UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
ESPECIALIZAÇÃO EM BIG DATA & DATA SCIENCE

ALEXANDRE K. S. MIYAZAKI

Otimizando StarVZ para carga de grandes volumes de dados

Monografia de Conclusão de Curso apresentada como requisito parcial para a obtenção do grau de Especialista em Big Data & Data Science

Orientador: Prof. Dr. Lucas Mello Schnorr

Porto Alegre
2019

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Miyazaki, Alexandre K. S.

Otimizando StarVZ para carga
de grandes volumes de dados / Alexandre K. S. Miyazaki. – Porto Alegre: PPGC da UFRGS, 2019.

46 f.: il.

Monografia (especialização) – Universidade Federal do Rio Grande do Sul. Especialização em Big Data & Data Science, Porto Alegre, BR–RS, 2019. Orientador: Lucas Mello Schnorr.

1. Spark. 2. Hadoop. 3. StarVZ. 4. Big Data. I. Schnorr, Lucas Mello. II. Título.

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

AGRADECIMENTOS

Primeiramente, agradecer minha esposa Nicole dos Reis pela compreensão e apoio não só durante o desenvolvimento deste trabalho mas também durante todo o período da pós-graduação. Sem dúvida seu apoio foi fundamental para eu conseguir completar esta fase.

Ao meu orientador, Prof. Dr. Lucas Mello Schnorr, agradecer pelo excelente trabalho. Como um dos melhores pesquisadores com quem eu já tive a satisfação de trabalhar, colocações pertinentes, revisão rápida e apoio durante todo o desenvolvimento desta monografia foram muito importantes para sua conclusão.

Por último, agradecer aos meus amigos e demais pessoas que de alguma forma me apoiaram, principalmente durante o desenvolvimento desta dissertação.

RESUMO

O StarVZ é um arcabouço para visualização de rastros de aplicações orientadas a tarefas no contexto de *High-Performance Computing*. Ele é separado em duas fases, a primeira voltada para o pré-processamento dos dados e a segunda para a exibição. Essa primeira fase sofre de problemas de desempenho ao trabalhar com grandes volumes de dados. Este trabalho se propõe a utilizar o Hadoop e o Spark para otimizar o processamento mais custoso desta fase. Este consiste em um fluxo de Ciência de Dados utilizando a linguagem R, o que facilitou bastante sua portagem para manipulação de dados via Spark. Nos resultados observados utilizando uma carga de trabalho de 12 gigabytes e 3 nós computacionais, cada um com 15 executores, chegamos a reduzir o tempo de execução em 74,11% comparado com a execução sequencial.

Palavras-chave: Spark. Hadoop. StarVZ. Big Data.

Improving StarVZ to handle Big Data

ABSTRACT

StarVZ is a framework for trace visualization of High-Performance Computing task-based applications. It is divided in two phases. The first one aims the data pre-processing whereas the second one aims the visualization. The first phase suffers from performance issues when handling large application traces and many gigabytes. This work proposes the utilization of Hadoop and Spark to optimize the most expensive step of this phase. It consists of a Data Science workflow using the R language, which eases its adaptation to Spark. In the observed results, with a 12 gigabytes workload and 3 nodes, each one with 15 executors, we managed to reduce the execution time in 74,11% comparing with the sequential execution.

Keywords: Spark, Hadoop, StarVZ.

LISTA DE ABREVIATURAS E SIGLAS

BSP	Bulk-Synchronous Parallel
CSV	Comma-Separated Value
CTF	Common Trace Format
DAG	Directed Acyclic Graph
DFS	Distributed Filesystem
GPU	Graphics Processing Unit
GFLOP	Gigaflop
GFS	Google File System
HDFS	Hadoop Distributed File System
HPC	High-Performance Computing
MPI	Message Passing Interface
OTF	Open Trace Format
OTF2	Open Trace Format version 2
PVM	Parallel Virtual Machine
RDD	Resilient Distributed Dataset
YARN	Yet Another Resource Negotiator

LISTA DE FIGURAS

Figura 2.1 Organização do Hadoop em camadas. 17
Figura 2.2 Fluxo de execução do MapReduce. 18
Figura 2.3 Arquitetura de uma aplicação Spark. 20

Figura 3.1 Fluxo de processamento do StarVZ. 22
Figura 3.2 Fluxo de pré-processamento do StarVZ. 22
Figura 3.3 DAG de execução do StarVZ. 25

Figura 4.1 Arquitetura da aplicação StarVZ. 28
Figura 4.2 Arquitetura da aplicação StarVZ utilizando Spark. 29
Figura 4.3 Fluxo de execução da aplicação. 30

Figura 5.1 Arquitetura de aplicações durante os experimentos. 35
Figura 5.2 Tempo total de execução da aplicação. 36
Figura 5.3 Tempos de execução segmentados por etapas. 37

LISTA DE TABELAS

Tabela 3.1 Arcabouço StarVZ - Tempos de execução da primeira Fase. 26

Tabela 4.1 Equivalências de operações. 32
Tabela 4.2 Número de linhas dos dados da tabela State agrupados pela coluna Type 32

Tabela 5.1 Configurações de um nó `draco`. 34
Tabela 5.2 Detalhamento da carga de trabalho. 35
Tabela 5.3 Tempos médios de execução, em segundos. 37

SUMÁRIO

1 INTRODUÇÃO **11**
2 FUNDAMENTAÇÃO **13**
2.1 Análise de desempenho de aplicações paralelas **13**
2.1.1 Ferramentas de visualização clássicas 13
2.1.2 Ferramentas de visualização orientadas a tarefas 15
2.2 Ferramental para Big Data **17**
2.2.1 Hadoop 17
2.2.2 Spark 19
3 O ARCABOUÇO STARVZ **21**
3.1 Visão Geral **21**
3.2 Fases **22**
3.3 Trabalhos Relacionados **23**
3.4 Motivação e Abordagem **26**
4 CONTRIBUIÇÃO: STARVZ SOBRE SPARK **28**
4.1 Arquitetura Proposta **28**
4.2 Implementação **30**
4.3 Sumário **33**
5 AVALIAÇÃO **34**
5.1 Metodologia de Avaliação **34**
5.2 Experimentos e Resultados **35**
5.3 Casos de Falha **38**
6 CONCLUSÃO E TRABALHOS FUTUROS **40**
REFERÊNCIAS **42**
APÊNDICE A — DOCUMENTAÇÃO DO STARVZ SOBRE HDFS/SPARK **45**

1 INTRODUÇÃO

A quantidade de dados gerados em todo o mundo diariamente é surpreendente. De acordo com World (2018), há uma estimativa que em 2020 para cada pessoa no mundo sejam produzidos em média média 1,7 MB de dados por segundo. Estes frequentemente precisam ser processados para ter algum valor, o que implica em uma constante necessidade por sistemas computacionais mais poderosos.

Plataformas de HPC (*High-Performance Computing*) costumam ser o estado da arte em poder computacional. Tais ambientes evoluíram para utilizar uma variabilidade de recursos de hardware, como processadores multicore e GPUs (*Graphics processing units*, comumente referenciadas como *accelerators*). Com a evolução do hardware, a abordagem de desenvolvimento de aplicações de HPC tradicional, chamada de *Bulk-synchronous parallel* (BSP) tornou-se obsoleta. Esta espera que os recursos de computação sejam homogêneos (idênticos), conectados por links estáveis e de alta vazão e portanto, é incapaz de utilizar a heterogeneidade dos recursos disponíveis no ambiente ao seu favor.

A abordagem que está sendo utilizada para o desenvolvimento de aplicações nesse cenário é orientada a tarefas. Nela, a aplicação é desenvolvida em alto nível, descrevendo as computações como um *Directed Acyclic Graph* (DAG). Ela é implementada em diverssos modelos de programação como OpenMP 4 (Ayguade et al., 2009), StarPU (AUGONNET et al., 2011), OmpSs (DURAN et al., 2011), ParSEC (Bosilca et al., 2013), etc. Neles a responsabilidade de escalonar e executar a aplicação de forma eficiente é atribuída para outra camada de software, denominada *runtime*. Ferramentas de análise de rastros, que antes auxiliavam o desenvolvedor da aplicação a fazer otimizações, como Paraver (PILLET et al., 1995) e Vampir (KNÜPFER et al., 2008) são pouco eficazes ao analisar aplicações baseadas em tarefas.

Com essa motivação, foi desenvolvido um arcabouço denominado StarVZ (GARCIA PINTO et al., 2018), cujo objetivo é fornecer uma visualização de rastros mais elaborada, provendo facilidade no entendimento e identificação de problemas de desempenho sutis, que dificilmente seriam identificados com abordagens clássicas. Esse arcabouço foi desenvolvido combinando *pj_dump*, a linguagem R (R Core Team, 2017), algumas bibliotecas expressivas dessa linguagem (`ggplot2`

(WICKHAM, 2009), `lpSolve` (BERKELAAR, 2016), `tidyverse` (TIDYVERSE, 2019)), Org-mode (DOMINIK, 2010) e `plotly` para análise de rastros gerados pelo modelo de programação StarPU.

Em um estudo de caso onde o StarVZ foi utilizado, ele conseguiu contribuir para a identificação de problemas de desempenho no StarPU. A primeira fase do arcabouço (pré-processamento de dados) levou cerca de 32 minutos para processar 18 GB. Tendo em vista que a tendência é que aplicações gerem cada vez mais dados, o objetivo deste trabalho é otimizar essa etapa do arcabouço na tentativa de viabilizar sua utilização com volumes maiores de dados, com tempos e recursos aceitáveis.

Mais especificamente, este trabalho será focado na etapa de manipulações realizadas sobre tabelas utilizando R e a biblioteca `dplyr`, que são operações comumente realizadas em fluxos de Ciência de Dados. Esta é a etapa mais custosa da fase de pré-processamento, no estudo de caso citado, ela foi responsável por 13 dos 32 minutos (40,62% do tempo total). Utilizaremos ferramentas de Big Data para atingir nossos objetivos, como o Hadoop (WHITE, 2015) e o Spark (CHAMBERS; ZAHARIA, 2018), além das facilidades oferecidas pela biblioteca `sparklyr`.

Observamos bons resultados nos experimentos comparativos da aplicação modificada em relação a original. No melhor caso, tivemos uma redução de 74,11% no tempo de execução ao processar uma entrada de 12 GB.

Este documento consiste em um trabalho de conclusão de curso da Especialização em Big Data & Data Science. Os próximos Capítulos são organizados da seguinte forma: o Capítulo 2 apresenta a fundamentação necessária para o entendimento do trabalho; o Capítulo 3 descreve com detalhes o arcabouço StarVZ; o Capítulo 4 disserta sobre as modificações propostas e realizadas; o Capítulo 5 apresenta os resultados obtidos; e o Capítulo 6 finaliza este trabalho e cita algumas possibilidades de trabalhos futuros.

O código fonte referente a este trabalho está no Github, e pode ser acessado neste link. Os arquivos LaTeXda monografia bem como alguns arquivos de apoio ao trabalho podem ser acessados neste link.

2 FUNDAMENTAÇÃO

Este Capítulo apresenta a fundamentação necessária para o entendimento do trabalho. A primeira Seção busca apresentar o contexto no qual o StarVZ está inserido, enumerando diversas ferramentas de análise de desempenho de aplicações paralelas. Na Segunda Seção é apresentado o ferramental que será utilizado na otimização do StarVZ.

2.1 Análise de desempenho de aplicações paralelas

Para melhor organização do trabalho, esta Seção foi dividida em duas partes. A primeira apresenta ferramentas que oferecem visualização de rastros de aplicações desenvolvidas no modelo BSP. A segunda contextualiza ferramentas voltadas para visualização de rastros de aplicações desenvolvidas no modelo orientado a tarefas.

2.1.1 Ferramentas de visualização clássicas

Essas ferramentas possuem o objetivo de prover visualizações de rastros para aplicações de HPC tradicionais. Estas eram desenvolvidas seguindo o modelo BSP, que consiste em uma série de super passos (computações, comunicações, sincronizações), executadas sobre um ambiente homogêneo. Como esta abordagem foi dominante durante muito tempo, suas necessidades balizaram o desenvolvimento da maior parte das ferramentas de análise de desempenho.

ViTE

ViTE (COULOMB et al., 2009) é uma ferramenta de visualização de rastros de código aberto. Suas entradas são arquivos na linguagem Pajé (STEIN et al., 2015) e para o processamento de grandes volumes ele conta com aceleração de hardware e OpenGL.

Essa ferramenta exibe os recursos de forma hierárquica, onde oferece a visualização das tarefas (eixo vertical) em função de tempo (eixo horizontal), simi-

lar a um Gantt. Na análise de aplicações distribuídas, também é possível incluir indicadores de transferências de dados.

Paraver

Paraver (PILLET et al., 1995) também objetiva a visualização e análise de rastros de execução. Suas entradas são geradas por vários modelos de programação, pela ferramenta Extrae. Para lidar com grandes volumes de dados, são realizadas agregações com uma abordagem definida pelo usuário via arquivo de configuração.

Vampir

Vampir (KNÜPFER et al., 2008) é uma ferramenta proprietária de código fechado para fins de análise de rastros. Ela traz uma abordagem de cliente-servidor, onde o primeiro pode ser executado no hardware de experimentação e o segundo é usualmente o computador do usuário.

Suas entradas são arquivos OTF2 (*Open Trace Format, version 2*) (ESCHWEILER et al., 2012). Ele fornece múltiplas visualizações como gráficos de espaço-tempo e estatísticas de execução.

Ravel

O objetivo da ferramenta Ravel (ISAACS, 2014) também é a visualização de rastros. Suas entradas são em formato OTF (*Open Trace Format*) e sua diferença em relação aos demais é que ele estrutura melhor as operações com a utilização de linhas de tempo lógicas.

FrameSoc e Ocelotl

FrameSoc (Pagano et al., 2013) é uma ferramenta de análise de desempenho, capaz de lidar com grandes volumes de dados. Como entrada, ela suporta diversos formatos como Pajé, CTF (*Common Trace Format*), Paraver e OTF2. Além dos rastros, é possível armazenar informações como metadados e anotações. A ferramenta converte tudo para um modelo de dados genérico e armazena em uma base de dados relacional.

Para visualizar grandes volumes de dados, a ferramenta baseia-se no Ocelotl (Dosimont et al., 2014). Este módulo gerencia uma agregação espaçotemporal dados via um arquivo de configuração parametrizável pelo usuário.

2.1.2 Ferramentas de visualização orientadas a tarefas

O modelo de computação orientado a tarefas segue o conceito de um conjunto de porções de trabalho independentes, de tamanhos variados, que podem ser executados em paralelo. Este insere uma camada intermediária entre o hardware e o software desenvolvido que facilita a implementação, pois provê uma certa independência de arquitetura. Além disso, quem analisa o desempenho de aplicações neste contexto não é mais o seu desenvolvedor, mas sim aqueles responsáveis pelo ambiente de execução (*runtime*).

As ferramentas de visualização para esse modelo de computação se inspiram em características daquelas desenvolvidas para o modelo BSP, pois este dominou o cenário de HPC por muito tempo. Todavia, como diversas premissas são quebradas nesse novo paradigma, as funcionalidades outrora utilizadas com eficiência para a análise de desempenho não são mais suficientes.

DAGViz

DAGViz (HUYNH et al., 2015) é composto por dois passos: extração do DAG dos arquivos de uma execução paralela; e visualização hierárquica do DAG. Essa ferramenta traz uma visualização diferente do modelo BSP, exibindo as tarefas como um grafo hierárquico. Nele, o analista pode colapsar e expandir os grupos de tarefas. Dados de tempo de execução não são tratados pela ferramenta.

Rastros de execução com dependências de tarefas

A ferramenta desenvolvida por Haugen et al. (2015) traz um gráfico no estilo espaço-tempo (Gantt). Há algumas outras funcionalidades como a identificação de dependências de tarefas (apenas o primeiro nível) a medida que o usuário passa o mouse sobre as caixas que representam as tarefas.

Como entradas, são utilizados a representação do DAG e os rastros de execução. Essa ferramenta é desenvolvida especificamente para o ambiente de execução PaRSEC.

Temanejo

Temanejo (KELLER et al., 2012) é um depurador para o modelo de programação baseado em tarefas, onde o analista visualiza um DAG. Ele suporta grande parte dos ambientes de execução de aplicações baseadas em tarefas, como OmpSs, StarPU e PaRSEC. Suas funcionalidades são focadas em depuração, permitindo que o usuário possa identificar e consertar parâmetros e dependências de tarefas.

Delay Spotter

Delay Spotter (Huynh; Taura, 2017) é uma ferramenta, construída sobre o DAGViz, que possibilita a identificação de atrasos em ambientes de execução. Ela divide os estados dos trabalhadores em três categorias, permitindo a identificação de atrasos decorrentes de problemas de escalonamento. Como em ambientes heterogêneos com tarefas variadas, a presença de atrasos faz parte da execução das aplicações, essa ferramenta é pouco efetiva.

TaskInsight

TaskInsight (CEBALLOS et al., 2018) é uma ferramenta que objetiva identificar o comportamento de memória e seu impacto na execução de tarefas. Apesar de prover algumas estatísticas e possibilitar algumas identificações de anomalias, apenas essa análise não é o suficiente para identificar a maioria dos pontos de otimização de aplicações baseadas em tarefas.

StarVZ

Arcabouço que é objeto deste trabalho, o StarVZ (GARCIA PINTO et al., 2018) possui a visualização de dados mais avançada dentre as ferramentas citadas. Construído com uma abordagem de script, ele possui um grande poder de customização. Ele será detalhado no Capítulo 3.

2.2 Ferramental para Big Data

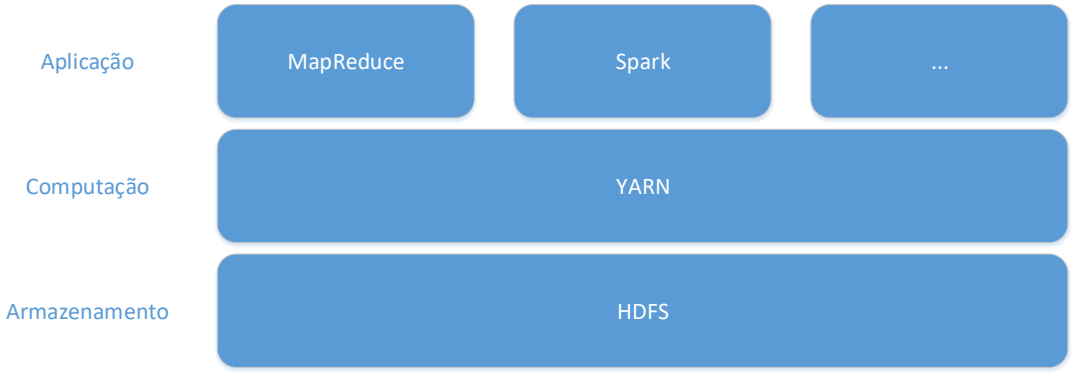
Esta Seção contextualiza as ferramentas para tratamento de grandes volumes de dados que serão utilizadas neste trabalho. Na Seção 2.2.1 contextualizamos a plataforma Hadoop e na Seção 2.2.2 falamos sobre Spark.

2.2.1 Hadoop

O Hadoop é um arcabouço que permite o processamento distribuído de grandes volumes de dados. A principal motivação que implicou em seu desenvolvimento foi o fato de que as velocidades de leitura e escrita dos discos rígidos não evoluíram da mesma forma que o seu tamanho. Ele resolve este problema lendo grandes volumes de dados de muitos discos, paralelizando a leitura.

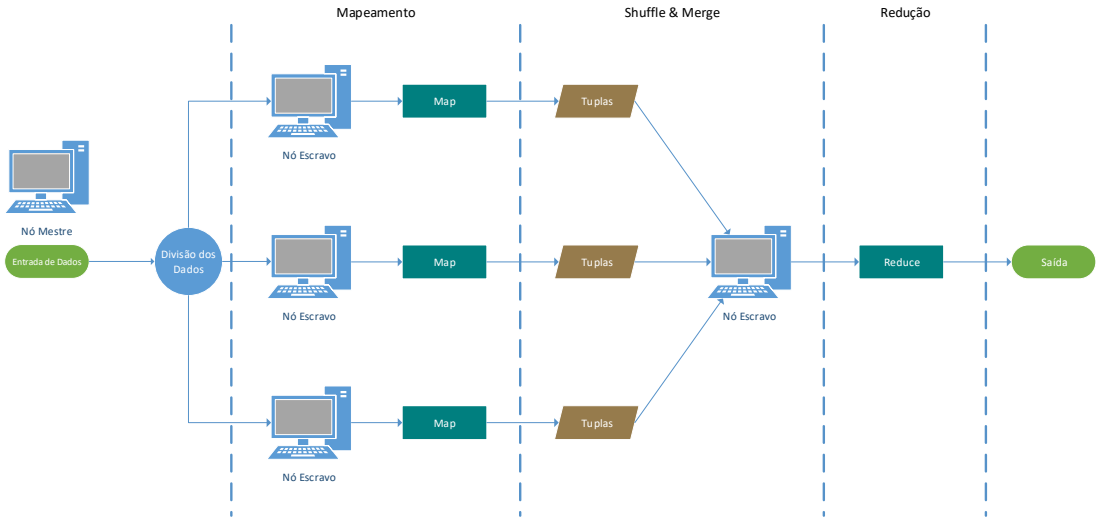
O Hadoop pode ser separado em camadas, como mostra a Figura 2.1. A primeira camada, de aplicação, é a de mais alto nível, a qual o usuário usualmente interage. Ele suporta o modelo de programação MapReduce, originalmente desenvolvido pelo Google com o objetivo de processar grandes volumes de dados. Tal modelo baseia-se em duas primitivas presentes em linguagens funcionais: *map* e *reduce*. Esta abordagem foi adotada pois constantemente era necessário mapear pedaços de dados de uma entrada à uma chave identificadora e, em seguida, realizar algum tipo de computação sobre os dados mapeados uma mesma chave (DEAN; GHEMAWAT, 2008).

Figura 2.1: Organização do Hadoop em camadas.



A Figura 2.2 mostra o fluxo de execução básico do MapReduce. A Fase de *Shuffle & Merge* não será detalhada no trabalho, mas é importante salientar que existe um processamento entre as fases de mapeamento e redução.

Figura 2.2: Fluxo de execução do MapReduce.



Primeiramente temos os dados sendo inseridos no Sistema de Arquivos Distribuído (*Distributed File System* ou DFS) da plataforma. Os arquivos são divididos em pedaços de tamanho fixo e distribuídos sobre a plataforma. Isso permite que os dados sejam tratados de forma rápida pois há paralelização em sua leitura.

Com os dados armazenados, pode se iniciar a fase de mapeamento, que resulta em conjuntos de tuplas intermediárias. Estas sofrem um pré-processamento na fase de *Shuffle* e são disponibilizados para os nós que irão executar a fase de redução. Finalmente, os executores da redução obtêm os dados e desempenham suas tarefas, gerando a saída da execução.

Voltando a Figura 2.1, na segunda camada encontra-se o gerenciador de recursos do arcabouço. O YARN (*Yet Another Resource Negotiator*) foi introduzido no Hadoop 2.0 e além de melhorar a implementação do modelo MapReduce, suporta outros paradigmas de computação. Ele fornece APIs para requisitar e manipular recursos, não sendo comum o usuário interagir diretamente com essa camada.

Como o Hadoop trabalha sobre dados muito grandes, é comum que tais cargas de trabalho ultrapassem a capacidade de uma única máquina física. Isso implica na necessidade de particionar os dados sobre múltiplas máquinas. Os sistemas de arquivos que lidam com tal tipo de armazenamento são denominados

sistemas de arquivos distribuídos. Hadoop possui sua própria implementação de um DFS, inspirado no *Google File System* (Comumente referenciado como *GFS*), denominado *Hadoop Distributed Filesystem* (*HDFS*), que é representado como a terceira camada da Figura 2.1.

De forma similar a sistemas de arquivos convencionais, é utilizada a abstração de blocos, que consistem no volume mínimo para leitura e escrita. Enquanto nestes, eles têm tipicamente poucos bytes, o tamanho de blocos no HDFS é na ordem de megabytes, sendo por padrão, 128 (configurável pelo usuário, por arquivo). Tal abstração permite ao Sistema: armazenar arquivos maiores do que qualquer disco no ambiente; simplifica o gerenciamento do armazenamento de dados; e facilita replicação para garantir alta disponibilidade.

Um HDFS possui dois tipos de nós: *namenode* e *datanodes*. Eles seguem uma política mestre-escravo, onde o nó do primeiro tipo atua como mestre e gerencia o sistema de arquivos e nós do segundo tipo recebem requisições (de clientes ou do *namenode*) e armazenam ou recuperam blocos. Também é possível executar um *secondary namenode* para evitar pontos únicos de falha na arquitetura. O principal papel deste nó é manter uma cópia dos dados de gerência do *namenode* principal, em caso de falhas, ele pode assumir o lugar deste.

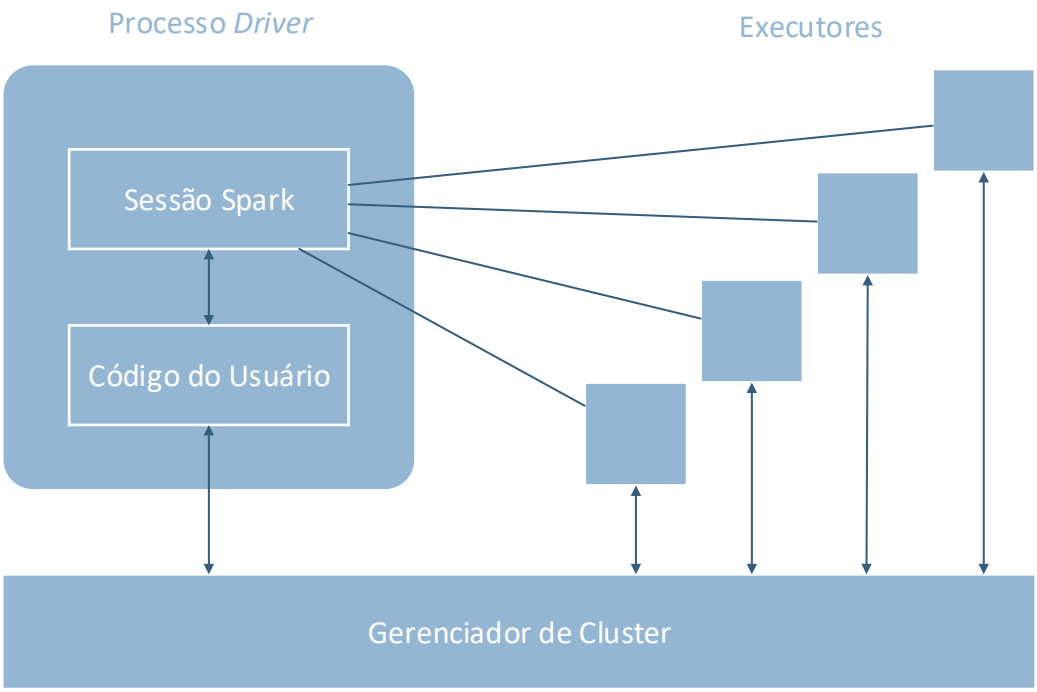
2.2.2 Spark

Spark é um conjunto de uma *engine* unificada e um conjunto de bibliotecas para processamento de dados distribuídos (CHAMBERS; ZAHARIA, 2018). A motivação para sua criação foi a necessidade de muitos tipos de processamento e a tendência do ferramental ser cada vez mais específico no cenário de *Big Data*.

A arquitetura do Spark pode ser visualizada na Figura 2.3. A *engine* usualmente é executada sobre um cluster de máquinas gerenciado por um `Gerenciador de Cluster`, que mantém o controle dos recursos disponíveis. Ele oferece suporte a três gerenciadores: um do próprio Spark, Hadoop YARN ou Mesos (HINDMAN et al., 2011).

O `Processo Driver` é responsável por manter o ciclo de vida da aplicação. Dentre suas responsabilidades estão: responder às entradas do programa do usuário; analisar, distribuir e agendar trabalho sobre os `Executores`; e manter informação relevante sobre a execução. Os `Executores` realizam duas funções:

Figura 2.3: Arquitetura de uma aplicação Spark.



executam tarefas que lhes são atribuídas e reportam o estado das computações para o `Processo Driver`.

O Spark é atrelado a um modelo de programação similar ao MapReduce, todavia, possui uma abstração para compartilhamento de dados, chamada *Resilient Distributed Datasets* (RDDs). Estas, consistem em coleções de objetos tolerantes a falhas e que podem ser manipulados em paralelo e são particionados sobre a infraestrutura que executa a *engine*.

Spark manipula RDDs através de uma API unificada disponibilizada para diversas linguagens, que facilita o desenvolvimento de aplicações. Nela, o usuário é capaz de passar funções locais para serem executadas de forma distribuída. A avaliação dos RDDs é realizada de forma *Lazy*, o que significa que as manipulações são efetivadas apenas quando é executada uma ação nos dados, que se trata de uma instrução para computar o resultado de uma série de transformações. Isso permite que ele encontre um plano de execução eficiente pois pode agregar operações, otimizando o tempo de execução.

3 O ARCABOUÇO STARVZ

Este Capítulo descreve em detalhes o arcabouço StarVZ. Este é apresentado genericamente na Seção 3.1, a Seção 3.2 apresenta de forma detalhada as fases de sua execução, a Seção 3.3 disserta sobre otimizações já propostas e testadas e a Seção 3.4 fala sobre a motivação e a abordagem adotadas neste trabalho.

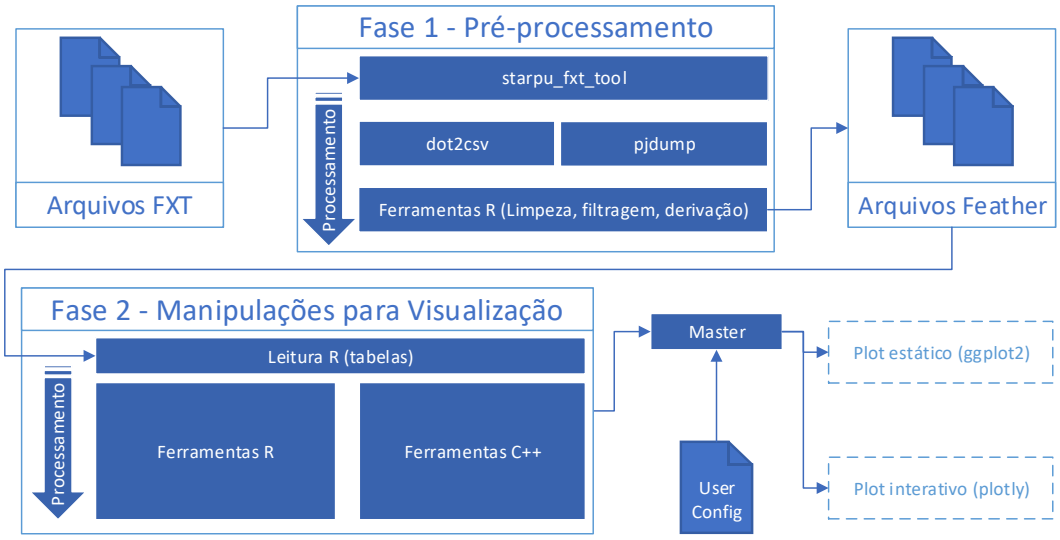
3.1 Visão Geral

O StarVZ (GARCIA PINTO et al., 2018) é um fluxo de processamento de análise de desempenho cujo objetivo é auxiliar na avaliação e na verificação de hipóteses sobre a execução de aplicações baseadas em tarefas em ambientes heterogêneos, executados sobre o ambiente de execução StarPU (AUGONNET et al., 2011). Dentre as ferramentas de visualização de rastros desse tipo de aplicação, na pesquisa realizada foi identificado que o StarVZ se destaca pela utilização de ferramentas de Ciência de Dados para análise de desempenho.

Construído com uma abordagem de script, ele possui grande poder de customização. No trabalho de Garcia Pinto et al. (2018), pode-se visualizar diversos gráficos de execução da aplicação: gráfico com comportamento de tarefas; gráfico com a quantidade de tarefas submetidas; o comportamento do ambiente de execução, com os estados dos trabalhadores StarPU; quantidade de tarefas prontas; taxa de GFlops (Gigaflops) do ambiente; tráfego de dados entre a memória das GPUs; transferências de rede MPI; e o número de operações MPI concorrentes.

Ele é composto de duas fases, que podem ser visualizadas de forma simplificada na Figura 3.1.Cada uma delas é formada por uma combinação de diversas ferramentas, resultando em um arcabouço rápido, consistente, flexível e versátil.

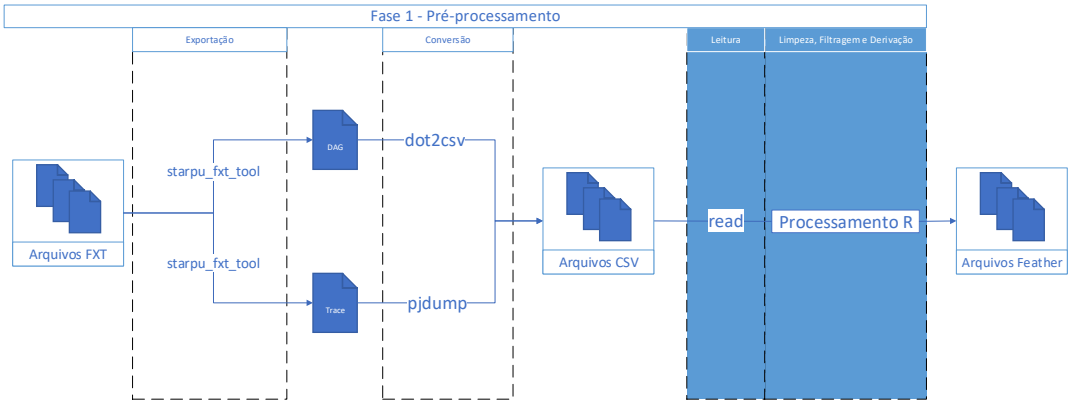
Figura 3.1: Fluxo de processamento do StarVZ.



3.2 Fases

Na primeira fase, que pode ser visualizada na Figura 3.2, os rastros de execução do StarPU, que são arquivos no formato binário FXT, são transformados e exportados através da ferramenta `starpu_fxt_tool` para dois arquivos: DAG no formato DOT e Trace no formato PAJÉ. Essa etapa gera eventos datados, que descrevem o comportamento da aplicação para todos os recursos envolvidos. Também são gerados dados sobre o ambiente de execução do StarPU, como número de tarefas submetidas, número de tarefas prontas, arquitetura da plataforma, etc.

Figura 3.2: Fluxo de pré-processamento do StarVZ.



Em seguida, é realizada uma verificação de integridade estrutural e temporal dos arquivos `trace`. Ela é executada via `pj_dump`, gerando cinco arquivos: o arquivo *states* possui informações sobre as tarefas executadas e seu comportamento no ambiente de execução; *variables* consiste em métricas de desempenho da plataforma e ambiente de execução; *events* possui informações sobre a utilização de recursos; *links* contém dados sobre comunicação MPI; e as informações de plataforma são registradas no arquivo *entities*. O DAG também passa por uma conversão e por fim, todos os arquivos são gravados no formato de Valores Separados por Vírgula (*Comma-Separated Value*, comumente abreviado para CSV).

Finalmente, os dados escritos em CSV são lidos, filtrados, agregados e combinados em uma ferramenta implementada na linguagem R, utilizando-se bibliotecas oriundas do pacote `tidyverse` para a manipulação de dados. As saídas são escritas no formato *Feather* (WICKHAM, 2019).

A segunda fase do fluxo de processamento inicia pela leitura das saídas da anterior. Cada arquivo torna-se uma tabela, e os dados são unificados em uma lista. É possível ter múltiplos rastros de aplicações sendo analisados em paralelo para comparação, basta multiplicar a leitura com entradas diferentes e elas podem posteriormente ser combinadas em uma única visualização. A criação dos gráficos ainda possui processamento de dados, o que traz certa flexibilidade para as visualizações. Como última etapa dessa fase, o usuário parametriza o sistema com um arquivo de configuração no formato YAML, para que a montagem da visualização seja configurável. Finalmente, o usuário pode analisar os gráficos gerados pela ferramenta.

Nos experimentos realizados em Garcia Pinto et al. (2018), em uma máquina equipada com um Intel(R) Xeon(R) CPU E3-1225 v3 @3.20GHz e 32GB de memória principal e com uma entrada de aproximadamente 18GB, a primeira fase do fluxo de processamento levou cerca de 32 minutos para executar. A segunda fase, para executar a leitura dos arquivos *Feather* e gerar a visualização levou em torno de 2 minutos.

3.3 Trabalhos Relacionados

O trabalho de Alles and Schnorr (2018) foi o primeiro a tentar otimizar o fluxo do StarVZ. Sua principal motivação foi a melhoria de desempenho da

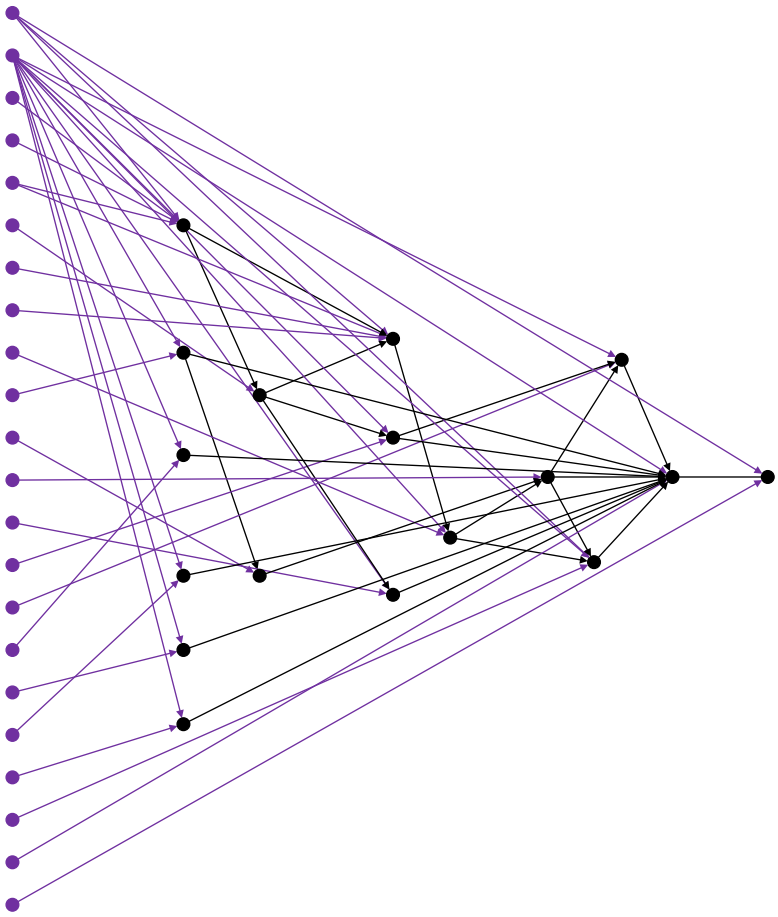
etapa de manipulação de dados, a mais custosa de acordo com os experimentos observados, com o objetivo de permitir o processamento de entradas maiores em um tempo aceitável. Nele utilizou-se `Drake` (LANDAU et al., 2019), uma biblioteca para a linguagem de programação R, cujo foco é executar apenas as partes necessárias de um fluxo de processamento de análise de dados, evitando os passos desnecessários que não mudarão suas saídas. Isso é realizado modelando as computações como um Grafo Acíclico Direcionado (*directed acyclic graph* ou DAG) de tarefas e armazenando em cache os resultados daquelas já executadas. Além disso, `Drake` também possui suporte a paralelismo (*Implicit parallelism*) para a execução de tarefas independentes.

Para modelar o StarVZ dessa forma, foram necessárias mudanças cujo objetivo era explicitar dependências e postergar junções de tabelas. Depois da identificação dos fluxos independentes no fluxo de processamento, foi utilizado `Drake` para criar um plano de execução, que consiste na declaração das tarefas onde cada uma é representada por uma função R. Na criação do plano, a biblioteca analisa cada uma das tarefas, suas entradas e saídas, para determinar dependências e gerar o DAG. O resultado da modelagem do StarVZ nessa abordagem pode ser visualizado na Figura 3.3.

Com esta modelagem, não foram observadas melhorias de desempenho no StarVZ. De acordo com Alles and Schnorr (2018), a cache de resultados intermediários da biblioteca acaba tendo que armazenar muitos dados em disco devido ao tamanho das tabelas geradas pelas entradas, prejudicando o tempo total de processamento. Em relação ao suporte a paralelismo, ele é limitado pela falta de suporte nativo a *multithread* da linguagem R. Por isso, `Drake` oferece essa funcionalidade instanciando múltiplas sessões R, que são processos separados. A comunicação entre esses processos é realizada pelo mesmo sistema de cache citado anteriormente, consequentemente, resultando nos mesmos problemas.

Outra abordagem que poderia ser adotada para melhorar o desempenho do StarVZ seria distribuir os fluxos independentes do fluxo de processamento pois elas podem trazer um ganho de desempenho em um ambiente viável. Ao simplesmente paralelizar, é possível que os mesmo problemas com o tamanho das tabelas sejam enfrentados, exigindo máquinas com muita memória para obter-se os resultados desejados.

Figura 3.3: DAG de execução do StarVZ.



Fonte: Inspirada na Figura de contida no trabalho de Alles and Schnorr (2018)

A primeira opção seria distribuir os fluxos independentes utilizando o pacote `Snow`, que significa *Simple Network of Workstations* (TIERNEY, 2018). Este é desenvolvido para linguagem R e permite a utilização de *Explicit parallelism*, oferecendo integração com três interfaces de baixo nível: PVM (*Parallel Virtual Machine*), através do pacote `rpvm`; MPI, através do pacote `rmpi` (YU, 2013); e uma integração com sockets, no caso de não ter nenhuma das outras opções disponíveis no ambiente. A utilização de `Snow` pode ser feita através da biblioteca `Snowfall` (KNAUS, 2015), que é uma camada cujo objetivo é melhorar a usabilidade, facilitando ainda mais a utilização.

A segunda alternativa seria a utilização da biblioteca `future` (BENGTSSON, 2019). Esta provê uma forma simples e uniforme de avaliar expressões de forma assíncrona, utilizando recursos diversos. Ela funciona utilizando o conceito de abstração *future*, que basicamente define um valor que poderá estar disponível no futuro. Tal valor pode ter seu estado como resolvido ou não resol-

vido, estado em que se ele for utilizado bloqueará o processo até sua resolução. Os *futures* podem ser resolvidos de diversas formas: sequential, onde são resolvidos sequencialmente no processo R corrente; multiprocess, que resolve utilizando *multicore*; cluster, que é o mais interessante para adoção pois resolve com sessões R na máquina local e/ou em máquinas remotas; e etc.

3.4 Motivação e Abordagem

Há algum tempo, vivemos na era dos dados. O volume armazenado em meios eletrônicos é difícil de medir, mas existem alguns estudos que podem nos auxiliar a ter uma noção disso. A IDC (*International Data Corporation*) (ZWOLENSKI; WEATHERILL, 2014) estima que, de acordo com o histórico de crescimento do mundo digital, este deve crescer entre 2013 e 2020 de 4.4 para 44 zettabytes [1].

Conforme explicitado na Seção 3.2, o tempo total de execução da primeira Fase do StarVZ, para uma carga de trabalho de 18 gigabytes levou cerca de 32 minutos. Para viabilizar o processamento de maiores volumes de dados em um tempo aceitável, trazendo as vantagens da utilização da ferramenta para esses cenários, é necessário otimizar esta Fase do fluxo de processamento.

Decompondo esse tempo entre todas as ferramentas utilizadas, a Tabela 3.1 exibe o tempo de cada ferramenta no fluxo de processamento para a execução mencionada no trabalho de Garcia Pinto et al. (2018). Como o volume de dados utilizado foram apenas 18 gigabytes, essas aplicações não possuem um tempo de execução razoável para tratamento de grandes volumes de dados.

Tabela 3.1: Arcabouço StarVZ - Tempos de execução da primeira Fase.

Ferramenta	**Tempo de Execução**	**Tempo Relativo**
starpu_fxt_tool	10 minutos	31,25%
pj_dump	9 minutos	28,12%
Ferramenta R	13 minutos	40,62%
Total	32 minutos	-

Como na segunda fase não é evidente um ponto de otimização urgente, tendo em vista que nos experimentos foram gerados resultados em poucos mi-

[1] Um zettabyte significa um trilhão de gigabytes.

nutos, a motivação deste trabalho é a otimização do ponto mais crítico do tempo de execução da primeira fase do fluxo de processamento. Como as ferramentas desenvolvidas em R manipulam tabelas, fazendo operações comuns a Ciência de Dados, este trabalho migrará as tabelas normais para tabelas Spark, com o intuito de distribuir o trabalho em diversas máquinas. Além disso, para esta adaptação suportar arquivos muito grandes, essa instância do Spark irá executar sobre o Hadoop, garantindo que os dados terão uma manipulação otimizada.

4 CONTRIBUIÇÃO: STARVZ SOBRE SPARK

A fase de pré-processamento do arcabouço StarVZ é similar a um fluxo de Ciência de Dados. Nele, são carregadas e manipuladas várias tabelas, utilizando bibliotecas do pacote `tidyverse`. Este Capítulo fala sobre as diferenças propostas e realizadas no StarVZ, com o objetivo de otimizar o tempo de execução da fase de pré-processamento. Resumidamente, no lugar de tabelas R, utilizaremos tabelas Spark (*Spark Dataframes*) para o carregamento e manipulação de dados. Na Seção 4.1 são apresentadas as mudanças arquiteturais propostas, na Seção 4.2, detalha-se a implementação dessas mudanças e na Seção 4.3 é realizado um resumo do que foi apresentado.

4.1 Arquitetura Proposta

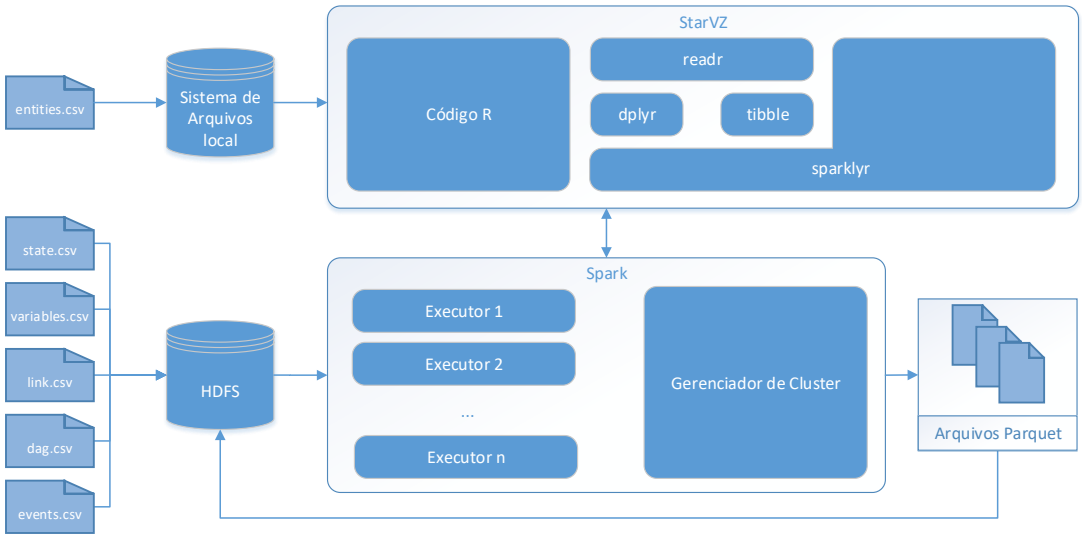
A arquitetura da aplicação que este trabalho se propõe a modificar pode ser visualizada na Figura 4.1. Os dados, já em formato CSV são lidos com a biblioteca `readr`. Com estes em memória, são realizadas filtragens e junções com o auxílio das bibliotecas `dplyr` e `tibble`. Finalmente, os dados são escritos em disco no formato *Feather*, usando a biblioteca de mesmo nome. Todo esse processo é conduzido através de um código R.

Figura 4.1: Arquitetura da aplicação StarVZ.

Durante o trabalho, identificou-se que não havia uma biblioteca que suportasse a escrita de tabelas Spark em arquivos *Feather*. Isso é essencial para compararmos a escrita do mesmo tipo de dados tanto na execução sequencial quanto na distribuída. Devido ao ecossistema Hadoop utilizar o formato *Parquet* (FOUNDATION, 2018) como padrão para armazenamento de dados colunares, o fluxo sequencial da aplicação foi adaptado para escrever suas saídas neste formato, utilizando a biblioteca `Apache Arrow`.

A Figura 4.2 ilustra a arquitetura da aplicação após as modificações propostas. Considerando o tamanho dos arquivos de entrada, utilizaremos o HDFS para armazená-los. Todavia, o arquivo entities não cresce muito com o aumento do tamanho das entradas (seu tamanho fica na ordem de KBytes). Para este, o custo de armazená-lo e tratá-lo no sistema de arquivos distribuído não é vantajoso pois os processamentos adicionais para realizar tais processos são mais caros que o tratamento do arquivo em si. Portanto, decidiu-se mantê-lo com seu armazenamento no sistema de arquivos local e processamento sequencial.

Figura 4.2: Arquitetura da aplicação StarVZ utilizando Spark.



A biblioteca escolhida para fazer a intermediação entre o código R e o Spark foi a `sparklyr`. De acordo com Chambers and Zaharia (2018), ela é baseada no pacote `dplyr` e sua abordagem prioriza a experiência com R, abstraindo alguns dos conceitos básicos como a seção Spark (*Spark Session*), diferindo da abordagem padrão das APIs. O processo driver, responsável por intermediar operações no código R e no Spark, é encapsulado nas tratativas dessa biblioteca.

O Spark utiliza diretamente o HDFS para leituras e escritas. Todas as tabelas de saída são escritas no HDFS. No caso da tabela entities, após seu processamento ela é convertida em uma tabela Spark. Com isso, padroniza-se o local de saída do arcabouço.

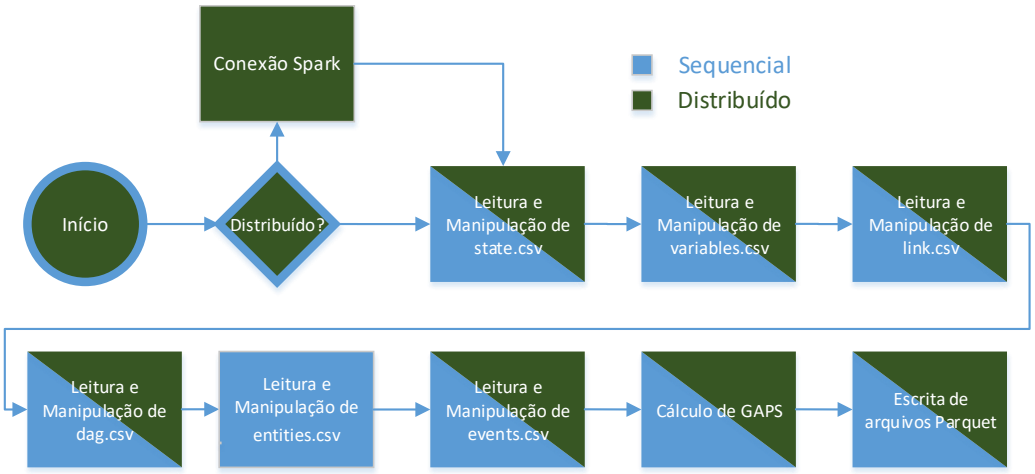
4.2 Implementação

Nesta Seção, começaremos dando uma visão geral das modificações realizadas no pacote StarVZ. Em seguida, aprofundaremos o fluxo de processamento da aplicação original (sequencial) e da modificada (distribuída), mostrando que ambas seguem praticamente o mesmo fluxo. Por fim as modificações realizadas são detalhadas e a forma que elas foram validadas é explicitada.

A execução do script da fase de pré-processamento agora conta com o parâmetro de tipo, onde `-S`, significa sequencial e `-D`, distribuído. Este foi criado dentro do próprio pacote StarVZ como um novo conjunto de funções, permitindo ao usuário a utilização de qualquer metodologia do arcabouço.

A Figura 4.3 representa os possíveis fluxos da aplicação. As formas que possuem o contorno `Sequencial` e o preenchimento `Distribuído` representam etapas que sempre são executadas. As formas que possuem preenchimento meio a meio, representam etapas que contam com uma implementação diferente para cada tipo de fluxo, executado conforme parametrizado. As formas que possuem apenas um preenchimento e nenhum contorno correspondem a etapas que são executadas sempre com a abordagem preenchida.

Figura 4.3: Fluxo de execução da aplicação.



Sobre o fluxo, a única modificação realizada foi a criação da `Conexão Spark` caso o usuário esteja utilizando o arcabouço de forma distribuída. Este trabalho teve como objetivo otimizar o processamento dentro de cada etapa, apesar de haver margem para melhorias a nível de fluxo. Destaca-se que, conforme

mencionado na Seção anterior, o processamento de entities foi mantido sempre no formato sequencial.

No fluxo original, a escrita de arquivos foi modificada para o formato *Parquet*, com o pacote `Apache Arrow`. Em termos de código, essa modificação é bastante pontual, adicionando uma importação de biblioteca e modificando os métodos de escrita de arquivos (de `write_feather` para `write_parquet`). É importante salientar que este pacote também conta com uma implementação de `write_feather`, que é mais eficiente que aquela do pacote `feather`.

Para utilizar a *engine* e executar a aplicação de forma distribuída, é necessário estabelecer uma conexão Spark, utilizando o método `spark_connect`. Com ela estabelecida, é possível ler arquivos em diversos formatos diretamente para tabelas no `sparklyr`. Esse processo é realizado na etapa de `Conexão Spark`, executada em caso de fluxo distribuído conforme representado na Figura 4.3. É importante ressaltar que com essa biblioteca, conseguimos ler arquivos armazenados no HDFS, o que permite que volumes de dados maiores que a quantidade de memória disponível na máquina sejam processados. Essa funcionalidade é muito importante pois desvincula o potencial da aplicação da limitação física da infraestrutura utilizada.

Com os arquivos carregados em tabelas Spark o restante do trabalho foi identificar equivalências entre as transformações realizadas com `dplyr` na `sparklyr`. Como esta foi desenvolvida baseada na primeira, esse processo foi facilitado. Ao encontrar uma manipulação, o seu resultado era observado na tabela original e reproduzido com funções suportadas pela `sparklyr`. Essa metodologia foi adotada pois, caso fosse necessário um tratamento que não fosse suportado pela biblioteca, seria preciso implementá-lo via funções definidas pelos usuários (*User defined function ou UDFs*). Essa funcionalidade sofre de problemas de desempenho ao utilizar R ou Python pois há o custo de instanciar um processo e compartilhar os dados de entradas e saídas.

A Tabela 4.1 exibe as principais equivalências utilizadas. As funções adaptadas com elas geraram as versões distribuídas de tratamento de todas as tabelas, com exceção de entities que ficou com seu processamento sequencial. Enquanto a maior parte das operações possui uma equivalência de um pra um, a função `separate` necessita de um conjunto de transformações para chegar no mesmo resultado.

Tabela 4.1: Equivalências de operações.

Operação `dplyr`	**Operação `sparklyr`**
distinct	unique
sort	sdf_sort
gsub	regexp_replace
rbind	union_all
grepl	rlike
separate	ft_regex_tokenizer + sdf_separate_column

O cálculo de GAPS é a última transformação de dados a ser realizada antes da escrita dos arquivos *Parquet*, sendo um conjunto de junções realizadas de forma recursiva. Durante o desenvolvimento, foi observada que a versão distribuída levava um tempo maior que a execução sequencial. Isso foi atribuído ao custo de comunicação de executores e aos processamentos que o Spark executa dentro desta etapa (lembrando sua execução *Lazy*) e foi um ponto de atenção levantado para ser observado durante os experimentos.

Logo após o cálculo de GAPS, os dados estão prontos para serem escritos. Neste ponto foi realizada a validação dos resultados, executando ambos os fluxos da aplicação com um conjunto de dados de 835 MB. Para cada um, validamos tabela por tabela, realizando diversas operações de sumarização e comparando seus resultados. Por exemplo, na tabela State, realizamos a comparação do número de linhas das tabelas (ambas tinham 5207577 linhas), além da comparação de 7 agrupamentos diferentes. A tabela 4.2 reproduz os dados observados em ambos os fluxos ao realizar-se o agrupamento dos dados pela coluna Type.

Tabela 4.2: Número de linhas dos dados da tabela State agrupados pela coluna Type

Agrupamento	**Número de Linhas**
Communication Thread State	199180
Memory Node State	326886
Worker State	952006
User Thread State	1623915
hread State	2105590

Por fim, as saídas são escritas no HDFS. Elas geram múltiplos arquivos, por exemplo se a saída se chamar *teste.parquet*, a aplicação criará um diretório

com esse nome e dentro dela haverá diversos arquivos, conforme abaixo. Isso ocorre pois por padrão as tabelas Spark são particionadas e escrevê-las gera um arquivo por partição.

```
part-00000-1c143c87-5361-427c-825e-792497a8fa61-c000.snappy.parquet
part-00001-1c143c87-5361-427c-825e-792497a8fa61-c000.snappy.parquet
...
```

4.3 Sumário

Neste Capítulo foram apresentadas as modificações realizadas no arcabouço StarVZ, desde a parte arquitetural até detalhes de implementação. No próximo Capítulo, é realizada a avaliação de tais modificações em testes comparativos da aplicação sequencial com a distribuída.

5 AVALIAÇÃO

Este Capítulo apresenta resultados comparativos da solução desenvolvida com a aplicação original. A Seção 5.1 apresenta detalhes da metodologia de avaliação, a Seção 5.2 disserta sobre os resultados e a Seção 5.3 fala sobre casos de falha encontrados durante os testes.

5.1 Metodologia de Avaliação

Os experimentos foram realizados no Parque Computacional de Alto Desempenho (PCAD) da UFRGS. Eles ocorreram nos nós de computação `draco`, cuja configuração de um nó é mostrada na Tabela 5.1. Todos os nós do cluster possuem a mesma configuração, com exceção de um que não foi utilizado.

Tabela 5.1: Configurações de um nó `draco`.

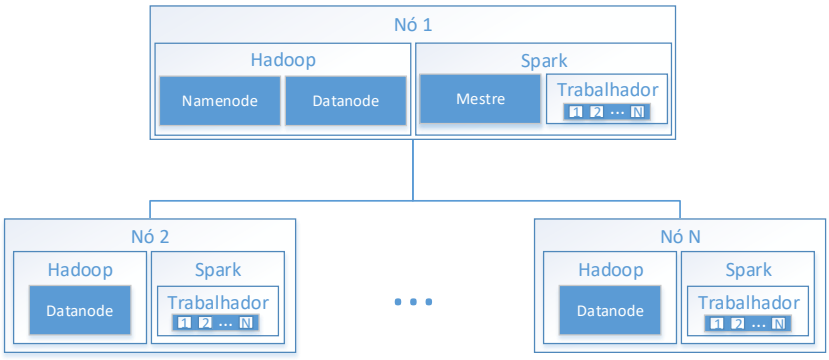
Parâmetro	**Configuração**
Processador	2 x Intel Xeon E5-2630 (Q1'12) Sandy Bridge, 2,5 GHz
Número de Núcleos Físicos	16
Número de Núcleos Lógicos	32
Memória	64 GB DDR3 RAM
Rede	Ethernet 10 Gigabit

A Figura 5.1 exibe a arquitetura que foi seguida durante os experimentos. Como o Spark acessa dados direto do HDFS, primeiramente executamos o Hadoop. Um dos nós era responsável por executar o *namenode* e também executava uma instância de *datanode*. Os demais executavam apenas *datanodes*.

O gerenciador de cluster utilizado foi o do próprio Spark ao invés do YARN. Essa decisão foi tomada durante os testes pois seus arquivos de registro mostraram informações mais claras sobre o que estava ocorrendo. A abordagem utilizada para ele foi similar àquela utilizada no Hadoop, um nó executou como mestre e trabalhador enquanto os demais executaram apenas como trabalhadores.

Cada trabalhador do Spark instancia N executores, responsáveis por processar tarefas. Cada um destes possui uma quantidade de núcleos e uma quantidade de memória dedicada. Durante os experimentos, parametrizamos a *engine*

Figura 5.1: Arquitetura de aplicações durante os experimentos.



para que cada trabalhador utilizasse 15 executores, cada um com 2 núcleos e 4 GB de memória para processamento de tarefas.

Realizamos testes com 1 nó, utilizando a aplicação original, 15 executores em 1 nó, 30 executores em 2 e 45 executores em 3 nós, utilizando a aplicação modificada. Cada teste foi configurado para executar 30 vezes para garantir a confiabilidade de seus resultados, todavia, houveram casos isolados de problemas de execução que não afetaram as tendências gerais observadas.

5.2 Experimentos e Resultados

A carga de trabalho dos experimentos já no formato CSV, entrada para a fase de pré-processamento no StarVZ, tinha um somatório de 12 GB. Ela consiste em rastros de execução de uma aplicação cholesky e foi escolhida pois era a maior entrada que tínhamos no momento. O tamanho de cada um dos arquivos pode ser visualizado na Tabela 5.2.

Tabela 5.2: Detalhamento da carga de trabalho.

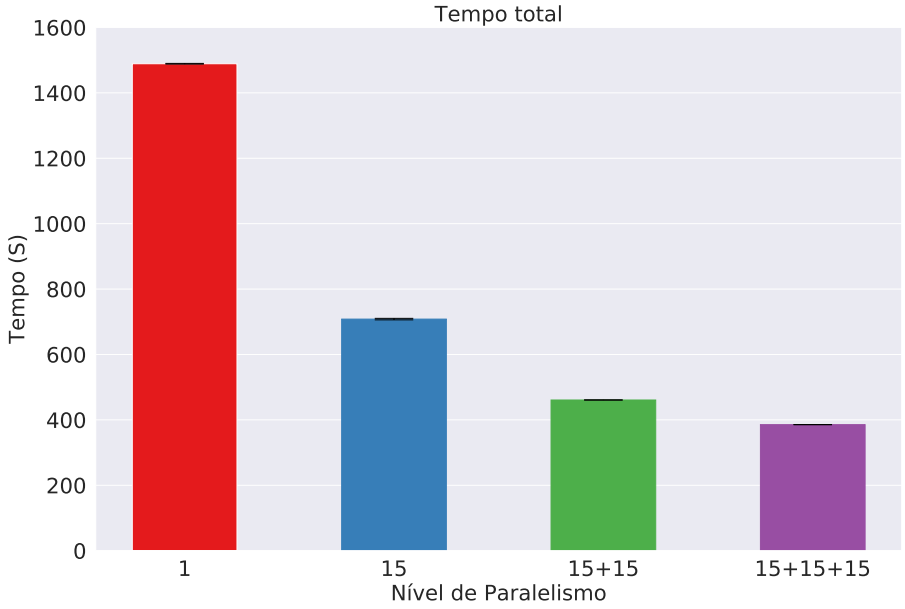
Arquivo	**Tamanho**
state.csv	6.8 GB
variables.csv	2.5 GB
link.csv	304 MB
dag.csv	270 MB
entities.csv	73 KB
events.csv	1.8 GB
Total	12 GB

Durante a implementação, optou-se por manter o processamento de entities no formato sequencial pois esse arquivo armazena apenas informações de plataforma e por isso, costuma não passar da ordem de tamanho de KB. Podemos observar que isso se confirma nesta carga de trabalho.

Foram executadas 30 repetições em cada teste, todavia, os experimentos com o Spark apresentaram problemas em algumas execuções, discutidos na Seção 5.3. Tivemos três testes com problemas ao executar com 15 executores enquanto com 30 e 45, tivemos apenas um. Nesses casos, consideramos apenas aquelas que processaram com sucesso (27, 29 e 29 repetições, respectivamente).

A Figura 5.2 mostra a média do tempo total de execução dos experimentos e o erro padrão no topo da barra (intervalo de confiança de 99,7%) em função do paralelismo em cada nó. A execução original de forma sequencial, levou em média 1489,02 segundos para completar o processamento. Já as execuções com a aplicação adaptada para utilizar o Spark, com 15 executores em 1 nó levou em média 708,17 segundos, com 30 em 2 nós, 460,81 segundos e 45 em 3, 385,44 segundos. Isso consiste em um *speedup* de respectivamente 2,10x, 3,23x e 3,86x em relação ao original.

Figura 5.2: Tempo total de execução da aplicação.



Segmentamos a análise pelas etapas exibidas na Figura 4.3, seus tempos de execução podem ser visualizados na Figura 5.3 e consultados com mais detalhes na Tabela 5.3. Analisando os resultados e o código, conseguimos separar as etapas em quatro grupos.

O primeiro e mais trivial deles, é o grupo em que não houve praticamente nenhuma alteração de código e tempo de execução. É o caso do tratamento de entities, que foi apenas convertido para uma tabela Spark para posterior gravação no HDFS.

Figura 5.3: Tempos de execução segmentados por etapas.

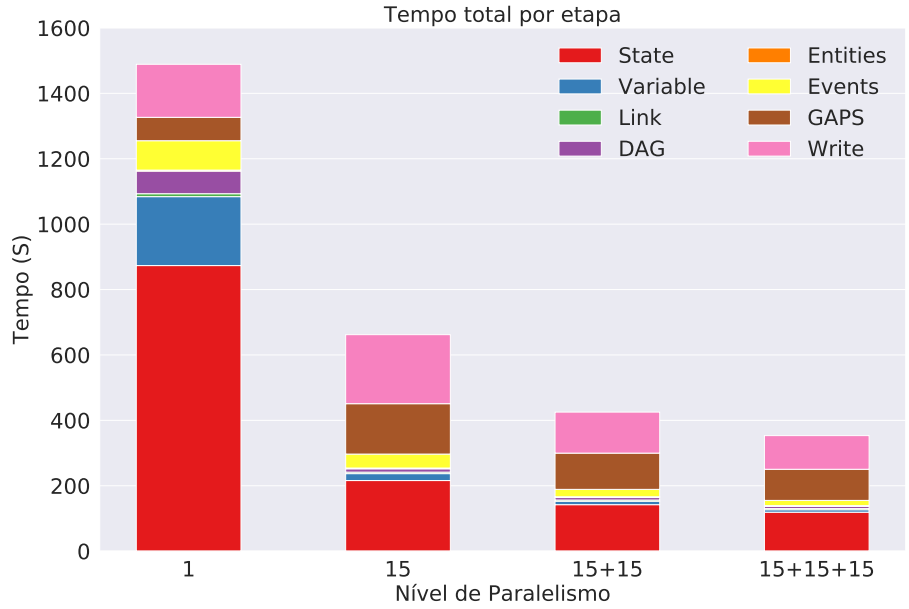


Tabela 5.3: Tempos médios de execução, em segundos.

Etapa	**1**	**15**	**15+15**	**15+15+15**
State	873.31	215.93	142.15	119.20
Variable	210.77	21.17	10.44	7.50
Link	8.93	4.59	3.84	3.53
DAG	69.14	10.10	7.58	6.76
Entities	3.07	2.42	2.31	2.30
Events	89.87	42.68	22.65	15.91
GAPS	71.51	154.03	110.83	95.22
Write	162.19	211.06	125.40	102.87

O segundo grupo identificado, foram manipulações que executaram apenas transformações (operações *Lazy*) no Spark (CHAMBERS; ZAHARIA, 2018) durante seu processamento. Estas constroem um plano de operações a serem aplicadas nos dados, todavia, a *engine* só irá executá-lo quando estritamente necessário (quando uma ação for realizada). Neste grupo enquadram-se as tabelas que observou-se um *speedup* muito grande, como é o caso de variable que apresentou 9,95x, 20,18x, 28,10x comparando-se as execuções com Spark com a execução Sequencial. A etapa que lida com o DAG também teve ganhos consideráveis, apresentando *speedups* de 6,84x, 9,12x e 10,22x. Os ganhos de link foram menos

significativos, pois seu tempo original já é pequeno, concluímos que ele pertence a este grupo pela análise do código de seu tratamento.

O terceiro grupo identificado foram transformações que de alguma forma ativaram uma ação no Spark durante seu processamento. Tais operações fazem com que a *engine* efetivamente manipule os dados (CHAMBERS; ZAHARIA, 2018) não sendo do tipo *Lazy* como as transformações. Esse grupo consiste nos tratamentos de state e events, nos quais tivemos um ganho inferior ao segundo grupo. Os ganhos observados para events foram de 2,10x, 3,96x e 5,64x, enquanto que para state tivemos 4,04x, 6,14x e 7,32x.

O último grupo, corresponde apenas ao Cálculo de GAPS. Essa operação foi levantada durante a implementação como um ponto de atenção, devido ao seu tempo ter aumentado de forma considerável em experimentos locais. Olhando para os tempos de execução temos, em segundos, 71,51 na execução sequencial, 154,03 na com 15 executores utilizando Spark, 110,83 com 30 executores e 95,22 utilizando 45 executores. Acreditamos que isso seja decorrente das ações Spark realizadas em tabelas geradas por junções de forma recursiva, utilizadas nesta etapa. Para identificar o ponto exato que causa essa perda de desempenho, são necessários mais testes.

Por fim, temos a escrita dos dados no sistema de arquivos distribuído. Nessa etapa pode-se observar que o ganho não é tão grande pois é nela que todos os planos de transformações montados nas etapas anteriores serão efetivamente executados sobre os dados. Isso é decorrente da escrita de uma tabela ser considerada uma ação e portanto, exige que transformações sejam realizadas. Pode-se observar, tanto na Tabela 5.3 quanto na Figura 5.3 que o tempo de execução com o Spark utilizando 15 executores é pior do que a execução sequencial. Depois dela, temos pequenos *speedups* de 1,29x e 1,57x.

Portanto, podemos concluir que a portagem do arcabouço StarVZ para executar sobre o Spark gerou ganhos consideráveis. Há outras diversas avaliações para serem realizadas, algumas serão enumeradas no próximo Capítulo.

5.3 Casos de Falha

Ocorreram algumas falhas de execução durante os testes comparativos entre a versão sequencial e distribuída. Nos logs da aplicação (no `Processo Dri-`

`ver` do Spark), é reportado que não foi possível criar uma conexão Spark. Nos logs do nó mestre não há registros de que houve a recepção de uma requisição para criar a aplicação, o que indica que a conexão efetivamente não foi criada. Devido ao comportamento observado, acreditamos que seja algum recurso que não foi liberado entre execuções, pois o tempo entre finalizar uma execução e iniciar outra é de apenas milissegundos. Para determinar a causa exata do problema, é necessário uma investigação mais aprofundada que deixamos como trabalhos futuros.

Foi disponibilizado uma carga de trabalho de 116 GB. Ao tentar executar a versão sequencial houve um erro de alocação de memória, o que era esperado tendo em vista que as máquinas possuem apenas 64 GB de memória. Todavia, executando a versão com Spark, ele também não completou a aplicação. Durante o processamento de State, o processo é bloqueado e encerrado por *timeout* (expiração de tempo de limite). Não houve tempo durante o trabalho para corrigir os problemas observados e rodar a aplicação com essa carga de trabalho, deixando isso também como um item para trabalhos futuros.

6 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho teve como objetivo otimizar o fluxo de Ciência de Dados realizado em R dentro da primeira fase (pré-processamento) do arcabouço StarVZ. Analisando trabalhos anteriores, identificamos que este seria o ponto de otimização mais interessante, tendo em vista que ele é o que mais contribui para esta fase, sendo responsável por 40,62% do tempo total.

Nos testes realizados com uma carga de trabalho de 12 GB, observamos um tempo de 1489,02 segundos ($\approx$ 24 minutos). Como esta é apenas uma parcela do tempo total da fase de pré-processamento, isso pode inviabilizar seu uso para maiores volumes de dados.

Nas adaptações realizadas, foi utilizado o ferramental para processamento de grandes volumes de dados, como o Hadoop e o Spark. Utilizamos o HDFS para armazenamento de entradas e saídas e o Spark para tratamento dos dados. Para adaptar a aplicação ao Spark, foi utilizada a biblioteca `sparklyr`, o que facilitou muito a adaptação pois ela é inspirada no `dplyr`, pacote utilizado no fluxo original.

Os experimentos mostraram redução no tempo de execução. Com 15 executores em um nó, onde apenas utilizamos o Spark para realizar o processamento paralelo, ele foi reduzido pela metade (2,10x de *speedup*). Ao realizar o processamento de forma distribuída com 30 e 45 executores, atingimos *speedups* de 3,23x e 3,86x respectivamente, no melhor caso, levando apenas 385,44 segundos. Embora esta seja apenas uma etapa da fase de pré-processamento, tal redução contribui para a viabilização de sua utilização com volumes de dados maiores.

Além disso, este fluxo é limitado pela quantidade física de memória presente nas máquinas. Ao utilizar o HDFS e o Spark, permitimos que a aplicação processe mais dados do que esta limitação, avanço importante para o processamento de grandes volumes de dados.

Como trabalhos futuros, dentro do fluxo de Ciência de Dados, é importante realizar mais avaliações (no final do trabalho foi disponibilizado uma entrada de centenas de Gigabytes, mas não tivemos tempo de realizar testes com ela). Diversificar as avaliações também é importante pois os rastros de outros tipos de aplicação (diferentes de cholesky) podem ter um comportamento dife-

rente. Além disso, as demais ferramentas utilizadas na fase de pré-processamento (`starpu_fxt_tool`, `dot2csv`, `pjdump`) também precisam ser trabalhadas.

Todo o código fonte desenvolvido neste trabalho está no Github. Ele pode ser acessado neste link, onde ainda deve receber atualizações referentes aos trabalhos futuros que foram expostos nesta Seção. Os arquivos LaTeXda monografia bem como alguns arquivos de apoio ao trabalho podem ser acessados neste link.

REFERÊNCIAS

ALLES, G. R.; SCHNORR, L. M. Parallel workflow support for starvz using drake. **WSPPD Proceedings**, p. 23 – 25, 2018. Available from Internet: <http://inf.ufrgs.br/gppd/wsppd/2018/proceedings.php>.

AUGONNET, C. et al. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. **Concurr. Comput. : Pract. Exper.**, John Wiley and Sons Ltd., Chichester, UK, v. 23, n. 2, p. 187–198, feb. 2011. ISSN 1532-0626. Available from Internet: <http://dx.doi.org/10.1002/cpe.1631>.

Ayguade, E. et al. The design of openmp tasks. **IEEE Transactions on Parallel and Distributed Systems**, v. 20, n. 3, p. 404–418, March 2009. ISSN 1045-9219.

BENGTSSON, H. **future Github Repository**. 2019. [Acessed: 2019-06-25]. Available from Internet: <https://github.com/HenrikBengtsson/future>.

BERKELAAR, M. **lp_solve reference guide Version 5.5**. 2016. [Acessed: 2019-05-05]. Available from Internet: <http://lpsolve.sourceforge.net/>.

Bosilca, G. et al. Parsec: Exploiting heterogeneity to enhance scalability. **Computing in Science Engineering**, v. 15, n. 6, p. 36–45, Nov 2013. ISSN 1521-9615.

CEBALLOS, G. et al. Analyzing performance variation of task schedulers with taskinsight. **Parallel Computing**, v. 75, p. 11 – 27, 2018. ISSN 0167-8191. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0167819118300346>.

CHAMBERS, B.; ZAHARIA, M. **Spark: The Definitive Guide : Big Data Processing Made Simple**. [S.l.]: O'Reilly Media, 2018. ISBN 9781491912218.

COULOMB, K. et al. **Visual Trace Explorer**. 2009. [Acessed: 2019-05-05]. Available from Internet: <http://vite.gforge.inria.fr/index.php>.

DEAN, J.; GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. **Commun. ACM**, ACM, New York, NY, USA, v. 51, n. 1, p. 107–113, jan. 2008. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/1327452.1327492>.

DOMINIK, C. **The Org Mode 7 Reference Manual - Organize Your Life with GNU Emacs**. [S.l.]: Network Theory Ltd., 2010. ISBN 1906966087, 9781906966089.

Dosimont, D. et al. A spatiotemporal data aggregation technique for performance analysis of large-scale execution traces. In: **2014 IEEE International Conference on Cluster Computing (CLUSTER)**. [S.l.: s.n.], 2014. p. 149–157. ISSN 1552-5244.

DURAN, A. et al. Ompss: a proposal for programming heterogeneous multi-core architectures. **Parallel Processing Letters**, v. 21, p. 173–193, 06 2011.

ESCHWEILER, D. et al. Open trace format 2: The next generation of scalable trace formats and support libraries. In: . [S.l.: s.n.], 2012. v. 22, p. 481 – 490. ISBN 9781614990406.

FOUNDATION, A. S. **Apache Parquet Website**. 2018. [Acessed: 2019-08-12]. Available from Internet: <https://parquet.apache.org/>.

GARCIA PINTO, V. et al. A visual performance analysis framework for task-based parallel applications running on hybrid clusters. **Concurrency and Computation: Practice and Experience**, v. 30, n. 18, p. e4472, 2018. E4472 cpe.4472. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4472>.

HAUGEN, B. et al. Visualizing execution traces with task dependencies. In: **Proceedings of the 2Nd Workshop on Visual Performance Analysis**. New York, NY, USA: ACM, 2015. (VPA '15), p. 2:1–2:8. ISBN 978-1-4503-4013-7. Available from Internet: <http://doi.acm.org/10.1145/2835238.2835240>.

HINDMAN, B. et al. Mesos: A platform for fine-grained resource sharing in the data center. In: **Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2011. (NSDI'11), p. 295–308. Available from Internet: <http://dl.acm.org/citation.cfm?id=1972457.1972488>.

Huynh, A.; Taura, K. Delay spotter: A tool for spotting scheduler-caused delays in task parallel runtime systems. In: **2017 IEEE International Conference on Cluster Computing (CLUSTER)**. [S.l.: s.n.], 2017. p. 114–125. ISSN 2168-9253.

HUYNH, A. et al. Dagviz: A dag visualization tool for analyzing task-parallel program traces. In: **Proceedings of the 2Nd Workshop on Visual Performance Analysis**. New York, NY, USA: ACM, 2015. (VPA '15), p. 3:1–3:8. ISBN 978-1-4503-4013-7. Available from Internet: <http://doi.acm.org/10.1145/2835238.2835241>.

ISAACS, K. **Ravel MPI trace visualization tool**. 2014. [Acessed: 2019-05-05]. Available from Internet: <https://github.com/LLNL/ravel>.

KELLER, R. et al. Temanejo: Debugging of thread-based task-parallel programs in starss. In: BRUNST, H. et al. (Ed.). **Tools for High Performance Computing 2011**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 131–137.

KNAUS, J. **Snowfall Cran Page**. 2015. [Acessed: 2019-06-25]. Available from Internet: <https://CRAN.R-project.org/package=snowfall>.

KNÜPFER, A. et al. The vampir performance analysis tool-set. In: RESCH, M. et al. (Ed.). **Tools for High Performance Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 139–155. ISBN 978-3-540-68564-7.

LANDAU, W. M. et al. **Drake Github Repository**. 2019. [Acessed: 2019-06-27]. Available from Internet: <https://github.com/ropensci/drake>.

Pagano, G. et al. Trace management and analysis for embedded systems. In: **2013 IEEE 7th International Symposium on Embedded Multicore Socs**. [S.l.: s.n.], 2013. p. 119–122.

PILLET, V. et al. Paraver: A tool to visualize and analyze parallel code. **WoTUG-18**, v. 44, 03 1995.

R Core Team. **R: A Language and Environment for Statistical Computing**. Vienna, Austria, 2017. [Acessed: 2019-05-04]. Available from Internet: <http://www.r-project.org/>.

STEIN, B. d. O. et al. **Pajé - trace file format**. 2015. [Acessed: 2019-05-07]. Available from Internet: <https://github.com/schnorr/pajeng/blob/master/doc/lang-paje/lang-paje.pdf>.

TIDYVERSE. 2019. [Acessed: 2019-05-05]. Available from Internet: <https://www.tidyverse.org/>.

TIERNEY, L. **Snow Home Page**. 2018. [Acessed: 2019-06-25]. Available from Internet: <http://homepage.divms.uiowa.edu/~luke/R/cluster/cluster.html>.

WHITE, T. **Hadoop: The Definitive Guide**. 4. ed. Beijing: O'Reilly, 2015. ISBN 978-1-4919-0163-2.

WICKHAM, H. **Ggplot2: Elegant Graphics for Data Analysis**. [S.l.: s.n.], 2009. ISBN 9780387981406.

WICKHAM, H. **feather: R Bindings to the Feather 'API'**. 2019. [Acessed: 2019-06-14]. Available from Internet: <https://cran.r-project.org/web/packages/feather/index.html>.

WORLD, D. I. **How Much Data Is Generated Per Minute?** 2018. [Acessed: 2019-08-19]. Available from Internet: <https://www.digitalinformationworld.com/2018/06/infographics-data-never-sleeps-6.html>.

YU, H. **RMPI Home Page**. 2013. [Acessed: 2019-06-24]. Available from Internet: <http://fisher.stats.uwo.ca/faculty/yu/Rmpi/>.

ZWOLENSKI, M.; WEATHERILL, L. The digital universe rich data and the increasing value of the internet of things. **Australian Journal of Telecommunications and the Digital Economy**, v. 2, 10 2014.

APÊNDICE A — DOCUMENTAÇÃO DO STARVZ SOBRE HDFS/SPARK

Nesta Seção serão mostrados os passos necessários para reproduzir os testes realizados. Existem diversas referências para o repositório deste trabalho no Github, onde encontram-se os scripts de automação de grande parte do setup e execução dos testes. É importante salientar que este é um repositório diferente do StarVZ.

O Setup do ambiente para executar os experimentos começa com o script `setup_environment.py`, dentro da pasta scripts do repositório. É necessário copiar também o script `setup_rpackages.R` para o mesmo diretório, pois este é chamado pelo primeiro.

Após executar o script `setup_environment.py`, para finalizar a instalação do Hadoop é necessário:

- Executar o comando `source ~/.bashrc`;
- Verificar se o parâmetro `fs.default.name` no arquivo `core-site.xml` está configurado corretamente (deve apontar para o *Namenode*);
- Verificar se o parâmetro `dfs.replication` no arquivo `hdfs-site.xml` está coerente. Nos experimentos, este foi definido como o número de nós;
- Configurar o arquivo `slaves`, que basicamente lista todos os nós do cluster Hadoop.

Feito isso, é necessário realizar as configurações do Spark. Para isso, deve-se configurar o arquivo `slaves`, da mesma forma que no Hadoop. Também é preciso configurar os parâmetros `SPARK_LOCAL_IP` e `SPARK_MASTER_HOST` no arquivo `spark-env.sh`, com o IP local e o IP do nó mestre respectivamente. Revise também os parâmetros no arquivo `spark-defaults.conf`, pois há uma série de apontamentos para o nó mestre.

Agora, vamos iniciar o Hadoop. Para isso, entre na pasta raiz de sua instalação e execute a sequência de comandos listada abaixo.

```
hadoop namenode -format
./sbin/start-dfs.sh
```

Para garantir que ele está rodando, pode ser executado o comando `hdfs dfsadmin -report`. Devem ser listados todos os *datanodes* do ambiente.

Agora, mude de diretório para onde estão os CSVs que serão utilizados como entrada. Execute a sequência de comandos listada abaixo. Isso criará a pasta `logs` e a pasta `inputs` no HDFS e copiará todos os arquivos CSV para a pasta inputs. Este comando pode demorar.

```
hdfs dfs -mkdir -p logs
hdfs dfs -mkdir -p inputs
hdfs dfs -put * inputs/
```

Com o Hadoop rodando, iremos iniciar o Spark. Para isso, no nó mestre, execute o seguinte comando: `./sbin/start-master.sh`. Nos nós trabalhadores, o comando é um pouco diferente: `./sbin/start-slave.sh <IP_MASTER>:7077`.

Se tudo ocorreu corretamente, agora temos o ambiente completo. Para executar o StarVZ, vá até sua pasta e execute o seguinte comando:

```
./R/phase1-workflow.R -D /inputs <PATH_PARA_ENTITIES>
spark://<IP_MESTRE>:7077 cholesky
```

Este deve iniciar a aplicação em modo distribuído e rodar a aplicação utilizando o Spark. Para repetições de testes, existe o script `experiment_execution.py`, que controla os experimentos, caso o usuário queira executar mais de uma repetição do mesmo teste.