dos cenários 2 e 3, foi realizada de uma MV para outra MV. Já no cenário 1 a transferência foi realizada de um processo para outro.

¹<https://cloud.google.com/compute/docs/regions-zones/regions-zones> [Acessado em 24/11/2017]

²<https://cloud.google.com/compute/docs/machine-types> [Acessado em 24/11/2017]

Figura 4.1: Gráfico com medições da transferência de arquivo e do cálculo do Pi no cenário 1.

Pelos resultados obtidos no Cenário 1, que pode ser visto na Figura 4.1, podemos ver que as execuções tanto do cálculo do Pi como da transferência de arquivo permaneceram bastante estáveis durante os quatro turnos do dia (madrugada, manhã, tarde e noite). O mesmo ocorreu tanto no Cenário 2 como no Cenário 3, mantendo a estabilidade mostrada no gráfico da Figura 4.1, mas com um pouco de diferença nos tempos de execução. Tomando em consideração todos as 400 amostras de cada Cenário, temos 99,5% de confiança que a média obtida nas execuções (3.963s para a computação do Pi e 0.299s para a transferência de arquivo).

Dada a estabilidade dos valores encontrados nos experimentos preliminares, vamos partir da hipótese que a nuvem é bastante estável. Como em alguns casos houveram poucos pontos fora do esperado, todos os experimentos seguintes foram repetidos ao menos 30 vezes.

Com isso podemos concluir que os ambientes de execução são estáveis e com tempos de execuções próximos, sendo possível realizar uma comparação entre eles nos experimentos seguintes.

4.2 Experimentos

Os experimentos foram realizados em dois cenários com memória distribuída e um cenário com memória compartilhada. Todos os cenários experimentais sem falhas utilizaram máquinas da *Google Compute Engine* (GCE) como pode ser visto nas Tabelas 4.1 e 4.2. Dos cenários com memória distribuída, um utiliza 8 Máquinas Virtuais (MVs) com uma vCPU cada e o outro 4 MVs com duas vCPUs cada. O cenário com memória compartilhada consiste em uma MV com 8 vCPUs. A Tabela 4.2 apresenta os detalhes dos cenários.

Cada experimento consiste na realização de dois tipos de execuções em cada ambiente. No primeiro tipo realizamos uma *execução alternada* entre os arcabouços HPAT e *Apache Spark*. A intenção da execução alternada é ignorar tempo de compilação decorrente da utilização da linguagem Julia no HPAT.

No segundo tipo realizamos uma *execução sequencial*, onde cada arcabouço executa suas iterações sequencialmente. Cada tipo de execução foi realizado 30 vezes em cada cenário, e medimos o tempo de execução dos dois arcabouços em cada aplicação e ambiente de execução, gerando os dados apresentados na Seção 4.3.

Para os experimentos também foram utilizadas as versões 2.0.1 do Spark e a versão do HPAT é a obtida a partir de uma versão do código em seu site do *GitHub*³.

Cenário	SO	Máquinas Virtuais(MVs)	vCPUs	Memória / MV	Disco / MV
1	Ubuntu 14.04	1	8	52GB	1TB
2		4	2	13GB	300GB
3		8	1	6.5GB	200GB

Tabela 4.2: Configuração dos cenários de execução

4.2.1 Experimento 1: Soma dos Elementos de um Vetor

Este algoritmo simples consiste na soma de todos os elementos em um vetor unidimensional, gerando um único valor como saída. Escolhemos este algoritmo pois ele não permite que o HPAT se beneficie de seus mecanismos para manipulação de *cache* de forma mais sofisticada.

Um arquivo no formato `csv` foi utilizado como entrada para as execuções do *Apache Spark*, e um arquivo *HDF5* para as execuções do HPAT. Esses arquivos representam vetores com 10^9 números, cuja soma é a saída da aplicação.

A entrada para a aplicação foi gerada utilizando um programa, criado pela equipe de desenvolvimento do HPAT, que cria como saída um arquivo `csv` e outro no formato *HDF5*. Para a execução deste programa é passado como parâmetro o tamanho do vetor. Em cada iteração realizada neste programa, um número aleatório é gerado e gravado tanto no arquivo `csv` como no arquivo *HDF5*, garantindo que ambos arquivos tenham a mesma representação do vetor de números.

Este algoritmo, em um ambiente MapReduce, funciona da seguinte maneira: Os números do vetor são particionados, ou mapeados, em cada um dos *workers*. Após o mapeamento, é executada a uma redução, que tem como objetivo somar os números presentes nas partições, terminando em um único valor que será a resposta da soma.

4.2.2 Experimento 2: K-means

A segunda aplicação utilizada nos experimentos foi o algoritmo de clusterização *K-means*. O algoritmo cria K grupos, ou *clusters*, baseando-se na distância entre os dados e seus pontos centrais. Os pontos centrais são reajustados até n vezes, sendo este n a quantidade máxima de iterações que o algoritmo irá realizar.

Os dados para a entrada da aplicação [KYJ13] contêm 434874 instâncias. Cada uma das 30 execuções realizou 5000 iterações e usou 15 pontos centrais gerados aleatoriamente pelos arcabouços. Para este experimento foi necessário a alteração de parte do código do *Spark MLLib* para o algoritmo ficar com a execução similar a utilizada pelo HPAT. Esta alteração foi necessária pois o algoritmo disponibilizado pelo arcabouço HPAT, para o *k-means*, não utiliza validações de convergência, executando a quantidade de iterações que é fornecida como parâmetro. Enquanto a execução originalmente fornecida pelo Spark utiliza a validação de convergência, podendo parar a execução antes de terminar todas as iterações. Então foi necessário modificar o código para que ambos os arcabouços realizem uma execução semelhante e seja possível a comparação por tempo de execução entre elas neste experimento.

4.2.3 Simulação de Ambiente com Falhas

Para realizar a simulação de um ambiente com falhas, foi executado o algoritmo da soma dos elementos de um vetor unidimensional, o mesmo utilizado no Experimento 1, descrito na Subseção 4.2.1, em um ambiente real para coletar o tempo de adicional para o Spark se recuperar de uma falha.

³Versão do código do HPAT utilizada nos experimentos: <https://github.com/IntelLabs/HPAT.jl/commit/cad25a142c6ccd747335c7e4dd14e09df0b328d6>

O cenário utilizado para as execuções foi uma máquina virtual com uma configuração similar a do Cenário 2 apresentada na tabela 4.2, no entanto esta máquina possui algumas diferenças na configuração como o processador *Intel Core i7 4850HQ* 2,3 GHz. Esta execução também possui a diferença de que não foi feita no *Google Compute Engine*, mas a máquina virtual foi executada no computador pessoal do autor.

Primeiro realizamos as execuções sem falhas para termos os tempos de execução do algoritmo sem falhas e termos ideia de quanto tempo seria acrescentado depois de adicionarmos uma falha. Então separamos a média dos tempos obtidos em três diferentes pedaços, categorizando em início, meio e fim da execução. A intenção desta separação era para saber o quanto a execução seria afetada dependendo de que fatia de tempo fosse causada a falha. Esta etapa do experimento foi realizada somente com o Spark, porque o HPAT não possui um sistema de tolerância a falhas, ou seja, cada vez que uma execução do HPAT falha ela tem que ser reinicializada, não sofrendo somente um atraso na execução.

O tempo obtido desta execução será utilizada como entrada para o simulador, assim como o tempo de execução dos outros dois experimentos. A falha utilizada pelo simulador segue a distribuição de Weibull [RRB10]. A probabilidade de ocorrer falha é definida, então em cada passo da simulação (que representa um segundo de execução) um valor aleatório seguindo uma distribuição uniforme é gerado. Se o valor gerado for menor ou igual ao valor de probabilidade definido, então a falha ocorre. Nesta simulação há a ocorrência de muitas falhas.

4.3 Resultados

Esta Seção apresenta os dados obtidos com as medições de tempo de execução do *Apache Spark* e HPAT nas aplicações apresentadas na Seção 4.2.

4.3.1 Experimento 1

Figura 4.2: Gráficos com as medições de tempo de execução no Experimento 1 Cenário 1. Os gráficos superiores apresentam boxplots para os dois tipos de execuções. Os Gráficos inferiores apresentam a duração de cada iteração para os dois tipos de execuções.

Este cenário consiste em 1 Máquina Virtual com 8 vCPUs e 52GB de memória, como descrito

na Tabela 4.2. Como pode ser observado nos gráficos superiores, no ambiente de memória compartilhada, pode ser concluído que o HPAT é mais eficiente do que o Spark, em termos de tempo de execução. Como podemos ver nos gráficos inferiores da figura 4.2, o HPAT também apresenta mais estabilidade do que os tempos de execução apresentados pelo Spark.

A estabilidade do HPAT, neste cenário, pode ser observada por seu desvio padrão de 0.02, obtendo um tempo de execução máximo 1.45 minutos e um mínimo de 1.37 minutos. Já o Spark possui um desvio padrão de 0.77, um tempo de execução máximo de 12.21 minutos e um mínimo de 9.52 minutos na execução sequencial.

Na execução alternada temos que o desvio padrão do HPAT se mantém em 0.02 com o mínimo caindo para 1.30 minutos e o máximo em 1.37 minutos. O Spark, na execução alternada, tem um desvio padrão 0.7, menor do que o da execução sequencial, e com tempo de execução máximo e mínimo menores, de 12.14 minutos e 9.3 minutos respectivamente.

Figura 4.3: Gráficos com as medições de tempo de execução no Experimento 1 Cenário 2. Os gráficos superiores apresentam boxplots para os dois tipos de execuções. Os Gráficos inferiores apresentam a duração de cada iteração para os dois tipos de execuções.

No entanto, no cenário 2 ocorre uma perda de estabilidade no arcabouço HPAT com relação ao Spark, mesmo que mantendo a eficiência mostrada nos outros dois cenários. Pelo *boxplot* de fluxo alternado, gráfico superior à esquerda da figura 4.3, podemos ver que o HPAT é mais eficiente que o arcabouço Spark. Isto também pode ser observado pelos tempos de execuções máximo e mínimo. O HPAT obteve um tempo de execução máximo de 2.07 minutos e um mínimo de 0.99 minutos, já o Spark obteve 3.67 e 3.4 minutos respectivamente. Isto mostra que o HPAT foi mais eficiente que o Spark para este cenário.

Já no gráfico inferior à esquerda da figura 4.3 podemos ver a progressão dos tempos de execuções dos arcabouços, pode ser observado que o Spark se manteve com uma variância baixa entre as suas execuções, com um desvio padrão de 0.08, já o HPAT obteve um desvio padrão de 0.31. Isso mostra que o Spark neste cenário obteve uma estabilidade maior do que a do HPAT.

Através do boxplot, mostrado no gráfico superior à direita da figura 4.3 podemos ver que o resultado das execuções mostradas anteriormente se mantém, porém o HPAT teve uma grande melhora em seu tempo de execução, tendo a maioria de suas execuções abaixo dos 2 minutos de tempo de execução. Já o Spark se manteve entre os 4 minutos, mesmo tempo obtido na execução alternada, mostrando que o tempo de compilação da aplicação não afeta as execuções em Spark,

mas afeta as execuções do HPAT.

Já com o gráfico em linha, mostrado no gráfico inferior à direita da figura 4.3 podemos observar que o tempo de execução em Spark é bem similar ao da execução alternada. Já a execução do HPAT se mantém mais estável nesta execução, se mantendo com uma variância mais baixa e com tempo de execução próximo de 1 minuto. Assim como na execução alternada, o Spark apresenta uma execução mais estável do que o HPAT para este cenário, no entanto permanecendo menos eficiente considerando o tempo de execução.

No cenário 3 ocorre um fenômeno semelhante ao observado no primeiro cenário. Onde o HPAT possui um desempenho e uma estabilidade maior do que a apresentada pelo Spark. O Spark, neste cenário, possui um desvio padrão, maior do que o obtido no cenário 1, de 1.78 enquanto o HPAT possui desvio padrão de 0.18 na execução sequencial e 1.73 do Spark contra 0.28 do HPAT na execução alternada. Isto mostra a estabilidade do HPAT e a maior instabilidade do Spark.

4.3.2 Experimento 2

Figura 4.4: Gráficos com as medições de tempo de execução no Experimento 2 Cenário 1. Os gráficos superiores apresentam boxplots para os dois tipos de execuções. Os Gráficos inferiores apresentam a duração de cada iteração para os dois tipos de execuções.

No gráfico superior à esquerda da figura 4.4 podemos observar que o HPAT é mais eficiente do que o Spark, levando em consideração o tempo de execução destes arcabouços. Com o Spark com uma variação maior do seu tempo de execução em comparação ao HPAT.

Esta variação pode ser melhor vista no gráfico inferior à esquerda da figura 4.4. Neste gráfico pode ser observado que a execução do HPAT está mais estável do que a execução Spark, se mantendo estável em uma faixa de tempo, enquanto a execução do Spark tem uma grande variação no decorrer das execuções.

A partir do gráfico superior à direita da figura 4.4, podemos deduzir que o HPAT é mais eficiente do que o Spark, considerando o tempo de execução. Com o Spark tendo média, aproximadamente 11.5 minutos, enquanto o HPAT tem uma média de aproximadamente próximo a 7 minutos.

Já o Spark mostra uma grande mudança no tempo de execução na quinta e na vigésima primeira execuções, no entanto entre as outras execuções também existe uma variação maior do que a apresentada pelo HPAT.

Resultados similares ao apresentado no cenário 1 foram obtidos no cenário 2. Com o HPAT sendo mais estável e mais eficiente, levando em consideração o tempo de execução, do que o Spark.

Figura 4.5: Gráficos com as medições de tempo de execução no Experimento 2 Cenário 3. Os gráficos superiores apresentam boxplots para os dois tipos de execuções. Os Gráficos inferiores apresentam a duração de cada iteração para os dois tipos de execuções.

No entanto, no cenário 3 temos um cenário onde o HPAT é mais eficiente em ambos os tipos de execução como pode ser observado pelos boxplots da figura 4.5. No entanto, nos gráficos inferiores temos que tanto o Spark como o HPAT possuem grandes variações nos tempos de execuções.

No gráfico de linha da execução alternada temos que o desvio padrão do HPAT é de 0.52, já o desvio padrão do arcabouço Spark é de 0.68, sendo possível afirmar que para este tipo de execução temos que o HPAT possui mais estabilidade do que o Spark. Já na execução sequencial, temos que o desvio padrão do HPAT é de 0.88, enquanto a execução do Spark possui um desvio padrão de 0.76, podendo afirmar que o Spark, na execução sequencial, é mais estável do que o HPAT.

4.3.3 Resumo dos Resultados

Pelos resultados apresentados nos dois experimentos e pela tabela 4.3 podemos observar que em ambos os experimentos o arcabouço HPAT obteve melhores resultados do que o *Spark*. Esta conclusão pode ser obtida pelos valores de Max, Min e desvio padrão mostrados na tabela 4.3.

Com os números mostrados na tabela, pode ser concluído que o arcabouço HPAT obteve em todos os experimentos uma eficiência maior do que a obtida nas execuções do *Spark*. Já na questão da estabilidade, o HPAT foi menos estável do que *Spark* em todo o cenário 2 do experimento 1 e na execução sequencial do cenário 3 do experimento 2.

4.3.4 Simulação de Ambiente com Falhas

Este experimento é importante para observar como os arcabouços se comportam em um ambiente no qual existe a possibilidade de ocorrer falha a cada segundo. Também vale notar que o HPAT não possui um mecanismo para a tolerância de falhas, enquanto o *Spark* possui este tipo de mecanismo.

Para a simulação do Experimento 1, foi utilizado a probabilidade de falha começando em 0,0% até a probabilidade de 6,0% aumentando 0,5% por vez. Já na simulação do Experimento 2 a pro-

Experimento	Cenário	Tipo de Execução	Arcabouço	Max	Min	Desvio Padrão
1	1	Sequencial	HPAT	1.45	1.37	0.02
		Sequencial	Spark	12.21	9.52	0.77
		Alternada	HPAT	1.37	1.3	0.02
		Alternada	Spark	12.14	9.3	0.7
1	2	Sequencial	HPAT	1.23	0.9	0.1
		Sequencial	Spark	3.49	3.37	0.04
		Alternada	HPAT	2.07	0.99	0.31
		Alternada	Spark	3.67	3.4	0.08
1	3	Sequencial	HPAT	2.55	0.94	0.28
		Sequencial	Spark	10.42	6.34	1.73
		Alternada	HPAT	1.71	0.74	0.18
		Alternada	Spark	10.6	6.25	1.78
2	1	Sequencial	HPAT	6.99	6.84	0.04
		Sequencial	Spark	12.61	7.87	0.94
		Alternada	HPAT	7.07	6.83	0.06
		Alternada	Spark	12.4	8.28	0.87
2	2	Sequencial	HPAT	8.08	6.96	0.25
		Sequencial	Spark	21.35	9.55	3.8
		Alternada	HPAT	8.35	6.87	0.35
		Alternada	Spark	21.14	13.01	2.36
2	3	Sequencial	HPAT	9.96	6.23	0.88
		Sequencial	Spark	14.78	12.44	0.76
		Alternada	HPAT	7.46	5.64	0.52
		Alternada	Spark	14.5	12.31	0.68

Tabela 4.3: *Resumo dos resultados obtidos nos experimentos*

Figura 4.6: Gráfico da simulação do Experimento 1 Cenário 1 e Experimento 2 Cenário 2. Cada ponto representa 100 execuções e possui uma barra representando o erro padrão.

babilidade começou também em 0,0% e foi até uma probabilidade de 0,5% aumentando 0,05% por vez. Para cada probabilidade de falha o respectivo experimento foi executado 100 vezes.

Como pode ser visto na Imagem 4.6, o HPAT começou a ter perda de desempenho, ou aumento no tempo de execução, após uma pequena porcentagem de probabilidade de falha. Isto acontece por causa da falta de mecanismo de tolerância a falhas neste arcabouço, então o HPAT tem que re-executar a aplicação desde o início.

A simulação mostra que apesar do incrível desempenho do HPAT nos Experimentos 1 e 2, onde foram executados em um ambiente sem falhas, a falta de um mecanismo de tolerância a falhas do HPAT permite o Spark ter um melhor desempenho e estabilidade em um ambiente com probabilidade de ocorrer falhas. No entanto, quando o HPAT se encontra em um ambiente com falhas, continua com um desempenho melhor do que o Spark. Nestes ambientes ocorrem muitas falhas.

Capítulo 5

Conclusões

A análise dos resultados apresentados na Seção 4.3 nos ajuda a responder: "o HPAT é uma alternativa para o Spark?" levando em consideração o tempo de execução dos arcabouços. Nos dois tipos de ambientes, sem falha, podemos observar um desempenho e estabilidade melhor do HPAT em todos os cenários. Já na simulação de ambiente com falhas, o HPAT apresenta um crescimento do tempo de execução com o aumento da probabilidade de ocorrência de falhas, enquanto o Spark apresenta um pequeno aumento no seu tempo de execução. Os casos, em que o HPAT fica com um tempo maior do que o apresentado pelo Spark, possuem uma quantidade de falhas grande.

O HPAT apresentou uma melhora significativa de desempenho em relação ao *Apache Spark* nos cenários utilizados. Além disso, o HPAT apresentou uma estabilidade maior em quase todos os cenários e tipos de execuções analisados. Pudemos observar também o efeito do tempo de geração do código MPI C++ no tempo de execução do HPAT que, apesar de impactar no tempo de execução, permite atingir *speedup* significativo em relação à abordagem do *Apache Spark*.

Apesar de uma tendência claramente positiva em relação ao uso do HPAT, é importante ressaltar que no cenário mais favorável o ganho do HPAT foi da ordem de 10 vezes em relação ao Spark. Sendo que os criadores do arcabouço tiveram ganhos entre 14 e 400 vezes. Logo, sim há uma vantagem na utilização de HPAT em um ambiente sem falhas, mas no nosso caso, um pouco menor do que a apresentada inicialmente.

Já em um ambiente com uma probabilidade de ocorrerem falhas a cada segundo, descobrimos que o HPAT tem um aumento no tempo de execução, tendo um desempenho pior do que o Spark mesmo com uma probabilidade relativamente baixa de falha. Este fato pode ser explicado pelo fato do HPAT não possuir um mecanismo de tolerância a falhas e o Spark possuir este mecanismo. Com isso, o HPAT realiza uma re-execução da aplicação a cada ocorrência da falha, enquanto o Spark tem um atraso em sua execução sem a necessidade de re-inicializar a execução.

Com isso, temos que para ambientes que não ocorrem falhas ou com uma probabilidade muito baixa de ocorrência de falhas, o HPAT é uma boa alternativa para o Spark, levando em consideração o tempo de execução para esta comparação. Já em ambientes com falhas, temos que o HPAT tem uma perda de desempenho que leva o Spark a ser uma melhor opção.

A implementação do HPAT na linguagem Julia visa permitir que o usuário escreva código em alto-nível de abstração. Apesar da maior velocidade de implementação, programas escritos em alto-nível comumente sacrificam seu desempenho em favor da facilidade de prototipação e da legibilidade de código. A abordagem do HPAT busca contornar essa limitação ao gerar código MPI C++ a partir do código em Julia. A homiconicidade da linguagem Julia, característica que compartilha com linguagens como LISP e Prolog, torna simples e eficiente usá-la na implementação de geradores de código e compiladores.

Os resultados apresentados mostram que o *overhead* da geração de código do HPAT foi praticamente desprezível em relação ao tempo de execução das aplicações medidas. Soma-se a isso o significativo *speedup* atingido por esse código gerado em relação ao *Apache Spark*. Podemos concluir portanto que a abordagem do HPAT oferece grandes vantagens de desempenho em relação à execução na JVM adotada pelo *Apache Spark*, nos cenários e aplicações apresentados neste trabalho.

O melhor desempenho do HPAT nas execuções sequenciais também pode ser explicado por suas estratégias de manipulação de *cache*.

Os códigos e dados utilizados para esta pesquisa podem ser encontrados no *GitHub* do autor¹.

5.1 Trabalhos Futuros

Alguns trabalhos futuros podem ser recomendados para melhorar a análise realizada por este trabalho. Experimentos utilizando ambientes com um maior número de máquinas virtuais e memória é recomendado para uma análise mais realista ao que se diz respeito a análise de *Big Data*.

Também seria interessante realizar experimentos utilizando diferentes aplicações, como *Word-Count*, *PageRank* e *TeraSort*. Outras aplicações, que foram utilizadas para comparação de arcabouços de *Big Data*, podem ser encontradas no Capítulo 3.

O HPAT, desde que foram realizados os experimentos deste trabalho, já lançou novas versões, também possuindo uma versão que utiliza Python como linguagem para a criação das aplicações. Seria interessante realizar uma comparação utilizando as duas versões deste arcabouço, em Julia e em Python. Spark também lançou novas versões desde então, podendo ter realizado alguma melhoria em seu desempenho e otimizações, podendo também ser incluído em futuras comparações.

¹<https://github.com/rafaelcarv/mestrado>

Apêndice A

Dificuldades Encontradas

Durante o período inicial da utilização do arcabouço HPAT algumas dificuldades foram encontradas em sua instalação e futuras atualizações. Por falta de uma documentação bem escrita e mais intuitiva poderia ter poupado algum tempo. Para solucionar a maior parte dos problemas em que o autor obteve foi a partir de trocas de *e-mails* e *tickets* no *GitHub* do arcabouço.

A primeira dificuldade foi conseguir instalar, de maneira apropriada, o HPAT no computador pessoal com Ubuntu 16.04, pela falta de uma variável de ambiente na configuração padrão do HPAT. Onde foi descoberta após trocas de *e-mail* com os desenvolvedores do arcabouço. Esta variável era necessária no Ubuntu 16.04 e não era necessária no 14.04.

Após realizar a instalação e configuração do HPAT, problemas foram encontrados após a realização de atualizações do repositório. Após diversas tentativas, as atualizações só eram possíveis quando era realizada a remoção das pastas do HPAT e depois a reinstalação do arcabouço. Foi então quando foi utilizada uma versão fixa do HPAT utilizando *checkout* pelo Git.

Outro problema encontrado durante as execuções dos experimentos, e depois confirmado pelos desenvolvedores, foi um problema quando os dados utilizados na aplicação fosse maior do que a memória RAM. Quando os dados utilizados, ou gerados, durante a execução ultrapassavam a quantidade de memória RAM total compartilhada entre os *workers* ocorria um estouro de memória e a execução falhava. Após entrar em contato com os desenvolvedores foi descoberto que o HPAT não trata *swap* na memória, ocorrendo este erro.

Referências Bibliográficas

- [ASS⁺17] Michael Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capota, Zheguang Zhao, Subramanya Duloor, Nadathur Satish e Theodore L Willke. Bridging the gap between hpc and big data frameworks. *Proceedings of the VLDB Endowment*, 10(8), 2017. 18
- [BEKS14] Jeff Bezanson, Alan Edelman, Stefan Karpinski e Viral B Shah. Julia: A fresh approach to numerical computing. *arXiv preprint arXiv:1411.1607*, 2014. 19
- [BKSE12] Jeff Bezanson, Stefan Karpinski, Viral B Shah e Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012. 19
- [Bor12] Christine L Borgman. The conundrum of sharing research data. *Journal of the American Society for Information Science and Technology*, 63(6):1059–1078, 2012. 18
- [DeV16] Ashley DeVan. The 7 v’s of big data. <https://www.impactradius.com/blog/7-vs-big-data/>, 2016. Accessed: 16/11/2017. 1, 5
- [DG08] Jeffrey Dean e Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 1, 6, 7, 8
- [DHL15] Iain Dunning, Joey Huchette e Miles Lubin. Jump: A modeling language for mathematical optimization. *arXiv preprint arXiv:1508.01982*, 2015. 19
- [FHK⁺11] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal e Dana Robinson. An overview of the hdf5 technology suite and its applications. Em *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, páginas 36–47. ACM, 2011. xi, 13, 14, 15
- [Fir17] George Firican. The 10 vs of big data. <https://tdwi.org/articles/2017/02/08/10-vs-of-big-data.aspx>, 2017. Accessed: 16/11/2017. 1, 5
- [GGL03] Sanjay Ghemawat, Howard Gobioff e Shun-Tak Leung. The google file system. Em *ACM SIGOPS operating systems review*, volume 37, páginas 29–43. ACM, 2003. 1
- [IAMJ16] Wissem Inoubli, Sabeur Aridhi, Haithem Mezni e Alexander Jung. Big data frameworks: A comparative study. *arXiv preprint arXiv:1610.09962*, 2016. 17, 18
- [KB13] Peter M Kogge e David A Bayliss. Comparative performance analysis of a big data nora problem on a variety of architectures. Em *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, páginas 22–34. IEEE, 2013. 17
- [Kno14] Tobias Knopp. Experimental multi-threading support for the julia programming language. Em *Proceedings of the 1st First Workshop for High Performance Technical Computing in Dynamic Languages*, páginas 1–5. IEEE Press, 2014. 19
- [KYJ13] Manohar Kaul, Bin Yang e Christian S Jensen. Building accurate 3d spatial networks to enable next generation intelligent transportation systems. Em *2013 IEEE 14th International Conference on Mobile Data Management*, volume 1, páginas 137–146. IEEE, 2013. 23

- [Lan01] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6:70, 2001. 1, 5
- [LD15] Miles Lubin e Iain Dunning. Computing in operations research using julia. *INFORMS Journal on Computing*, 27(2):238–248, 2015. 19
- [LLW⁺14] Xiaoyi Lu, Fan Liang, Bing Wang, Li Zha e Zhiwei Xu. Datampi: extending mpi to hadoop-like big data computing. Em *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, páginas 829–838. IEEE, 2014. 18
- [MCB⁺11] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh e Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011. 1
- [RRB10] Mustafizur Rahman, Rajiv Ranjan e Rajkumar Buyya. Reputation-based dependable scheduling of workflow applications in peer-to-peer grids. *Computer Networks*, 54(18):3341–3359, 2010. 24
- [SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia e Robert Chansler. The hadoop distributed file system. Em *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, páginas 1–10. IEEE, 2010. 1, 17
- [TAS17] Ehsan Totoni, Todd A Anderson e Tatiana Shpeisman. Hpat: high performance analytics with scripting ease-of-use. Em *Proceedings of the International Conference on Supercomputing*, página 9. ACM, 2017. xi, 2, 11, 12, 13, 18
- [Tro12] Mark Troester. Big data meets big data analytics. *Cary, NC: SAS Institute Inc*, 2012. 1, 5
- [TTS⁺14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham et al. Storm@ twitter. Em *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, páginas 147–156. ACM, 2014. 17
- [VMD⁺13] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth et al. Apache hadoop yarn: Yet another resource negotiator. Em *Proceedings of the 4th annual Symposium on Cloud Computing*, página 5. ACM, 2013. xi, 8, 9, 17
- [Whi12] Tom White. *Hadoop: The definitive guide*. "O'Reilly Media, Inc.", 2012. xi, 1, 8, 9
- [XGFS13] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin e Ion Stoica. Graphx: A resilient distributed graph system on spark. Em *First International Workshop on Graph Data Management Experiences and Systems*, página 2. ACM, 2013. 18
- [ZCD⁺12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker e Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Em *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, páginas 2–2. USENIX Association, 2012. xi, 1, 9, 10, 11
- [ZCF⁺10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker e Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010. 1, 2, 9, 10, 17, 18
- [ZDL⁺13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker e Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. Em *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, páginas 423–438. ACM, 2013. 18