DARLON VASATA

FERRAMENTA DE PROGRAMAÇÃO E PROCESSAMENTO PARA EXECUÇÃO DE APLICAÇÕES COM GRANDES QUANTIDADES DE DADOS EM AMBIENTES DISTRIBUÍDOS

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Doutor em Ciências.

São Paulo
2018

DARLON VASATA

FERRAMENTA DE PROGRAMAÇÃO E PROCESSAMENTO PARA EXECUÇÃO DE APLICAÇÕES COM GRANDES QUANTIDADES DE DADOS EM AMBIENTES DISTRIBUÍDOS

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Doutor em Ciências.

Área de Concentração:

Engenharia de Computação

Orientadora:

Prof.ª Doutora Liria Matsumoto Sato

São Paulo
2018

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catalogação-na-publicação

Vasata, Darlon

Ferramenta de programação e processamento para execução de aplicações com grandes quantidades de dados em ambientes distribuídos / D. Vasata, L. M. Sato -- versão corr. -- São Paulo, 2018.

175 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Big Data 2.Sistemas Distribuídos 3.Programação Paralela I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t. III.Sato, Liria Matsumoto

DEDICATÓRIA

Dedico este trabalho aos meus queridos avós Desiderio e Ivanil, João e Lúcia.

AGRADECIMENTOS

Primeiramente agradeço a Deus pelo dom da vida, e por toda a saúde e força fornecida para superar as dificuldades.

À minha orientadora, Dra. Liria Matsumoto Sato pelos ensinamentos e paciência durante toda a pesquisa. Ser seu aluno é um privilégio, e os conhecimentos adquiridos são para a vida toda.

Minha família... ah, a minha família. Obrigado por entenderem o tempo dedicado a este trabalho, que muito dele foi retirado do tempo em que eu deveria estar com vocês. Pai, Mãe e Daniel: Eu amo vocês.

Aos amigos que me acompanharam nesta caminhada, por toda a sua compreensão e apoio. Sejam aqueles que me apoiaram moralmente, ou então aqueles que me tiravam daqui. Uma fuga mental do trabalho, e companhia nos momentos de lazer. Àqueles que me abriram as portas de suas casas e reservaram um cantinho, sempre aconchegante, em seus lares para que esse amigo que estava em São Paulo pudesse descansar. Não teria chegado até aqui sem vocês. Serei eternamente grato.

Agradeço também este trabalho à ele mesmo, recursivamente. Graças a ele conheci pessoas maravilhosas, de todos os lugares. A experiência com o doutorado me trouxe conhecimento e amizades que jamais alcançaria sem ele, ou quem sabe, não teria dado o devido valor se estivesse em um ambiente diferente.

Agradeço também àqueles que não me acompanharam nesta caminhada até o final, por desencontros da vida. Este trabalho não seria o mesmo sem a ajuda e apoio de vocês.

Agradeço à Universidade de São Paulo, e principalmente à Escola Politécnica por viabilizar a estrutura necessária para a pesquisa e as atividades inerentes ao projeto de doutorado.

Agradeço também ao IFPR, por permitir que o doutorado fosse realizado com o afastamento de minhas atividades como docente da instituição.

A todos, meu muito obrigado.

“All you touch and all you see,

is all your life will ever be”

Breathe - Pink Floyd

RESUMO

A temática envolvendo o processamento de grandes quantidades de dados é um tema amplamente discutido nos tempos atuais, envolvendo seus desafios e aplicabilidade. Neste trabalho é proposta uma ferramenta de programação para desenvolvimento e um ambiente de execução para aplicações com grandes quantidades de dados. O uso da ferramenta visa obter melhor desempenho de aplicações neste cenário, explorando o uso de recursos físicos como múltiplas linhas de execução em processadores com diversos núcleos e a programação distribuída, que utiliza múltiplos computadores interligados por uma rede de comunicação, de forma que estes operam conjuntamente em uma mesma aplicação, dividindo entre tais máquinas sua carga de processamento.

A ferramenta proposta consiste na utilização de blocos de programação, de forma que tais blocos sejam compostos por tarefas, e sejam executados utilizando o modelo produtor consumidor, seguindo um fluxo de execução definido. A utilização da ferramenta permite que a divisão das tarefas entre as máquinas seja transparente ao usuário. Com a ferramenta, diversas funcionalidades podem ser utilizadas, como o uso de ciclos no fluxo de execução ou no adiantamento de tarefas, utilizando a estratégia de processamento especulativo.

Os resultados do trabalho foram comparados a duas outras ferramentas de processamento de grandes quantidades de dados, Hadoop e Spark. Tais resultados indicaram que o uso da ferramenta proporciona aumento no desempenho das aplicações, principalmente quando executado em *clusters* homogêneos.

Palavras-chave: *Big Data*. Sistemas Distribuídos. Programação Paralela.

ABSTRACT

The topic involving the processing of large amounts of data is widely discussed subject currently, about its challenges and applicability. This work proposes a programming tool for development and an execution environment for applications with large amounts of data. The use of the tool aims to achieve better performance of applications in this scenario, exploring the use of physical resources such as multiple lines of execution in multi-core processors and distributed programming, which uses multiple computers interconnected by a communication network, so that they operate jointly in the same application, dividing such processing among such machines.

The proposed tool consists of the use of programming blocks, so that these blocks are composed of tasks, and the blocks are executed using the producer consumer model, following an execution flow. The use of the tool allows the division of tasks between the machines to be transparent to the user. With the tool, several functionalities can be used, such as cycles in the execution flow or task advancing using the strategy of speculative processing.

The results were compared with two other frameworks, Hadoop and Spark. These results indicate that the use of the tool provides an increase in the performance of the applications, mostly when executed in homogeneous clusters.

Keywords: Big Data. Distributed Systems. Parallel Programming.

SUMÁRIO

Lista de Ilustrações

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviaturas e Siglas

Lista de Símbolos

1 Introdução **25**

1.1 Objetivo 28

1.2 Justificativa 29

1.3 Contribuições 29

1.4 Métodos 31

1.5 Convenções adotadas 33

1.6 Organização do trabalho 34

2 Conceitos e ferramentas **36**

2.1 Sistemas paralelos e distribuídos 36

2.1.1 *Clusters* de computadores 37

2.1.2 MPI 38

2.1.3 Sistemas de arquivos distribuídos 39

2.1.3.1 HDFS 39

2.1.4 Desenvolvimento de aplicações paralelas 42

2.2 *Big Data* 43

2.2.1 O modelo Mapreduce 45

3 Revisão bibliográfica **50**

3.1 Hadoop 50

3.2 Spark 54

3.3 Outros trabalhos 56

3.4 Considerações sobre os trabalhos citados 60

4 Ferramenta de programação e processamento de grandes quantidades de dados em ambiente distribuído **61**

4.1 Modelo de programação 62

4.2 Execução em blocos 63

4.2.1 Funções definidas pelo usuário 64

4.2.2 Blocos de execução 65

4.2.3 Suporte a múltiplas entradas e saídas entre os blocos 66

4.2.4 Múltiplas entradas e saídas 66

4.2.5 Execução iterativa 67

4.3 Execução de aplicações 68

4.4 Arquitetura 71

4.4.1 Blocos de entrada de dados e blocos de saída de dados 73

4.5 Outras funcionalidades 74

4.5.1 Processamento especulativo 74

4.5.2 Armazenamento intermediário 77

4.5.3 Execução iterativa 78

4.6 Comparação com os trabalhos relacionados 79

5 Implementação do sistema de execução **80**

5.1 Estruturas de dados 80

5.1.1 Módulos de armazenamento de valores e controle de granularidade 82

5.2 Sistema de execução 84

5.3 Banco de memória 85

5.4 Funções do usuário 86

5.4.1 Funções definidas pelo usuário 86

5.4.2 Configurações do sistema e da execução de aplicações 87

5.5 Blocos de execução 87

5.5.1 Execução paralela das tarefas 89

5.6 Múltiplas entradas e saídas 90

5.7 Trocas de dados 91

5.8 Agrupamento de valores em duas etapas 93

5.9 Blocos de entrada de dados e blocos de saída de dados 94

5.10 Blocos de armazenamento intermediário 95

5.11 Processamento especulativo 95

5.12 Execução utilizando ciclos 98

6 Resultados e discussão **99**

6.1 Configurações utilizadas nos testes 99

6.1.1 Ambiente de execução 99

6.1.2 Arquivos de entrada 100

6.2 Descrição dos testes realizados 101

6.2.1 Contagem de palavras 102

6.2.2 Contagem de palavras com contagem de palíndromos 102

6.3 Resultados 104

6.3.1 Programação em blocos 105

6.3.1.1 Contagem de palavras 105

6.3.1.2 Contagem de palavras com contagem de palíndromos 110

6.3.1.3 Blocos de armazenamento intermediário 111

6.3.2 Processamento especulativo 112

6.3.3 Ambiente heterogêneo 115

6.3.4 Análise de tráfego 120

6.3.5 Outras quantidades de dados 121

6.3.6 Processamento iterativo 124

7 Conclusão **128**

7.1 Trabalhos futuros 131

7.2 Publicações realizadas 131

Apêndice A – Funções do usuário e exemplos **133**

A.1 Documentação 133

A.1.1 udf.h 133

A.1.2 keyvalue.h 135

A.2 Contador de palavras 147

A.2.1 Configurações da aplicação 147

A.2.2 map 147

A.2.3 reduce 148

Apêndice B – Métricas **150**

B.1 Tempo de execução 150

B.2 *Speedup* 150

B.3 Eficiência 150

Apêndice C – Dados obtidos **152**

Referências **162**

Glossário **170**

LISTA DE ILUSTRAÇÕES

1 Exemplo de armazenamento de arquivos utilizando HDFS 40

2 Pilares do *Big Data* 44

3 Execução do modelo de programação Mapreduce 47

4 Arquitetura do *framework* Hadoop versão 1 51

5 Arquitetura do *framework* Hadoop versão 2 52

6 Processos Hadoop versão 1 52

7 Processos Hadoop versão 2 53

8 Exemplo de RDD realizando contagem de palavras 55

9 Modelo de programação em blocos 62

10 Arquivos de configuração e programação das tarefas pelo usuário . . . 63

11 Fluxo de execução em blocos 64

12 Bloco de execução 65

13 Bloco de execução em ambiente distribuído 66

14 Fluxo de execução com blocos com múltiplas entradas e saídas 67

15 Estrutura interna do bloco Reduce A 67

16 Execução utilizando múltiplas entradas e saídas 67

17 Exemplo de execução utilizando iteratividade 68

18 Blocos e fluxo de execução no sistema 69

19 Exemplo de fluxo de execução de aplicações 70

20 Máquinas no ambiente distribuído executando blocos 71

21 Arquitetura do Sistema 72

22 Estrutura interna do bloco de entrada de dados 73

23 Estrutura interna do bloco de saída de dados 74

24 Exemplo de dependência de dados em tarefas de reduce 75

25 Bloco de armazenamento intermediário 77

26 Grafo direcionado e grafo direcionado acíclico 78

27 Estrutura de dados utilizada para representar chaves 81

28 Estrutura de dados utilizada para representar valores 81

29 Estrutura de dados utilizada para representar chaves e valores 81

30 Exemplo de armazenamento de estrutura 82

31 Módulo de armazenamento das estruturas 83

32 Grupo de pares KV 84

33 Código do cabeçalho de função para funções UDF 86

34 Código do cabeçalho da função `Emit` utilizada nas funções UDF . . . 86

35 *Thread* passando parâmetros e executando chamadas às funções . . . 88

36 *Thread* retirando elementos KV de saída das execuções UDF e inserindo-os nas estruturas de saída do bloco 89

37 Estrutura de múltiplas saídas 90

38 Armazenamento em sequência de valores KV para trocas de dados . . 91

39 Envio dos dados de forma assíncrona utilizando MPI 92

40 Confirmação de recebimento dos dados enviados 93

41 Recebimento dos dados de forma assíncrona utilizando MPI 93

42 Bloco de entrada 94

43 Bloco de saída 95

44 Busca de dados utilizando estrutura multinível 96

45 Exemplo de fluxo de execução utilizando ciclos 98

46 Aplicação de contagem de palavras 102

47 Contagem de palavras e palíndromos, versão 1 103

48 Contagem de palavras e palíndromos, versão 2 103

49 Contagem de palavras e palíndromos, versão 3 104

50 Contagem de palavras e palíndromos, versão 4 104

51 Tempo de execução 105

52 Tempo de execução utilizando diferentes valores para máscara de granularidade 107

53 Tempo de execução utilizando diferentes números de *threads* 107

54 Speedup e eficiência 109

55 Tempo de execução de contagem de palavras e palíndromos 110

56 Execuções utilizando blocos de armazenamento temporário, com múltiplos blocos e múltiplas *threads* 113

57 Processamento especulativo 114

58 Processamento especulativo com diferentes quantidades de bits 115

59 Comparação entre sistemas em ambiente heterogêneo 117

60 Execuções utilizando apenas uma máquina separadas por máquina . . 118

61 Tempo de execução em ambiente heterogêneo utilizando diferentes quantidades de *threads* 119

62 Tempo de execução em ambiente heterogêneo utilizando a mesma quantidade de dados de entrada 120

63 Quantidade de bytes trafegados 121

64 Quantidade de pacotes trafegados 122

65 Quantidade de pacotes de diferentes tamanhos trafegados 122

66 Tempo de execução com entrada utilizando blocos menores 123

67 Tempo de execução utilizando diversos blocos por máquina 124

68 Fluxo de execução do algoritmo *K-Means* implementado 125

69 Saída da execução do algoritmo *K-Means* utilizando 1 milhão de pontos 126

70 Exemplo de arquivo com informações da execução da aplicação do algoritmo *K-Means* implementado 127

71 Exemplo de arquivo com informações da execução da aplicação . . . 148

72 Exemplo de função map em C que realiza a contagem de palavras . . 149

73 Exemplo de função reduce em C que realiza a contagem de palavras . 149

LISTA DE TABELAS

1 Número de máquinas e tamanho da entrada utilizado em cada teste, expresso em MiB 100

2 Parâmetros utilizados por padrão 101

3 Dados utilizados no gráfico da Figura 51 - Tempo de execução (pg. 105) 152

4 Desvios padrão dos valores utilizados no gráfico da Figura 51 - Tempo de execução (pg. 105) 152

6 Desvios padrão dos dados utilizados no gráfico da Figura 52 - Tempo de execução utilizando diferentes valores para máscara de granularidade (pg. 107) 153

7 Dados utilizados no gráfico da Figura 53 - Tempo de execução utilizando diferentes números de *threads* (pg. 107) 153

5 Dados utilizados no gráfico da Figura 52 - Tempo de execução utilizando diferentes valores para máscara de granularidade (pg. 107) . . . 153

8 Desvios padrão dos dados utilizados no gráfico da Figura 53 - Tempo de execução utilizando diferentes números de *threads* (pg. 107) 154

9 Dados da entrada individual utilizados no gráfico da Figura 54 - *Speedup* e eficiência (pg. 109) 154

10 Dados da entrada distribuída utilizados no gráfico da Figura 54 - *Speedup* e eficiência (pg. 109) 154

11 Dados utilizado no gráfico da Figura 55 - Tempo de execução de contagem de palavras e palíndromos (pg. 110) 154

12 Desvios padrão dos dados utilizado no gráfico da Figura 55 - Tempo de execução de contagem de palavras e palíndromos (pg. 110) 154

13 Dados do teste utilizando armazenamento intermediário utilizados no gráfico da Figura 56 - Execuções utilizando blocos de armazenamento temporário, com múltiplos blocos e múltiplas *threads* (pg. 113) 155

14 Desvios padrão dos dados do teste utilizando armazenamento intermediário utilizados no gráfico da Figura 56 - Execuções utilizando blocos de armazenamento temporário, com múltiplos blocos e múltiplas *threads* (pg. 113) 155

15 Dados do teste sem utilizar armazenamento intermediário utilizados no gráfico da Figura 56 - Execuções utilizando blocos de armazenamento temporário, com múltiplos blocos e múltiplas *threads* (pg. 113) 155

16 Desvios padrão dos dados do teste sem utilizar armazenamento intermediário utilizados no gráfico da Figura 56 - Execuções utilizando blocos de armazenamento temporário, com múltiplos blocos e múltiplas *threads* (pg. 113) 155

17 Dados utilizados no gráfico da Figura 57 - Processamento especulativo (pg. 114) 155

18 Desvio padrão dos dados utilizados no gráfico da Figura 57 - Processamento especulativo (pg. 114) 155

19 Dados da entrada individual utilizados no gráfico da Figura 58 - Processamento especulativo com diferentes quantidades de bits (pg. 115) 156

20 Desvios padrão dos dados da entrada individual utilizados no gráfico da Figura 58 - Processamento especulativo com diferentes quantidades de bits (pg. 115) 156

21 Dados da entrada distribuída utilizados no gráfico da Figura 58 - Processamento especulativo com diferentes quantidades de bits (pg. 115) 156

22 Desvios padrão dos dados da entrada distribuída utilizados no gráfico da Figura 58 - Processamento especulativo com diferentes quantidades de bits (pg. 115) 156

23 Dados utilizados no gráfico da Figura 59 - Comparação entre sistemas em ambiente heterogêneo (pg. 117) 157

24 Desvios padrão dos dados utilizados no gráfico da Figura 59 - Comparação entre sistemas em ambiente heterogêneo (pg. 117) 157

25 Dados de 1 a 4 *threads* do gráfico da Figura 60 - Execuções utilizando apenas uma máquina separadas por máquina (pg. 118) 157

26 Dados de 5 a 8 *threads* do gráfico da Figura 60 - Execuções utilizando apenas uma máquina separadas por máquina (pg. 118) 157

27 Desvios padrão dos dados de 1 a 4 *threads* do gráfico da Figura 60 - Execuções utilizando apenas uma máquina separadas por máquina (pg. 118) 158

28 Desvios padrão dos dados de 5 a 8 *threads* do gráfico da Figura 60 - Execuções utilizando apenas uma máquina separadas por máquina (pg. 118) 158

29 Dados de 1 a 4 *threads* utilizados no gráfico da Figura 61 - Tempo de execução em ambiente heterogêneo utilizando diferentes quantidades de *threads* (pg. 119) 158

30 Dados de 5 a 8 *threads* utilizados no gráfico da Figura 61 - Tempo de execução em ambiente heterogêneo utilizando diferentes quantidades de *threads* (pg. 119) 159

31 Desvios padrão dos dados de 1 a 4 *threads* utilizados no gráfico da Figura 61 - Tempo de execução em ambiente heterogêneo utilizando diferentes quantidades de *threads* (pg. 119) 159

32 Desvios padrão dos dados de 5 a 8 *threads* utilizados no gráfico da Figura 61 - Tempo de execução em ambiente heterogêneo utilizando diferentes quantidades de *threads* (pg. 119) 159

33 Dados utilizados no gráfico da Figura 62 - Tempo de execução em ambiente heterogêneo utilizando a mesma quantidade de dados de entrada (pg. 120) 159

34 Desvio padrão dos dados utilizados no gráfico da Figura 62 - Tempo de execução em ambiente heterogêneo utilizando a mesma quantidade de dados de entrada (pg. 120) 160

35 Dados utilizado no gráfico da Figura 63 - Quantidade de bytes trafegados (pg. 121) 160

36 Dados utilizado no gráfico da Figura 64 - Quantidade de pacotes trafegados (pg. 122) 160

37 Dados utilizado no gráfico da Figura 65 - Quantidade de pacotes de diferentes tamanhos trafegados (pg. 122) 160

38 Dados utilizado no gráfico da Figura 66 - Tempo de execução com entrada utilizando blocos menores (pg. 123) 160

39 Desvio padrão dos dados utilizado no gráfico da Figura 66 - Tempo de execução com entrada utilizando blocos menores (pg. 123) 161

40 Dados utilizado no gráfico da Figura 67 - Tempo de execução utilizando diversos blocos por máquina (pg. 124) 161

41 Dados utilizado no gráfico da Figura 67 - Tempo de execução utilizando diversos blocos por máquina (pg. 124) 161

LISTA DE ALGORITMOS

1 Pseudocódigo da função *map* 48

2 Pseudocódigo da função *reduce* 48

3 Execução de aplicações no sistema de execução 85

4 Pseudocódigo do processamento especulativo 97

5 Pseudocódigo do algoritmo *K-Means* 125

LISTA DE ABREVIATURAS E SIGLAS

API *Application Programming Interface*

B Byte

CPU *Central Processing Unit*

DAG *Directec Acyclic Graph*

DG *Directed Graph*

EB Exabyte

ERAD-SP Escola Regional de Alto Desempenho do estado de São Paulo

FSB *Front Side Bus*

GFS *Google File System*

HPC *High Performance Computing*

ICMP *Internet Control Message Protocol*

IFPR Instituto Federal do Paraná

KB Kilobyte

KV *Key-Value*

KVG *Key-Value Group*

MB Megabyte

MPI *Message Passing Interface*

NFS *Network File System*

PB Petabyte

PPGEE Programa de Pós-Graduação em Engenharia Elétrica da Escola Politécnica da USP

SQL *Structure Query Language*

TB Terabyte

UDF *User Defined Function*

USP Universidade de São Paulo

XML *eXtended Markup Language*

YARN *Yet Another Resource Negotiator*

LISTA DE SÍMBOLOS

c	Número de bits utilizados para indexação em sub árvores.
g	Índice de granularidade. Consiste no número de bits utilizados para agrupar pares KV.
g_{id}	Identificação do grupo de KVG.
p	Número de instâncias de execução simultâneas em blocos de execução.
$H(x)$	Função de *hash*.
h	Valor de *hash*.
l	Número de blocos que fazem parte da janela deslizante, também dito como tamanho da janela.
m	Máquina do ambiente de execução.
n	Número de máquinas que compõem o ambiente de execução.
p	Número de *threads* operando simultaneamente em blocos de execução.
r	Fator de replicação utilizado no HDFS.
s	Tamanho do vetor utilizado na busca em árvore multinível.

1 INTRODUÇÃO

Nos últimos anos o cenário da computação viu-se com um novo desafio, derivado dos avanços tecnológicos que resultaram na geração de grandes conjuntos de dados. Entre tais avanços pode-se citar o acesso em grande escala da população às diversas mídias digitais e os progressos envolvendo tecnologias de sensores e monitoramento (ANAGNOSTOPOULOS; ZEADALLY; EXPOSITO, 2016).

A quantidade aproximada de dados gerados até o ano de 2003 foi de 5 exabytes (EB[1]). Estima-se que atualmente essa quantidade de dados dobra a cada ano, chegando a 8 zettabytes (ZB[2]) ao final de 2015 (SAGIROGLU; SINANC, 2013). Informações disponibilizadas em redes sociais, dados sobre geolocalizações e resultados de experimentos científicos são exemplos de dados gerados diariamente.

Com a análise desses dados em larga escala, informações de valor podem ser obtidas. Porém, a capacidade de processamento necessária para esse tipo de análise transcende os limites tradicionais da computação. Este desafio de explorar um grande volume de dados popularizou-se pelo nome de *Big Data* (CHEN; MAO; LIU, 2014), que trata de lidar com quantidades de dados que superam os limites conhecidos para bancos de dados, assim como ferramentas de análise, gerenciamento e armazenamento de informações (MGI, 2011; KAISLER et al., 2013). Estudos já tratam da computação em *exaescala* (AMARASINGHE et al., 2009; REED; DONGARRA, 2015; SARKAR; HARROD; SNAVELY, 2009), com o processamento de dados na ordem dos exabytes

[1] 1 EB equivale a 10^8 bytes
[2] 1 ZB equivale a 10^9 bytes

considerando aspectos como concorrência, eficiência energética e resiliência.

A análise de dados envolvendo *Big Data* pode trazer diversos benefícios, como indicam trabalhos discutindo sua aplicação em setores como saúde pública (MAGALHÃES et al., 2018; KHOURY; IOANNIDIS, 2014; BOLLIER; FIRESTONE, 2010), educação (GIBSON; WEBB, 2015; WEST, 2012; VAITSIS; NILSSON; ZARY, 2014), indústria (KUSIAK, 2015), meio ambiente (BIN et al., 2015), comércio eletrônico (ZHANG; ZHU, 2014; RAGUNATHAN et al., 2015), segurança digital (TERZI; TERZI; SAGIROGLU, 2017), aprendizagem de máquina (SINGH; JAISWAL, 2018) e visualização de informações (SHNEIDERMAN, 2014).

Para que tais benefícios sejam obtidos utilizando a análise desses dados, desafios ainda precisam ser solucionados. Os recentes avanços tecnológicos fizeram com que a capacidade de geração de dados aumente mais rápido do que a tecnologia para processamento de informações (REED; DONGARRA, 2015). O baixo custo no armazenamento de dados viabilizou que grandes volumes de dados fossem armazenados. Porém, o poder computacional necessário para a manipulação desses dados aliado ao custo e tempo envolvidos para transferi-los de seu local de armazenamento à unidade de processamento são problemas em aberto no contexto de *Big Data*. Desta forma, este cenário demanda a criação de novos paradigmas para o desenvolvimento de aplicações que tratem essa grande quantidade de dados.

Uma abordagem com que este problema vem sendo tratado utiliza modelos que consistem em executar as aplicações em máquinas no qual dados já se encontram previamente armazenados. Esta abordagem tem como base que, em um cenário em que é tratada uma grande quantidade de dados, o custo de enviar a aplicação aos dados é inferior ao custo de enviar os dados à aplicação. Esta prática possibilita melhor desempenho na execução das aplicações devido ao menor tráfego de informações utilizando a rede de comunicação entre as máquinas do ambiente distribuído.

Modelos do tipo Mapreduce e suas variações, oferecidas por ferramentas (SINGH;

REDDY, 2014), como Hadoop, Yarn, Spark, Stratosphere, entre outras, abordam o problema do processamento envolvendo grandes quantidades de dados. Tais modelos e ferramentas permitem que o processamento de alto desempenho seja aliado à facilidade de programação do ponto de vista do usuário, devido à distribuição automática de tarefas no ambiente de execução.

Porém, o modelo de programação Mapreduce possui limitações, relacionadas principalmente em relação a seu desempenho de execução e à gama de possíveis aplicações a serem executadas utilizando o modelo (LIN, 2012).

Os modelos e sistemas disponíveis permitem que o fluxo de execução das tarefas que compõem uma aplicação seja definido por um grafo acíclico direcionado (DAG[3]) (SEDGEWICK; WAYNE, 2014; CAO et al., 2017). Utilizando este tipo de representação, os vértices do grafo representam as tarefas executadas e as arestas indicam a troca de informações entre as diversas tarefas. O uso de um DAG permite que cada tarefa seja executada apenas uma única vez, devido à ausência de ciclos no grafo. Assim, tarefas que necessitem de iterações devem contornar esse problema armazenando temporariamente resultados intermediários e realizar diversas submissões da aplicação ao sistema, utilizando tais resultados armazenados como entrada para uma nova execução da aplicação. Esta abordagem possui desvantagens, como o grande número de acessos ao sistema de armazenamento para a escrita e leitura dos resultados intermediários e a necessidade de recarregar as tarefas a serem executadas em memória a cada início de execução de aplicação.

Para suprir algumas dessas limitações é necessária a disponibilidade de um suporte eficiente para a execução de operações que envolvem os seguintes aspectos:

- processamento iterativo, de forma a possibilitar a realização de operações que envolvam repetições em um fluxo de execução (MAZUR et al., 2011), com a utilização de grafo direcionado (DG);

[3] Acrônimo do termo em inglês *Directed Acyclic Graph*

- processamento com múltiplas entradas e saídas, possibilitando que diversas fontes de dados sejam utilizadas, mesmo que em diferentes instantes da execução das aplicações;
- adiantamento no processamento de tarefas que seguem um fluxo de execução, utilizando processamento especulativo.

1.1 Objetivo

O suporte a uma gama maior de algoritmos para processamento de informações em grandes quantidades vem se tornando cada vez mais necessário. Frente a isto, este trabalho objetiva apresentar um modelo de programação para cenário de *Big Data*, e desenvolver um ambiente de execução tal que aplicações desta natureza possam ser desenvolvidas e executadas de maneira eficiente, com a distribuição de suas tarefas pelo ambiente de forma transparente ao usuário.

De forma específica, o modelo apresentado visa possibilitar que uma gama maior de algoritmos seja aplicada à área de Big Data, com o oferecimento de funcionalidades que permitem a execução de tarefas seguindo um fluxo de execução de operações utilizando ciclos (SEDGEWICK; WAYNE, 2014).

Como objetivos secundários, estão a execução paralela e distribuída transparente ao usuário e a melhora de desempenho. Para atingi-los, o modelo foi desenvolvido com as diversas tarefas do fluxo de execução divididas em blocos, possibilitando que a execução de cada bloco seja realizada de forma distribuída, com a orquestração de trocas de dados e sincronização de tarefas sendo realizadas de forma automática. Quanto à melhora de desempenho é implementada a execução especulativa, que consiste em realizar o adiantamento do processamento de tarefas do fluxo de execução. Ainda em relação ao desempenho, a ferramenta é desenvolvida proporcionando maior aproveitamento dos dados em memória e múltiplas linhas de execução, utilizando recursos

proporcionados com o uso de *hardwares* dotados de diversas unidades de processamento.

1.2 Justificativa

As pesquisas envolvendo a temática do *Big Data* atualmente encontram-se em ampla expansão, com muitas discussões e resultados envolvendo o assunto, principalmente em relação à sua aplicabilidade em diversos setores da sociedade.

Porém, a pesquisa envolvendo *Big Data* demanda que novos limites para a computação sejam alcançados. A manipulação de grandes quantidades de dados requer novas técnicas de análise, considerando barreiras atuais como memória, poder de processamento, capacidade de armazenamento e tráfego de dados (KAISLER et al., 2013; REED; DONGARRA, 2015; EMANI; CULLOT; NICOLLE, 2015). Modelos de programação que promovam alto desempenho são necessários para que a computação seja possível neste cenário.

O uso de execuções utilizando DG permite que novos algoritmos sejam executados seguindo o modelo proposto neste trabalho, devido à possibilidade de utilização de ciclos no fluxo de execução. No modelo Mapreduce original, este tipo de aplicação deve ser decomposta utilizando diversas submissões ao sistema, devido à sua característica de não suporte a repetições. Com a utilização do modelo proposto no trabalho, estas aplicações podem ser desenvolvidas completamente e submetidas ao sistema uma única vez utilizando os recursos fornecidos, ou até mesmo fazerem parte de uma aplicação mais complexa.

1.3 Contribuições

O trabalho realizado nesta tese pretende contribuir nos seguintes aspectos da área de alto desempenho e *Big Data*:

- desenvolvimento de aplicações para *Big Data* em blocos seguindo um fluxo de execução, permitindo que blocos sejam executados de forma coordenada;
- utilização de diversas linhas de execução simultâneas para processamento de tarefas, fazendo melhor proveito de recursos computacionais disponíveis;
- aplicação de computação distribuída, realizando de forma automática a comunicação entre diversas máquinas em um ambiente distribuído, com a divisão de tarefas entre as máquinas;
- dados de entrada para máquinas no ambiente realizado utilizando dados do sistema de arquivos distribuído armazenado localmente na máquina, diminuindo a transferência de dados entre máquinas;
- processamento utilizando múltiplas entradas e saídas nos diversos blocos de execução;
- processamento em memória, repassando para os próximos blocos apenas endereços para os dados, diminuindo a necessidade de cópias das informações;
- reaproveitamento de espaço em memória previamente alocado, diminuindo o número de chamadas de sistema para alocação e liberação de memória, otimizando o desempenho das aplicações;
- maior desempenho de aplicações, através de otimizações possíveis com o compartilhamento da memória local entre os diversos blocos de execução;
- utilização de processamento especulativo, possibilitando a computação de tarefas do bloco seguinte no fluxo de execução antes da finalização das tarefas do bloco atualmente sendo processado;
- suporte a aplicações com ciclos de repetição.

1.4 Métodos

Para que os objetivos descritos fossem atingidos, foram realizadas as seguintes atividades:

1. pesquisa bibliográfica sobre os aspectos e características da área de *Big Data*;
2. busca sobre os desafios em aberto para a área;
3. estudo sobre como a computação paralela e distribuída poderia contribuir para a solução do problema;
4. elaboração da estratégia da programação em blocos, a ser utilizada como solução para o problema do processamento de grandes quantidades de dados;
5. projeto computacional da estratégia de programação em blocos, com estruturas de dados, modelo de comunicação e modelo de processamento;
6. divisão do projeto computacional em diversos módulos, como armazenamento, comunicação, processamento, dentre outros;
7. elaboração dos testes a serem realizados para validação da solução para o problema;
8. implementação computacional de cada módulo, com testes isolados de funcionamento;
9. implementação computacional da estratégia de programação em blocos, com todos os módulos implementados operando conjuntamente;
10. preparação e documentação do ambiente de execução;
11. testes utilizando a implementação do modelo de programação em blocos;
12. análise e comparação dos resultados com outras ferramentas;

13. otimizações na implementação, e novos testes;
14. testes utilizando diferentes configurações da aplicação, variando o nível de granularidade no agrupamento de pares (k, v);
15. análise dos resultados obtidos;
16. testes utilizando diferentes quantidades de *threads*;
17. análise dos resultados obtidos;
18. projeto da estratégia de processamento especulativo;
19. implementação da estratégia de processamento especulativo;
20. análise dos resultados obtidos;
21. projeto, implementação e testes da estratégia de armazenamento intermediário;
22. análise de resultados e otimizações na estratégia de armazenamento intermediário;
23. projeto, implementação e testes da estratégia de utilização de ciclos no fluxos de execução;
24. análise de resultados e otimização na estratégia de utilização de ciclos;
25. captura e pacotes e análise de tráfego de dados na aplicação;
26. análise geral dos resultados obtidos.

O desenvolvimento da ferramenta foi realizado utilizando a linguagem de programação C, com testes em um ambiente distribuído em um *cluster* de computadores.

1.5 Convenções adotadas

Algumas convenções foram utilizadas neste trabalho, com objetivo de facilitar sua leitura e entendimento.

Termos destacados com fonte do tipo `typewriter` referem-se a aspectos puramente computacionais, como comandos, nomes de arquivos ou trechos de código. Ex.: `int soma (int a, int b)`.

Os termos **Map** e **Reduce** (com letra inicial maiúscula) são referentes a estágios de execução do modelo Mapreduce, enquanto **map** e **reduce** (com letra inicial minúscula) referem-se às tarefas executadas nessas fases. Apresentam-se com letra inicial minúscula e em itálico (***map*** e ***reduce***) às tarefas definidas pelo usuário em cada operação. A implementação de códigos que realizam tais tarefas são identificados como **`map`** e **`reduce`**. Exemplos: a fase de **Reduce** ocorre após a fase de **Map**; as tarefas de **map** tratam da separação de palavras; o código ***map*** realiza a leitura de uma linha completa e separa cada palavra; o código **`map`** utiliza vetores do tipo `char` para o armazenamento das palavras lidas e números do tipo `int` para registrar a quantidade de ocorrências de cada palavra.

O termo **disco** é utilizado para referenciar sistemas de armazenamento em geral, não limitando-se apenas a armazenamento em disco rígido. Engloba outros sistemas, como discos de estado sólido (SSD[4]), por exemplo. A palavra Sistema (com S maiúsculo) é um substantivo, referindo-se ao sistema computacional desenvolvido como ambiente de execução seguindo as estratégias apresentadas neste trabalho. **Aplicação** é dito ser o *software* executado no Sistema, assim como **usuário** quem o executa e **desenvolvedor**, o programador da aplicação.

Para prefixos de unidades de medida de informação são utilizados os padrões do Sistema Internacional (SI) e da norma IEEE 1541 (IEEE, 2009). Em momentos em

[4] Acrônimo do termo em inglês *Solid State Disk* - livre trad.

que a quantidade de informação é aproximada, o SI é utilizado (Ex.: KB, MB e GB). Para situações de quantidade exata em potência de 2, a IEEE 1541 é adotada (Ex.: KiB, MiB e GiB).

Convencionou-se neste trabalho adotar a grafia utilizando fonte tipo `typewriter` a elementos diretamente relacionados à implementação, como estruturas de dados, trechos de código e parâmetros de funções.

Neste trabalho é utilizada uma abstração matemática de agrupamento de informações, definida pela associação entre uma chave e um valor ou grupo de valores, chamado aqui de par chave-valor. As estruturas de dados relativas a tal abstração denominam-se **elemento KV**[5] e **par KV**, sendo estas equivalentes. Da mesma forma, referencia-se como ***hash* do par** ao valor resultante da aplicação de uma função de *hash* à chave *k* presente em um par chave-valor.

1.6 Organização do trabalho

Este texto está organizado da seguinte forma: a introdução do trabalho consta no Capítulo 1, detalhando o objetivo do trabalho, justificativa, suas contribuições e métodos utilizados. Conceitos base para o trabalho são descritos no Capítulo 2, com o intuito de situar o leitor sobre as tecnologias utilizadas no trabalho.

De forma a situar o trabalho no cenário científico atual, a revisão bibliográfica realizada é apresentada no Capítulo 3, citando trabalhos relacionados e suas conexões. A proposta do trabalho é apresentada e discutida no Capítulo 4. Detalhes técnicos e ferramentas utilizadas na implementação da ferramenta e ambiente de execução são descritos no Capítulo 5. Os resultados obtidos e sua discussão são expostos no Capítulo 6, com a conclusão mostrada no Capítulo 7. Neste último, também são descritas as publicações que foram fruto deste trabalho, bem como sugestões de trabalhos futuros.

[5]KV é dado pelas letras inicias dos termos em inglês *key* (chave) e *value* (valor) - livre trad.

Complementar ao trabalho, estão os apêndices contendo informações adicionais. O Apêndice A contém a documentação das funções fornecidas ao desenvolvedor e exemplos de utilização, enquanto o Apêndice B explica as métricas utilizadas na análise de resultados. Ao final, o Apêndice C apresenta os valores utilizados nos gráficos de resultados e os desvios padrão das amostras de tempo realizadas.

2 CONCEITOS E FERRAMENTAS

O trabalho desenvolvido nesta tese trata da apresentação e implementação de um modelo de programação para aplicações que manipulem grandes quantidades de dados. Como requisito para a explicação de tal modelo, faz-se necessária a apresentação prévia de conceitos e ferramentas relacionados principalmente à área de sistemas distribuídos e tecnologias envolvidas na área de *Big Data*, com sistemas de armazenamento e modelos de programação. Este capítulo trata de apresentar esses conceitos, que são base para o desenvolvimento do trabalho.

2.1 Sistemas paralelos e distribuídos

Um sistema distribuído é uma coleção de computadores independentes que se apresenta ao usuário como um sistema único e coerente (TANENBAUM; STEEN, 2007). Tais computadores são interligados por uma rede de comunicação e executam uma aplicação integrada, tal que as aplicações que a compõem operem de forma simultânea compondo um único sistema.

As máquinas que compõem um sistema distribuído podem ser das mais diversas, como computadores pessoais, *mainframes*, sensores, dispositivos móveis, dentre outros. O ponto principal no desenvolvimento de um sistema distribuído está em como é realizada a colaboração entre estes diversos componentes de processamento.

Algumas características possuem destaque nos sistemas distribuídos. A transparência consiste em não apresentar ao usuário aspectos como localização dos recursos

físicos, a forma como a comunicação dos componentes é realizada e qual é a arquitetura destes dispositivos. A escalabilidade trata da quantidade de dispositivos existentes, e sua relação com o sistema como um todo. Esta característica deriva da independência dos computadores entre si e em qual proporção a quantidade de máquinas influencia no desempenho da execução das aplicações distribuídas. Outra característica é a tolerância a falhas: em um sistema distribuído, existem diversos pontos em que defeitos podem ocorrer, mas o não funcionamento de um determinado componente não implica necessariamente em falhas na execução do sistema. Po exemplo caso durante um sistema distribuído em operação uma máquina falhe, outras podem assumir suas funções e continuar a execução da aplicação.

2.1.1 *Clusters* de computadores

Um *cluster* pode ser definido como um sistema de processamento distribuído, consistindo em um aglomerado de computadores interconectados operando como um recurso computacional unificado. Cada computador pertencente ao *cluster* é chamado de nó, que podem ser computadores pessoais com um ou mais núcleos de processamento. Um *cluster* geralmente refere-se a dois ou mais computadores interconectados, e podem estar organizados fisicamente juntos ou separados e interconectados utilizando uma rede local (LAN[1]) (BUYYA, 1999). A vantagem da utilização de *clusters* de computadores dá-se pela possibilidade de obtenção de recursos como alto desempenho, escalabilidade, alta taxa de transferência de dados e alta disponibilidade, alcançados com a utilização de elementos de baixo custo. Computadores organizados desta maneira obtiveram destaque na década passada, principalmente devido à aplicação da computação de alto desempenho na área científica.

[1] Acrônimo do inglês *Local Area Network* - livre trad.

2.1.2 MPI

A interface de troca de mensagens (MPI[2]) (MPI, 2015a) é uma interface de biblioteca com diversas funções de trocas de mensagens envolvendo sistemas distribuídos. Foi constituído em um fórum aberto, com a participação de pesquisadores, programadores, usuários e fabricantes, representando diversas organizações.

O MPI possui sua base no modelo de programação de troca de mensagens, que consiste em um conjunto de processadores com memória local, de maneira que cada processador possua acesso direto apenas aos dados e instruções armazenados em sua própria memória. Neste modelo, são utilizadas instruções de envio e recebimento de mensagens entre os processadores utilizando a rede de comunicação, possibilitando que dados armazenados nas memórias locais sejam acessados pelos demais processadores.

A utilização do MPI consiste na identificação do paralelismo pelo programador, de forma que a utilização das rotinas de comunicação seja explícita no código da aplicação. O modelo possui funções coletivas e também de ponto a ponto, síncronas e assíncronas, como `MPI_Send`, `MPI_Recv`, `MPI_Isend`, `MPI_Irecv`, `MPI_Bcast`, dentre outras (QUINN, 2004).

Existem diversas implementações do padrão MPI, sendo que MPICH (MPICH, 2015) e OpenMPI (OPENMPI, 2015) são bastante populares, e ambas disponíveis para as linguagens de programação C e Fortran. Atualmente, o padrão MPI encontra-se na versão 3.1 (MPI, 2015b)

Uma recente implementação do MPI, porém ainda em fase de testes, é a MPI-ULFM[3] (ULFM, 2015), que trata de prover um mecanismo de tolerância a falhas em aplicações que utilizam este modelo de troca de mensagens.

[2]Do inglês *Message Passing Interface* - livre trad.
[3]Acrônimo do termo em inglês *Message Passing Interface - User-Level Failure Mitigation*

2.1.3 Sistemas de arquivos distribuídos

Os sistemas de arquivos distribuídos desempenham um importante papel no contexto deste trabalho, pois além de proporcionar o armazenamento dos arquivos permitem que uma camada de abstração seja adicionada ao sistema, e com ela diversos cuidados sobre os dados deixam de fazer parte das aplicações, simplificando o desenvolvimento de sistemas que realizam a manipulação de informações utilizando arquivos como entrada e saída de dados.

A arquitetura envolvida no funcionamento de um sistema de arquivos distribuído pode variar de acordo com as características de cada sistema, podendo ser uma arquitetura cliente/servidor, arquitetura com base em *clusters* ou simétrica (TANENBAUM; STEEN, 2007). Como exemplos desses sistemas de arquivos distribuídos tem-se o NFS[4] (SHEPLER et al., 2003), GFS[5] (GHEMAWAT; GOBIOFF; LEUNG, 2003) e Ivy (MUTHITACHAROEN et al., 2002), HDFS[6] (SHVACHKO et al., 2010) entre outros.

2.1.3.1 HDFS

O sistema de arquivos distribuídos utilizado no Hadoop, o HDFS (SHVACHKO et al., 2010) é um sistema de arquivos desenvolvido para prover tolerância a falhas, alta escalabilidade e suporte a arquivos de grandes tamanhos. Possui diversas características herdadas do GFS, principalmente em relação à sua arquitetura. É implementado utilizando a linguagem de programação Java, e opera em uma camada superior à do sistema de arquivos local.

Sua arquitetura é dividida basicamente de acordo com os dois sistemas que o com-

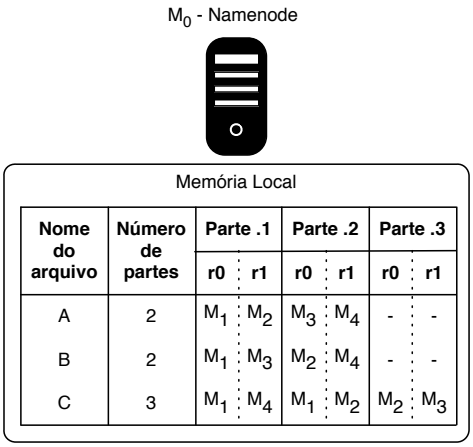
[4]Acrônimo do nome em inglês *Network File System* - Sistema de arquivos em rede - livre trad.
[5]Acrônimo do nome Sistema de Arquivos do Google em inglês, *Google File System* - livre trad.
[6]Acrônimo do termo em inglês *Hadoop Distributed File System*

põem, o Namenode[7][8] e o Datanode[9]. A máquina que executa o Namenode é responsável por armazenar a árvore de diretórios do sistema de arquivos e também a localização de onde os dados dos arquivos se encontram no sistema distribuído, enquanto que o Datanode cuida de armazenar os dados localmente em cada máquina.

Cada arquivo armazenado no HDFS é dividido em diversas partes de tamanho fixo[10], que são replicados e distribuídos para as máquinas do ambiente distribuído. O fator de replicação[11] r e o tamanho de cada parte do arquivo podem ser modificados nas configurações do sistema de arquivos.

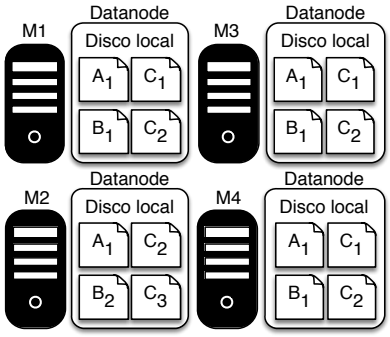
Figura 1: Exemplo de armazenamento de arquivos utilizando HDFS

(a) Namenode



Nome do arquivo	Número de partes	Parte .1		Parte .2		Parte .3	
		r0	r1	r0	r1	r0	r1
A	2	M_1	M_2	M_3	M_4	-	-
B	2	M_1	M_3	M_2	M_4	-	-
C	3	M_1	M_4	M_1	M_2	M_2	M_3

(b) Datanode



Fonte: próprio autor (2018)

A Figura 1 exemplifica uma situação em que três arquivos são armazenados no HDFS, utilizando um ambiente distribuído composto por cinco máquinas, nomeadas de M_0 a M_4. O Namenode possui em memória o controle dos arquivos e suas partes, e onde cada parte e réplica são armazenadas. No exemplo $r = 2$, ou seja, cada parte do arquivo possui uma réplica para prover a tolerância a falhas.

O HDFS possui uma interface de comunicação para aplicações que realizam leitura

[7]Termos utilizados na publicação e implementação originais do trabalho.
[8]Em inglês, *namenode* corresponde a máquina de nomes - livre trad.
[9]Em inglês, *datanode* corresponde a máquina de dados - livre trad.
[10]Na versão 1.0, o tamanho padrão para as partes dos arquivos é de $64MB$
[11]Na versão 1.0, o fator de replicação r possui como padrão o valor 2

e escrita de arquivos, utilizando uma interface de programação de aplicativos (API[12]) que deve ser utilizada pelas aplicações clientes.

Uma operação de leitura no HDFS ocorre em diversos passos: inicialmente, é contatado o Namenode, que responderá com a lista de partes e réplicas do arquivo a ser lido, e onde estes dados encontram-se armazenados pelo ambiente distribuído. Após isto, o cliente comunica-se diretamente com os servidores Datanode que possuem a parte do arquivo a ser lida. Este processo é transparente ao usuário, ficando a cargo do sistema de arquivos.

Quanto às operações de escrita, a API não comunica-se com o Namenode de maneira imediata. Inicialmente o cliente HDFS armazena temporariamente os dados em um arquivo local, e os dados são gravados neste arquivo. Assim que o arquivo temporário possuir um tamanho superior ao de um bloco de arquivo HDFS, o servidor Namenode é comunicado e responde com as localizações de onde cada bloco deve ser enviado. O cliente HDFS então envia os dados para as máquinas destino, e o Namenode registra o arquivo, seus blocos e as localizações em sua hierarquia de diretórios. Quanto à replicação de blocos, a máquina que recebe o bloco possui a informação de onde o bloco deve ser replicado, e, na medida em que recebe os dados já os repassa para a máquina onde a réplica será armazenada, que por sua vez repassa para a próxima máquina a armazenar a próxima réplica, seguindo uma estrutura de *pipeline*, até que o fator *r* definido seja satisfeito (BORTHAKUR, 2008, p. 11).

O projeto do HDFS não possibilita a atualização de dados já escritos no sistema de arquivos. Esta característica permite que diversas otimizações sejam aplicadas ao sistema, principalmente pela não necessidade de implementação de mecanismos de coerência de dados em aplicações que já realizaram leituras e atualização de valores em réplicas dos arquivos.

Além das funções comuns de acesso a dados de arquivos, a API do HDFS fornece

[12] Acrônimo do inglês *Application Programming Interface* - livre trad.

operações para acesso a metadados dos arquivos, com informações sobre como os dados estão armazenados no sistema distribuído, tais como número de réplicas, número de partes que cada arquivo está dividido e em qual máquina cada parte se encontra armazenada.

2.1.4 Desenvolvimento de aplicações paralelas

A utilização de processadores capazes de executar diversas linhas de execução em paralelo, como as tecnologias multinúcleo[13] e *hyperthreading* (BULPIN, 2004) por exemplo, possibilita que diversos componentes possam ser compartilhados (GEER, 2005) como memória, por exemplo. Desta forma, é possível que as aplicações sejam executadas de forma independente do processador ou núcleo de execução, visto que todos estes possuem acesso aos endereços de memória em que se encontra a aplicação.

A programação para os sistemas multiprocessadores e multinúcleos difere das aplicações tradicionais, e compete aos desenvolvedores de *software* a criação de aplicações que utilizem os recursos de multiprocessamento disponíveis. Para tal, é necessária a utilização de ferramentas que explorem o paralelismo fornecido pelo *hardware*. A utilização de processos e *threads* com esta finalidade já é amplamente difundida na computação, principalmente no desenvolvimento de sistemas operacionais (TANENBAUM; WOODHULL, 2000). Estes recursos também podem ser utilizados em sistemas monoprocessados, porém, neste caso, as diversas linhas de execução concorrem entre si pela CPU, que por sua vez alterna o processamento entre as diversas linhas de execução e o usuário possui a falsa impressão de paralelismo.

Um processo é um programa em execução, de forma que cada processo possui em memória o código da tarefa a ser executada e o conteúdo dos registradores relacionados ao processamento. Inclui também a pilha do processo, que contém dados temporários e uma seção de dados armazenando variáveis globais (SILBERSCHATZ; GALVIN;

[13] Do inglês *multicore* - livre trad.

GAGNE, 2004). As *threads* operam de forma semelhante, de forma que a utilização de diversas instâncias destas implica no compartilhamento da seção de código, de dados e outros recursos do sistema operacional em comum.

A construção de aplicações utilizando *threads* pode ser realizada com a utilização de APIs, com o objetivo de facilitar o desenvolvimento. APIs amplamente conhecidas são OpenMP (DAGUM; MENON, 1998) e Pthreads (NICHOLS; BUTTLAR; FARRELL, 1996).

2.2 *Big Data*

Atualmente, a noção de grande em quantidade de dados alcança a ordem de petabytes. Essa percepção vem sendo modificada no decorrer dos anos (BORKAR; CAREY; LI, 2012).

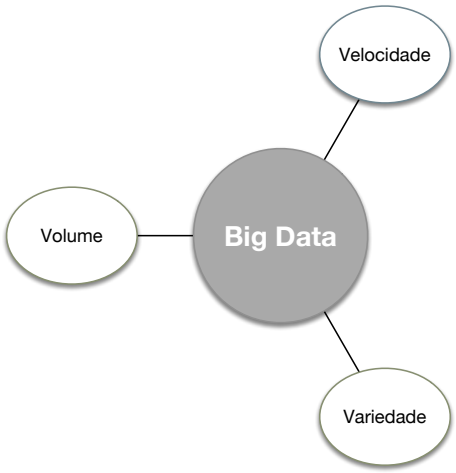
A diminuição dos custos de dispositivos de armazenamento em conjunto com o avanço de tecnologias que possibilitaram a geração de grandes volumes de informações permitiu que todos os dados resultantes de sistemas, experimentos, análises ou registros pudessem ser armazenados. Dados podem ser salvos em estado bruto ou de forma não estruturada, deixando o processamento dessas informações para um tempo posterior.

A quantidade de dados gerados nessas situações pode atingir a casa dos terabytes (TB[14]), petabytes (PB[15]) ou até mesmo exabytes, em casos de grandes empresas ou setores de governo.

Analisar esse conjunto de dados com a finalidade de extrair novas informações tornou-se um grande desafio, que extrapola diversos limites tecnológicos da atualidade, como capacidade de processamento, tráfego de rede, memória, tempo de acesso a disco, tolerância a falhas, escalabilidade, modelos de programação, entre outros.

[14] 1 TB equivale a 2^{40} bytes
[15] 1 PB equivale a 2^{50} bytes

Figura 2: Pilares do *Big Data*



Fonte: Próprio autor (2018)

Desta forma, operar com tecnologias de *Big Data* não significa apenas manipular dados a partir de um certo número de gigabytes (GB[16]) ou terabytes, mas sim a ideia de operar além dos limites atuais das tecnologias de computação (MGI, 2011; KAISLER et al., 2013).

Diversos trabalhos classificam os desafios a serem enfrentados em três grandes categorias (IBM; ZIKOPOULOS; EATON, 2011), apresentadas na Figura 2. Como analogia para esses desafios, considere uma aplicação que consiste em apresentar as condições climáticas para um tempo futuro. Tais desafios são:

- variedade: os dados envolvidos no *Big Data* podem possuir diversas origens e também estar em diversos formatos, como imagens, informações geográficas, dados de simulações científicas, áudio, texto, entre outros. Além destes, devem ser considerados também dados não estruturados. Como exemplo, para as condições climáticas podem ser considerados o histórico de chuvas e secas, níveis de temperatura e umidade nos últimos dias, direção dos ventos e também informações sobre as correntes marítimas em diversos pontos do oceano;

[16] 1 GB equivale a 2^{30} bytes

- volume: a grande quantidade de dados a serem processados, que podem chegar à casa dos petabytes ou exabytes. Dados nesta magnitude extrapolam as tecnologias de memória atualmente disponíveis. Para tal, diferentes mecanismos devem ser explorados para o armazenamento e acesso a esses dados. Para a previsão do tempo a quantidade de informações referente às correntes marítimas ou então o histórico de chuvas e níveis de temperatura podem conter bilhões de pontos a servir como entrada para a aplicação. O armazenamento e acesso a tais dados requer uma estrutura distribuída e tolerante a falhas para o armazenamento, com suporte a integridade e disponibilidade.

- velocidade: tratar a questão de *Big Data* com velocidade significa obter respostas das aplicações em um tempo hábil, antes que a informação perca o valor ou sentido da existência do seu resultado. No exemplo, considere a previsão do tempo para o dia de amanhã: caso o processamento necessário para prever as condições climáticas demore mais que um dia, o resultado torna-se insignificante, pois ao término do processamento não será mais uma previsão.

Dada a consolidação da área de computação de alto desempenho (HPC[17]), as tecnologias e estratégias desenvolvidas nesta área podem ser utilizadas para o processamento de aplicações de *Big Data*, considerando características, tais como, escalabilidade, tolerância a falhas e desempenho de entrada e saída.

2.2.1 O modelo Mapreduce

O Mapreduce (DEAN; GHEMAWAT, 2008) é um modelo de programação utilizado para realizar o processamento de grandes quantidades de dados, que obteve grande destaque nas áreas comercial e científica devido às suas características de simplicidade de programação e suporte a tolerância a falhas.

[17] Acrônimo do termo em inglês *High Performance Computing* - livre trad.

Seu funcionamento consiste na utilização de chaves e valores para o processamento das informações, com o emprego de duas funções definidas pelo usuário, *map* e *reduce*. Diversas instâncias dessas duas funções são executadas simultaneamente no ambiente distribuído, não havendo comunicação de dados entre elas.

A execução do Mapreduce ocorre em diversas fases (Figura 3), sendo que cada fase possui uma tarefa específica, tendo seus dados de saída servindo como dados de entrada para a próxima fase. As principais fases são Map, Shuffle e Reduce, as quais são apresentadas a seguir.

A fase de Map consiste em receber os dados da entrada, comumente proveniente de um sistema de arquivos. Nesta fase é executada a função *map* definida pelo usuário, tendo como saída pares de informações no formato (k, v), tal que k é uma chave e v um valor associado a essa chave. Esses pares serão utilizados para compor a entrada de dados para a fase de Reduce. Assim, tem-se:

$$map(entrada) \rightarrow lista(k, v)$$

A entrada da fase de Reduce é dada no formato (k, V), tal que V é o conjunto de todos os valores v associados à mesma chave k. Estes dados servem como entrada para a função *reduce* definida pelo usuário, tendo como saída r o resultado do processamento dos valores em V. Desta forma, tem-se:

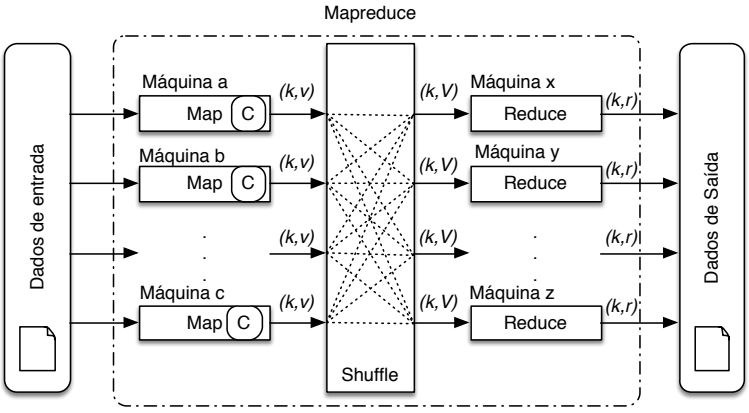
$$reduce(k, V) \rightarrow lista(k, r)$$

É necessário que exista a troca de dados no ambiente distribuído, de forma a estruturar as saídas da fase de Map como entradas para a fase de Reduce. Esta fase é chamada de Shuffle (Figura 3), e além do envio de informações é responsável também por formar o conjunto V, parte de (k, V), entrada para o Reduce. A definição de qual

reduce receberá quais pares (k, V) tem como base uma função de espalhamento (*hash*) aplicada na chave k. Com base no valor deste *hash*, os dados são distribuídos entre as diversas instâncias do reduce. Todo o ambiente distribuído aplica a mesma função de *hash*, tornando possível que o mesmo reduce receba todos os valores v relacionados a uma dada chave k, independente do local no ambiente distribuído em que estes tenham sido processados.

Outra fase que pode ser utilizada é a de Combine. Esta é aplicada ao final da execução de cada map, com a finalidade de pré-processar informações da fase de Reduce ou então agrupar resultados para diminuir a quantidade de informações trafegadas pela rede. Esta fase consiste na utilização de uma função *combine*, que em diversas situações pode ser a mesma função aplicada na fase de Reduce.

Figura 3: Execução do modelo de programação Mapreduce



Fonte: Próprio autor (2018)

Um exemplo do funcionamento do Mapreduce consiste em uma aplicação que realiza a contagem da quantidade de ocorrências de cada palavra, dado um texto de entrada. Com isto, para cada palavra da entrada do *map* é gerado um par (k, v), com k sendo a palavra em si, e v o registro da ocorrência da palavra, neste caso, 1. Após a fase de Shuffle, a entrada (k, V) do Reduce seria composta pela palavra k e V o conjunto de ocorrências, por exemplo para uma dada palavra $k = abacaxi$, com 4 ocorrências em todo o conjunto de entrada da aplicação, o conjunto V seria dado por $V = \{1, 1, 1, 1\}$.

Com isto, a computação da função *reduce* consistiria em somar todos os valores em V, gerando uma saída $(abacaxi, 4)$. O pseudocódigo das funções *map* e *reduce* pode ser visto nos Algoritmos 1 e 2, respectivamente.

```
1: função map( texto frase )
2: {
3:     vetor palavras = separar(frase, ' ' );
4:     para cada posição p em palavras
5:     {
6:         emitir( p, 1 );
7:     }
8: }
```

Algoritmo 1: Pseudocódigo da função *map*

```
1: função reduce( chave k, vetor V )
2: {
3:     inteiro soma = 0;
4:     para cada posição v em V
5:     {
6:         soma = soma + v;
7:     }
8:     emitir( k , soma );
9: }
```

Algoritmo 2: Pseudocódigo da função *reduce*

A execução de aplicações neste modelo permite que diversas vantagens sejam exploradas. Entre elas está a possibilidade de levar a aplicação até os dados, diferente do que comumente ocorre em aplicações tradicionais. Considerando que o sistema de execução Mapreduce esteja operando nas mesmas máquinas em que existe um sistema de arquivos distribuído, cada operação de Map pode executar diretamente na máquina onde os dados se encontram, permitindo que estes sejam lidos localmente, minimizando o tráfego de dados no meio de comunicação do ambiente distribuído. A não dependência de informações entre as diversas instâncias de map ou reduce possibilita que o paralelismo e a execução no ambiente distribuído seja automatizado, simplificando o desenvolvimento de aplicações.

Existem diversas limitações para a execução do Mapreduce. O modelo padrão consiste basicamente na execução de apenas um passo (leitura da entrada, Map, Shuffle,

Reduce e escrita do resultado de saída, nesta ordem), e caso seja necessário executar diversas iterações sobre passos, será preciso armazenar em disco a saída da execução atual e realizar uma próxima execução possuindo como entrada a saída armazenada na execução anterior. Outra limitação é a adaptabilidade de algoritmos já conhecidos: diversas aplicações possuem rotinas com base em convergência ou repetições, tendo como critério de parada informações globais à aplicação. A característica de descentralização do Mapreduce dificulta a execução deste tipo de aplicação.

3 REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta diversos trabalhos da literatura relacionados com a proposta apresentada. Estes envolvem extensões para o modelo Mapreduce ou outros modelos de execução derivados desse, suporte a processamento *online*, processamento iterativo e suporte a MPI.

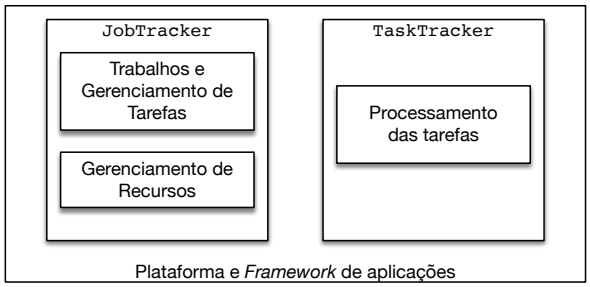
Diversas ferramentas implementam o modelo de programação Mapreduce, de forma que algumas seguem o modelo padrão e outras apresentam variações com base na versão original.

3.1 Hadoop

O Apache Hadoop (APACHE, 2014) está entre os *frameworks* de processamento em grandes quantidades de dados mais populares da atualidade. Consiste em uma plataforma de *software* desenvolvida utilizando a linguagem de programação Java (ORACLE, 2018), que aplica computação distribuída ao processamento de grandes quantidades de dados. Entre suas características estão o suporte a tolerância a falhas e a alta escalabilidade, podendo ser executado em *clusters* com milhares de máquinas (SHVACHKO, 2011) e ambientes de computação em nuvem (LU; HAI-SHAN; TING-TING, 2012).

Hadoop também aplica balanceamento de carga, utilizando uma técnica de compartimentos. Cada máquina que opera o sistema possui um número definido de compartimentos, onde cada um deles executa uma tarefa. Ao finalizar a tarefa, o compar-

Figura 4: Arquitetura do *framework* Hadoop versão 1



Fonte: Traduzido e adaptado de (MURTHY et al., 2014, p. 116, Fig. 7.1)

timento torna-se livre e uma nova tarefa é alocada. Desta forma, a carga de trabalho é dividida entre as máquinas na medida em que o processamento das tarefas em cada compartimento é finalizado, e máquinas com maior poder de processamento recebem mais tarefas para serem processadas. Caso o sistema possua compartimentos livres e identifique que determinada tarefa tarda em finalizar, esta pode ser alocada novamente em outro compartimento, e assim que a tarefa for finalizada em alguma das máquinas, a outra tem seu resultado descartado.

Ao processar cada tarefa, os dados resultantes são gravados em disco, sendo esta uma das principais influências para a degradação de desempenho das aplicações que utilizam este *framework*.

A arquitetura do Hadoop segue o modelo mestre-escravo, e difere nas versões 1 e 2, como mostrado nas Figuras 4 e 5. Na versão 2 é incluído o YARN[1], possibilitando o gerenciamento de recursos no *cluster*. Na versão 1, aspectos referentes à execução da plataforma e o *framework* de execução das aplicações são integrados (Figura 4), enquanto que na versão 2 estes elementos são tratados separadamente (Figura 5).

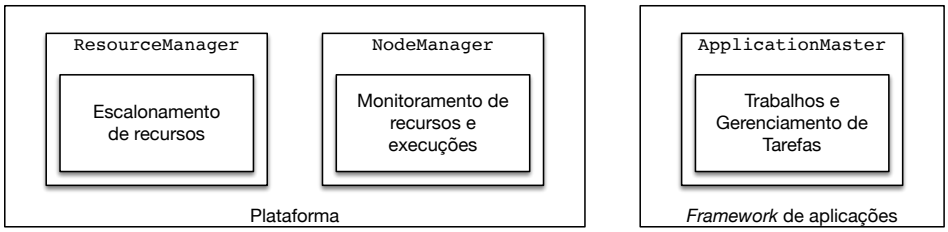
Para a execução das aplicações na versão 1, o processo `JobTracker`[2] comunica as máquinas que devem realizar o processamento. Já na versão 2, o gerenciador de recursos inicializa o processo `ApplicationMaster`[3], responsável por inicializar con-

[1] Acrônimo dos termos em língua inglesa *Yet Another Resource Negociator*, ou "mais um gerenciador de recursos- livre trad.

[2] "Rastreador de trabalhos", em inglês - livre trad.

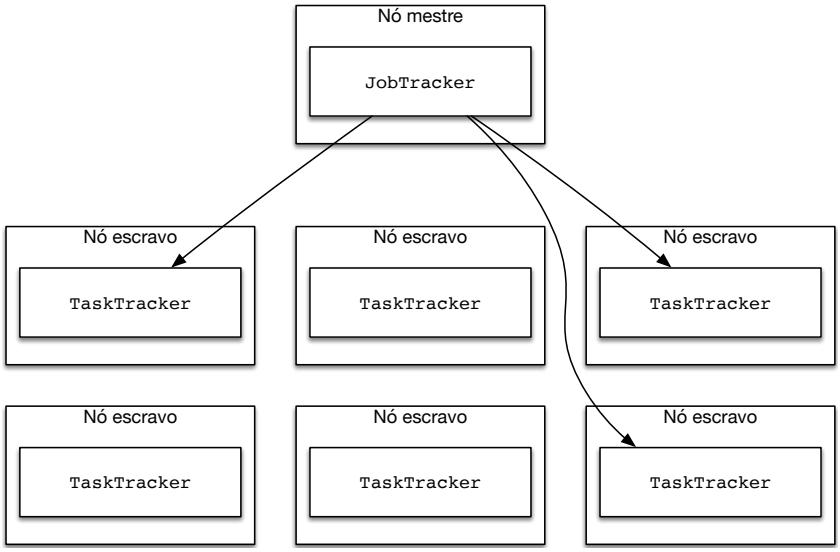
[3] "Aplicação Mestre", em inglês - livre trad.

Figura 5: Arquitetura do *framework* Hadoop versão 2



Fonte: Traduzido e adaptado de (MURTHY et al., 2014, p. 116, Fig. 7.1)

Figura 6: Processos Hadoop versão 1

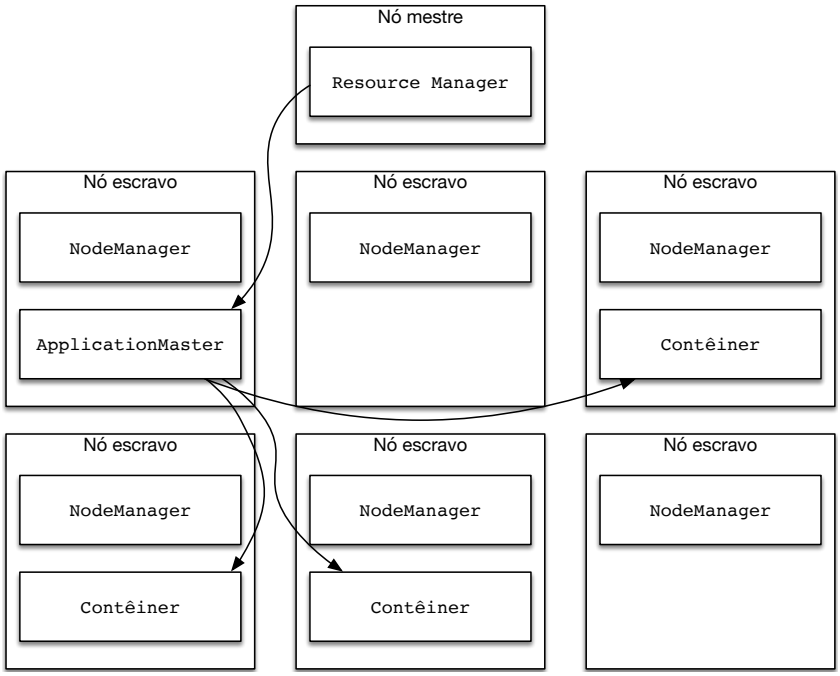


Fonte: Próprio autor (2018)

têineres, que por sua vez irão realizar o processamento das tarefas. A escolha das máquinas que terão tais processos alocados é feita pelo gerenciador de recursos, de acordo com os recursos disponíveis em cada máquina e sua carga de processamento. As Figuras 6 e 7 ilustram um exemplo de aplicação sendo alocada e executada em um *cluster* com Hadoop nas versões 1 e 2, respectivamente. Nestas, a aplicação é disparada no nó mestre, que distribuem as tarefas para os nós escravos. Comparando tais Figuras, é possível observar as diferenças no modo de execução entre as diferentes versões.

Quanto ao sistemas de arquivos utilizado, o HDFS, ambas as versões operam da

Figura 7: Processos Hadoop versão 2



Fonte: Traduzido e adaptado de (MURTHY et al., 2014, p. 116, Fig. 7.1)

mesma forma, com os processos `Namenode`[4] e `Datanode`[5] operando nas máquinas.

Diversas ferramentas utilizam o Hadoop como uma camada base para execução de tarefas ou então interagem com suas execuções, compondo o chamado Ecossistema Hadoop[6]. Entre elas temos:

Hive: (APACHE, 2014; THUSOO et al., 2009): é uma ferramenta que utiliza conceitos de Mapreduce e HDFS, de maneira que seus comandos sejam semelhantes aos utilizados em operações SQL (ELMASRI; NAVATHE, 2005), utilizando uma linguagem de programação denominada HiveQL (THUSOO et al., 2009). É uma plataforma que opera sobre o sistema Hadoop, com objetivo de receber funções HiveQL (com sintaxe semelhante à linguagem de consulta SQL) e convertê-las em operações Mapreduce. Possui um sistema próprio, responsável por organizar e disparar diversas tarefas junto ao Hadoop;

[4]Processo responsável por manter a estrutura de diretórios e localização dos arquivos e seus blocos.
[5]Processo responsável por ler e escrever os dados no disco local.
[6]Do inglês *Hadoop Ecosystem* - livre trad.

Pig: (APACHE, 2015b): esta ferramenta é executada em uma camada de abstração superior ao Hadoop, com o intuito de processar operações utilizando uma linguagem semelhante à SQL. Suas aplicações são desenvolvidas utilizando a linguagem interpretada PigLatin, com funções que realizam a distribuição e paralelização de tarefas em um ambiente de execução distribuído. Pig, assim como o Hive, está relacionada a este trabalho devido a utilizar uma camada inferior para realizar o processamento de suas tarefas;

Tez: (APACHE, 2018; SAHA et al., 2015): permite várias tarefas compondo um grafo acíclico direcionado e executá-las de uma única vez, possibilitando também que o fluxo de dados dentro do grafo seja alterado dinamicamente, no decorrer da execução das aplicações. Sua relação com este trabalho dá-se pela possibilidade de integração de diversas tarefas e com isso compôr uma aplicação mais complexa, sendo executada de uma única vez. Tez permite ainda execuções iterativas ou em *batch*.

Atualmente, o Hadoop encontra-se em sua versão 2, com suporte a gerenciamento de recursos em *clusters*, chamada de YARN[7] (MURTHY et al., 2014). Este utiliza o conceito de contêineres para a alocação de recursos para a execução de diversas tarefas.

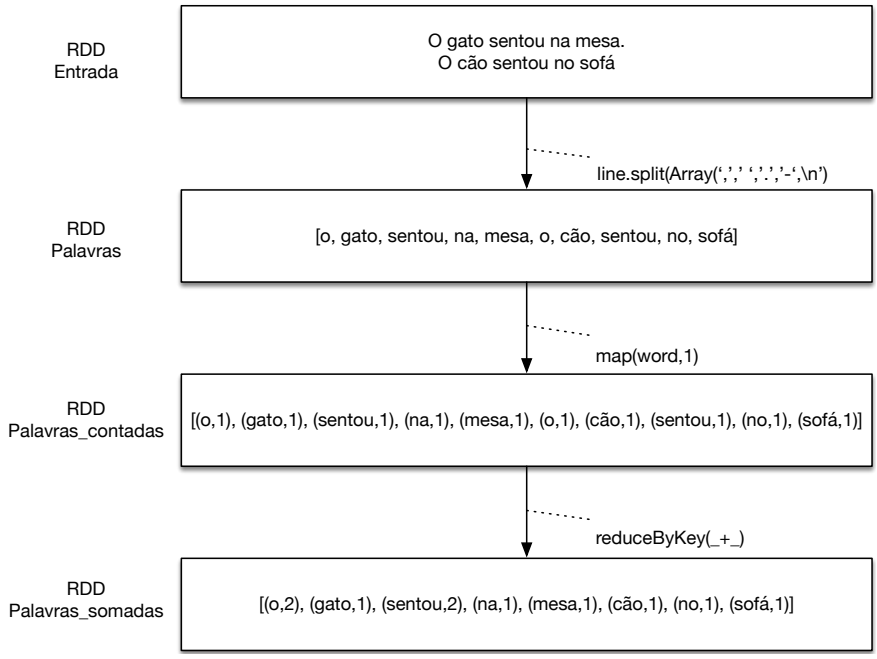
3.2 Spark

O Apache Spark (APACHE, 2015a; KARAU et al., 2015) é uma ferramenta desenvolvida para a execução de aplicações com grandes quantidades de dados, que utiliza um mecanismo de processamento em memória evitando que os dados sejam gravados em disco ao final de cada iteração.

É dividido em diversos componentes como o Spark Core e os conjuntos de dados resilientes distribuídos (RDD[8]) (ZAHARIA et al., 2012). O Spark Core é responsável

[7] Acrônimo do inglês *Yet Another Resource Negotiator*
[8] Acrônimo do termo em inglês *Resilient Distributed Dataset* - livre trad.

Figura 8: Exemplo de RDD realizando contagem de palavras



Fonte: Próprio Autor (2018)

por distribuir e coordenar as tarefas, e os RDD's são estruturas que possibilitam a reutilização dos dados nela armazenados. Seu funcionamento é tolerante a falhas, e também permite que dados persistam durante o funcionamento das aplicações.

As estruturas RDD operam de maneira distribuída e são imutáveis. Sobre elas são aplicadas operações chamadas ações[9] ou transformações[10], resultando em novos RDD's. Em situações de falha, a cadeia de RDD's pode ser reconstruída a partir do último RDD disponível, aplicando as mesmas operações. Operações de transformação podem possuir diversos RDD's como entrada ou saída, além de poder realizar comunicação de dados entre diversas máquinas, visto que RDD's podem possuir instâncias em diferentes máquinas de um ambiente distribuído. A Figura 8 ilustra um exemplo de aplicação em que dados são lidos de um texto e a operação de contagem de palavras é realizada. Observe que em tal Figura os vértices representam RDD's e as arestas, transformações (CHAMBERS; ZAHARIA, 2008).

[9]Do inglês *action* - livre trad.
[10]Do inglês *transformation* - livre trad.

Spark possui ainda, em uma camada de abstração superior, API e bibliotecas para suporte a operações utilizando diversas linguagens de programação, como Java, Scala (ÉCOLE POLYTECHNIQUE FÉDÉRALE, 2018) (utilizada no exemplo da Figura 8), Python (PYTHON SOFTWARE FOUNDATION, 2018) e R (THE R FOUNDATION, 2018).

Spark está relacionado com este trabalho devido à sua versatilidade e organização de suas tarefas, com operações que podem ser aplicadas aos RDDs. Nele, os RDD's são executados de maneira coordenada, semelhante à execução em blocos proposta neste trabalho. Porém, os RDDs atuam de maneira bloqueante, em que o próximo RDD não pode ser executado antes que o atual seja finalizado, e em contraste, neste trabalho é proposta a estratégia de processamento especulativo, visando a diminuição do tempo de execução das aplicações.

3.3 Outros trabalhos

O trabalho de Rehmann e Schoettner (2011) apresenta um modelo de execução utilizando memória transacional distribuída (DTM[11]), de modo a prover consistência de dados entre diversas máquinas. Utiliza o modelo de programação Mapreduce, e procede mantendo todos os resultados intermediários de suas operações em memória. Sua relação com a proposta desenvolvida nesta tese dá-se pela estratégia de armazenamento de dados temporários do processamento das tarefas, como será utilizado no mecanismo de processamento especulativo.

Li et al. (2011) discute quais aspectos em relação à arquitetura serão necessários ao Mapreduce para dar suporte à execução de algoritmos de análises de dados realizando um passo, considerando duas propostas diferentes no modelo tradicional de *hash* utilizados na ferramenta Hadoop. Apresenta o MR-Hash, que consiste em diversas funções de *hash* executadas de maneira encadeada tal que o primeiro nível define a distribuição

[11]Do inglês *Distributed Transactional Memory* - livre trad.

do processamento com base na chave k e os demais níveis distribuem recursivamente esses dados entre diversas operações de reduce, e também realizam armazenamento temporário em disco de pares (k, v). Apresenta também o método INC-Hash, composto por funções de inicialização, combinação e finalização, de forma a possibilitar o processamento de dados de maneira incremental. A relação do trabalho citado com esta tese dá-se pelo reconhecimento se a informação presente no par (k, v) já foi processada, armazenando-a em uma estrutura de dados em memória específica, como ocorre com a estratégia de processamento especulativo.

Uma otimização no Mapreduce é realizada diminuindo as dependências entre as diversas fases é apresentada no trabalho de Heintz, Chandra e Weissman (2014). Para tal, considera diferentes custos de comunicação na fase de Shuffle para máquinas que executarão a operação de Reduce e também leva em conta diferentes quantidades de dados processados nas operações de Map, analisando a carga de processamento de cada nó. Em sua estratégia, considera para a escolha de quais máquinas realizarão as operações de Map, o custo de envio do dado na máquina onde a informação está armazenada até a máquina onde será executada a operação de Map e também avalia o custo de comunicação para a fase de Shuffle. Sua conexão com o trabalho proposto dá-se pela utilização de estratégias de otimização na troca de dados, como utilizado com o MPI na implementação deste trabalho.

Seguindo a linha de suporte à computação incremental, a ferramenta Incoop (BHATOTIA et al., 2011) apresenta modificações no Hadoop para prover suporte ao processamento de dados incluídos no sistema em tempo de execução. Adiciona uma fase de contração[12] para controlar a granularidade das tuplas enviadas na fase de Shuffle. Otimiza também o processamento de funções armazenando resultados e detectando entradas já computadas, diminuindo a quantidade de operações com a utilização de um servidor externo à aplicação, chamado de *memoization server*. Sua relação com este

[12]Do inglês *contraction* - livre trad.

trabalho dá-se principalmente pela técnica de armazenamento de dados intermediários às operações, assim como ocorre com a estratégia de processamento especulativo.

Com o objetivo de permitir iterações na plataforma Mapreduce, iHadoop (ELNIKETY; ELSAYED; RAMADAN, 2011) apresenta modificações no Hadoop possibilitando que operações Map e Reduce sejam encadeadas, seguindo o modelo produtor consumidor, de forma assíncrona. Permite que iterações sejam realizadas de forma assíncrona e especulativa, dado que uma iteração pode iniciar antes mesmo da anterior ter finalizado seu processamento. O trabalho também realiza uma análise das dependências, classificando-as em dependências de execução, dados e controle nas diversas fases do modelo Mapreduce, de forma a possibilitar a otimização na execução de tarefas. Tem relação com o presente trabalho pelo uso de estratégias que permitem a utilização de iterações em tarefas de map e reduce, como também ocorre na execução utilizando múltiplas entradas e saídas e no armazenamento temporário em disco, que possibilitam a execução de ciclos em aplicações.

Considerando a entrada contínua de dados, a ferramenta M^3, descrita no trabalho de Aly et al. (2012), aplica o suporte a esse tipo de entrada implementando operações Map e Reduce que permanecem em memória aguardando dados e mantendo seu estado de execução. Além disso, aplica otimizações que possibilitam que todo o fluxo de dados seja mantido em memória, sem utilizar acesso a disco em operações intermediárias durante o fluxo de execução da aplicação. Utiliza um esquema de modificação de dados, considerando funções de adição, remoção e atualização de entradas de dados, tal que estas funções são aplicadas de maneira encadeada nas diversas fases de execução do modelo Mapreduce. Sua relação com o trabalho nesta tese dá-se pela utilização de mecanismos de propagação de atualização de pares (k, v) nos blocos de execução aplicado na proposta de processamento especulativo.

O suporte a computação de entrada contínua de dados é tratado no Mapreduce Online (CONDIE, 2010), permitindo que operações de Reduce sejam executadas assim

que as saídas das operações de Map sejam computadas, seguindo um modelo de execução encadeada. Porém, este possui limitações nas execuções, ocorrendo situações em que o processamento não pode ser adiantado. Utiliza um mecanismo que armazena em disco todos os dados intermediários relacionados ao processamento, permitindo que uma prévia da computação final seja calculada. Com esta funcionalidade, o trabalho indica que os resultados apresentados como saída não necessariamente refletem a computação final da aplicação, mas sim o progresso da computação em um determinado instante. Um dos apontamentos negativos desta técnica trata-se de que, caso os dados computados necessitem de atualização devido a novas entradas, todo o conjunto deverá ser recalculado. Sua relação com o trabalho desta tese dá-se pela utilização de mecanismos de armazenamento de dados intermediários ao processamento das informações, como ocorre na proposta da estratégia de processamento especulativo e armazenamento de dados intermediário em disco.

Desenvolvido para processamento em máquinas *multicore*, o trabalho de Chen, Chen e Zang (2010) explora os diversos núcleos de processamento presentes na máquina local. Apresenta uma implementação do Mapreduce que consiste em processar localmente partes menores das entradas de maneira iterativa, utilizando a memória compartilhada entre os diversos núcleos de processamento. A implementação possui o nome de Ostrich, e tem sua base em uma outra implementação de Mapreduce para *multicore*, chamada Phoenix (RANGER et al., 2007). Sua relação com este trabalho dá-se pela utilização de paralelismo na execução de tarefas, explorando os diversos núcleos de processamento. De forma específica, é proposto que diversas *threads* executem as funções das tarefas map e reduce propostas pelo usuário utilizando diversas linhas de execução.

A ferramenta Percolator apresentada no trabalho de Peng, Dabek e Inc (2010) realiza o processamento de entradas de forma incremental, visto que foi desenvolvida especialmente para essa funcionalidade, tendo sua utilização principal em buscas em

sítios da Internet.

A otimização aplicada na ferramenta MATRIX (WANG; RAJENDRAN; RAICU, 2013), apresentada no trabalho de Wang et al. (2014) utiliza estratégias de balanceamento de carga considerando a localidade dos dados, no cenário de processamento de grandes quantidades de dados. Sua relação com este trabalho dá-se por considerar a localização dos dados no processamento de informações.

3.4 Considerações sobre os trabalhos citados

Após o levantamento bibliográfico, verificou-se que existem diversas lacunas envolvendo o modelo de programação Mapreduce. Tais lacunas abordam questões envolvendo melhora de desempenho e eliminação de bloqueios durante a execução de aplicações. Outro ponto observado também foi a possibilidade de execução de tarefas Mapreduce seguindo um fluxo de execução com a presença de ciclos.

Outra funcionalidade também ausente é a possibilidade de execução de diferentes operações simultâneas com múltiplas entradas e saídas, em operações como Map ou Reduce. A ferramenta Spark permite esse tipo de execução, porém possui a limitação de que o início da próxima operação deve aguardar até que todo o processamento das anteriores tenha sido finalizado.

4 FERRAMENTA DE PROGRAMAÇÃO E PROCESSAMENTO DE GRANDES QUANTIDADES DE DADOS EM AMBIENTE DISTRIBUÍDO

Este trabalho apresenta um modelo de programação e um ambiente para execução de tarefas com suporte a manipulação de grandes entradas de dados, tal que o ambiente seja distribuído e compatível com sistemas de arquivos já existentes.

O modelo de execução proposto permite que as aplicações sejam desenvolvidas em blocos, tendo sua base no modelo de programação Mapreduce. Desta forma, permite-se que as tarefas sejam executadas seguindo um fluxo de execução composto por esses blocos.

Os recursos de detecção e recuperação de falhas possuem grande relevância no cenário que envolve a manipulação de grandes quantidades de dados em ambientes distribuídos. Porém, visto que a tese apresentada neste trabalho é voltada para a programação, ambiente de execução e aumento de desempenho, questões envolvendo tolerância a falhas não são abordadas neste trabalho.

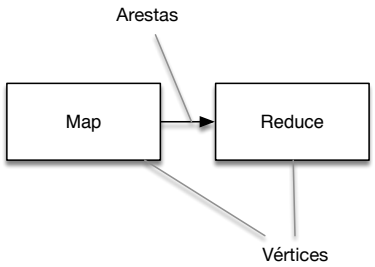
De forma mais específica esta tese têm suas contribuições implementadas em execução em blocos, suporte a múltiplas entradas e saídas, processamento especulativo, armazenamento em disco de dados intermediários e execução iterativa. Um dos pontos chave do trabalho é manter por mais tempo os dados em memória, diminuindo o número de acessos a disco, e com isso melhorando o desempenho das aplicações. Para situações em que não seja possível manter todos os dados em memória, pode-se apli-

car o armazenamento intermediário em disco, no qual todos os dados existentes em um bloco são gravados em disco e posteriormente carregados em memória.

4.1 Modelo de programação

A execução consiste no processamento paralelo e distribuído de tarefas definidas pelo usuário. Tais tarefas são organizadas em blocos, que em seu conjunto definem o fluxo de execução da aplicação, representado por um grafo direcionado (DG), de forma que os vértices do grafo denotam blocos de execução, e as arestas direcionadas definem a ordem em que os blocos serão executados. A Figura 9 demonstra um exemplo de DG, em que uma operação de Map é executada, seguida de uma operação de Reduce.

Figura 9: Modelo de programação em blocos



Fonte: Próprio autor (2018)

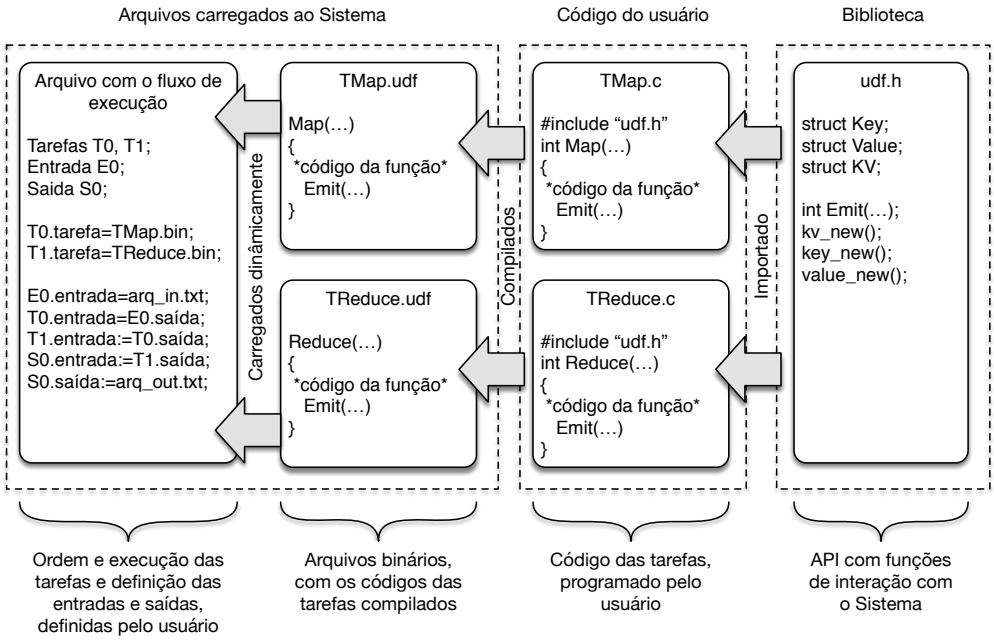
O modelo apresentado possui sua base no modelo de programação Mapreduce, porém inclui características distintas, com o propósito de otimizar a execução de aplicações e transpor limitações existentes no modelo Mapreduce.

Neste trabalho o ambiente de execução, responsável de executar as tarefas programadas pelo usuário de acordo com o modelo de execução proposto, é definido como Sistema.

A Figura 10 descreve como é realizada a programação de aplicações, para um exemplo seguindo o DG apresentado na Figura 9. À direita da Figura 10 é visualizada a biblioteca fornecida ao usuário, com as funções para a manipulação de dados e interações com o ambiente de execução. A programação das tarefas map e reduce

é realizada pelo usuário, de forma que cada arquivo possua o código fonte referente a uma tarefa, que posteriormente serão compilados com a ligação com a biblioteca, gerando um arquivo binário a ser carregado em memória para cada tarefa. Na área pontilhada posicionada à esquerda da Figura 10, podem ser vistos os arquivos binários com os códigos das tarefas compiladas e também o arquivo com a definição de como será dado o fluxo de execução da aplicação, de como as entradas e saídas serão interligadas e a ordem de execução das tarefas. A programação da aplicação é feita de modo que o código a ser executado em cada tarefa seja separado da descrição do fluxo a ser seguido pelas tarefas durante a execução.

Figura 10: Arquivos de configuração e programação das tarefas pelo usuário



Fonte: Próprio autor (2018)

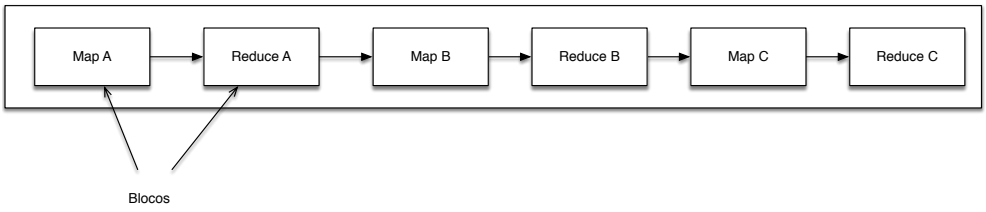
4.2 Execução em blocos

Uma das limitações do modelo Mapreduce é que aplicações que necessitem aplicar diversas vezes o modelo devem ser executadas em sequência utilizando várias submissões ao sistema, sendo que a saída de uma submissão é gravada em disco, e este mesmo arquivo é utilizado como entrada para a próxima etapa.

Utilizando o modelo de execução em blocos, o desenvolvedor da aplicação poderá incorporar funções a serem executadas em cada bloco e definir um fluxo de execuções a ser seguido. Com isto será possível desenvolver uma única aplicação que seja equivalente a diversas tarefas Mapreduce, e executá-las utilizando uma única submissão da aplicação ao sistema.

A Figura 11 ilustra a execução em blocos. Nesta, diversas tarefas map e reduce podem ser executadas de forma encadeada seguindo um modelo produtor consumidor, de forma que a saída de um bloco seja utilizada como entrada do próximo, assim a memória local pode ser melhor utilizada, minimizando o número de acessos ao disco.

Figura 11: Fluxo de execução em blocos



Fonte: Próprio autor (2018)

Utilizando estas estruturas disponíveis, o usuário poderá combiná-las para o desenvolvimento de aplicações, ou então desenvolver mecanismos de mais alto nível, operando como um *framework* em uma camada superior ao Sistema, utilizando blocos e fluxos de execução previamente desenvolvidos e utilizando-os para finalidades específicas. Para tal, pode-se incorporar uma nova camada, que converta determinado código de entrada para os arquivos utilizados como entrada para o Sistema.

4.2.1 Funções definidas pelo usuário

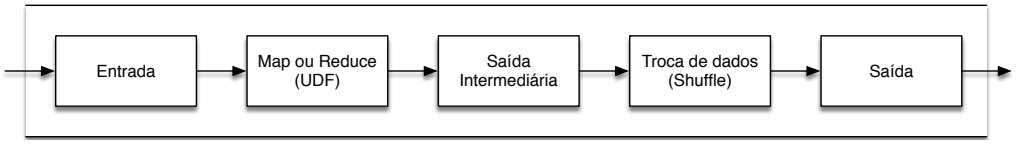
Na composição do fluxo de execução, cada bloco executa também uma função definida pelo usuário (UDF). Tais funções são carregadas em tempo de execução no Sistema, e cada bloco é responsável por carregar e executar suas respectivas funções. A inclusão desta funcionalidade possibilita que o código do usuário seja separado do có-

digo do sistema, não necessitando que o código do usuário seja adicionado ao Sistema, para então ser recompilado ou ligado a cada execução da aplicação.

4.2.2 Blocos de execução

Os blocos de execução consistem em unidades que definem tarefas a serem realizadas no sistema. O uso destes blocos possibilita que a execução dessas tarefas ocorra de forma distribuída pelo ambiente computacional e transparente ao usuário. Cada bloco de execução possui uma entrada de dados, uma operação a ser realizada, uma saída intermediária, trocas de dados e uma saída. Estes elementos são executados de forma ordenada e encadeada, como indicado na Figura 12.

Figura 12: Bloco de execução

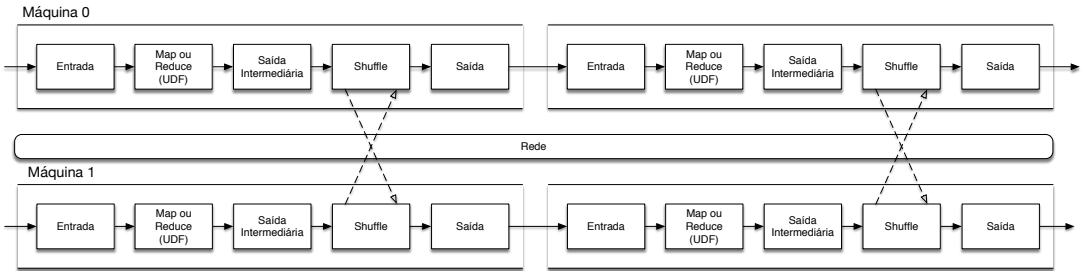


Fonte: Próprio autor (2018)

Durante a inicialização do sistema, cada máquina do ambiente possui instâncias de todos os blocos do fluxo de execução, de forma que estes inicialmente encontram-se em estado de espera, até que dados sejam incluídos em suas estruturas de entrada para então serem processados e repassados à estrutura de saída. As execuções se iniciam com os blocos de entrada, lendo dados de arquivos e repassando para as estruturas de entrada dos blocos seguintes.

A Figura 13 ilustra dois blocos de execução em um ambiente distribuído, composto por duas máquinas. Cada máquina possui instâncias de todos os blocos, e a comunicação entre estas ocorre com instâncias do mesmo bloco, durante a execução da operação de Shuffle. Com a utilização de funções de *hash* é possível conhecer previamente a quais máquinas os dados resultantes do processamento das tarefas map e reduce devem ser destinados.

Figura 13: Bloco de execução em ambiente distribuído



Fonte: Próprio autor (2018)

A comunicação entre as instâncias dos blocos é realizada de forma assíncrona, com o uso de troca de mensagens. Após a informação haver passado pela troca de dados no processo de Shuffle, os valores armazenados na estrutura de saída serão o resultado final referente à execução global de todas as instâncias do mesmo bloco no ambiente distribuído.

4.2.3 Suporte a múltiplas entradas e saídas entre os blocos

De forma a possibilitar uma maior gama de aplicações possíveis de serem desenvolvidas utilizando o modelo, é proposto que a aplicação permita a execução de blocos considerando múltiplas entradas e saídas.

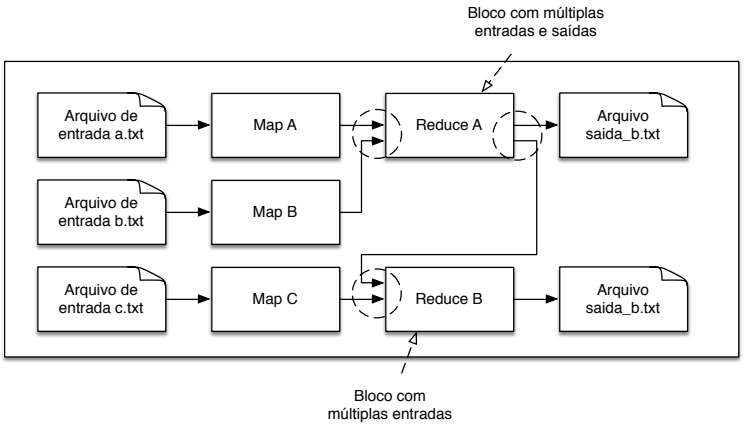
Com esta funcionalidade, uma aplicação complexa pode ser decomposta em diversas aplicações, que realizem processamentos específicos.

A Figura 14 representa um cenário de múltiplas entradas, de forma que os blocos *ReduceA* e *ReduceB* possuem como entrada vários blocos, de forma simultânea. Internamente, cada saída é registrada em uma estrutura separada. A Figura 15 apresenta as estruturas internas do bloco Reduce A, da Figura 14.

4.2.4 Múltiplas entradas e saídas

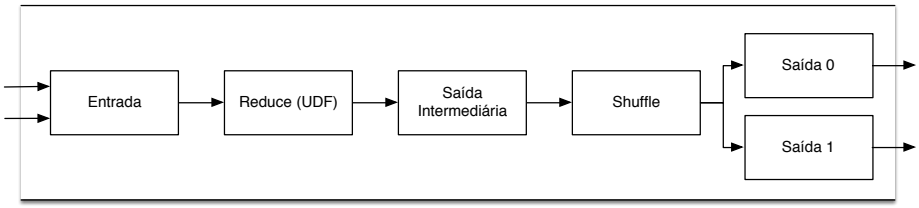
A utilização de múltiplas entradas e saídas permite que seja definido mais de um bloco a ser utilizado como saída, e a escolha de qual bloco os dados devem ser enviados

Figura 14: Fluxo de execução com blocos com múltiplas entradas e saídas



Fonte: Próprio autor (2018)

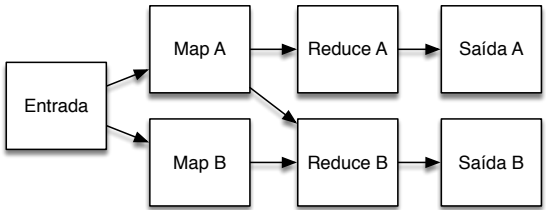
Figura 15: Estrutura interna do bloco Reduce A



Fonte: Próprio autor (2018)

é definida em tempo de execução. Com esta funcionalidade, as aplicações podem ser decompostas em diversos blocos, como pode ser visualizado na Figura 16.

Figura 16: Execução utilizando múltiplas entradas e saídas



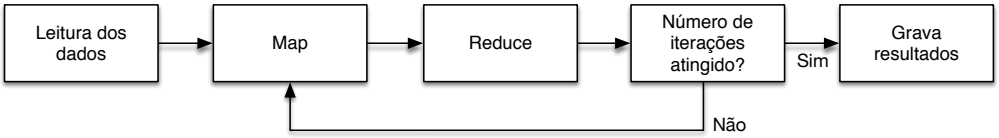
Fonte: Próprio autor (2018)

4.2.5 Execução iterativa

A implementação de algoritmos iterativos requer que existam mecanismos de execução de ciclos no decorrer do fluxo de execução. No modelo proposto, é possível que

sejam utilizados recursos de iteração com base em critérios de parada. A Figura 17 mostra um exemplo, em que o critério de parada é definido com base em uma função de contagem de ciclos.

Figura 17: Exemplo de execução utilizando iteratividade



Fonte: Próprio autor (2018)

Na utilização de ciclos, as aplicações podem fazer uso dos dados em memória, como normalmente ocorre com os blocos em sequência.

4.3 Execução de aplicações

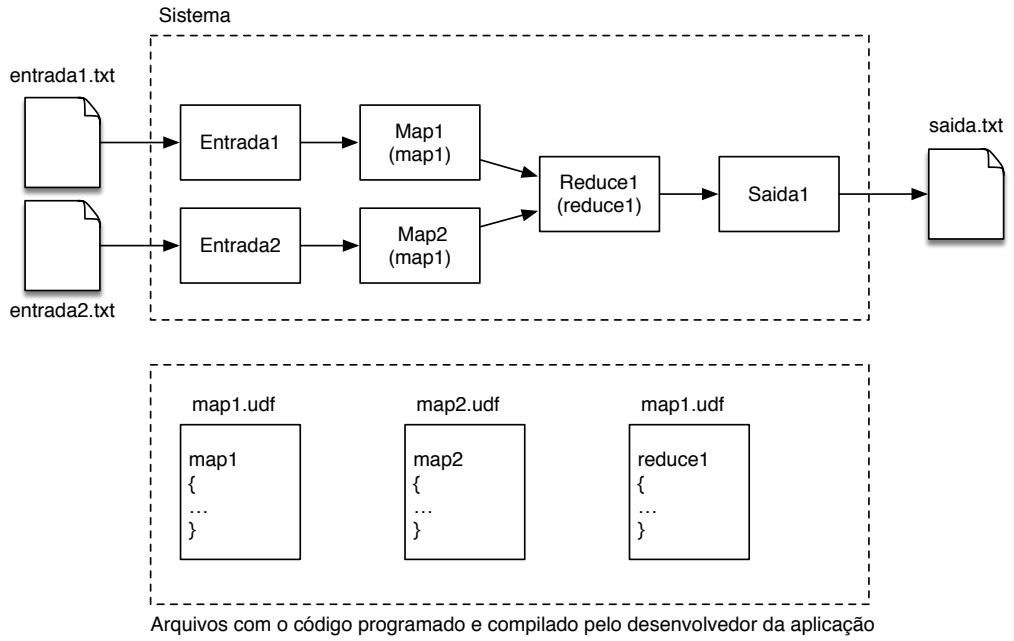
Para que a aplicação seja executada, inicialmente o usuário elabora um arquivo seguindo a linguagem de marcação estendida (XML[1]) (W3C, 2018) contendo as definições da execução da aplicação. A inicialização da aplicação é feita a partir da leitura deste arquivo, que realizará a alocação de recursos e execução das tarefas definidas. O Apêndice A apresenta um exemplo de arquivo de configuração deste tipo.

Assim que a leitura do arquivo com as informações das execuções é realizada, o sistema realiza a busca pelas partes dos arquivos, inicia a alocação dos blocos e configura o fluxo de execução, como exemplificado na Figura 18. Após os blocos serem alocados, é iniciada a execução com a leitura de dados referente a cada bloco de entrada. Assim que processadas, as informações de saída de cada bloco são repassadas como entrada para o próximo bloco, até que os dados cheguem ao bloco de saída, que será responsável por salvar os dados em arquivo.

O desenvolvedor de aplicações também deve compilar os códigos responsáveis pelas operações a serem processadas em cada bloco. Estes serão alocados em memória

[1] Acrônimo da nomenclatura em inglês *Extendend Markup Language*

Figura 18: Blocos e fluxo de execução no sistema



Fonte: Próprio autor (2018)

pelo Sistema de forma dinâmica, com chamadas às funções definidas sendo realizada pelo próprio Sistema.

Cada bloco lido do arquivo com as configurações da execução da aplicação corresponde internamente no sistema a uma estrutura de dados que possui ponteiros para as entradas e saídas, bem como ponteiro para a função correspondente para ser executada, e ponteiros para as *threads* responsáveis pela execução do bloco.

Para cada bloco da aplicação é instanciado um bloco do sistema, no qual é chamada a função correspondente e sua respectiva estrutura como parâmetro. Os blocos map e reduce instanciam blocos de execução. Todos os blocos são instanciados e colocados em execução, sendo que inicialmente permanecem suspensos, aguardando a chegada de dados em suas respectivas entradas.

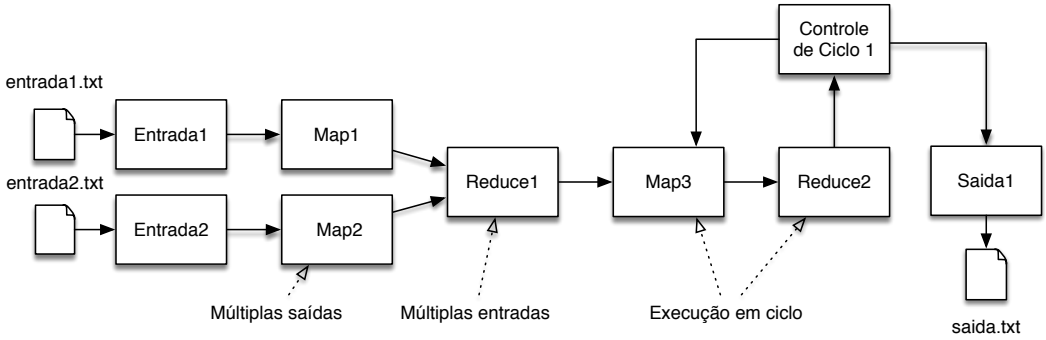
Por exemplo, na Figura 18, a execução é iniciada com a leitura de dados do arquivo `entrada1.txt`, que serão repassados para a saída do bloco Entrada1. Assim que houverem dados na saída desse bloco, o bloco Map1 sai do estado de espera, lê esses

dados e os processa, gerando novos dados e armazenando-os em sua respectiva saída, que será a entrada para o próximo bloco.

A interação do usuário com o Sistema será dada por arquivos compilados com o código a ser executado, e por arquivos de configuração contendo a definição do fluxo de execução e parâmetros do Sistema. Cada código programado pelo desenvolvedor é executado por um bloco específico. Os blocos são carregados dinamicamente e compatíveis entre si, utilizando as mesmas estruturas de entrada e saída para cada bloco. O fluxo de execução dos blocos pode conter ciclos, de maneira que blocos sejam executados mais de uma vez. A Figura 19 ilustra um exemplo de fluxo de execução em blocos de uma aplicação utilizando o Sistema e suas funcionalidades.

No exemplo da Figura 18, são utilizados dois blocos de entrada (Entrada1 e Entrada2), três blocos de execução (Map1, Map2 e Reduce1) e um bloco de saída (Saida1). Além destes, também é possível utilizar blocos de controle de ciclo e armazenamento intermediário de dados, e compor uma aplicação com fluxo de execução mais complexo, como exemplificado na Figura 19.

Figura 19: Exemplo de fluxo de execução de aplicações



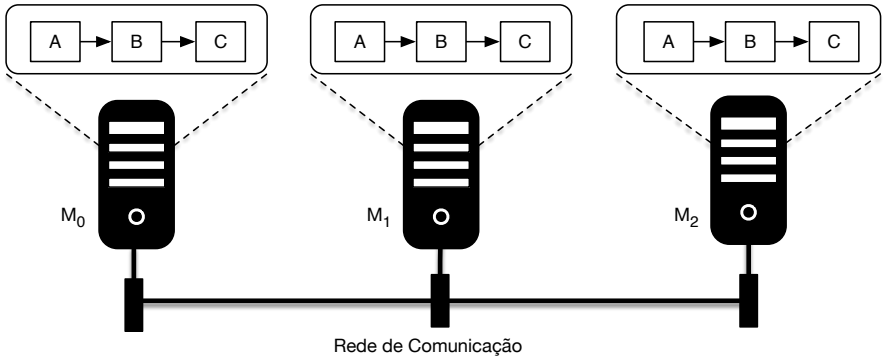
Fonte: Próprio autor (2018)

As tarefas consistem em operações map ou reduce, de forma que as tarefas executadas em determinado bloco não dependem das tarefas do próximo bloco. Por exemplo, operações de reduce podem ser encadeadas sem a necessidade de operações map.

As tarefas definidas no fluxo de execução são executadas em todas as máquinas do

ambiente distribuído. Cada máquina executa todos os blocos programados. Na Figura 20, é apresentado um exemplo em que três máquinas interligadas em rede executam uma aplicação composta por três blocos.

Figura 20: Máquinas no ambiente distribuído executando blocos



Fonte: Próprio autor (2018)

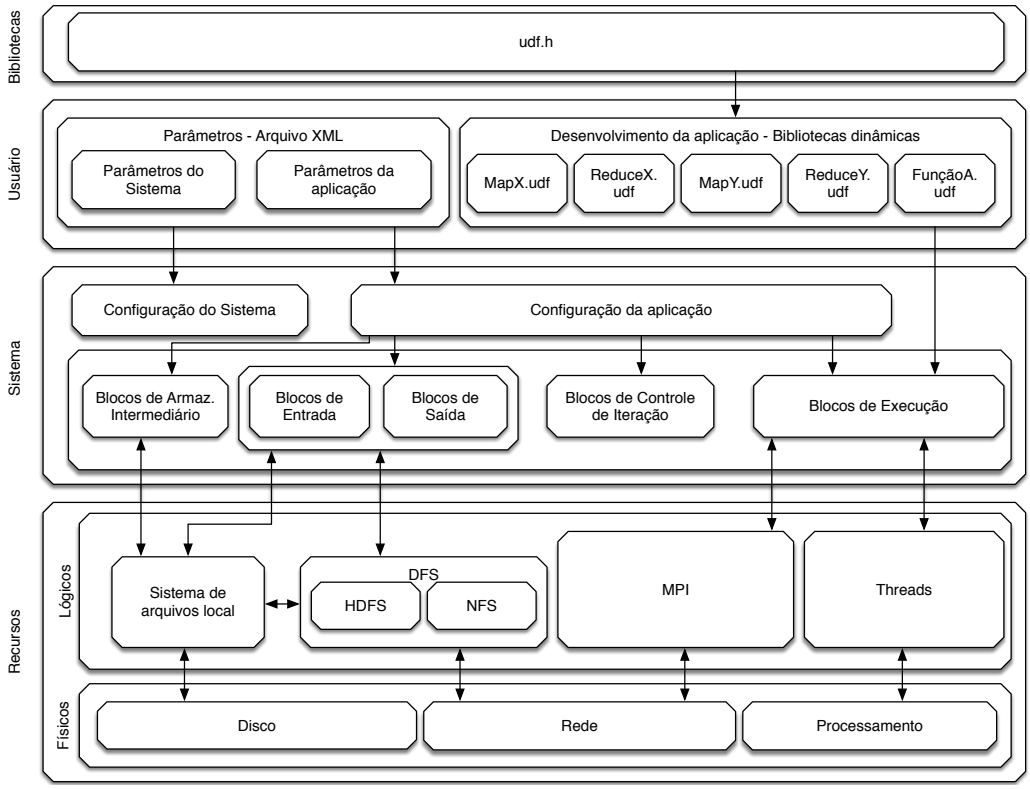
As tarefas de map e reduce a serem executadas na aplicação são definidas pelo usuário. Os códigos de cada tarefa são armazenados em arquivos a serem carregados pelo Sistema, assim como, a ordem de execução. A programação de cada tarefa é realizada utilizando uma API fornecida ao usuário, e o arquivo contendo o código da tarefa a ser executada é carregado dinamicamente, em tempo de execução.

4.4 Arquitetura

O ambiente de programação e execução é constituído por diversos componentes, cada um com responsabilidades específicas. O conjunto desses módulos compõe a sua arquitetura e define suas relações com os recursos computacionais e como se dará a execução de aplicações do usuário utilizando as estratégias propostas.

A arquitetura é apresentada de forma gráfica na Figura 21 e está estruturada em camadas. Na parte superior da Figura 21 têm-se as bibliotecas fornecidas ao usuário, responsáveis pela interação das funções que compõem as tarefas com o Sistema. Logo abaixo está a camada do usuário, com os arquivos cujos nomes devem ser informados

Figura 21: Arquitetura do Sistema



Fonte: Próprio autor (2018)

pelo usuário no início da execução da aplicação. O arquivo contendo os parâmetros do Sistema possui configurações relacionadas ao ambiente de execução, enquanto o arquivo de parâmetros da execução possui as especificações da aplicação em si, com o fluxo de execução a ser utilizado, definição das entradas e saídas e configurações de cada bloco. Ambos arquivos devem seguir o formato XML. Na mesma camada, estão os arquivos com os códigos já compilados a serem executados em cada tarefa, as funções UDF. Estas, são arquivos binários com bibliotecas dinâmicas, que posteriormente serão carregadas e executadas em tempo de execução pelo Sistema.

O Sistema é responsável por carregar os arquivos apresentados pelo usuário e executar a aplicação de acordo com os parâmetros definidos. Nesta camada, os blocos são alocados e os arquivos com as funções UDF são carregadas e executadas. A camada Recursos, com os recursos necessários para a execução é dividida em recursos

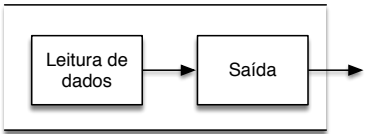
lógicos e físicos, onde os recursos lógicos abordam aspectos relacionados ao sistema operacional, sistemas de arquivos e comunicação, enquanto recursos físicos indicam os componentes de *hardware* presentes nas máquinas utilizados diretamente.

4.4.1 Blocos de entrada de dados e blocos de saída de dados

A finalidade dos blocos de entrada e saída de dados é proporcionar o interfaceamento dos blocos de execução com os sistemas de armazenamento.

Um bloco de entrada de dados consiste em uma estrutura composta por uma operação de leitura de dados e uma estrutura de armazenamento de dados em memória. A leitura de dados consiste em carregar informações no Sistema a partir de um sistema de arquivos definido, como o HDFS ou o sistema de arquivos utilizado no disco físico da máquina local. Com o uso desta operação, a leitura de dados torna-se facilitada para o usuário do sistema, bastando indicar apenas informações de acesso ao arquivo, como nome, tipo, localização e modo de leitura (binário ou texto). Desta forma, as bibliotecas necessárias para a leitura são carregadas e as operações de movimentação de dados para a memória são realizadas, guardando as informações na estrutura de saída, já utilizando uma estrutura de dados seguindo o par (k, v). A Figura 22 destaca a estrutura interna utilizada nos blocos de entrada de dados.

Figura 22: Estrutura interna do bloco de entrada de dados

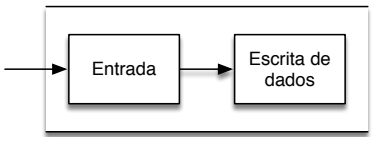


Fonte: Próprio autor (2018)

Os blocos de saída de dados operam de forma semelhante, porém estes tratam de armazenar em disco as informações existentes na estrutura de entrada. A Figura 23 mostra o esquema interno de um bloco deste tipo.

No ambiente de execução distribuído, ambos os blocos de entrada e saída de dados

Figura 23: Estrutura interna do bloco de saída de dados



Fonte: Próprio autor (2018)

possuem instâncias em todas as máquinas. Porém, diferente dos blocos de execução, estes não realizam comunicação de dados entre as diversas instâncias dos blocos.

4.5 Outras funcionalidades

O uso do processamento em blocos é projetado de maneira que possibilite outras funcionalidades, como o processamento especulativo, o armazenamento intermediário e a execução iterativa.

4.5.1 Processamento especulativo

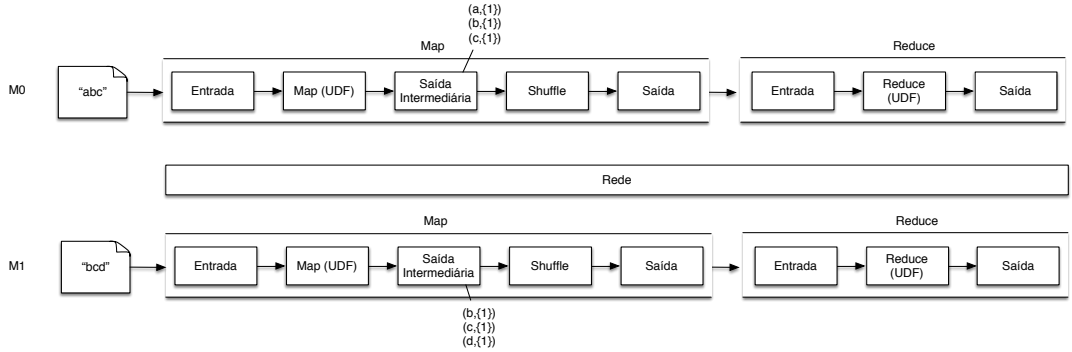
O uso de diversas operações sendo realizadas utilizando um fluxo de execução permite que recursos sejam aproveitados de melhor maneira, como a memória local, por exemplo. Porém algumas operações atuam de forma bloqueante, impossibilitando que o processamento do próximo bloco seja iniciado antes do término do bloco anterior. Esta dependência de dados pode levar ao mal uso de recursos, visto que blocos devem aguardar o término da execução do bloco anterior em todas as instâncias em execução no ambiente distribuído, ocasionando em ociosidade de unidades de processamento durante o tempo de espera.

Na medida em que resultados forem recebidos das demais instâncias, as informações de entrada em cada bloco seguinte serão atualizadas, para já serem processadas.

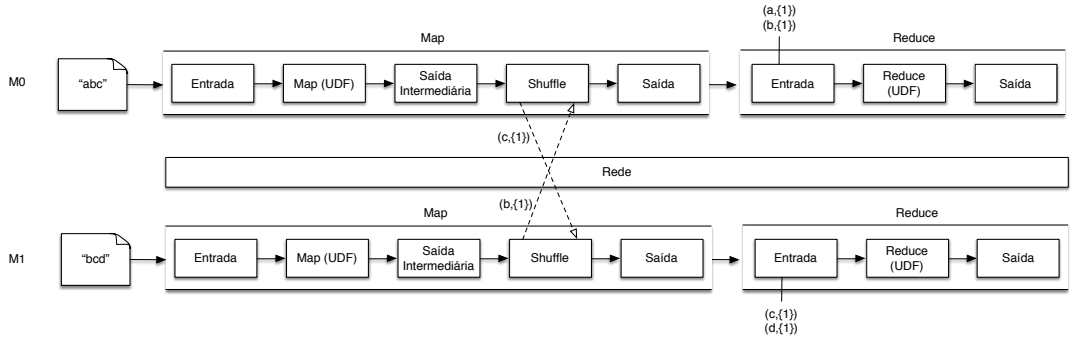
A vantagem do processamento especulativo é o adiantamento no fluxo de execução, possibilitando o melhor uso de recursos devido à não necessidade de espera da finalização do processamento do bloco anterior.

Figura 24: Exemplo de dependência de dados em tarefas de reduce

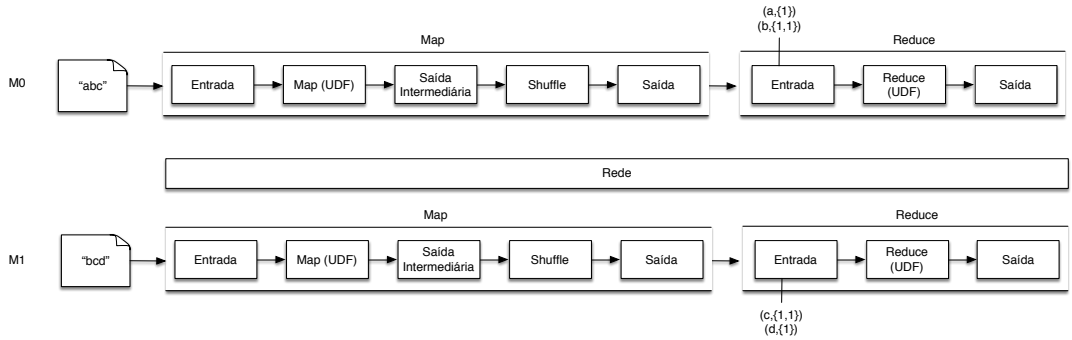
(a) Operação de Map finalizada



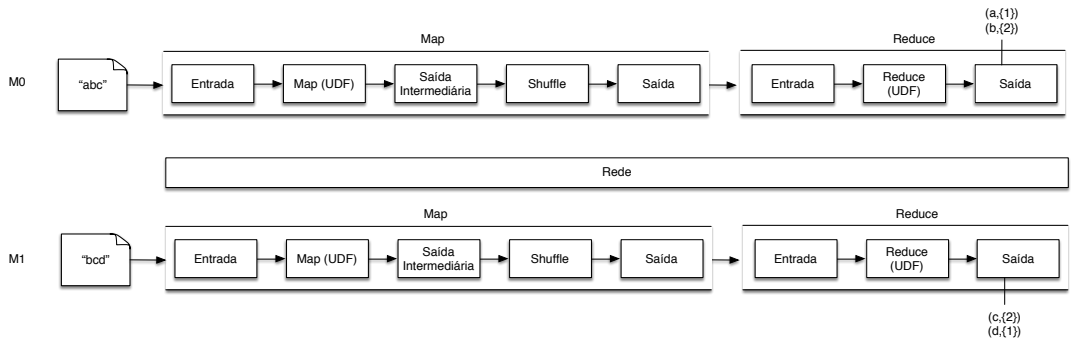
(b) Troca de dados com o Shuffle



(c) Reduce recebeu todos os dados



(d) Operação de Reduce finalizada



Fonte: Próprio autor (2018)

Como exemplo da aplicação da estratégia, a Figura 24 apresenta um cenário em que a dependência de dados nas operações de reduce é destacada. Nesta, é utilizada uma aplicação que realiza a contagem do número de ocorrências de cada caractere da entrada. Para o exemplo, considere que a função de particionamento utilizada segue o *hash* h_1, tal que

$$h_1(x) = \begin{cases} 0, & \text{se } x \in \{a, b\} \\ 1, & \text{se } x \in \{c, d\} \end{cases}$$

A Figura 24 mostra quatro instantes do processamento da aplicação. Em 24(a) é apresentado o instante em que as tarefas da operação de Map finalizaram o processamento das entradas, emitindo os pares (k, v). Em 24(b) é acompanhado o passo seguinte à operação de Map, desempenhado pela fase de Shuffle. Nesta, dados que devem ser processados pela própria máquina são repassados diretamente ao bloco de Reduce e dados que devem ser processados por outras máquinas são enviados utilizando a rede. Em 24(c) é visto o instante em que as tarefas de Reduce receberam todos os dados processados pelas tarefas de Map. A Figura 24(d) apresenta o resultado final da aplicação com o processamento realizado nas tarefas de Reduce.

Observa-se que durante a espera da chegada dos dados pela rede, que ocorre nos instantes de tempo entre as Figuras 24(b) e 24(c), não é realizado nenhum processamento nas operações de Reduce. Este tempo de espera pelos dados resulta em ociosidade no processamento das máquinas do ambiente distribuído.

A solução de otimização de desempenho para este caso consiste no processamento dos dados pelo reduce na medida em que forem disponíveis pelo bloco de map, seja processado localmente ou recebido de outras máquinas do ambiente distribuído.

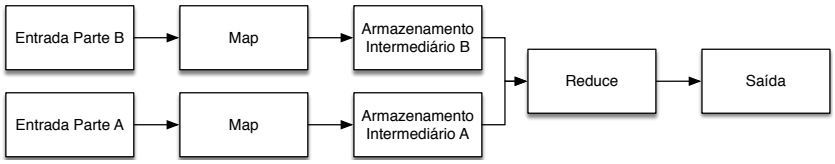
4.5.2 Armazenamento intermediário

À medida que um volume maior de informações é processado por uma máquina, a quantidade de memória utilizada cresce proporcionalmente. Além dos dados da aplicação, também é necessária memória para as estruturas de dados referentes ao processamento do ambiente de execução.

O uso do mecanismo de memória virtual (TANENBAUM; WOODHULL, 2000) torna possível a execução de aplicações com tamanho superior à capacidade da memória física. Para tal armazena páginas de processos em disco e as carrega em memória conforme vão sendo executadas. Porém quando não há espaço suficiente na memória física, é necessário que um alto número de buscas de páginas em disco seja realizado, e com isso, o desempenho das aplicações executadas é degradado, visto que o tempo de acesso a disco é muito maior que o tempo de acesso à memória (STALLINGS, 2010).

Como alternativa a evitar o uso da memória virtual, propõe-se o uso de um bloco que armazene em disco resultados intermediários da aplicação, de uma única vez. Desta forma, a quantidade de dados de entrada para uma mesma máquina pode ser maior, visto que o consumo de memória é diminuído. A Figura 25 apresenta um exemplo de utilização de bloco de armazenamento intermediário de dados.

Figura 25: Bloco de armazenamento intermediário



Fonte: Próprio autor (2018)

A utilização de blocos de armazenamento intermediário faz parte da lógica de programação da aplicação, e deve ser prevista pelo usuário ou desenvolvedor de aplicações, considerando o tamanho previsto das entradas de dados. Blocos de armazenamento intermediário também podem ser utilizados em estruturas de repetição (Seção 4.5.3), abordando esquemas de divisão e conquista no processamento de grandes ar-

quivos.

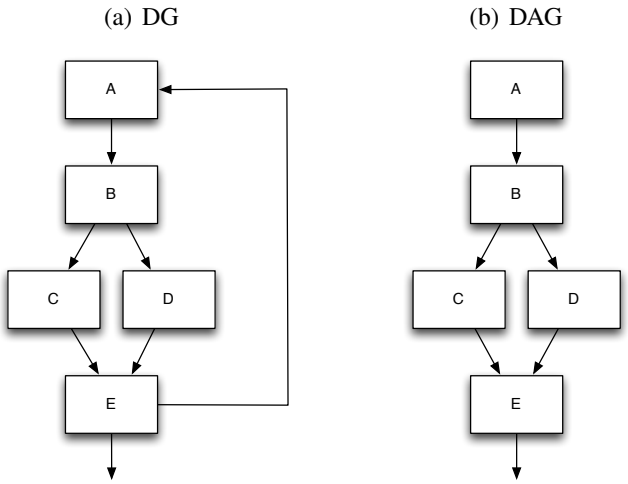
4.5.3 Execução iterativa

O fluxo de execução consiste na execução dos diversos blocos que compõem o sistema, de forma ordenada e representado por um grafo direcionado (DG).

Propõe-se o uso de DG de maneira que cada vértice do grafo representa um bloco sendo executado, e as arestas do grafo representam a comunicação entre os blocos, realizada na máquina local, seguindo o modelo produtor consumidor.

Diversas ferramentas que se propõem a manipular dados em *Big Data* disponibilizam a utilização de fluxo de execução modelado por um grafo direcionado acíclico (DAG) (APACHE, 2015a; APACHE, 2018). A utilização deste tipo de grafo impossibilita a execução de mecanismos de iteratividade, devido à necessidade de realização de ciclos no decorrer do fluxo de execução. Na Figura 26 é ilustrada a comparação de dois grafos, DG e DAG. Na Figura 26(a) os vértices $\overline{ABDE}$ formam um ciclo, devido à aresta $\overline{EA}$ retornar a um vértice já visitado anteriormente.

Figura 26: Grafo direcionado e grafo direcionado acíclico



Fonte: Próprio autor (2018)

Na execução utilizando DAG, os blocos são executados uma única vez, simplifi-

cando a implementação deste tipo de execução devido a não necessidade de utilização de estruturas que diferenciem o estado de execução em que as informações se encontram.

4.6 Comparação com os trabalhos relacionados

As estratégias propostas neste capítulo objetivam solucionar algumas deficiências existentes no modelo de programação Mapreduce.

Com a utilização de um fluxo de execução utilizando o conceito de blocos será possível executar diversas operações Map e Reduce, de forma encadeada, sem a necessidade de iniciar uma nova execução do sistema utilizando dados intermediários, como ocorre com o modelo Mapreduce.

A proposta também contempla estratégias para problemas de otimização, como o bloqueio em operações de Reduce, utilizando processamento especulativo. Com isto, será possível que a execução de blocos seja iniciada antes do término da execução do bloco anterior.

A utilização de ciclos no fluxo de execução será possível utilizando uma combinação das múltiplas entradas e saídas e dos blocos de armazenamento intermediário, onde o último bloco pertencente ao ciclo possui duas saídas: (a) uma nova execução do ciclo e outra (b) para a continuidade do fluxo de execução.

5 IMPLEMENTAÇÃO DO SISTEMA DE EXECUÇÃO

Neste capítulo são apresentados aspectos de implementação do sistema de execução que aplica as estratégias definidas para a utilização de grandes quantidades de dados, como descrito no Capítulo 4.

O sistema foi desenvolvido utilizando a linguagem de programação C, esta escolhida devido a sua ampla utilização em aplicações de alto desempenho, com bibliotecas e estruturas de comunicação voltadas para a execução no sistema operacional Linux, com exceção do algoritmo que localiza partes de arquivos locais, este implementado utilizando a linguagem de programação Java (ORACLE, 2018) e *scripts* Bash (GNU, 2018).

O ambiente desenvolvido é composto por um sistema distribuído e paralelo, utilizando diversos módulos responsáveis por executar códigos programados pelo usuário.

5.1 Estruturas de dados

O gerenciamento das estruturas de entrada e saída de dados em cada bloco é implementado de maneira que as estruturas de dados utilizadas como entrada nos blocos sejam equivalentes às estruturas da saída, tornando os blocos compatíveis entre si. A implementação dos pares (k, V) é apresentada nos códigos das Figuras 27, 28 e 29. Com isto, o par (k, v) é representado como um caso particular de (k, V) em que V possui um único elemento. Denota-se como par KV ou estrutura KV à estrutura de dados

apresentada na Figura 29.

Figura 27: Estrutura de dados utilizada para representar chaves

```
struct Key
{
    unsigned long long int tam_key;
    void *key;
}
```

Fonte: Próprio autor (2018)

Figura 28: Estrutura de dados utilizada para representar valores

```
struct Value
{
    unsigned long long int tam_value;
    void *value;
}
```

Fonte: Próprio autor (2018)

Figura 29: Estrutura de dados utilizada para representar chaves e valores

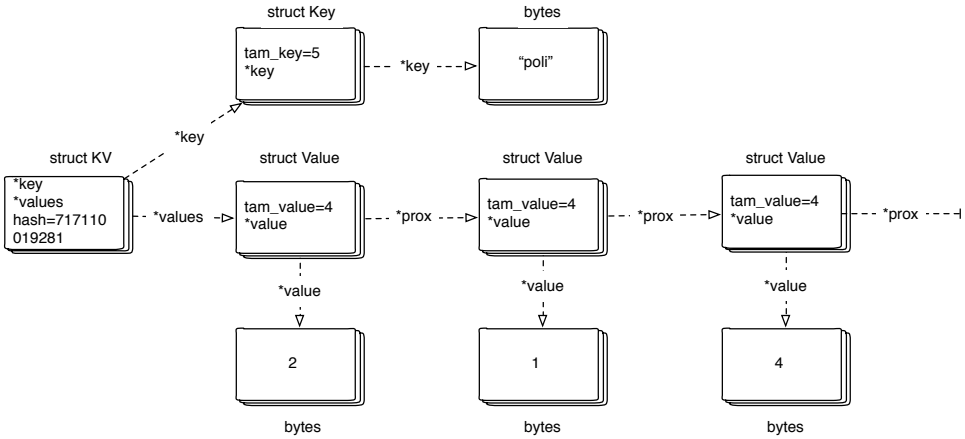
```
struct KV
{
    struct Key *key;
    struct List *values;
    uint32_t hash;
}
```

Fonte: Próprio autor (2018)

O armazenamento do par (k, V) ocorre utilizando ponteiros para estruturas em memória do tipo `void`, permitindo a utilização de qualquer tipo de dado, sendo sua definição especificada no código das funções definidas pelo usuário. A Figura 30 ilustra a alocação em memória para o exemplo (*"poli"*, {2,1,4}). Nesta, cada elemento representa uma região de memória alocada, e pode-se observar a estrutura KV e ponteiros para a estrutura `Key` e valores do tipo estrutura `Value`.

Nos blocos utilizados no Sistema, as estruturas de armazenamento de dados (entrada, saída intermediária e saída) são compostas por estruturas `KV` organizadas em árvores do tipo rubro-negras (CORMEN et al., 2001, pp. 275), ordenadas de acordo com o valor do *hash* da chave. Cada nó da árvore pode suportar e armazenar diversas estruturas `KV`, caso estas possuam o mesmo valor de *hash*. Com isto, colisões de *hash* são tratadas diretamente na busca, remoção e inserção de valores na estrutura de árvore. A escolha de utilização de árvores rubro-negras dá-se devido às suas características de busca e inclusão de informações serem realizadas de forma eficiente.

Figura 30: Exemplo de armazenamento de estrutura



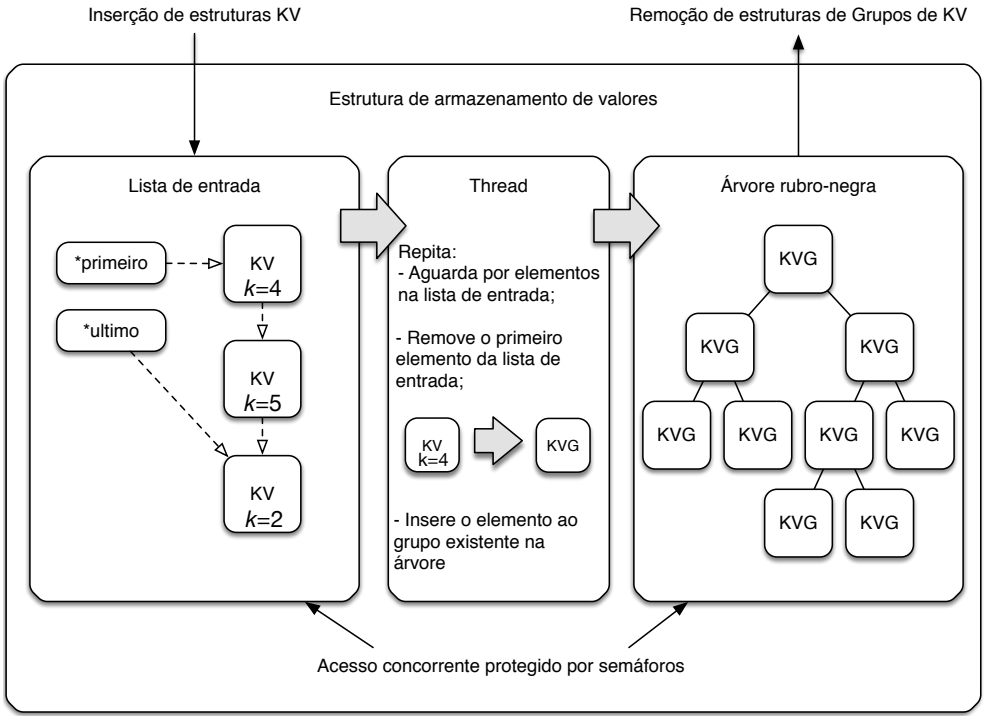
Fonte: Próprio autor (2018)

5.1.1 Módulos de armazenamento de valores e controle de granularidade

O gerenciamento das estruturas de dados com os pares (k, V), além da estrutura em árvore possui também uma estrutura em lista de entrada e uma *thread*. A estrutura de entrada consiste em uma lista com acesso concorrente protegido por semáforos, com o intuito de evitar inconsistências na lista. Cada elemento da árvore é composto por um grupo de pares KV (KVG) organizados em lista. A *thread* consome um par da entrada e verifica se já existe um grupo para aquele par. Caso o grupo já exista, o par é inserido na lista, caso contrário um novo grupo contendo o par é criado e inserido na árvore.

Cada grupo é composto por pares que possuem a mesma sequência de g bits de *hash* iniciais, tal que g é definido previamente à execução da aplicação. Desta forma, a busca por um grupo na árvore consiste em encontrar um grupo que possua os mesmos g bits iniciais que o *hash* do par que será inserido. Cada grupo possui uma identificação única *gid*, dada pela sequência de g bits que seus pares possuem em comum. Este mecanismo de controle de granularidade está ilustrado na Figura 32.

Figura 31: Módulo de armazenamento das estruturas

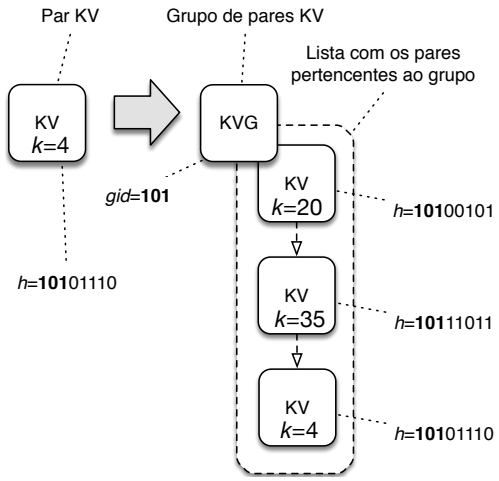


Fonte: Próprio autor (2018)

Na Figura 31 é exemplificada uma situação com três elementos KV na lista de entrada temporária. Na ilustração é exibido um valor em cada elemento KV, indicando que cada elemento é referente a um par (k, v) distinto. No exemplo, o primeiro elemento da lista é retirado pela *thread* e posteriormente inserido em um grupo na árvore rubro-negra, processo este ilustrado na Figura 32. Na árvore, o grupo pode ser localizado de maneira eficiente, devido às propriedades de busca na árvore rubro-negra.

A adoção de uma mesma estrutura de dados para o armazenamento de dados de

Figura 32: Grupo de pares KV



Fonte: Próprio autor (2018)

entrada e saída de todos os blocos presentes no Sistema simplifica e otimiza a implementação do fluxo de execução, visto que repassar um dado para o próximo bloco significa remover o endereço da estrutura da árvore de saída do bloco anterior e inseri-lo na estrutura da árvore de entrada do bloco seguinte, dispensando a movimentação de dados em memória.

5.2 Sistema de execução

A execução no ambiente distribuído é dada pela utilização de todas as máquinas presentes, e também considera-se que o sistema de arquivos HDFS esteja presente nas máquinas. Uma máquina é responsável por realizar a leitura dos arquivos de entrada e iniciar a execução no ambiente, aqui denotada como máquina Mestre.

O Algoritmo 3 descreve como é realizada a execução das aplicações no Sistema, com a leitura dos arquivos que definem o fluxo de execução. Após ter seus blocos instanciados em todas as máquinas do ambiente distribuído, a execução da aplicação é iniciada.

```
1  início
2      Leitura do arquivo de configuração geral
3      Leitura do arquivo de configuração da execução
4      Verificação de quais máquinas possuem os arquivos de entrada
5      Distribui para as máquinas os arquivos com as funções UDF
6      Instancia todos os blocos do fluxo de execução em todas as máquinas do
         ambiente de execução distribuído
7      Configura os blocos, de acordo com as entradas e valores calculados
8      Inicia a execução da aplicação
9      Aguarda os blocos de saída finalizarem.
10 fim
```

Algoritmo 3: Execução de aplicações no sistema de execução

5.3 Banco de memória

Visando otimizar o desempenho do ambiente de execução, foi implementado um banco de memória compartilhado entre os blocos presentes no fluxo de execução. Seu funcionamento consiste em utilizar estruturas de dados previamente alocadas e não mais utilizadas, dispensando o uso de chamadas de sistema responsáveis por alocações de memória. O uso de funções do tipo `malloc` requerem que o *kernel* do sistema operacional realize diversas varreduras em sua área de memória livre e reserve a área requerida para uso exclusivo da aplicação (LINUX QUESTIONS, 2009).

A implementação do banco de memória consiste em alocar vetores de estruturas e armazenar os endereços de memória em uma estrutura de dados do tipo pilha. Desta forma, quando um bloco requisitar uma estrutura, um endereço é desempilhado, e, caso a pilha esteja vazia, um novo vetor de estruturas é alocado. Neste mecanismo, solicitar ao Sistema uma nova estrutura consiste em retirar um endereço da pilha, e remover valores previamente existentes no local de memória apontado. De forma análoga, informar que uma estrutura não será mais utilizada pela aplicação consiste em inserir seu endereço na pilha.

5.4 Funções do usuário

A interface do usuário com o Sistema consiste basicamente no cabeçalho da função a ser implementada, e na utilização das estruturas e funções fornecidas ao usuário na biblioteca UDF, com os protótipos de funções especificados no arquivo `udf.h`.

O cabeçalho da função a ser implementada pelo usuário é apresentado no código da Figura 33, que possui o nome da função, o ponteiro para a estrutura de retorno, ponteiros para as saídas e ponteiro para a estrutura de entrada a ser processada pela função. As saídas de dados para o Sistema ocorrem com o uso da função `Emit`, apresentada no código da Figura 34. Os endereços `aloc_kv` e `aloc_lista` apontam para os bancos de memória de estruturas KV e elementos de lista, respectivamente. As estruturas e funções fornecidas podem ser vistas no Apêndice A, Seção A.1 .

Figura 33: Código do cabeçalho de função para funções UDF

```
1: int <nome da função> ( void *retorno, void *aloc_kv, void
  *aloc_lista, struct KV *kv_input);
```

Fonte: Próprio autor (2018)

Figura 34: Código do cabeçalho da função `Emit` utilizada nas funções UDF

```
1: int Emit ( struct KV *kv_output, void *retorno);
```

Fonte: Próprio autor (2018)

5.4.1 Funções definidas pelo usuário

A operação a ser realizada pelos diversos blocos de execução no ambiente é individual a cada bloco e deve ser definida e programada pelo usuário (UDF). Deve ser implementada e compilada gerando bibliotecas dinamicamente carregáveis (KERRISK, 2010, p. 279). Com a utilização destas, torna-se possível executar as funções UDF sem a necessidade de recompilar o Sistema. Aqui, estas funções são referentes

a operações de map e reduce. O Apêndice A apresenta e descreve um exemplo de aplicação utilizando o código apresentado.

Quanto às estruturas de dados utilizadas e funções para manipulação de dados, durante a execução da aplicação, as rotinas disponibilizadas em `udf.h` encontram-se carregadas em memória como parte do Sistema. No decorrer da execução do código do usuário são realizadas apenas chamadas a essas rotinas. De forma semelhante é realizado o acesso aos dados dos parâmetros, com a utilização de ponteiros para os dados no sistema. Com isto, o número de cópias de dados em memória é reduzido. A documentação das funções e estruturas disponíveis em `udf.h` pode ser observada no anexo A.

5.4.2 Configurações do sistema e da execução de aplicações

As operações especificadas pelo usuário são executadas pelo Sistema, que acessa um arquivo da aplicação contendo os parâmetros a serem utilizados, bem como o fluxo de execução. Para tal, são utilizados arquivos padronizados seguindo o formato XML. Tais arquivos são escritos pelo usuário e utilizados como entrada para o Sistema. A definição completa dos arquivos de configuração, bem como exemplos de utilização são apresentados no apêndice A.

5.5 Blocos de execução

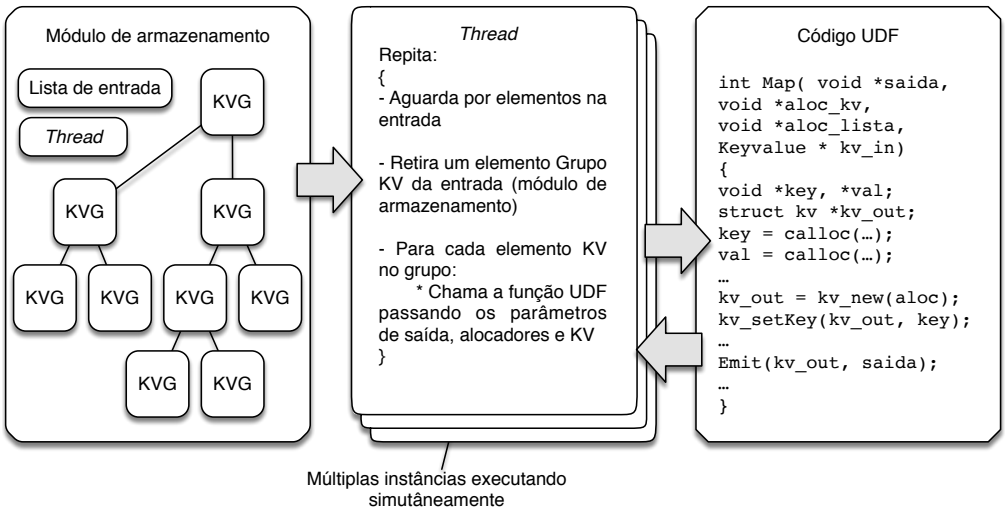
A função dos blocos de execução é executar o código definido pelo usuário, com as operações que realizam a transformação dos dados de entrada em dados de saída. Os módulos com o armazenamento em memória da entrada, saída intermediária e saída de cada bloco operam de acordo como descrito na Seção 5.1.1. A saída intermediária corresponde aos resultados dos dados processados localmente, antes da execução da operação de Shuffle. Após a execução deste, têm-se o resultado global, contendo os

resultados de todas as instâncias de execução desse mesmo bloco, que serão salvos no módulo de saída. O fluxo dos dados nesses módulos segue o modelo produtor consumidor (DIJKSTRA, 1968, p. 31).

Para que a função UDF seja executada, a biblioteca é carregada para a memória em um espaço do usuário e serão realizadas chamadas de funções neste espaço. Na linguagem de programação C, este mecanismo de carregamento e execução é efetuado com a utilização das funções `dlopen` e `dlsym`, da biblioteca `dlfcn.h` (IEEE; THE OPEN GROUP, 2001).

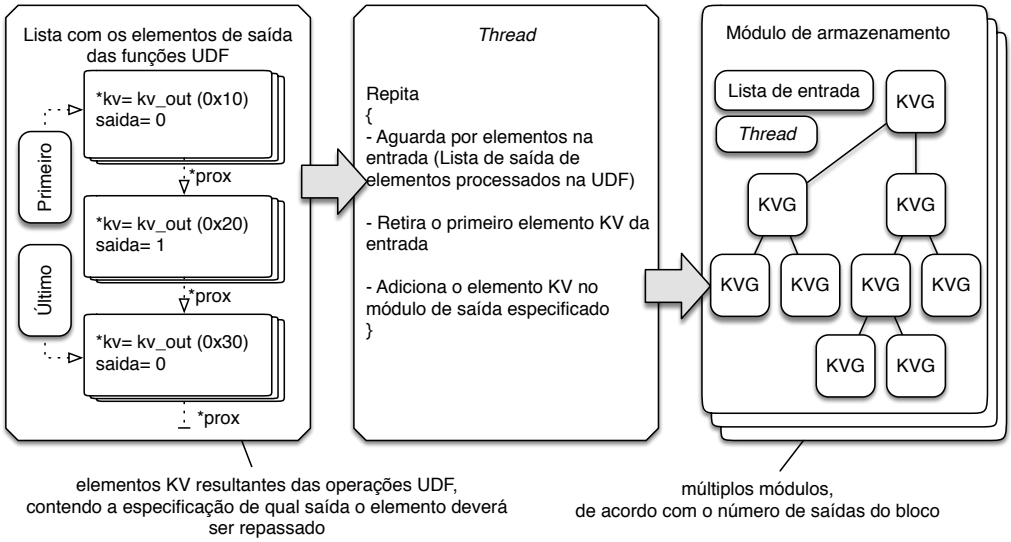
Em cada bloco de execução, uma *thread* retira um elemento KV da entrada de dados e o utiliza como parâmetro para a função UDF correspondente ao bloco, como indicado na Figura 35. No momento em que a função `Emit` do código UDF for executada, seu resultado será salvo em uma lista temporária de saída. Outra *thread* será responsável por retirar os dados desta lista e repassá-los ao módulo de armazenamento de dados intermediários, como indicado na Figura 36.

Figura 35: *Thread* passando parâmetros e executando chamadas às funções



Fonte: Próprio autor (2018)

Figura 36: *Thread* retirando elementos KV de saída das execuções UDF e inserindo-os nas estruturas de saída do bloco



Fonte: Próprio autor (2018)

5.5.1 Execução paralela das tarefas

De forma a utilizar melhor as diversas unidades de processamento nas determinadas máquinas, a execução das funções UDF é realizada de forma paralela utilizando diversas *threads* operando de forma simultânea. A *thread* apresentada na Figura 35 possui p instâncias de execução, tal que p é um valor definido pelo usuário no arquivo de configuração do ambiente de execução.

As diversas *threads* consomem dados da árvore de entrada de forma concorrente, visto que apenas um elemento pode ser retirado da árvore simultaneamente, necessário para que a consistência da árvore seja mantida. O elemento consumido é um grupo de pares KV onde cada par presente no grupo será processado sequencialmente pela função UDF.

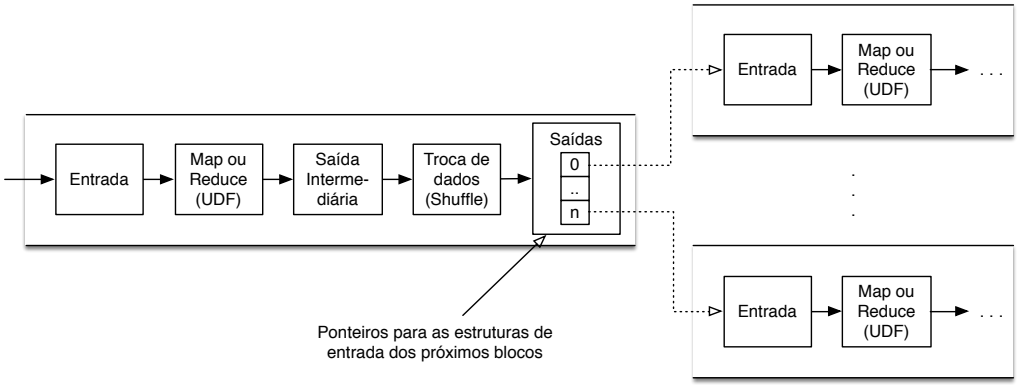
5.6 Múltiplas entradas e saídas

A possibilidade de uso de diversas entradas e saídas é implementada utilizando ponteiros nos blocos de execução. Internamente, as estruturas de saída não são alocadas, de forma que cada bloco possui apenas ponteiros para as estruturas de entrada do próximo bloco. Este mecanismo possibilita que a transferência de dados para o próximo bloco seja realizada de maneira eficiente.

As múltiplas saídas de um bloco são definidas por um vetor de ponteiros, de forma que cada posição do vetor aponte para uma das saídas definidas para o bloco, e durante a execução da função UDF uma saída deve ser especificada como destino para o par KV. Este esquema é ilustrado na Figura 37.

As múltiplas entradas não possuem um mecanismo de acesso específico para cada entrada, visto que os módulos de entrada possuem controle de concorrência, conforme mostra a Figura 31, e basta apenas que os blocos apontem suas saídas para o próximo bloco.

Figura 37: Estrutura de múltiplas saídas



Fonte: Próprio autor (2018)

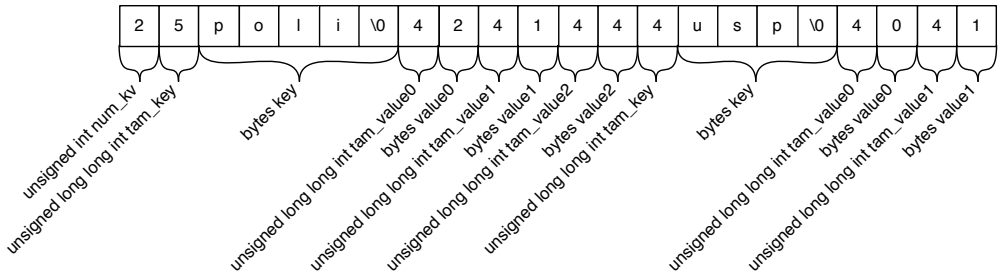
5.7 Trocas de dados

A troca de dados entre as diversas instâncias de execução dos blocos no ambiente distribuído é realizada durante a operação de Shuffle, interna a cada bloco de execução. Esta operação atua como consumidor da estrutura de dados com a saída intermediária, e insere seus resultados na estrutura de entrada do bloco seguinte.

A comunicação é realizada utilizando a biblioteca de troca de mensagens MPI.

Cada estrutura KV tem seus dados agrupados, de forma a ocupar um espaço contíguo em memória. Como estes dados são enviados para outras máquinas, o desenvolvedor não deve utilizar ponteiros no espaço de dados para chaves e valores. A Figura 38 apresenta um exemplo de armazenamento em sequência em que são enviadas duas estruturas KV, compostas por (*"poli"*, {2,1,4}) e (*"usp"*, {0,1}). Com a utilização deste mecanismo, todos os dados da estrutura KV serão enviados pela rede utilizando uma única mensagem, diminuindo a sobrecarga necessária para a troca de informações.

Figura 38: Armazenamento em sequência de valores KV para trocas de dados



Fonte: Próprio autor (2018)

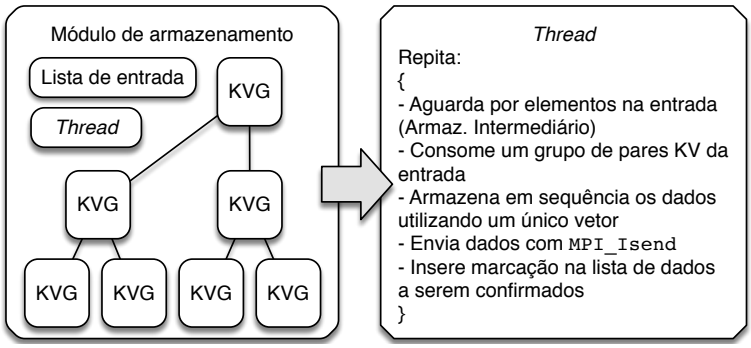
A troca de mensagens ocorre de forma assíncrona, com o uso das funções `MPI_Isend` e `MPI_Irecv`. A vantagem da utilização de mensagens assíncronas é permitir que as instâncias dos blocos continuem a sua execução, sem o bloqueio das tarefas enquanto as mensagens são enviadas para as demais instâncias do bloco no ambiente distribuído. Para que estas tarefas possam ser executadas, a tarefa de Shuffle possui 3 *threads*, definidas a seguir.

- Envio: Responsável por retirar as estruturas KV da saída intermediária e enviar para as máquinas utilizando a rede (Figura 39). A quantidade de dados submetidos à função `MPI_Isend` é dada por todos os pares KV presentes no grupo de pares consumidos da entrada. Todos os pares são armazenados em uma mesma sequência, e então após isso, os dados são enviados ao destino;

- Controle de recebimento: Após o envio de uma mensagem, a *thread* de envio inclui um registro de marcação em uma lista de mensagens a confirmar, com o objetivo de conferir se todas as mensagens já foram recebidas (Figura 40). A *thread* de controle de recebimento consiste em reconhecer se a mensagem já foi recebida e retirar o registro de marcação da lista. A realização desta tarefa é necessária para reconhecer se o bloco encontra-se ativo ou inativo, com seu processamento já finalizado;

- Recebimento: Cuida do recebimento dos dados pela rede. Opera aceitando o recebimento de qualquer máquina, e repassando o dado recebido à estrutura de saída especificada no par (Figura 41).

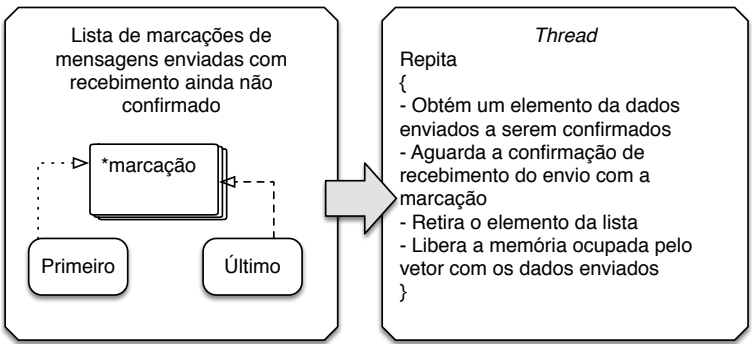
Figura 39: Envio dos dados de forma assíncrona utilizando MPI



Fonte: Próprio autor (2018)

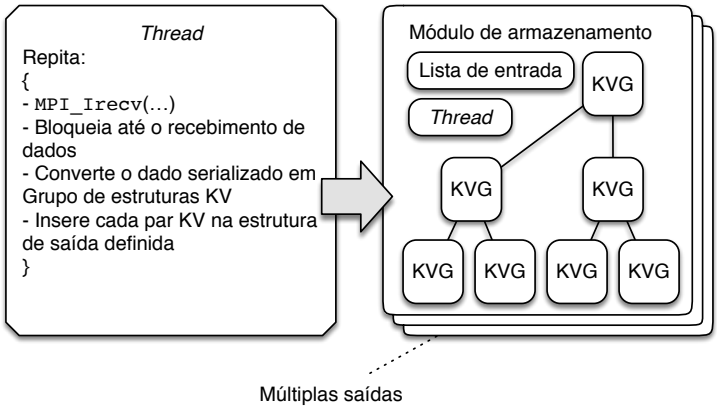
De forma a diminuir a sobrecarga de processamento e memória necessária à transferência de dados, no passo anterior ao envio é verificado se o destino é a própria máquina. Caso sim, o dado é repassado diretamente à estrutura de saída, sem a necessidade de acesso à rotinas de comunicação.

Figura 40: Confirmação de recebimento dos dados enviados



Fonte: Próprio autor (2018)

Figura 41: Recebimento dos dados de forma assíncrona utilizando MPI



Fonte: Próprio autor (2018)

5.8 Agrupamento de valores em duas etapas

O processo de agrupamento de valores *v* associados à mesma chave *k* é realizado em duas etapas, tendo em vista a diminuição do tempo necessário neste processo. A primeira etapa consiste no próprio agrupamento de pares KV[1], dividindo o conjunto total de chaves em subconjuntos menores. Adicionar um novo par ao grupo consiste em incluir uma nova estrutura na lista de pares, cuja operação possui baixo custo computacional.

Após a troca de dados, grupos de pares KV são unidos em um único grupo contendo os pares recebidos de outras máquinas. Este passo, além de unir os grupos que

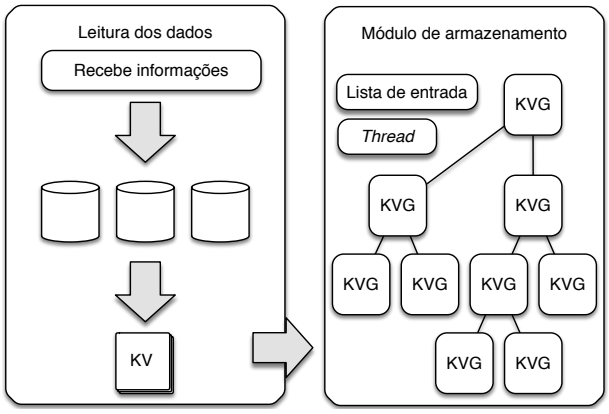
[1]Descrito na Seção 5.1.1 e exemplificado na Figura 32.

possuem bits de *hash* em comum, também é responsável por agrupar todos os valores v associados à mesma chave k. O agrupamento de pares que possuem a mesma chave é realizado com a ordenação dos pares de acordo com k, e após ordenados, os pares em sequência que possuem a mesma chave têm seus valores v unificados em um único par KV.

5.9 Blocos de entrada de dados e blocos de saída de dados

Os blocos de entrada e saída de dados desempenham o interfaceamento do Sistema com o sistema de arquivos. O bloco de entrada de dados realiza a leitura de cada registro do arquivo de entrada definido, e o repassa à estrutura de saída do bloco, como indicado na Figura 42. O par KV de entrada possui como chave k o número do registro local lido, e o valor v corresponde aos dados lidos da entrada referente ao registro. O primeiro registro lido possui valor $k = 0$, e k é incrementado na medida em que novos registros são lidos do arquivo de entrada.

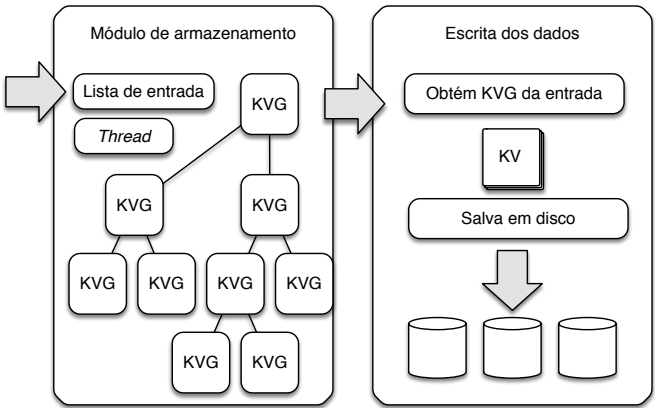
Figura 42: Bloco de entrada



Fonte: Próprio autor (2018)

O bloco de saída de dados desempenha atividade semelhante à do bloco de entrada, porém este trata de salvar o dado em disco a partir de valores armazenados na estrutura de entrada. A Figura 43 ilustra esse processo.

Figura 43: Bloco de saída



Fonte: Próprio autor (2018)

5.10 Blocos de armazenamento intermediário

A implementação dos blocos de armazenamento intermediário consiste basicamente em enviar ou receber do disco todo o conteúdo do módulo de armazenamento. A leitura ou escrita de dados utiliza a mesma abordagem do armazenamento de dados em sequência utilizada na troca de dados (Seção 5.7).

5.11 Processamento especulativo

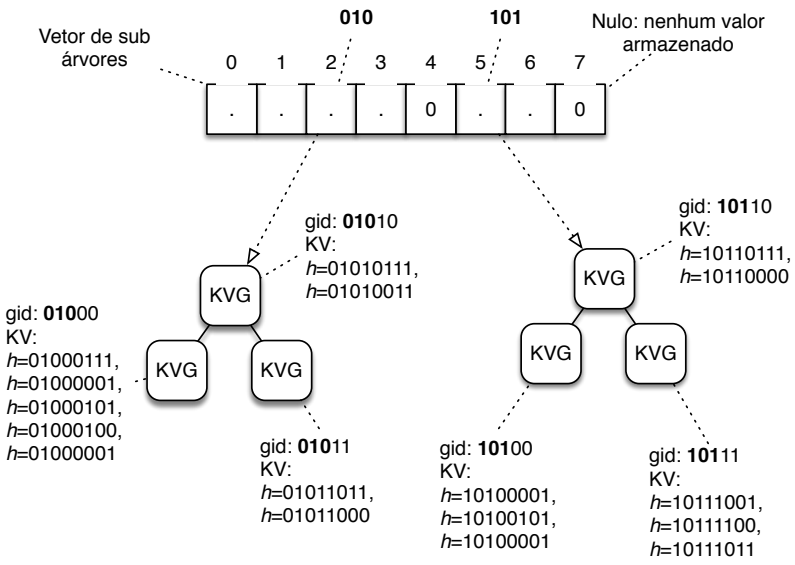
A implementação da estratégia de processamento especulativo consiste no armazenamento temporário das estruturas de entrada já processadas de cada bloco, com a relação de quais dados na estrutura de saída foram originados de cada estrutura KV na entrada. Desta forma, será possível efetuar o cálculo dos próximos blocos e atualizá-los de acordo com a necessidade.

Porém, esta técnica demanda uma maior quantidade de memória, e devido a isso, limita-se a dois o número de blocos consecutivos que possam fazer parte dessa simultaneamente.

Para que seja possível aplicar esta estratégia, é necessário verificar na saída do bloco de execução se um par com a mesma chave que a existente no par de entrada

já foi processado. Esta tarefa adiciona uma sobrecarga de processamento considerável ao sistema, visto que este processo deve ser realizado para todos os pares a serem processados. Desta forma, um mecanismo de busca multinível foi implementado, com o intuito de diminuir o tempo necessário para tais buscas. A busca multinível consiste em dividir a árvore de saída em s sub árvores rubro-negras, tal que as raízes das árvores estão armazenadas em um vetor de tamanho s. Com isto, a árvore com os dados de saída tem seu tamanho reduzido, otimizando a busca, e o acesso a cada sub árvore é realizado de maneira direta, visto que a posição do vetor em que a sub árvore se encontra é calculada com base nos *hashes* dos dados nela armazenados. O tamanho do vetor, s, é um valor tal que $s = 2^c$, onde c é o número de bits do *hash* a serem utilizados como índice no vetor. A Figura 44 ilustra este procedimento, para um caso em que $c = 3$. Caso um par KV procurado não possua uma sub árvore endereçada, a busca é finalizada, visto que nenhum grupo referente àquele par foi processado, e com isso o mecanismo de busca é otimizado, dado que diminui o número de comparações durante a pesquisa.

Figura 44: Busca de dados utilizando estrutura multinível



Fonte: Próprio autor (2018)

Caso já exista um par com a mesma chave k na estrutura de saída, os valores

associados a *k* são utilizados como parâmetro de entrada novamente para função UDF, onde o resultado da nova computação atualizará os valores correspondentes ao par na árvore de saída. O Algoritmo 4 apresenta o processamento e as atualizações de dados.

```
1  início
2      repita
3          Aguarda por elementos na árvore de entrada
4          Consome um elemento da árvore de entrada
5          se Já existe na estrutura de saída um elemento com a mesma chave
6           então
7              Retira o par da estrutura de saída
8              Chama função UDF utilizando os valores de entrada e o valor
                retirado da saída
9              Atualiza o valor na saída
10         fim
11         senão
12             Chama a função UDF utilizando apenas os valores de entrada
13             Inclui o par resultado na estrutura de saída
14         fim
15     até sempre;
16 fim
```

Algoritmo 4: Pseudocódigo do processamento especulativo

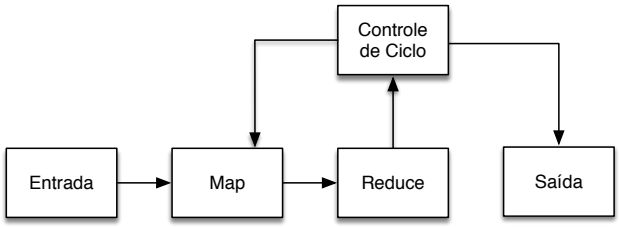
A otimização de desempenho desenvolvida na estratégia de processamento especulativo impõe algumas limitações às funções UDF. Para que a estratégia seja aplicada, é necessário que a função UDF possua propriedade associativa e também que a chave *k* do par resultante saída de seu processamento não tenha seu valor modificado. Tais limitações são necessárias porque o valor de saída é utilizado como parâmetro de entrada em um novo cálculo da função, e seu resultado deve ser equivalente ao cálculo realizado com todos os valores sendo utilizados como parâmetro em um único cálculo da função. Já a exigência de chaves iguais na entrada e saída dá-se pela necessidade de busca do par na saída da função. Esta limitação pode ser substituída por outra função que realize tal associação, porém neste trabalho optou-se por limitar a chaves iguais, por simplicidade de implementação.

5.12 Execução utilizando ciclos

Dentre as propostas do trabalho está a possibilidade de execução de ciclos no decorrer do fluxo de execução. Para tal, faz-se necessário que o fluxo de execução seja coordenado com a utilização de um Grafo Direcionado (DG).

A implementação do suporte a ciclos permite que mecanismos de iteração sejam utilizados, aplicando um critério de parada do ciclo. Um exemplo de fluxo de execução com ciclos é ilustrado na Figura 45.

Figura 45: Exemplo de fluxo de execução utilizando ciclos



Fonte: Próprio autor (2018)

Um bloco de controle de ciclos possui apenas duas saídas, sendo uma delas a continuidade no ciclo e a outra a saída do ciclo. A definição de qual caminho a informação percorrerá será dada pelo resultado do critério de parada, implementado utilizando contadores de iterações.

6 RESULTADOS E DISCUSSÃO

Neste capítulo são apresentados e discutidos os testes realizados com a implementação da estratégia desenvolvida.

6.1 Configurações utilizadas nos testes

As seções a seguir abordam informações necessárias para a interpretação dos resultados obtidos, como configurações das máquinas utilizadas e arquivos de entrada.

6.1.1 Ambiente de execução

Os testes foram executados em um *cluster* homogêneo, composto por 5 máquinas interligadas por uma rede de dados Gigabit Ethernet (FRAZIER, 1998). Cada máquina possui as seguintes especificações:

- Processador Intel Core^{TM} i7-3770 (INTEL, 2012), com frequência de 3.40 GHz, 8 MiB de memória cache, 4 núcleos de processamento e suporte a *hiperthreading*;
- 16 GiB de memória RAM;
- Disco rígido de 1.8 TiB, com barramento SATA 3.0;
- Sistema operacional Linux, versão 3.16.7, com distribuição OpenSuse (SUSE, 2018) versão 13.2.

6.1.2 Arquivos de entrada

As entradas foram consideradas de duas formas distintas: (a) cada máquina possui o seu arquivo de entrada único, e esse arquivo é lido apenas quando a máquina que o armazena está incluída no ambiente distribuído da aplicação. Aqui rotulamos esta entrada como entrada individual; e (b) a entrada de dados completa é processada, independente da quantidade de máquinas utilizada, que aqui chamaremos de entrada distribuída, visto que a carga de trabalho é dividida entre as máquinas presentes. A Tabela 1 apresenta a quantidade de dados considerada em cada execução, indicando o tamanho da entrada utilizada em cada máquina, de acordo com o número de processos sendo executados. Na Tabela 1, os valores 1_m a 5_m indicam que a execução é realizada utilizando de 1 a 5 máquinas no ambiente distribuído. Os valores apresentados são expressos em MiB.

Tabela 1: Número de máquinas e tamanho da entrada utilizado em cada teste, expresso em MiB

	Entrada individual					Entrada distribuída				
Máquina	1_m	2_m	3_m	4_m	5_m	1_m	2_m	3_m	4_m	5_m
1	64	64	64	64	64	320	192	128	128	64
2	x	64	64	64	64	x	128	128	64	64
3	x	x	64	64	64	x	x	64	64	64
4	x	x	x	64	64	x	x	x	64	64
5	x	x	x	x	64	x	x	x	x	64
Total	64	128	192	256	320	320	320	320	320	320

Por exemplo, uma execução utilizando duas máquinas possui 128 MiB de dados processados na entrada individual, dado que as máquinas 1 e 2 processariam 64 MiB cada uma. Por outro lado, na entrada distribuída são processados 320 MiB, com 192 MiB processados na máquina 1 e 128 MiB na máquina 2.

A utilização destes diferentes tipos de entrada justifica-se pelo contexto de aplicação da estratégia. A entrada individual não é prática para execuções reais de *Big Data*, devido às limitações de memória e barramentos. Seu propósito neste trabalho é apenas para comparações de desempenho.

Os parâmetros utilizados como padrão para todos os testes estão apresentados na Tabela 2. Para situações em que valores diferentes destes forem empregados, tais informações serão explicitamente descritas em cada caso.

Tabela 2: Parâmetros utilizados por padrão

Parâmetro	**Configuração**
Máscara de granularidade	8 bits
Número de *threads* processando Map	4 *threads*
Número de *threads* processando Reduce	4 *threads*

6.2 Descrição dos testes realizados

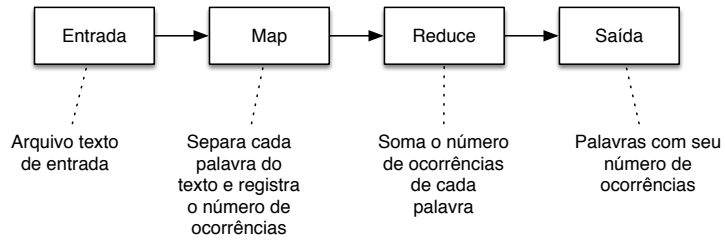
Foram realizados dois testes distintos para a análise de resultados. O primeiro teste consiste em uma aplicação responsável por contar a quantidade de palavras existentes em uma dada entrada. O segundo teste, além do resultado do primeiro informa também a quantidade de palíndromos presentes na entrada, gravando esse resultado em uma outra saída. O objetivo do primeiro teste é observar parâmetros de desempenho e compará-los com o *framework* Apache Hadoop, enquanto o segundo permite observar a versatilidade e desempenho do sistema utilizando as múltiplas entradas e saídas, com diferentes combinações na ordem dos blocos executados. Para o teste que explora o comportamento da aplicação utilizando os blocos de armazenamento intermediário, a funcionalidade de ciclos de execução é utilizada, visto que há um ciclo que percorre os diversos blocos armazenados na máquina e o processa. Utilizando a aplicação de contagem de palavras, também foram realizados testes em um ambiente de máquinas

heterogêneo, cujas configurações das máquinas é apresentada junto aos resultados do teste. Adicionalmente foram realizados testes e análises especificamente sobre a transferência de dados nas aplicações, bem como com blocos de outros tamanhos, com o intuito de observar o comportamento do Sistema nessas condições.

6.2.1 Contagem de palavras

O primeiro teste consiste em utilizar uma aplicação Mapreduce, de maneira que na fase de Map o texto da entrada é separado em palavras, de forma que a palavra é considerada como k e a quantidade ocorrências como v. Posteriormente na fase de Reduce são somados todos os valores v_i associados à k, resultando na quantidade de ocorrências de cada palavra. A Figura 46 ilustra tal funcionamento.

Figura 46: Aplicação de contagem de palavras



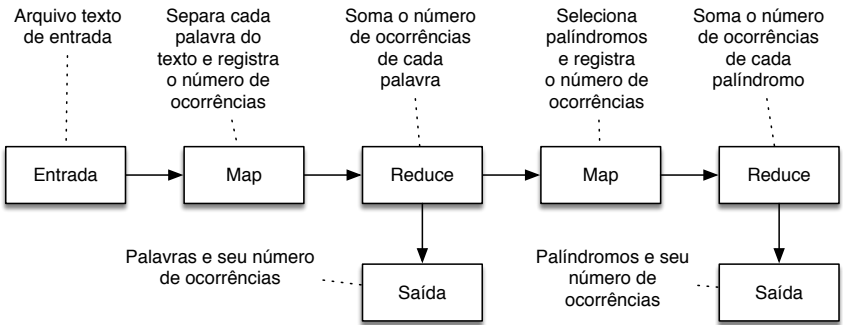
Fonte: Próprio autor (2018)

6.2.2 Contagem de palavras com contagem de palíndromos

O teste envolvendo a contagem de palavras e palíndromos objetiva testar o funcionamento do modelo de programação e o desempenho do ambiente de execução utilizando diferentes configurações e variações dos blocos no fluxo de execução. Além da contagem de palavras, a aplicação deve apresentar uma saída contendo os palíndromos encontrados. Um palíndromo é uma sequência de caracteres simétrica, ou seja, a mesma sequência é obtida se estes forem acessados na ordem inversa. Nos testes realizados, são contadas as ocorrências de palavras da entrada que têm esta característica.

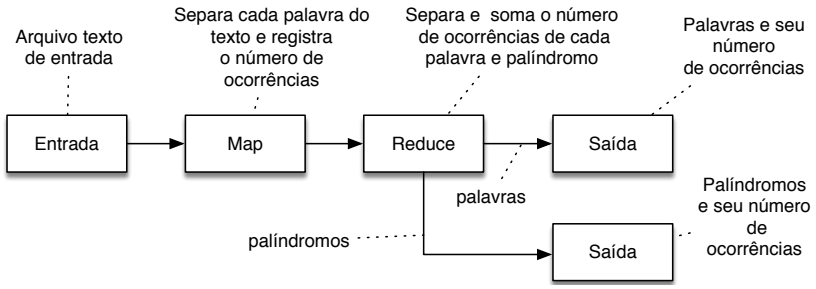
Foram desenvolvidas quatro versões desta aplicação, rotuladas de v_1, v_2, v_3 e v_4, apresentadas nas Figuras 47, 48, 49 e 50. A versão v_1 utiliza duas aplicações Mapreduce encadeadas, gravando em disco o resultado intermediário do contador de palavras. A versão v_2 incorpora o reconhecimento do palíndromo na mesma UDF utilizada no processo de Reduce da contagem de palavras, emitindo os resultados diretamente para duas saídas distintas. A versão v_3 reconhece se a palavra de entrada é palíndromo na operação de Map, e caso seja, repassa o resultado para um Reduce específico. Esse Reduce conta as ocorrências e grava o resultado em uma saída. A versão v_4 ocorre de maneira semelhante à v_3, porém o mesmo código FDU é instanciado para as duas tarefas de Reduce. O objetivo é reutilizar códigos do usuário em diferentes pontos da aplicação.

Figura 47: Contagem de palavras e palíndromos, versão 1



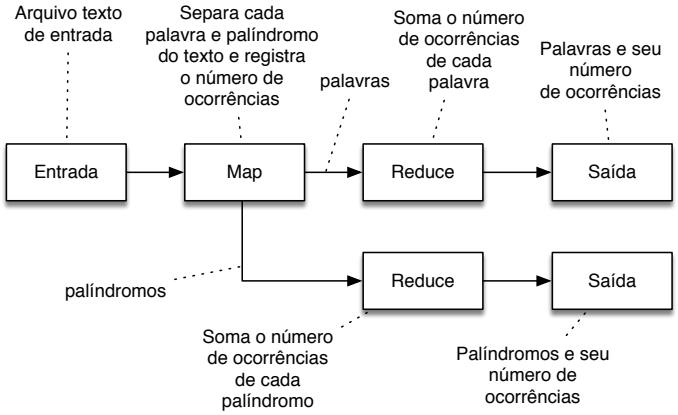
Fonte: Próprio autor (2018)

Figura 48: Contagem de palavras e palíndromos, versão 2



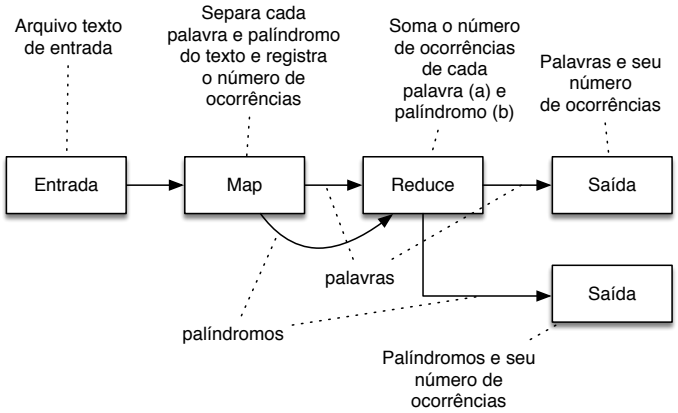
Fonte: Próprio autor (2018)

Figura 49: Contagem de palavras e palíndromos, versão 3



Fonte: Próprio autor (2018)

Figura 50: Contagem de palavras e palíndromos, versão 4



Fonte: Próprio autor (2018)

6.3 Resultados

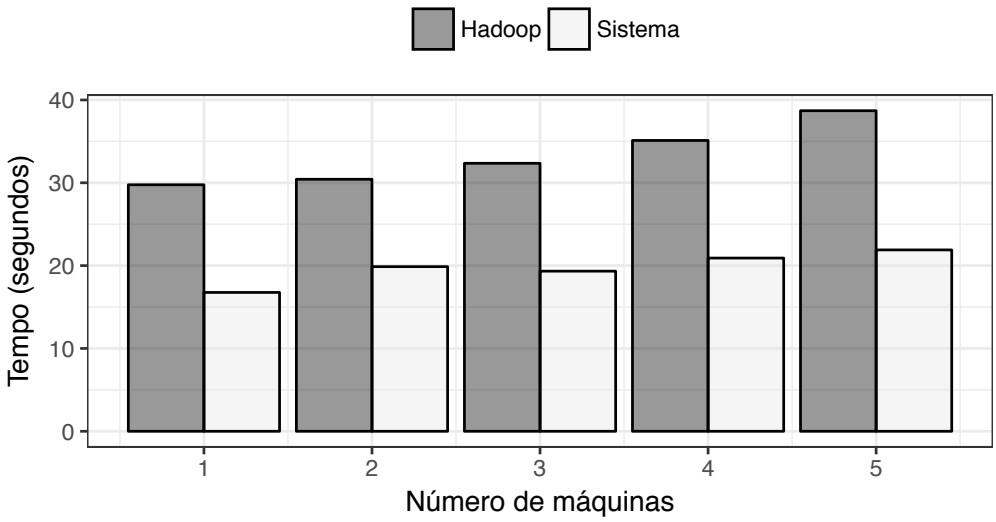
As seções a seguir apresentam os resultados obtidos nos testes, separados de acordo com diferentes aspectos da estratégia desenvolvida. As métricas utilizadas foram tempo de execução, *speedup* e eficiência. A definição de cada métrica utilizada pode ser conferida no Apêndice B.

6.3.1 Programação em blocos

Os resultados dos testes utilizando a programação em blocos visam observar o comportamento da aplicação utilizando diferentes parâmetros do ambiente de execução e seu desempenho.

6.3.1.1 Contagem de palavras

Figura 51: Tempo de execução



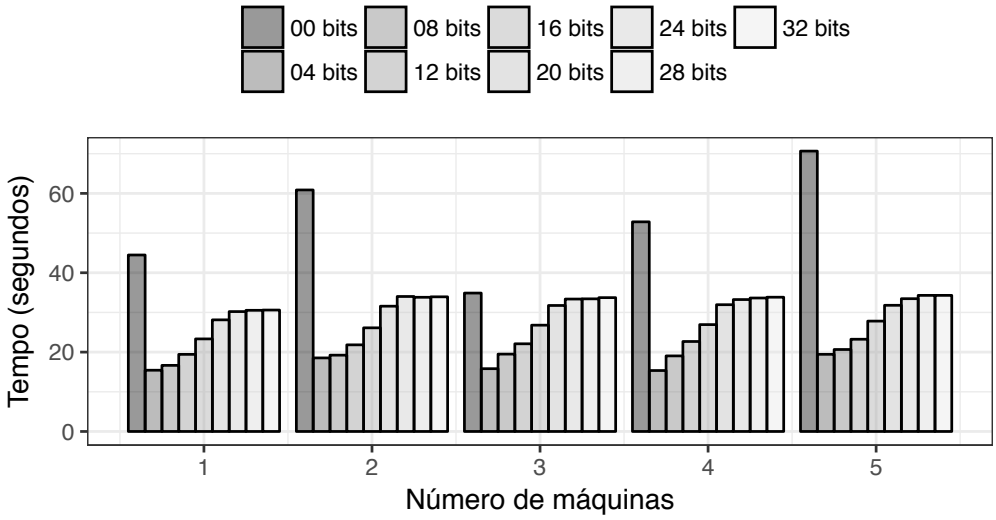
Fonte: Próprio autor (2018)

No gráfico apresentado na Figura 51 observa-se que o tempo de execução utilizando o modelo proposto toma de 56 a 65% do tempo de execução da aplicação realizada utilizando o Apache Hadoop. Isto dá-se principalmente pelo processamento de dados em memória, otimizações nas trocas de dados, e também pela não utilização de mecanismos de tolerância a falhas, que implicam em maior quantidade de acessos a disco e dados transferidos, fatores estes associados à degradação de desempenho. Observa-se também que o tempo de execução é proporcional ao número de máquinas utilizadas. Isto dá-se pelo aumento na carga de trabalho à medida que mais máquinas são utilizadas, como indicado na Tabela 1.

A Figura 52 apresenta o gráfico do tempo de execução utilizado pela aplicação considerando diferentes quantidades de bits do *hash* para o agrupamento de estruturas KV. No gráfico observa-se que os menores tempos de execução foram alcançados com máscara de granularidade com 4 bits. Isto dá-se principalmente à limitação no tamanho da árvore que armazena as estruturas. Máscaras com um menor número de bits indicam que uma menor quantidade de elementos na árvore será utilizada, e consequentemente, mais rápido será o acesso aos dados no primeiro nível da fase de agrupamento. A quantidade de pares a serem agrupados no segundo nível é inversamente proporcional ao tamanho da árvore, porém não é necessário realizar buscas na estrutura de lista presente no KVG, visto que o agrupamento de valores com base na chave é realizado com um vetor indexado pelo *hash* da chave *k*.

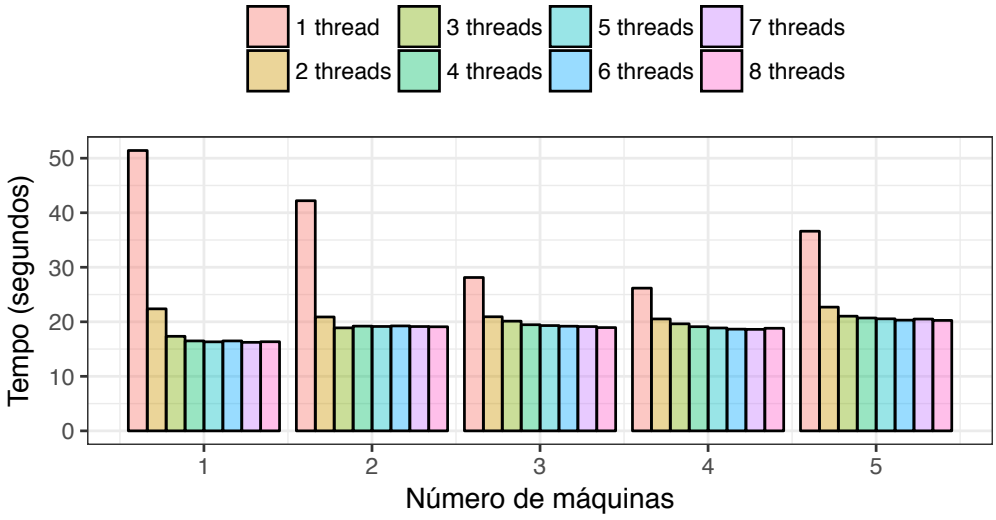
Os tempos de execução mais baixos foram atingidos utilizando os extremos da máscara de controle de granularidade. Com 0 bits, a árvore possui um único elemento, sobrecarregando o segundo nível da fase de agrupamento. Neste nível são tratadas as colisões de *hash*, bem como possíveis colisões no vetor utilizado para o agrupamento, e a resolução dessas colisões ocasionou em queda de desempenho. Entretanto, utilizando 32 bits na máscara, o primeiro nível de agrupamento é sobrecarregado, pois a árvore atinge seu tamanho máximo, e o tempo necessário para percorrer e balancear a árvore, inserir e remover elementos torna-se fator negativo para o desempenho da aplicação.

Figura 52: Tempo de execução utilizando diferentes valores para máscara de granularidade



Fonte: Próprio autor (2018)

Figura 53: Tempo de execução utilizando diferentes números de *threads*



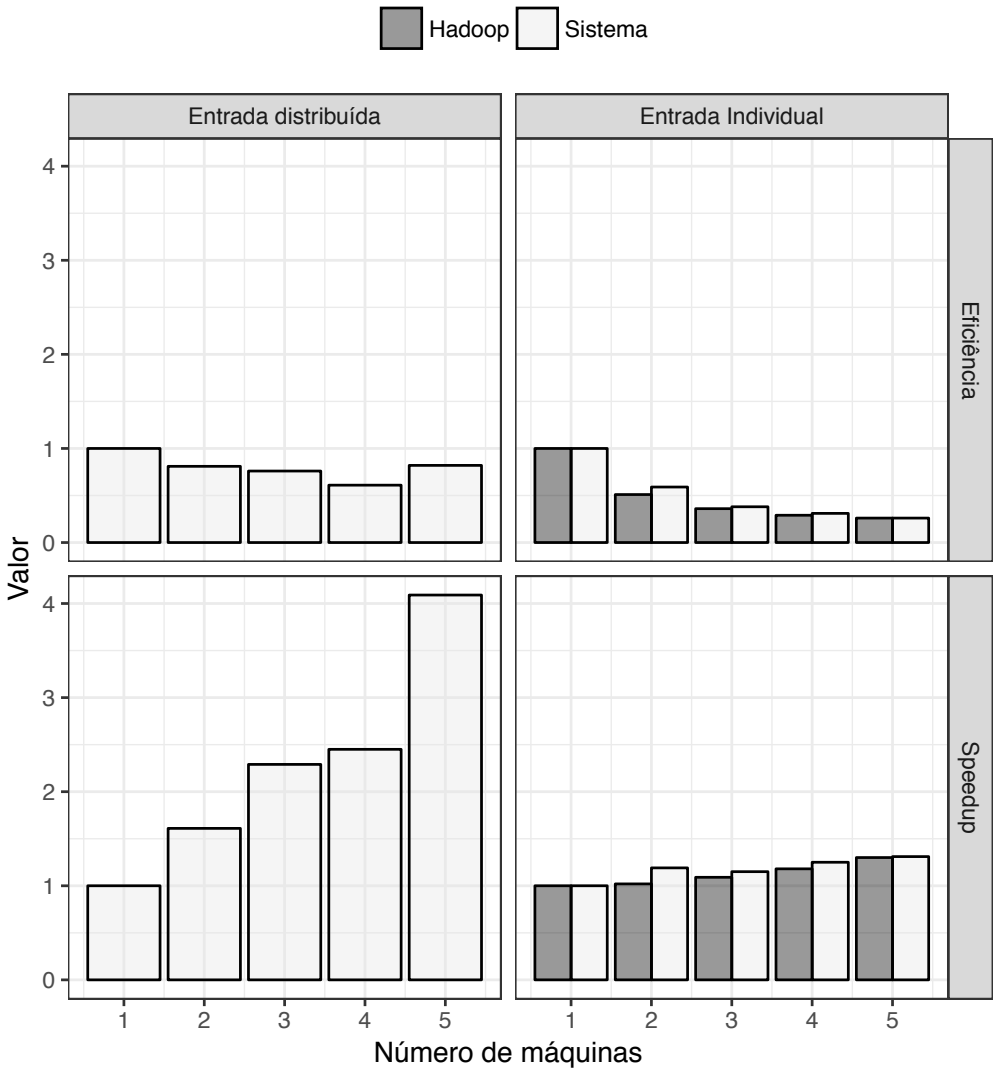
Fonte: Próprio autor (2018)

A quantidade de *threads* presente na aplicação também influencia no desempenho, visto que um baixo número destas pode levar a unidades de processamento ociosas, e um alto número pode acarretar em concorrência no acesso a recursos, realizando

excessivas trocas de contexto para intercalar o acesso à unidade de processamento. No gráfico apresentado na Figura 53 pode-se observar que os menores desempenhos ocorreram utilizando 1 e 2 *threads*. Por outro lado, utilizando de 4 a 8 *threads* não há uma diferença significativa no desempenho, visto que o *hardware* de processamento das máquinas possui 4 núcleos com suporte a *hiperthreading*, otimizando execuções paralelas e concorrentes. Outro fator limitante para o processamento *multithread* na aplicação é o acesso concorrente das *threads* que executam as UDFs à árvore de entrada com os pares KV em cada bloco de execução.

Internamente no ambiente de execução implementado há diversas outras *threads* operando, em tarefas como envio e recebimento de dados, sincronização de blocos, dentre outras atividades. Porém o acesso concorrente à árvore de entrada faz com que estas *threads* internas tenham o acesso às unidades de processamento com baixa concorrência. Espera-se que este comportamento possa ser diferente para aplicações em que o processamento realizado pelas operações Map e Reduce possuam maior carga de processamento.

Figura 54: Speedup e eficiência



Fonte: Próprio autor (2018)

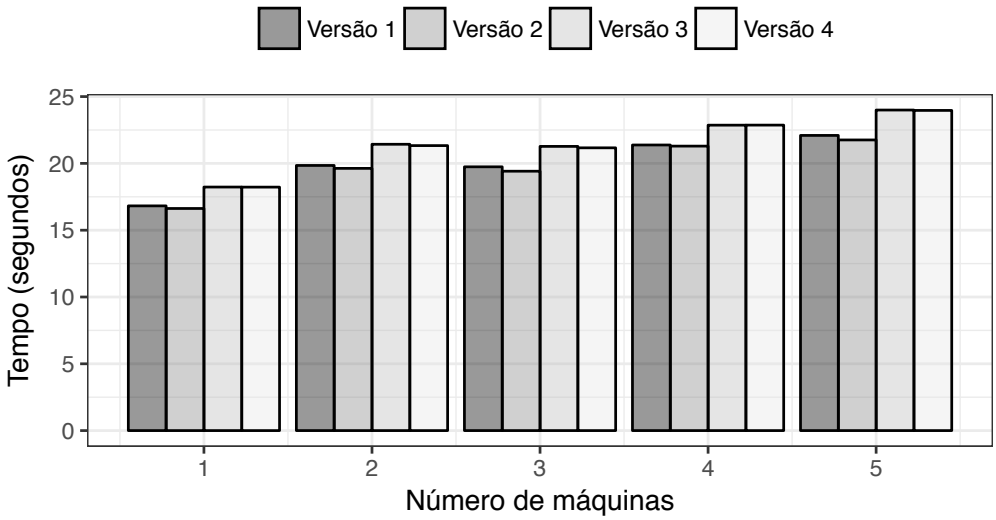
Os gráficos da Figura 54 apresentam os valores de *speedup* e eficiência para a aplicação utilizando o modelo proposto e o Apache Hadoop, empregando a entrada individual. Observa-se que o *speedup* é crescente para ambas as aplicações testadas, sendo superior para a aplicação que utiliza o modelo apresentado, e a diferença de *speedup* diminui à medida que o número de máquinas aumenta. Ressalta-se que o *speedup* é calculado utilizando os tempos paralelo e sequencial de cada aplicação, e apesar dos valores serem semelhantes, o tempo de execução da aplicação utilizando a

estratégia proposta é inferior, como mostrado na Figura 51.

Ainda na Figura 54 são apresentados os valores de *speedup* e eficiência, executados utilizando a entrada distribuída. Visto que a carga de trabalho é partilhada entre as máquinas, os valores de *speedup* são muito maiores que os vistos na Figura 54. Observa-se também que para 3 e 4 processos o *speedup* não teve aumento significativo e a eficiência apresentou a maior queda. Isto ocorre porque a máquina 1 possui o maior tamanho de arquivo em ambos os casos, e o processamento destes dados atua como gargalo no sistema, mantendo o mesmo tempo de execução. Assim, ao utilizar 4 máquinas, mais recursos de processamento mantêm-se ociosos, justificando a queda de eficiência.

6.3.1.2 Contagem de palavras com contagem de palíndromos

Figura 55: Tempo de execução de contagem de palavras e palíndromos



Fonte: Próprio autor (2018)

Os tempos de execução das aplicações de contagem de palavras e palíndromos estão expressos no gráfico da Figura 55. Neste, observa-se as diferenças no desempenho das diversas versões implementadas. As versões v_1 e v_2 possuem desempenho

semelhante, sendo a v_2 ligeiramente melhor. Verifica-se que as principais diferenças entre v_1 e v_2 são o número de blocos de execução e a concorrência no acesso a disco. v_1 e v_2 possuem 4 e 2 blocos de execução, respectivamente. Um maior número de blocos indica mais árvores com pares (k, v), e consequentemente, mais agrupamentos. Quando à escrita de dados, v_1 escreve os dados em momentos diferentes, enquanto v_2 escreve-os no mesmo instante, concorrendo pelo acesso ao disco local. Observando estes pontos, percebe-se que o maior número de blocos degrada mais o desempenho que a concorrência no acesso a disco, neste caso.

Comparando os resultados obtidos com as versões v_3 e v_4 verifica-se que possuem desempenho equivalentes, e a reutilização de códigos UDF pôde ser utilizada. Comparadas à v_1 e v_2, v_3 e v_4 possuem desempenho inferior, devido ao maior número de dados trafegados pela rede, visto que na operação de Map os dados são enviados para reduces diferentes.

6.3.1.3 Blocos de armazenamento intermediário

O desempenho obtido nas aplicações com a utilização dos blocos de armazenamento intermediário foi testado utilizando diversos blocos de entrada por máquina e também diferentes quantidades de *threads* realizando o processamento da aplicação. Os resultados de tais testes estão expressos nos gráficos da Figura 56. Nos gráficos, a linha pontilhada indica o tempo obtido com o mesmo teste utilizando a ferramenta Hadoop. O número de blocos indicado corresponde à quantidade de blocos de entrada por máquina, como também o número de blocos de armazenamento intermediário utilizados.

Utilizando apenas um bloco, o tempo de execução não difere significativamente entre as execuções com ou sem a estratégia de armazenamento intermediário. Apenas para o caso em que uma *thread* é utilizada a execução com Hadoop obteve menor tempo de execução. Para execuções utilizando dois blocos, a estratégia torna-se vanta-

josa com a utilização de 4 e 5 nós, onde os menores tempos de execução obtidos foram utilizando 4, 5 e 6 *threads*. Esta diferença dá-se principalmente pela melhor utilização de recursos de processamento, possibilitando execuções de tarefas paralelamente. As execuções utilizando 4 blocos tornam-se vantajosas para processamentos utilizando 5 máquinas com 2 ou mais *threads*.

Ao comparar as execuções do Sistema com e sem o armazenamento intermediário, observa-se que o impacto no tempo de execução não é muito significativo, viabilizando a utilização da estratégia. Para algumas execuções, em maioria aquelas com 4 blocos e 4, 6 e 8 *threads* o tempo de execução com o armazenamento intermediário é inferior ao das demais execuções. Isto dá-se pelo menor uso de operações envolvendo alocação de memória, visto que com o uso dos blocos de armazenamento intermediário uma menor quantidade de estruturas de dados são mantidas simultaneamente em memória, levando a um melhor uso da estratégia de banco de memória. Neste caso, o tempo envolvido com leituras e escritas a disco é menor que o tempo utilizado em alocações de estruturas de memória.

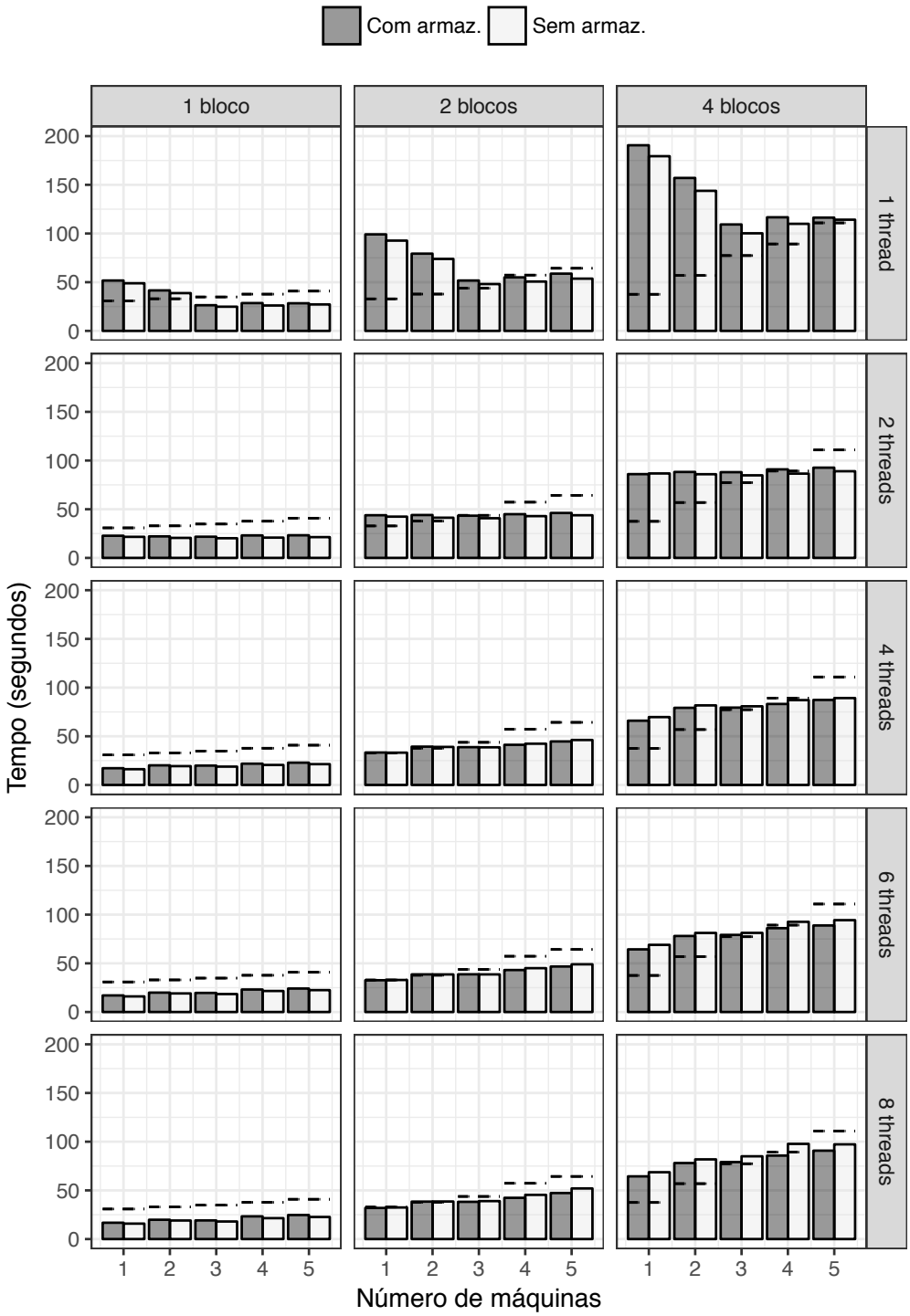
6.3.2 Processamento especulativo

Os experimentos realizados contaram com a execução da aplicação que realiza a contagem de palavras, utilizando como entrada arquivos de 64 MiB, de maneira que cada máquina presente no ambiente distribuído possui um arquivo deste tamanho, e com isto, a quantidade total de dados processados é obtida em função do número total de máquinas presentes na execução da aplicação.

A Figura 57 apresenta os tempos de execução obtidos utilizando o processamento especulativo, comparando com a ferramenta Hadoop e com a própria ferramenta sem a utilização da estratégia proposta, executando uma aplicação que realiza a contagem de palavras.

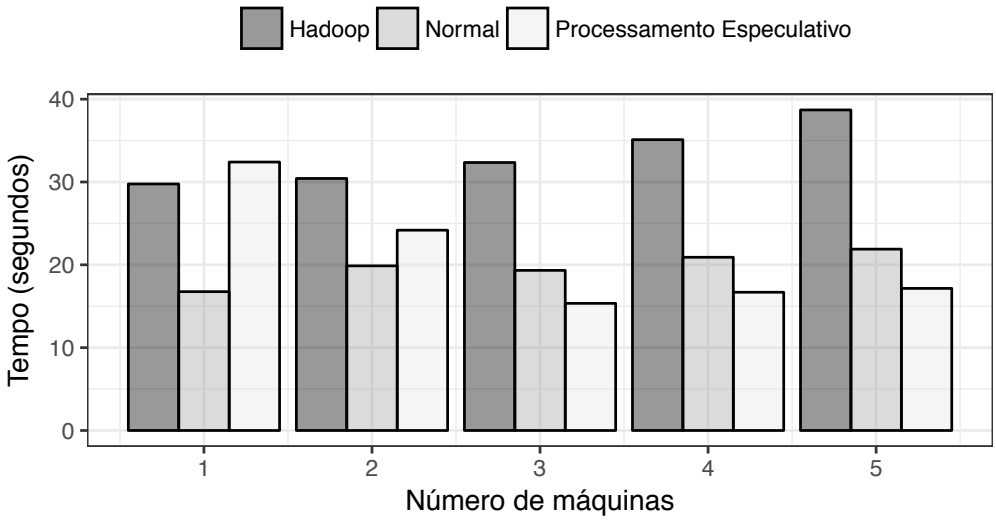
No gráfico é possível observar que o tempo de execução utilizando a ferramenta

Figura 56: Execuções utilizando blocos de armazenamento temporário, com múltiplos blocos e múltiplas *threads*



Fonte: Próprio autor (2018)

Figura 57: Processamento especulativo

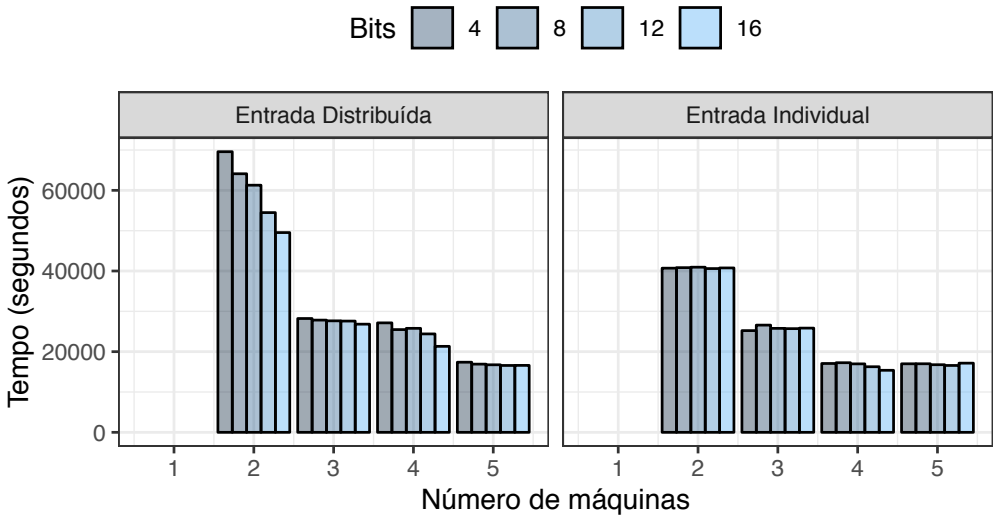


Fonte: Próprio autor (2018)

desenvolvida é menor que o tempo utilizado pela ferramenta Hadoop. Também verifica-se o tempo obtido utilizando processamento especulativo, no qual o maior desempenho obtido foi $21,7\%$ de ganho em relação ao tempo de execução sem a estratégia.

Pode-se observar também a influência que a comunicação de dados possui na estratégia aplicada. Visto que o tempo de espera durante o recebimento de dados é aproveitado para a verificação se o par sendo processado atualmente já foi processado em tempo anterior. Desta forma, para 1 e 2 processos a estratégia não mostrou-se eficiente, visto o pequeno número de troca de dados e grande tamanho das árvores que armazenam os pares (k, v), aumentando assim o tempo de busca na árvore. Considerando que a quantidade de dados aumenta e o tamanho da árvore diminui na medida em que mais máquinas são utilizadas, este equilíbrio atingiu o ponto com menor tempo de execução utilizando 3 máquinas. Com 4 e 5 máquinas, o tempo de execução foi ligeiramente maior se comparado ao tempo obtido utilizando 3 processos, mas expressivamente menor se comparado às execuções sem a utilização da estratégia de processamento especulativo.

Figura 58: Processamento especulativo com diferentes quantidades de bits



Fonte: Próprio autor (2018)

A quantidade de bits utilizada no vetor de busca de pares já processados pode influenciar o desempenho das aplicações, e o menor tempo obtido ocorreu com a execução utilizando 16 bits com entrada individual e 4 processos, como apresentado na Figura 58. Para a entrada distribuída, o impacto causado pela diferença no tamanho do vetor é significativo, porém esta diferença diminui na medida em que mais máquinas são utilizadas. Já para o processamento individual tal diferença não possui o mesmo impacto, visto que grande parte do tempo de execução é concentrado na execução das tarefas, e não na busca em si. Considerando que as execuções são feitas utilizando 6 *threads*, a aplicação não permanece bloqueada durante a busca, dado que outras *threads* realizam o processamento das tarefas nesse tempo.

6.3.3 Ambiente heterogêneo

Com objetivo de testar a escalabilidade do Sistema em ambiente heterogêneo, três máquinas foram adicionadas ao *cluster*. Além das 5 máquinas já utilizadas nos testes anteriores, aqui denotadas como M_1, M_2, M_3, M_4 e M_5, foram adicionadas as máquinas M_6, M_7 e M_8, com as seguintes configurações:

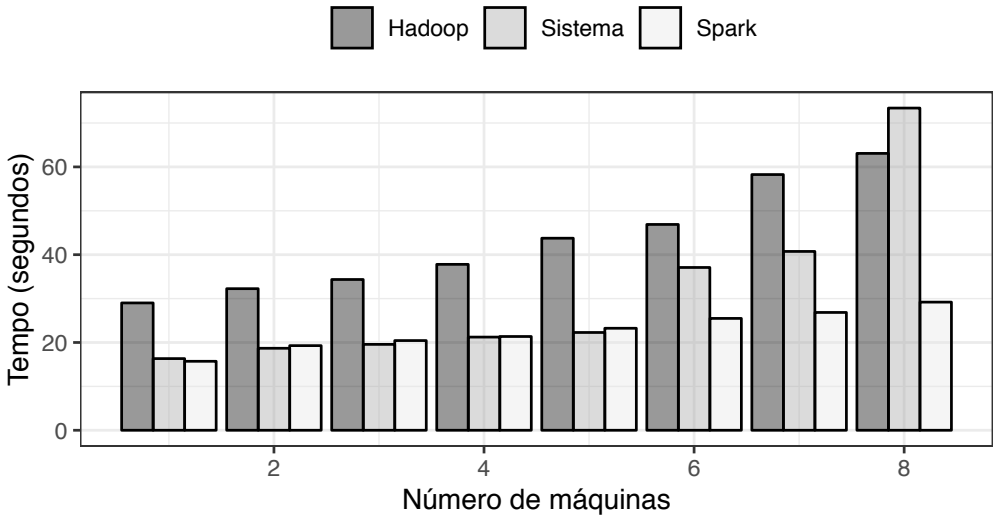
M_6 e M_7:
- Processador Intel Xeon E52620 (INTEL, 2012), com frequência de 2.00 Ghz, 15 MiB de memória cache e suporte a *hiperthreading*;
- 32 GiB de memória RAM;
- Disco rígido de 1 TiB;
- Sistema operacional Linux, versão 3.10.0, com distribuição Centos versão 7.3.

M_8:
- Processador Intel Xeon E5335 (INTEL, 2007), com frequência de 2.00 Ghz, 4 MiB de memória cache e suporte a *hiperthreading*;
- 8 GiB de memória RAM;
- Disco rígido de 250 GiB;
- Sistema operacional Linux, versão 2.6.32, com distribuição Centos versão 6.5.

As máquinas estão interligadas em uma mesma rede local, interconectados por um *switch* Gigabit Ethernet. Um teste empírico de transferência de dados foi realizado entre todas as máquinas, com o envio e recebimento de um arquivo de 10 GiB, utilizando a ferramenta `scp`. As máquinas M_1 a M_7 mostraram velocidade de comunicação de 110 MiB/s, enquanto que a máquina M_8 apresentou velocidade de 50 MiB/s. Desta forma, esta característica também foi utilizada na análise dos resultados obtidos.

Observando os resultados apresentados no gráfico da Figura 59, verifica-se que em ambientes homogêneos, o Sistema possui o menor tempo de execução, com exceção da execução utilizando uma única máquina. Nas execuções utilizando de 6 a 8 máquinas, o uso das estratégias de balanceamento de carga, principalmente do Spark torna as execuções utilizando tal *framework* mais rápidas neste cenário. Para o caso da execução utilizando 8 máquinas, o Hadoop também teve desempenho superior ao do Sistema, dado sua aplicação de balanceamento de carga utilizando compartimentos.

Figura 59: Comparação entre sistemas em ambiente heterogêneo

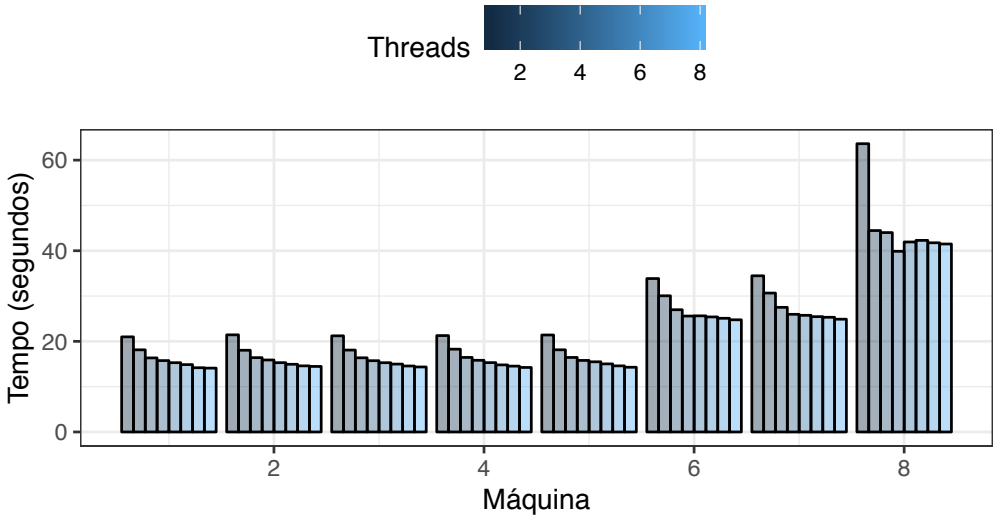


Fonte: Próprio autor (2018)

Visando observar o comportamento da aplicação sob diferentes características físicas das máquinas, o gráfico da Figura 60 demonstra o tempo de execução do Sistema utilizando uma única máquina e diversas *threads*. Neste teste, não há comunicação entre as máquinas, ou seja, o tempo gasto para a execução das aplicações é composto majoritariamente por acessos ao disco e processamento. Observa-se que nesta situação, a utilização de *threads* leva ao aumento do desempenho, considerando que em todos os casos, o tempo de execução diminui à medida que o número de *threads* aumenta. Convém ressaltar que o número de núcleos de processamento disponível nas máquinas é maior ou igual que o número de *threads* utilizadas para o processamento das tarefas. Nas execuções destes testes, é processado um arquivo de 64 MiB.

Na Figura 61, o gráfico demonstra a influencia do número de *threads* no tempo de execução do Sistema executado no *cluster* heterogêneo. Observa-se que no ambiente homogêneo, utilizando até três máquinas, manter um número alto de *threads* torna-se uma abordagem vantajosa. Porém, a partir de quatro máquinas esta situação se inverte, visto que os dados são processados, e as *threads* do Sistema responsáveis pela comunicação entre as máquinas e a organização dos pares KV nas estruturas em

Figura 60: Execuções utilizando apenas uma máquina separadas por máquina

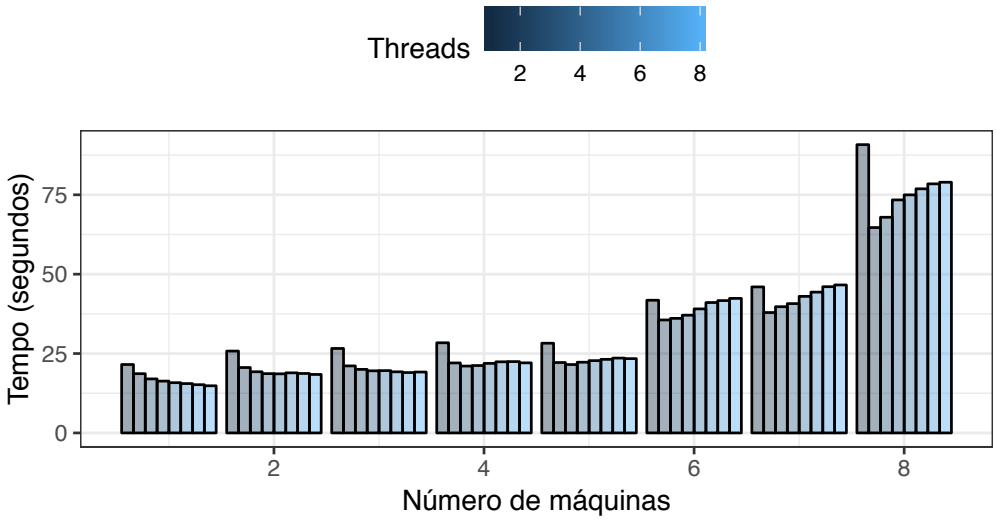


Fonte: Próprio autor (2018)

árvore utilizam maior tempo de CPU, fator este que influencia diretamente no tempo de execução das aplicações, visto a maior concorrência das *threads* pela unidade de processamento. Quanto às execuções com mais de seis máquinas, a utilização de duas *threads* obteve os menores tempos de execução. Isto ocorre devido ao pouco processamento nos códigos UDF, processados pelas *threads* nas tarefas. Pouco processamento nesta situação ocasiona uma maior concorrência das *threads* às estruturas de armazenamento, indicando que existe um gargalo no processamento do Sistema para o teste realizado. Outro ponto também observado é a diferença nos tempos de execução entre máquinas heterogêneas. Utilizando de 1 a 5 máquinas, o tempo de execução difere pouco. Ao utilizar 6 e 7 máquinas, ou seja, as máquinas M_6 e M_7 são incluídas no processamento da aplicação, e ao utilizar 8 máquinas, a máquina M_8 é também é adicionada. A discrepância de tempos entre as execuções com 5 e 6 máquinas e com 7 e 8 máquinas indicam que o tempo de processamento é limitado pelas máquinas mais lentas, visto que as demais máquinas com maior capacidade de processamento permanecem ociosas aguardando as demais finalizarem suas tarefas.

Os resultados apresentados no gráfico da Figura 62 correspondem a um teste rea-

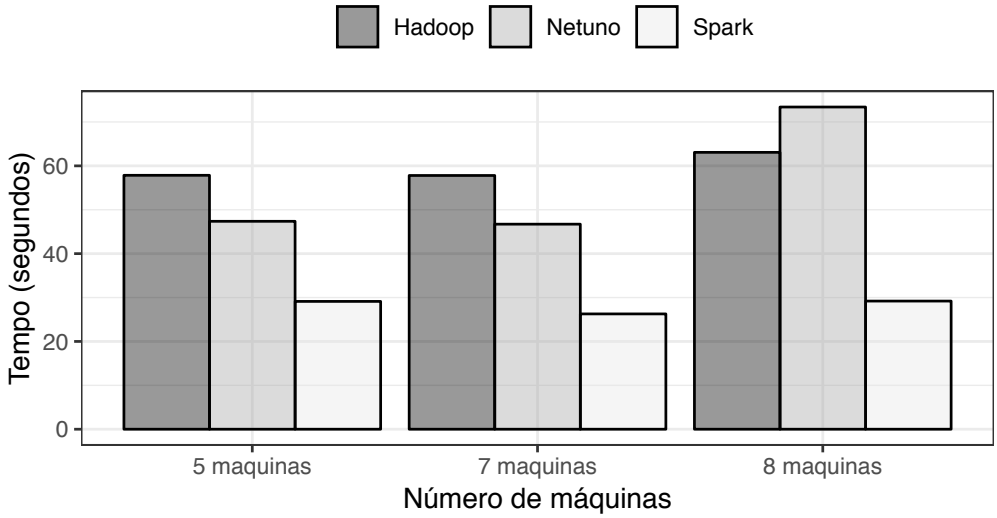
Figura 61: Tempo de execução em ambiente heterogêneo utilizando diferentes quantidades de *threads*



Fonte: Próprio autor (2018)

lizado com a finalidade de descobrir se o desempenho aumenta com a inclusão de uma máquina de menor processamento, ou se é mais vantajoso incluir uma maior carga de entrada de dados às máquinas já existentes. Neste caso, foi processada uma entrada de 8 blocos, onde na execução com 8 máquinas cada máquina processa 1 bloco, na execução com 7 máquinas a máquina M_1 processa 2 blocos e na execução com 5 máquinas, as máquinas M_1, M_2 e M_3 processam 2 blocos. No gráfico é observado que os tempos de execução que não utilizaram a máquina M_8 obtiveram os melhores desempenhos. Tal resultado ocorre porque ao incluir a máquina M_8, as demais permanecem ociosas por um período até que esta processe suas tarefas. O resultado mostrou que, para esta situação, com processos utilizando uma menor quantidade de máquinas, melhor desempenho foi obtido, visto que o tempo de ociosidade das máquinas foi utilizado para processamento das tarefas. Ressalta-se ainda outro fator degradante para o processamento incluindo a máquina M_8, que é sua velocidade de transferência de dados mais baixa, influenciando diretamente o desempenho.

Figura 62: Tempo de execução em ambiente heterogêneo utilizando a mesma quantidade de dados de entrada



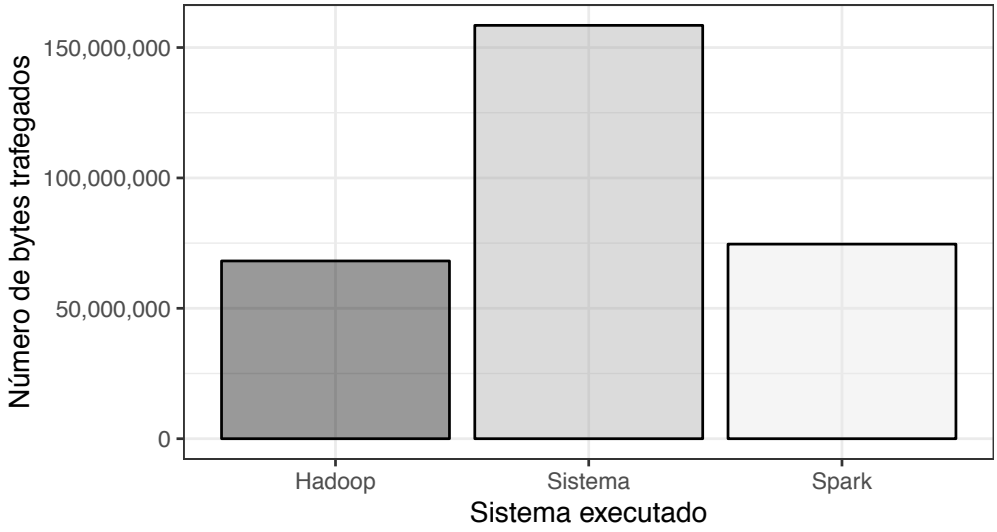
Fonte: Próprio autor (2018)

6.3.4 Análise de tráfego

Outro aspecto observado no Sistema é a quantidade de dados trafegados, visto que o uso da comunicação de dados comumente possui grande impacto no desempenho das aplicações. As informações foram coletadas em uma execução utilizando 5 máquinas, onde foi utilizada o *software* `Tcpdump` (TCPDUMP, 2018) para coletar os dados trafegados pela rede na máquina M_1. Os resultados apresentados nas Figuras 63, 64 e 65 são referentes à comunicação entre as máquinas M_1 e M_4, escolhidas aleatoriamente. Após a coleta dos dados trafegados, foi utilizado o software `Wireshark` (WIRESHARK, 2018) para análise das informações, e observou-se a quantidade de bytes trafegados pelas conexões TCP, da camada de transporte.

A quantidade de bytes trafegados pelo Sistema é muito maior que a quantidade trafegada pelo Spark e Hadoop, como pode ser observado na Figura 63. Este alto número de dados dá-se pelas informações adicionais que são incluídas, como mostrado na Figura 38. Outro ponto observado considera o número de pacotes TCP trafegados. O gráfico da Figura 64 mostra que o número de pacotes trafegados pelo Sistema é

Figura 63: Quantidade de bytes trafegados



Fonte: Próprio autor (2018)

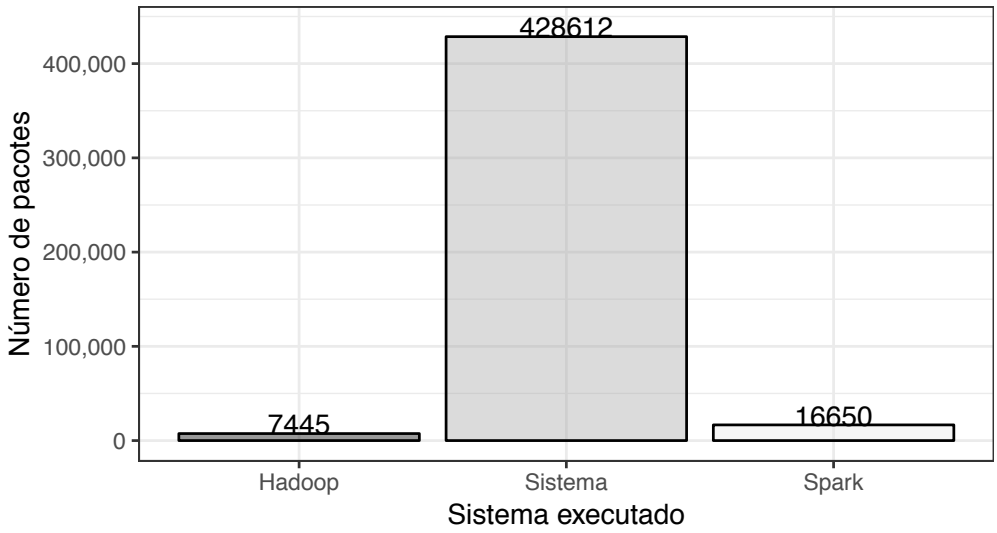
muito maior que pelo Hadoop ou Spark. Isto ocorre porque o Sistema não possui um gerenciamento de tráfego de dados dedicado, apenas retirando os elementos KVG na árvore com os dados, convertendo-os em vetores de bytes enviando-os pela rede.

Sabendo que uma grande quantidade de dados foi trafegada pelo Sistema e também um grande número de pacotes foi enviado, observou-se também qual o tamanho desses pacotes trafegados. A Figura 64 mostra um gráfico (com eixo y em escala logarítmica de base 10) contendo a comparação de tamanho de pacotes trafegados pelo Sistema, Hadoop e Spark. Em todos os tamanhos de pacotes observados, o Sistema trafegou uma quantidade muito superior aos demais *frameworks*, principalmente com pacotes de 80 a 1279 bytes. Observando o gráfico, percebe-se que o problema da alta quantidade de dados trafegados está associado principalmente aos dados adicionais incluídos no pacote, e não ao número de envio relativos a um tamanho de pacote específico.

6.3.5 Outras quantidades de dados

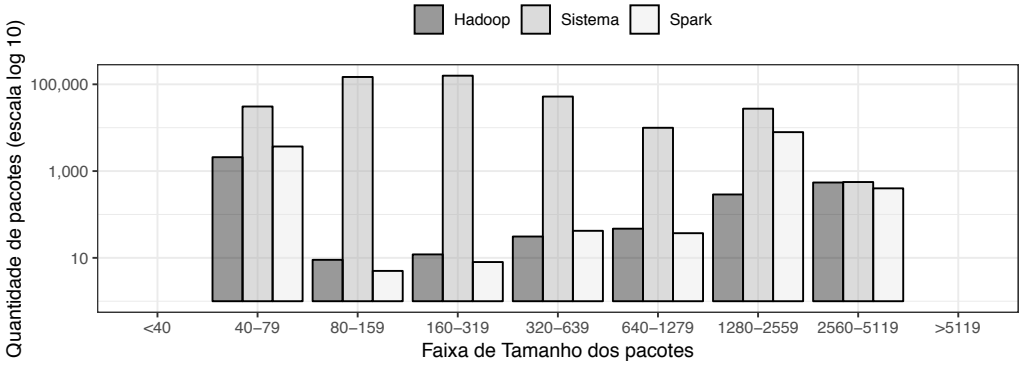
Também foram realizados testes considerando entradas de dado com blocos de tamanho menor. Para blocos de 32 MiB em ambiente de execução homogêneo, o tempo

Figura 64: Quantidade de pacotes trafegados



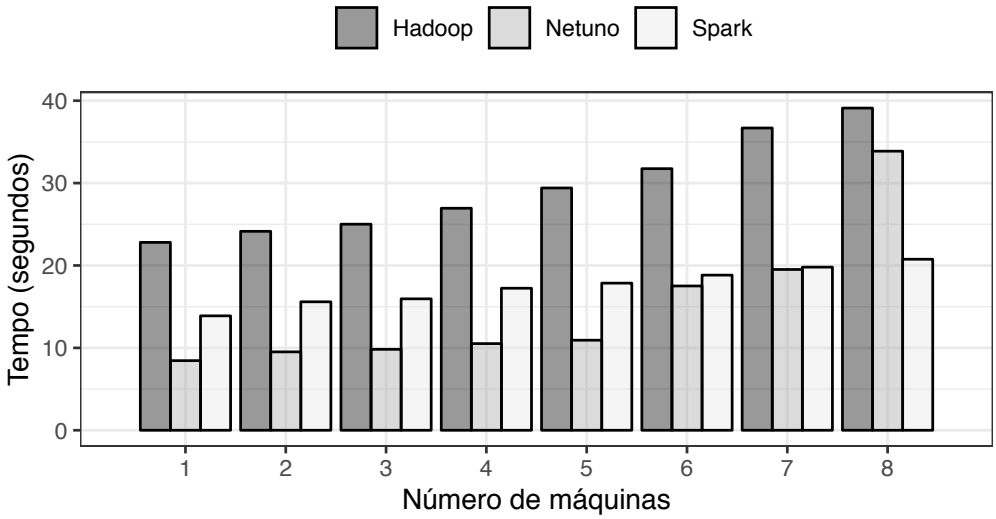
Fonte: Próprio autor (2018)

Figura 65: Quantidade de pacotes de diferentes tamanhos trafegados



Fonte: Próprio autor (2018)

Figura 66: Tempo de execução com entrada utilizando blocos menores

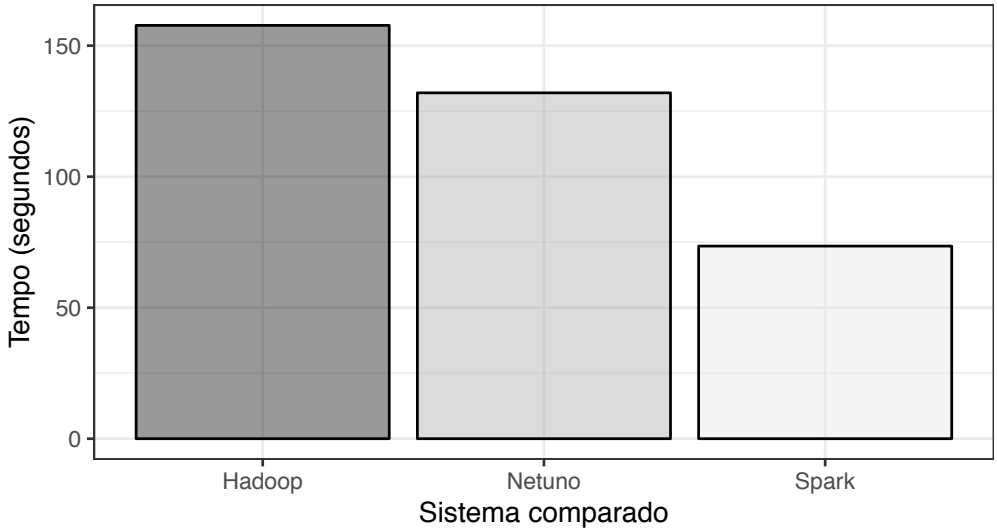


Fonte: Próprio autor (2018)

de execução do Sistema obteve o melhor desempenho, comparado aos demais. Para ambiente heterogêneo, o tempo de execução do Sistema obteve o melhor desempenho, comparado aos demais. Para ambiente heterogêneo com 6 e 7 máquinas o tempo de execução do Sistema continua maior que os demais, porém com uma discrepância menos significativa em relação ao Spark. Na execução com 8 máquinas, as estratégias de balanceamento de carga e a menor quantidade de dados transferidos do Spark fizeram que ele obtivesse o menor tempo de execução neste cenário.

Outro teste realizado consistiu em utilizar um número maior de blocos por máquina como entrada. O gráfico da Figura 67 apresenta o resultados dos testes em que são utilizados 6 blocos por máquina, executados utilizando 5 máquinas. Neste teste, para realizar a leitura de cada máquina foram utilizados os blocos de armazenamento intermediário, e o tempo necessário para escrever e carregar dados do disco fizeram com que o Sistema obtivesse tempo de execução maior que o obtido com o Spark.

Figura 67: Tempo de execução utilizando diversos blocos por máquina



Fonte: Próprio autor (2018)

6.3.6 Processamento iterativo

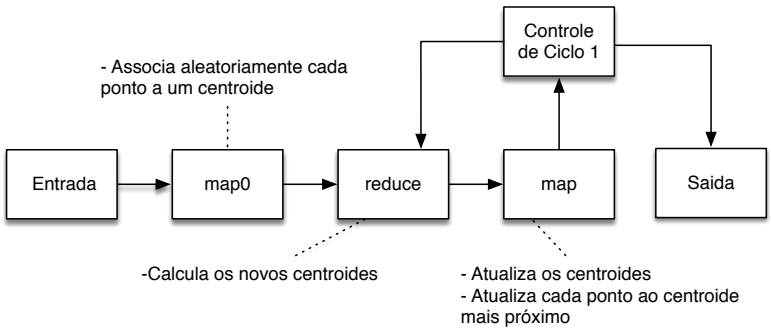
Outro teste realizado consistiu em verificar a utilização da ferramenta em outros algoritmos, de forma que um processo iterativo seja realizado. Foi implementado o algoritmo de agrupamento das K-Médias (*K-Means*) (ZHAO; MA; HE, 2009). Este algoritmo consiste em associar, para um determinado conjunto de pontos de entrada, cada ponto da entrada a um ponto k determinado de centroide, de maneira que cada centroide seja o ponto médio dos pontos associados a ele. O pseudocódigo do algoritmo *K-Means* é apresentado no Algoritmo 5.

O algoritmo *K-Means* é iterativo, e se repete até que os centroides não sejam modificados entre cada iteração. A implementação utilizando a ferramenta aqui proposta foi realizada utilizando 2 blocos map e um bloco reduce, como indicado na Figura 68. O bloco `map0` realiza a associação inicial dos pontos de entrada a um centróide definido aleatoriamente. O `reduce` consiste em calcular os novos centroides e enviar o novo centroide calculado às demais máquinas, enquanto `map` verifica para cada ponto se o centroide a que o ponto está associado continua sendo o centroide mais próximo. Caso não, o ponto é associado ao novo centroide.

```
1: Defina valores iniciais para os centroides $k_0$ a
   $k_(nk-1)$, onde n é o número de grupos a serem formados
2: Enquanto não houverem mudanças nos centroides, ou I
   iterações sejam executadas, faça
3: {
4:     Para cada ponto P da entrada, faça
5:         Associe o ponto ao centroide mais próximo
6:         Para i de 1 até nk
7:         {
8:             Calcule o ponto médio dos pontos associados
               a $k_i$
9:             Atualize $k_i$
10:        }
11: }
```

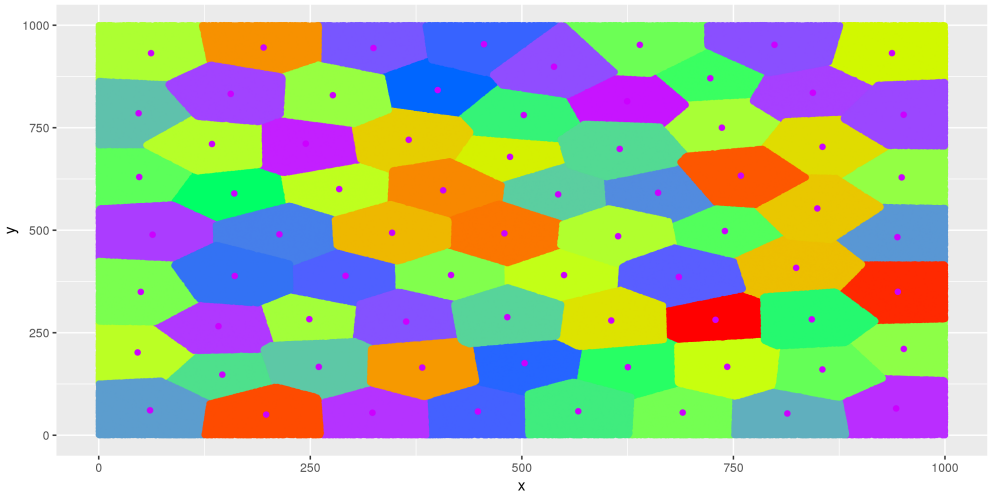
Algoritmo 5: Pseudocódigo do algoritmo *K-Means*

Figura 68: Fluxo de execução do algoritmo *K-Means* implementado



Fonte: Próprio autor (2018)

Figura 69: Saída da execução do algoritmo *K-Means* utilizando 1 milhão de pontos



Fonte: Próprio autor (2018)

Os testes foram realizados utilizando 1000000 (um milhão) de pontos em um plano bidimensional como entrada, e 100 iterações. A Figura 69 apresenta no plano cartesiano o resultado da execução. Internamente a cada grupo, o ponto em destaque consiste no centroide do respectivo grupo, de forma que cada grupo é representado por uma cor diferente. Os pontos de entrada utilizados pertencem ao intervalo $[0, 1000]$.

O arquivo com informações da execução apresentado na Figura 70 demonstra o uso do bloco de controle de iteração, em destaque. Nele são informados quais blocos compõem o início e fim do ciclo, o número de iterações e qual bloco deve continuar com o fluxo de execução após o número de iterações ter sido atingido. Na Figura, os blocos de entrada e saída foram omitidos, para melhor visualização dos blocos de execução e controle de iteração.

Figura 70: Exemplo de arquivo com informações da execução da aplicação do algoritmo *K-Means* implementado

```
<job>
...
        <exec>
                <var>map0</var>
                <udf>kmeans.udf</udf>
                <num_threads>4</num_threads>
                <output>Reduce</output>
        </exec>
        <exec>
                <var>reduce</var>
                <udf>kmeans.udf</udf>
                <num_threads>4</num_threads>
                <output>Saida</output>
        </exec>
        <exec>
                <var>map</var>
                <udf>kmeans.udf</udf>
                <num_threads>4</num_threads>
                <output>iteracao</output>
        </exec>
        <iteration>
                <var>iteracao</var>
                <from>reduce</from>
                <to>map</to>
                <output>Saida</output>
                <num_iter>100</num_iter>
        </iteration>
...
</job>
```

Fonte: Próprio autor (2018)

7 CONCLUSÃO

Neste trabalho foi proposta uma ferramenta de programação para desenvolvimento e ambiente de execução para aplicações com grandes quantidades de dados. O uso da ferramenta visa obter melhor desempenho em aplicações neste cenário, utilizando recursos disponíveis como diversas linhas de execução e programação distribuída. O uso da ferramenta consiste em utilizar diversos blocos de programação, de forma que cada bloco é composto por uma tarefa, e estes blocos são executados de acordo com o modelo produtor-consumidor, de acordo com um fluxo de execução definido. A execução dos blocos é realizada simultaneamente por todas as máquinas no ambiente distribuído, e sua troca de dados é realizada de maneira transparente ao usuário.

A execução utilizando processamento em blocos mostrou-se vantajosa no teste da aplicação que realiza a contagem de palavras, com tempos de execução inferiores ao de uma aplicação equivalente executada utilizando Hadoop e Spark.

O controle de como os diversos pares KV são agrupados nas estruturas de dados em árvore também possui influência no desempenho. O agrupamento é realizado com base no *hash* da chave k, e o melhor desempenho obtido deu-se utilizando 4 bits. Os desempenhos menos satisfatórios deram-se com o emprego dos valores extremos, utilizando 0 e 32 bits, visto que com 0 a árvore possui um único elemento, sobrecarregando o agrupamento dos pares que possuem o mesma chave. Ao utilizar 32 bits, a árvore atinge seu tamanho máximo, e o tempo necessário para percorre, balancear, inserir e remover elementos na árvore torna-se um fator degradante no desempenho

da aplicação. O uso de diversas *threads* também interfere no tempo de execução. Ao utilizar uma única *thread* para tarefas, recursos físicos deixam de ser utilizados. Desta forma, os melhores desempenhos obtidos foram operando com 4 a 6 *threads*. Utilizando um maior número de *threads* o desempenho não aumenta, ficando limitado pela concorrência no acesso à estrutura que possui os dados de entrada para o bloco.

A maneira que os diversos blocos são organizados para a execução das aplicações também possui relação com o desempenho. Os testes realizados com uma aplicação que realiza a contagem de palavras e palíndromos, apresentaram diferenças no tempo de execução entre quatro diferentes variações da aplicação, de forma que a versão que possui a menor quantidade de troca de dados obteve o melhor desempenho.

A estratégia de processamento especulativo no modelo de programação em blocos possibilita que a execução do bloco seguinte seja iniciada antes do término da execução do bloco anterior. A aplicação desta estratégia consiste em verificar se algum par com a mesma chave k já tenha sido processado, e o resultado é calculado e atualizado caso seja necessário. Este processamento é realizado utilizando o tempo em que a máquina aguarda pela comunicação de dados. Os resultados indicaram que o uso do processamento especulativo torna-se vantajoso quando executado em ambiente distribuído a partir de 3 máquinas. O baixo desempenho da estratégia dá-se pela menor quantidade de dados trafegados e o grande tamanho da estrutura de árvore que armazena os dados, visto que é necessário realizar diversas buscas na árvore de saída dos dados, de forma a verificar se o par já foi processado. Quanto à troca de dados, a estratégia utiliza o tempo de espera pela chegada dos pares KV para realizar o processamento especulativo, e, desta forma, mostrou-se que é necessário um tempo mínimo de espera para que seu uso possua vantagens em relação ao desempenho.

Para situações em que o a quantidade de memória necessária para a processamento da aplicação é superior à memória disponível, pode-se utilizar os blocos de armazenamento intermediário, que consistem em armazenar em disco todos os pares existentes

em uma estrutura em árvore, para posteriormente serem carregados. Visto que todos os dados são gravados ou carregados de uma única vez, seu impacto é mínimo para a degradação do desempenho, mostrando ser uma estratégia vantajosa para situações com pouca memória disponível.

Um cenário contendo ambiente de execução heterogêneo foi testado, e o mesmo teste utilizado no ambiente homogêneo foi aplicado. Foram utilizadas máquinas com processadores com diferentes quantidades de núcleos, frequências, quantidade de memória e velocidade de transmissão de dados. Neste cenário, a não presença de mecanismos de balanceamento de carga e a grande quantidade de dados trocados pelo Sistema tornam este menos vantajoso que o Spark.

A análise da quantidade de dados trafegados mostrou que a comunicação realizada pelo Sistema é muito superior à feita pelo Hadoop e Spark. Esta análise demonstrou que este é um ponto de gargalo, e que otimizações no módulo de comunicação podem levar a um aumento de desempenho do Sistema.

Os resultados dos testes realizados indicaram que o uso da estratégia pode levar ao aumento de desempenho em aplicações, principalmente em ambiente homogêneos e situações em que o tamanho das estruturas em árvore em que armazenam os pares KV seja pequena. Estes fatores são influenciados principalmente pelo tamanho da entrada de dados e o número de máquinas presentes no ambiente distribuído utilizadas para o processamento da aplicação.

As contribuições gerais do trabalho deram-se pelo aumento de desempenho em aplicações para a área de *Big Data*, utilizando blocos de execução e possibilitando que diversos blocos sejam executados de maneira coordenada e distribuída, aproveitando recursos múltiplos de processamento disponíveis, como indicado no Capítulo 1.

7.1 Trabalhos futuros

Como continuidade para este trabalho, recomenda-se abranger os pontos listados a seguir.

- Utilização de blocos de armazenamento intermediário dinâmicos, automatizados de acordo com a necessidade;
- otimizações no módulo de troca de dados, considerando outros tamanhos de dados enviados, e não apenas o KVG completo;
- otimizações no gerenciamento de memória, levando em conta aspectos como fragmentação de memória;
- atualização da estratégia de execução especulativa considerando mais de um bloco de execução;

7.2 Publicações realizadas

A partir de resultados intermediários com as pesquisas deste trabalho, diversas publicações foram realizadas. A lista a seguir apresenta tais publicações:

- Vasata e Sato (2014): Utilização de MPI com o sistema de arquivos distribuído HDFS;
- Vasata e Sato (2015): Mapreduce com múltiplos blocos de execução atuando de forma coordenada utilizando entradas e saídas interligadas;
- Vasata e Sato (2016a): *Cross Join Operation Over Large Quantities of Data Considering Data Locality and Data Replication*;
- Vasata e Sato (2016b): Execução de aplicações Mapreduce utilizando paralelismo local e distribuído;

- Vasata e Sato (2017): Estratégia para fase de Merge utilizando agrupamento local.

APÊNDICE A - FUNÇÕES DO USUÁRIO E EXEMPLOS

Este apêndice apresenta a documentação das funções disponíveis ao usuário nas UDFs, e também apresenta um exemplo de código de utilização da ferramenta.

A.1 Documentação

Esta seção apresenta as funções disponíveis, com a documentação gerada utilizando a ferramenta Doxygen (HEESCH, 2018). Ambos os códigos documentando `udf.h` e `Keyvalue.h` possuem adaptações de estrutura em relação à documentação original.

A.1.1 udf.h

Esta biblioteca possui a rotina de interação com o ambiente de execução, a função `Emit`.

Referência do arquivo udf.h

Funções

- int Emit (void ∗saida, Keyvalue ∗kv)

  Função que envia uma estrutura KV ao Sistema, para posterior processamento no fluxo de execução.

Documentação das funções

int Emit (void ∗ *saida,* Keyvalue ∗ *kv*)

Função que envia uma estrutura KV ao Sistema, para posterior processamento no fluxo de execução.

Parâmetros

saida	Endereço para a estrutura de saída, onde os dados devem ser retornados.
kv	Endereço para a estrutura que será retornada.

Retorna

0 para ok.
1 para erros.

A.1.2 keyvalue.h

Esta biblioteca possui rotinas de manipulação de estruturas KV. A estrutura interna da implementação de um par KV é apresentada no capítulo 5, e ilustrada na Figura 30.

Referência do arquivo Keyvalue.h

Componentes

- struct keyvalue

  Estrutura Keyvalue, que armazena o par KV.

Funções

- struct keyvalue keyvalue_printArmaz (void)

  Estrutura Keyvalue, que armazena o par KV.
- unsigned int keyvalue_getArmazDados (void)

  Retorna a quantidade de dados armazenados em estruturas Keyvalue.
- unsigned int keyvalue_getArmazMetadados (void)

  Retorna a quantidade de metadados utilizados em estruturas Keyvalue.
- Keyvalue ∗ keyvalue_new (Keyvalue_Alocador ∗alocador_kv)

  Obtém uma nova estrutura, a partir do banco de memória;.
- bool keyvalue_free (Keyvalue_Alocador ∗alocador_kv, Keyvalue ∗kv)

  Devolve a área de memória ocupada por uma estrutura ao banco de memória.
- bool keyvalue_init (Keyvalue ∗kv, void ∗key, unsigned int size)

  Inicializa uma estrutura Keyvalue.
- bool keyvalue_destroyFree (Keyvalue ∗kv, ListaSizeElem_Alocador ∗alocador_lse)

  Libera a memória utilizada em uma estrutura Keyvalue.
- bool keyvalue_setKey (Keyvalue ∗kv, void ∗key, unsigned int key_size)

  Seta a chave em uma estrutura Keyvalue.
- bool keyvalue_pushValue (Keyvalue ∗kv, void ∗value, unsigned int value_size, ListaSizeElem_Alocador ∗alocador_lse)

  Insere um valor em uma estrutura Keyvalue.
- void ∗ keyvalue_popValue (Keyvalue ∗kv, unsigned int ∗value_size_ret, ListaSizeElem_Alocador ∗alocador_lse)

  Remove um valor de uma estrutura Keyvalue.
- void ∗ keyvalue_popValueAt (Keyvalue ∗kv, unsigned int ∗value_size_ret, unsigned int pos, ListaSizeElem_Alocador ∗alocador_lse)

  Remove um valor de uma estrutura Keyvalue em uma posição específica.
- void ∗ keyvalue_getValueAt (Keyvalue ∗kv, unsigned int ∗value_size_ret, unsigned int pos)

  Obtém um valor armazenado na estrutura, sem removê-lo do conjunto de valores. É retornado uma cópia do valor armazenado internamente.
- void ∗ keyvalue_getKey (Keyvalue ∗kv, unsigned int ∗keysize_ret)

  Obtém a chave armazenada em uma estrutura Keyvalue.
- unsigned int keyvalue_getNumValues (Keyvalue ∗kv)

  Obtém a pilha de valores.
- unsigned int keyvalue_getSizeData (Keyvalue ∗kv)

  Obtém a quantidade de dados armazenados na estrutura.
- unsigned long int keyvalue_getSizeMetadata (Keyvalue ∗kv)

  Obtém a quantidade de dados e metadados armazenados na estrutura.
- int keyvalue_compareKey (Keyvalue ∗kv1, Keyvalue ∗kv2)

  Compara se duas estruturas possuem a mesma chave.
- int keyvalue_compareKeySaida (Keyvalue ∗kv1, Keyvalue ∗kv2)

  Compara duas estruturas keyvalue, considerando também a saída do par kv.
- int keyvalue_compareKeySaidaVoid (const void ∗a, const void ∗b)

  Função de comparação, utilizando ponteiros void.

- bool keyvalue_mergeValues (Keyvalue ∗kv_to, Keyvalue ∗kv_from)

  Agrupa os valores de duas estruturas Keyvalue.
- bool keyvalue_setIdSaida (Keyvalue ∗kv, unsigned char id_saida)

  Define o id do bloco de saída de uma estrutura Keyvalue.
- unsigned char keyvalue_getIdSaida (Keyvalue ∗kv)

  Obtém o id de saída de uma estrutura Keyvalue.
- void keyvalue_print (Keyvalue ∗kv, Funcao_Keyvalue_Imprimir f_print_key, Funcao_Keyvalue_Imprimir f_print_value)

  Imprime os valores armazenados em uma estrutura Keyvalue.
- void keyvalue_printVetores (Keyvalue ∗kv)

  Imprime a estrutura de vetores interna do Keyvalue.
- unsigned char ∗ keyvalue_pack (Keyvalue ∗kv, unsigned int ∗size_pack_ret)

  Converte a estrutura de Keyvalue em um vetor de bytes.
- bool keyvalue_unpack (Keyvalue ∗kv_output, unsigned char ∗pack, unsigned int size_pack, ListaSizeElem_Alocador ∗alocador_lse)

  Converte um vetor de bytes em uma estrutura Keyvalue.

Variáveis

- void ∗ key

  Endereço para a chave.
- unsigned int key_size

  Tamanho da chave.
- ListaSize lista

  Lista com os valores.
- unsigned char saida

  Bloco de saída do Keyvalue.
- unsigned int hash

  Hash da chave.

Documentação das funções

int keyvalue_compareKey (Keyvalue ∗ *kv1*, Keyvalue ∗ *kv2*)

Compara se duas estruturas possuem a mesma chave.

Parâmetros

kv1	Endereço de estrutura Keyvalue.
kv2	Endereço de estrutura Keyvalue.

Retorna

0 para iguais.
1 para kv1 > kv2.
-1 para kv1 < kv2
2 para erros.

int keyvalue_compareKeySaida (Keyvalue * *kv1,* Keyvalue * *kv2*)

Compara duas estruturas keyvalue, considerando também a saída do par kv.

Compara duas estruturas keyvalue, considerando também a saída do par kv. A prioridade de comparação é:

- tamanho da chave
- chave
- saída

Parâmetros

kv1	Endereço para 1ra estrutura a ser comparada
kv2	Endereço para 2da estrutura a ser comaparada

Retorna

0 para estruturas iguais
1 para kv1 > kv2
-1 para kv1 < kv2

int keyvalue_compareKeySaidaVoid (const void * *a,* const void * *b*)

Função de comparação, utilizando ponteiros void.

Esta função não utiliza apenas ponteiros void, como também são considerados os endereços para ponteiros de estrutura keyvalue. Internamente, os valores nos endereços apontados por a e b chamam a função keyvalue_compareKeySaida.

Parâmetros

a	Endereço para o ponteiro da 1ra estrutura
b	Endereço para o ponteiro da 2da estrutura

Retorna

0 para estruturas iguais
1 para kv1 > kv2
-1 para kv1 < kv2

bool keyvalue_destroyFree (Keyvalue * *kv,* ListaSizeElem_Alocador * *alocador_lse*)

Libera a memória utilizada em uma estrutura Keyvalue.

Aviso

Libera os dados armazenados.

Parâmetros

kv	Endereço para a estrutura Keyvalue com os dados a serem liberados.
alocador_lse	Endereço para o banco de memória de estruturas de lista.

Retorna

true para ok.
false para erros.

bool keyvalue_free (Keyvalue_Alocador * *alocador_kv,* Keyvalue * *kv*)

Devolve a área de memória ocupada por uma estrutura ao banco de memória.

Parâmetros

alocador_kv	Endereço para o banco de memória de estruturas Keyvalue.
kv	Endereço para a estrutura a ser liberada a memoria.

unsigned int keyvalue_getArmazDados (void)

Retorna a quantidade de dados armazenados em estruturas Keyvalue.

Retorna

Quantidade de dados armazenados.

unsigned int keyvalue_getArmazMetadados (void)

Retorna a quantidade de metadados utilizados em estruturas Keyvalue.

Retorna

Quantidade de metadados utilizados.

unsigned char keyvalue_getIdSaida (Keyvalue * *kv*)

Obtém o id de saída de uma estrutura Keyvalue.

Parâmetros

kv	Endereço para a estrutura Keyvalue.

Retorna

Id de saída da estrutura.

void* keyvalue_getKey (Keyvalue * *kv*, unsigned int * *keysize_ret*)

Obtém a chave armazenada em uma estrutura Keyvalue.

Parâmetros

kv	Endereço para a estrutura Keyvalue.
keysize_ret	Retorno: Tamanho da chave retornada.

Retorna

Endereço para a chave (Cópia da chave armazenada internamente).
NULL para erros.

unsigned int keyvalue_getNumValues (Keyvalue * *kv*)

Obtém a pilha de valores.

Parâmetros

kv	Endereço para a estrutura Keyvalue.

Retorna

Pilha de valores Obtém a quantidade de valores armazenados.

Parâmetros

kv	Endereço para a estrutura Keyvalue.

Retorna

Quantidade de valores armazenados.

unsigned int keyvalue_getSizeData (Keyvalue * *kv*)

Obtém a quantidade de dados armazenados na estrutura.

Parâmetros

kv	Endereço para a estrutura Keyvalue.

Retorna

Quantidade de dados armazenados.
0 para erros.

unsigned long int keyvalue_getSizeMetadata (Keyvalue * *kv*)

Obtém a quantidade de dados e metadados armazenados na estrutura.

Parâmetros

kv	Endereço para a estrutura Keyvalue.

Retorna

Quantidade de metadados armazenados.
0 para erros.

void* keyvalue_getValueAt (Keyvalue * *kv,* unsigned int * *value_size_ret,* unsigned int *pos*)

Obtém um valor armazenado na estrutura, sem removê-lo do conjunto de valores. É retornado uma cópia do valor armazenado internamente.

Parâmetros

kv	Endereço para a estrutura Keyvalue.
value_size_ret	Retorno: Tamanho do valor retornado.
pos	Posição do valor a ser obtido.

Retorna

Valor armazenado em pos.
NULL para erros.

bool keyvalue_init (Keyvalue * *kv,* void * *key,* unsigned int *size*)

Inicializa uma estrutura Keyvalue.

Parâmetros

kv	Endereço para a estrutura a ser inicializada.
key	Endereço para a chave.
size	Tamanho da chave

Retorna

true para ok.
false para erros.

bool keyvalue_mergeValues (Keyvalue * *kv_to*, Keyvalue * *kv_from*)

Agrupa os valores de duas estruturas Keyvalue.

Parâmetros

kv_to	Estrutura que terá os valores adicionados.
kv_from	Estrutura que terá seus valores adicionado a kv_to.

Retorna

true para ok.
false para erros.

Keyvalue* keyvalue_new (Keyvalue_Alocador * *alocador_kv*)

Obtém uma nova estrutura, a partir do banco de memória;.

Parâmetros

alocador_kv	Endereço para o banco de memória de estruturas Keyvalue.

Retorna

Endereço para a estrutura alocada.
NULL para erros.

unsigned char* keyvalue_pack (Keyvalue * *kv*, unsigned int * *size_pack_ret*)

Converte a estrutura de Keyvalue em um vetor de bytes.

Parâmetros

kv	Endereço para a estrutura Keyvalue.
size_pack_ret	Retorno: Tamanho do vetor de bytes.

Retorna

Vetor de bytes.

void* keyvalue_popValue (Keyvalue * *kv*, unsigned int * *value_size_ret*, ListaSizeElem_Alocador * *alocador_lse*)

Remove um valor de uma estrutura Keyvalue.

Parâmetros

kv	Endereço para a estrutura Keyvalue.
value_size_ret	Retorno: Tamanho do valor retornado.
alocador_lse	Endereço para o banco de memória de estruturas de lista.

Retorna

Endereço para o valor removido. É retornado uma cópia do valor, em outra posição de memória, diferente do local armazenado internamente na estrutura.
NULL para erros.

void* keyvalue_popValueAt (Keyvalue * *kv,* unsigned int * *value_size_ret,* unsigned int *pos,* ListaSizeElem_Alocador * *alocador_lse*)

Remove um valor de uma estrutura Keyvalue em uma posição específica.

Parâmetros

kv	Endereço para a estrutura Keyvalue.
value_size_ret	Retorno: Tamanho do valor retornado.
pos	Índice da posição do valor a ser removido.
alocador_lse	Endereço para o banco de memória de estruturas de lista.

Retorna

Endereço do valor retornado. É retornado uma cópia do valor, em uma outra posição de memória, diferente do local armazenado internamente na estrutura.
NULL para erros.

void keyvalue_print (Keyvalue * *kv,* Funcao_Keyvalue_Imprimir *f_print_key,* Funcao_Keyvalue_Imprimir *f_print_value*)

Imprime os valores armazenados em uma estrutura Keyvalue.

Parâmetros

kv	Endereço para a estrutura Keyvalue.
f_print_key	Ponteiro para função para imprimir a chave.
f_print_value	Ponteir para função para imprimir os valores.

struct keyvalue keyvalue_printArmaz (void)

Estrutura Keyvalue, que armazena o par KV.

Imprime a quantidade de dados e metadados armazenados na estrutura

void keyvalue_printVetores (Keyvalue * *kv*)

Imprime a estrutura de vetores interna do Keyvalue.

Parâmetros

kv	Endereço para a estrutura Keyvalue.

bool keyvalue_pushValue (Keyvalue * *kv,* void * *value,* unsigned int *value_size,* ListaSizeElem_Alocador * *alocador_lse*)

Insere um valor em uma estrutura Keyvalue.

Parâmetros

kv	Endereço para a estrutura Keyvalue.
value	Endereço para o valor. O conteúdo será copiado.
value_size	Tamanho do valor.
alocador_lse	Endereço para o banco de memória de estruturas de lista.

Retorna

true para ok.
false para erros.

bool keyvalue_setIdSaida (Keyvalue * *kv,* unsigned char *id_saida*)

Define o id do bloco de saída de uma estrutura Keyvalue.

Parâmetros

kv	Endereço para a estrutura Keyvalue.
id_saida	Id da saída.

Retorna

true para ok.
false para erros.

bool keyvalue_setKey (Keyvalue * *kv,* void * *key,* unsigned int *key_size*)

Seta a chave em uma estrutura Keyvalue.

Parâmetros

kv	Endereço para a estrutura Keyvalue.
key	Endereço para a chave. O conteúdo será copiado.
key_size	Tamanho da chave.

Retorna

true para ok.
false para erros.

bool keyvalue_unpack (Keyvalue * *kv_output,* unsigned char * *pack,* unsigned int *size_pack,* ListaSizeElem_Alocador * *alocador_lse*)

Converte um vetor de bytes em uma estrutura Keyvalue.

Parâmetros

kv_output	Retorno: Endereço para a estrutura Keyvalue não inicializada.
pack	Vetor de bytes.
size_pack	Tamanho do vetor de bytes.
alocador_lse	Endereço para o banco de memória de estruturas de lista.

Retorna

true para ok.
false para erros.

A.2 Contador de palavras

Como exemplo e teste do modelo apresentado foi desenvolvido uma aplicação contadora de palavras, como descrito no Capítulo 6, na Figura 46. As próximas seções demonstram um arquivo de configuração e códigos `map` e `reduce` dessa aplicação. Nas seções seguintes, as figuras 72 e 73 com os códigos das funções `map` e `reduce`, a declaração de bibliotecas C como `stdlib.h`, `stdio.h`, `string.h` e `unistd.h` foram omitidas para melhora de legibilidade do código.

A.2.1 Configurações da aplicação

O código XML apresentado na Figura 71 corresponde ao fluxo de execução e às funções UDF que devem ser executadas pela aplicação, assim como arquivos de entrada e saída. Cada bloco possui uma *tag* `var`, que o identifica entre os demais blocos. Além desta, cada bloco possui a *tag* `next`, informando qual é o bloco seguinte, ao qual os dados devem ser repassados para que o fluxo de execução siga a ordem definida.

A.2.2 map

A Figura 72 mostra a função `map` que recebe os dados da entrada com uma sequência de caracteres contendo a linha de entrada e uma função que separa cada palavra da entrada (linha 13). Posteriormente, para cada palavra (linha 14) um novo KV é criado (linha 16) contendo a palavra como chave (linha 21) e o número de ocorrências como valor (linha 22). Após isso, o par KV criado é devolvido ao Sistema para processamento com a função `Emit` (linha 23). O endereço para retorno utilizado na função `Emit` é o mesmo passado como parâmetro no cabeçalho da função (linha 4). Esta execução corresponde ao mecanismo ilustrado na Figura 35, onde a *thread* retira os elementos da árvore que contém os elementos KV da entrada de dados e repassa-os

Figura 71: Exemplo de arquivo com informações da execução da aplicação

```
<job>
	<input>
		<var>Entrada</var>
		<filename>entrada.txt</filename>
		<type>Text</type>
		<format>Line</format>
		<next>Map</next>
	</input>
	<exec>
		<var>Map</var>
		<udf>map.bin</udf>
		<num_threads>4</num_threads>
		<output>Reduce</output>
	</exec>
	<exec>
		<var>Reduce</var>
		<udf>Reduce.bin</udf>
		<num_threads>4</num_threads>
		<output>Saida</output>
	</exec>
	<output>
		<var>Saida</var>
		<filename>saida.txt</filename>
	</output>
</job>
```

Fonte: Próprio autor (2018)

como parâmetro para a função UDF, na parte direita da figura, seguindo o cabeçalho de função e a função `Emit` apresentados nos códigos das figuras 33 e 34.

A.2.3 reduce

A Figura 73 apresenta o exemplo de código `reduce`. Assim como na função `map`, a função `reduce` utiliza o mesmo protótipo da função apresentada no código da Figura 33 (linha 4). Cada valor recebido na estrutura KV de entrada é percorrida (linhas 18 a 21). Os valores são somados (linha 19) e adicionados ao par KV (linha 22), e então retornados ao Sistema (linha 23).

Figura 72: Exemplo de função map em C que realiza a contagem de palavras

```
1: #include "udf.h"
2: #include "Keyvalue.h"
3: #include "Palavras.h"
4: int map ( void *retorno, void *alocador_kv, void *alocador_lista, struct KV * kv_input)
5: {
6:        struct KV *kv;
7:        struct Value *val;
8:        char *palavra;
9:        char **vetor_palavras;
10:       unsigned int n_palavras, vetor_tam_palavras, i ;
11:       unsigned long long int *ocorrencias;
12:       val = keyvalue_getValue(kv_input, 0);
13:       separaPalavras(&n_palavras, *vetor_palavras, *vetor_tam_palavras, val->tam_value,
   val->value);
14:       for(i=0; i<n_palavras; i++)
15:       {
16:              kv = keyvalue_new(alocador_kv);
17:              palavra = calloc( vetor_tam_palavras[i] +1, sizeof(char) );
18:              memcopy(palavra, vetor_palavras[i], vetor_tam_palavras[i]);
19:              ocorrencias = calloc(1,
                sizeof( unsigned long long int));
20:              *ocorrencias = 1;
21:              keyvalue_setKey(kv, palavra);
22:              keyvalue_addValue(kv, ocorrencias, alocador_lista);
23:              Emit(retorno, kv);
24:       }
25:       return 0;
26: }
```

Fonte: Próprio autor (2018)

Figura 73: Exemplo de função reduce em C que realiza a contagem de palavras

```
1: #include "udf.h"
2: #include "Keyvalue.h"
3: #include "Palavras.h"
4: int reduce(void *saida, void *alocador_kv, void *alocador_lista, Keyvalue *kv)
5: {
6:        char *palavra;
7:        unsigned int palavra_size;
8:        int *val, *sum;
9:        unsigned int val_size, n_values;
10:       Keyvalue *kv_new;
11:       palavra = keyvalue_getKey(kv, &palavra_size);
12:       n_values = keyvalue_getNumValues(kv);
13:       sum = malloc(sizeof(int));
14:       *sum = 0;
15:       kv_new = keyvalue_new(alocador_kv);
16:       keyvalue_init(kv_new, palavra, palavra_size); //percorre e retira os valores do par KV
17:       while( ( val = keyvalue_popValue(kv, &val_size, alocador_lista)) && (val!=NULL) )
18:       {
19:              (*sum)+=(*val);    //soma as ocorrências
20:              free(val);         //libera a memória ocupada pelo valor
21:       }
22:       keyvalue_pushValue(kv_new, sum, sizeof(int), alocador_lista);
23:       Emit(saida, kv_new);
24:       return 0;
25: }
```

Fonte: Próprio autor (2018)

APÊNDICE B - MÉTRICAS

Este apêndice apresenta as métricas de avaliação de desempenho utilizadas no trabalho, como o tempo de execução, *speedup* e eficiência.

B.1 Tempo de execução

O tempo de execução é a métrica que consiste em quanto tempo leva para a realização do trabalho. A medida é tomada com a utilização da função `time` presente no sistema Bash. Nos testes deste trabalho, são realizadas dez execuções de cada teste, e então a média aritmética e o desvio padrão são calculados.

B.2 *Speedup*

O *speedup* é uma medida de grau de desempenho de aplicações paralelas. Consiste na divisão do tempo de execução sequencial de uma aplicação pelo tempo de execução da mesma aplicação em paralelo, como apresentado na Equação B.1.

$$S(p) = \frac{T(1)}{T(p)} \tag{B.1}$$

B.3 Eficiência

Assim como o *speedup*, a eficiência é métrica voltada à aplicações paralelas. Sua medida é o grau de aproveitamento dos recursos computacionais, dado pela divisão

entre o grau de desempenho (*speedup*) e os recursos computacionais disponíveis. A Equação B.2 apresenta a forma de cálculo para eficiência.

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{p \times T(p)} \tag{B.2}$$

APÊNDICE C - DADOS OBTIDOS

Este apêndice apresenta os valores utilizados na confecção dos gráficos apresentados no Capítulo 6, bem como os desvios padrão das tomadas de tempo realizadas.

Tabela 3: Dados utilizados no gráfico da Figura 51 - Tempo de execução (pg. 105)

Sistema	Hadoop
16s 761ms	29s 761ms
19s 874ms	30s 425ms
19s 332ms	32s 351ms
20s 912ms	35s 109ms
21s 896ms	38s 699ms

Tabela 4: Desvios padrão dos valores utilizados no gráfico da Figura 51 - Tempo de execução (pg. 105)

Sistema	Hadoop
137ms	795ms
307ms	693ms
183ms	542ms
364ms	584ms

Tabela 6: Desvios padrão dos dados utilizados no gráfico da Figura 52 - Tempo de execução utilizando diferentes valores para máscara de granularidade (pg. 107)

Máscara (bits)	1 nó	2 nós	3 nós	4 nós	5 nós
0	1s 229ms	2s 264ms	1s 643ms	820ms	1s 488ms
4	176ms	276ms	94ms	232ms	371ms
8	155ms	236ms	145ms	128ms	299ms
12	158ms	151ms	211ms	230ms	208ms
16	130ms	168ms	185ms	156ms	216ms
20	129ms	292ms	114ms	206ms	322ms
24	223ms	416ms	191ms	176ms	275ms
28	348ms	220ms	202ms	169ms	243ms
32	180ms	220ms	201ms	214ms	400ms

Tabela 7: Dados utilizados no gráfico da Figura 53 - Tempo de execução utilizando diferentes números de *threads* (pg. 107)

Número de *Threads*	1 nó	2 nós	3 nós	4 nós	5 nós
1	51s 414ms	42s 203ms	28s 124ms	26s 176ms	36s 611ms
2	22s 380ms	20s 887ms	20s 925ms	20s 524ms	22s 692ms
3	17s 341ms	18s 888ms	20s 123ms	19s 628ms	21s 33ms
4	16s 493ms	19s 208ms	19s 465ms	19s 105ms	20s 697ms
5	16s 327ms	19s 144ms	19s 311ms	18s 871ms	20s 551ms
6	16s 494ms	19s 246ms	19s 194ms	18s 653ms	20s 306ms
7	16s 237ms	19s 134ms	19s 127ms	18s 617ms	20s 505ms
8	16s 344ms	19s 92ms	18s 939ms	18s 818ms	20s 256ms

Tabela 5: Dados utilizados no gráfico da Figura 52 - Tempo de execução utilizando diferentes valores para máscara de granularidade (pg. 107)

Máscara (bits)	1 nó	2 nós	3 nós	4 nós	5 nós
0	44s 475ms	1m 0s 867ms	34s 882ms	52s 835ms	70s 660ms
4	15s 436ms	18s 539ms	15s 823ms	15s 360ms	19s 449ms
8	16s 664ms	19s 235ms	19s 498ms	19s 46ms	20s 657ms
12	19s 431ms	21s 835ms	22s 99ms	22s 675ms	23s 244ms
16	23s 325ms	26s 120ms	26s 789ms	26s 934ms	27s 814ms
20	28s 129ms	31s 576ms	31s 756ms	31s 945ms	31s 798ms
24	30s 213ms	34s 15ms	33s 371ms	33s 246ms	33s 461ms
28	30s 538ms	33s 818ms	33s 426ms	33s 640ms	34s 299ms
32	30s 601ms	33s 937ms	33s 729ms	33s 837ms	34s 302ms

Tabela 8: Desvios padrão dos dados utilizados no gráfico da Figura 53 - Tempo de execução utilizando diferentes números de *threads* (pg. 107)

Número de *Threads*	1 nó	2 nós	3 nós	4 nós	5 nós
1	1s 34ms	961ms	1s 779ms	1s 607ms	1s 56ms
2	297ms	485ms	248ms	237ms	411ms
3	168ms	339ms	239ms	223ms	351ms
4	268ms	229ms	191ms	104ms	239ms
5	171ms	184ms	157ms	221ms	332ms
6	189ms	172ms	204ms	110ms	302ms
7	97ms	215ms	248ms	172ms	256ms
8	232ms	258ms	197ms	314ms	267ms

Tabela 9: Dados da entrada individual utilizados no gráfico da Figura 54 - *Speedup* e eficiência (pg. 109)

Speedup	1 processo	2 processos	3 processos	4 processos	5 processos
Sistema	1,00	1,19	1,15	1,25	1,31
Hadoop	1,00	1,02	1,09	1,18	1,30

Tabela 10: Dados da entrada distribuída utilizados no gráfico da Figura 54 - *Speedup* e eficiência (pg. 109)

	1 processo	2 processos	3 processos	4 processos	5 processos
Speedup Sistema	1,00	1,61	2,29	2,45	4,09
Eficiência Sistema	100,00%	80,55%	76,46%	61,31%	81,87%

Tabela 11: Dados utilizado no gráfico da Figura 55 - Tempo de execução de contagem de palavras e palíndromos (pg. 110)

Versão do teste	1 processo	2 processos	3 processos	4 processos	5 processos
versão 1	16s 821ms	19s 846ms	19s 745ms	21s 378ms	22s 94ms
versão 2	16s 624ms	19s 627ms	19s 414ms	21s 294ms	21s 754ms
versão 3	18s 229ms	21s 431ms	21s 272ms	22s 861ms	23s 993ms
versão 4	18s 223ms	21s 332ms	21s 164ms	22s 865ms	23s 965ms

Tabela 12: Desvios padrão dos dados utilizado no gráfico da Figura 55 - Tempo de execução de contagem de palavras e palíndromos (pg. 110)

Versão do teste	1 processo	2 processos	3 processos	4 processos	5 processos
versão 1	189ms	291ms	342ms	245ms	426ms
versão 2	233ms	328ms	158ms	322ms	509ms
versão 3	171ms	199ms	245ms	394ms	367ms
versão 4	374ms	278ms	397ms	226ms	491ms

Tabela 13: Dados do teste utilizando armazenamento intermediário utilizados no gráfico da Figura 56 - Execuções utilizando blocos de armazenamento temporário, com múltiplos blocos e múltiplas *threads* (pg. 113)

Número de blocos	1 processo	2 processos	3 processos	4 processos	5 processos
1	52s 757ms	42s 325 ms	2 6s 719ms	27s 959ms	27s 848ms
2	99s 068ms	81s 561ms	52s 343ms	54s 31ms	57s 81ms
4	191s 311ms	157s 515ms	106s 509ms	115s 647ms	116s 929ms

Tabela 14: Desvios padrão dos dados do teste utilizando armazenamento intermediário utilizados no gráfico da Figura 56 - Execuções utilizando blocos de armazenamento temporário, com múltiplos blocos e múltiplas *threads* (pg. 113)

Número de blocos	1 processo	2 processos	3 processos	4 processos	5 processos
1	961ms	1s 215ms	1s 307ms	1s 375ms	1s 703ms
2	2s 163ms	1s 608ms	1s 893ms	2s 575ms	4s 335ms
4	2s 986ms	3s 712ms	2s 891ms	5s 269ms	4s 288ms

Tabela 15: Dados do teste sem utilizar armazenamento intermediário utilizados no gráfico da Figura 56 - Execuções utilizando blocos de armazenamento temporário, com múltiplos blocos e múltiplas *threads* (pg. 113)

Número de blocos	1 processo	2 processos	3 processos	4 processos	5 processos
1	49s 642	39s 365ms	25s 865ms	27s 128ms	25s 522ms
2	95s 086	74s 328ms	48s 305ms	50s 858ms	53s 041ms
4	181s 2623	147s 292ms	99s 027ms	103s 140ms	107s 106ms

Tabela 16: Desvios padrão dos dados do teste sem utilizar armazenamento intermediário utilizados no gráfico da Figura 56 - Execuções utilizando blocos de armazenamento temporário, com múltiplos blocos e múltiplas *threads* (pg. 113)

Número de blocos	1 processo	2 processos	3 processos	4 processos	5 processos
1	661ms	0s 935ms	0s 973ms	0s 926ms	1s 676ms
2	2s 638ms	2s 026ms	1s 661ms	3s 384ms	2s 923ms
4	2s 654ms	2s 078ms	3s 875ms	6s 729ms	5s 145ms

Tabela 17: Dados utilizados no gráfico da Figura 57 - Processamento especulativo (pg. 114)

	1p	2 processos	3 processos	4 processos	5 processos
Hadoop	29s 761ms	30s 425ms	32s 351ms	35s 109ms	38s 699ms
Sistema sem proc. especulativo	16s 761ms	19s 874ms	19s 332ms	20s 912ms	21s 896ms
Sistema com proc. especulativo	32s 408ms	24s 183ms	15s 351ms	16s 689ms	17s 154ms

Tabela 18: Desvio padrão dos dados utilizados no gráfico da Figura 57 - Processamento especulativo (pg. 114)

	1 processos	2 processos	3 processos	4 processos	5 processos
Hadoop	616ms	795ms	693ms	542ms	584ms
Sistema sem proc. especulativo	194ms	137ms	307ms	183ms	364ms
Sistema com proc. especulativo	904ms	557ms	396ms	899ms	707ms

Tabela 19: Dados da entrada individual utilizados no gráfico da Figura 58 - Processamento especulativo com diferentes quantidades de bits (pg. 115)

bits	2 processos	3 processos	4 processos	5 processos
1	40s 706ms	25s 218ms	17s 72ms	16s 991ms
4	40s 828ms	26s 581ms	17s 261ms	17s 2ms
8	40s 942ms	25s 776ms	16s 976ms	16s 766ms
12	40s 604ms	25s 710ms	16s 265ms	16s 583ms
16	40s 764ms	25s 841ms	15s 379ms	17s 150ms

Tabela 20: Desvios padrão dos dados da entrada individual utilizados no gráfico da Figura 58 - Processamento especulativo com diferentes quantidades de bits (pg. 115)

bits	2 processos	3 processos	4 processos	5 processos
1	1s 937ms	1s 163ms	633ms	671ms
4	1s 341ms	1s 686ms	583ms	455ms
8	1s 149ms	995ms	536ms	653ms
12	1s 890ms	741ms	714ms	568ms
16	2s 290ms	1s 514ms	578ms	1s 162ms

Tabela 21: Dados da entrada distribuída utilizados no gráfico da Figura 58 - Processamento especulativo com diferentes quantidades de bits (pg. 115)

bits	2 processos	3 processos	4 processos	5 processos
1	69s 580ms	28s 218ms	27s 145ms	17s 384ms
4	64s 107ms	27s 827ms	25s 473ms	16s 913ms
8	61s 275ms	27s 642ms	25s 786ms	16s 745ms
12	54s 479ms	27s 586ms	24s 402ms	16s 581ms
16	49s 526ms	26s 819ms	21s 317ms	16s 601ms

Tabela 22: Desvios padrão dos dados da entrada distribuída utilizados no gráfico da Figura 58 - Processamento especulativo com diferentes quantidades de bits (pg. 115)

bits	2 processos	3 processos	4 processos	5 processos
1	1s 670ms	717ms	1s 948ms	571ms
4	801ms	921ms	1s 363ms	529ms
8	857ms	516ms	1s 534ms	567ms
12	963ms	1s 91ms	782ms	648ms
16	1s 61ms	348ms	450ms	740ms

Tabela 23: Dados utilizados no gráfico da Figura 59 - Comparação entre sistemas em ambiente heterogêneo (pg. 117)

Número de processos	Hadoop	Spark	Sistema
1	29s 7ms	15s 712ms	16s 325ms
2	32s 256ms	19s 279ms	18s 678ms
3	34s 348ms	20s 436ms	19s 567ms
4	37s 797ms	21s 360ms	21s 229ms
5	43s 756ms	23s 240ms	22s 274ms
6	46s 892ms	25s 482ms	37s 84ms
7	58s 257ms	26s 842ms	40s 738ms
8	63s 75ms	29s 195ms	73s 403ms

Tabela 24: Desvios padrão dos dados utilizados no gráfico da Figura 59 - Comparação entre sistemas em ambiente heterogêneo (pg. 117)

Número de processos	Hadoop	Spark	Sistema
1	1s 737ms	562ms	179ms
2	856ms	1s 306ms	369ms
3	639ms	511ms	304ms
4	653ms	523ms	286ms
5	1s 46ms	752ms	366ms
6	907ms	1s 66ms	471ms
7	607ms	960ms	558ms
8	1s 529ms	430ms	2s 23ms

Tabela 25: Dados de 1 a 4 *threads* do gráfico da Figura 60 - Execuções utilizando apenas uma máquina separadas por máquina (pg. 118)

Máquina	1 *thread*	2 *threads*	3 *threads*	4 *threads*
1	20s 996ms	18s 134ms	16s 348ms	15s 751ms
2	21s 442ms	18s 36ms	16s 404ms	15s 889ms
3	21s 221ms	18s 83ms	16s 372ms	15s 731ms
4	21s 277ms	18s 276ms	16s 455ms	15s 820ms
5	21s 405ms	18s 137ms	16s 448ms	15s 795ms
6	63s 627ms	44s 454ms	44s 7ms	39s 878ms
7	33s 864ms	30s 54ms	26s 980ms	25s 583ms
8	34s 474ms	30s 659ms	27s 508ms	25s 967ms

Tabela 26: Dados de 5 a 8 *threads* do gráfico da Figura 60 - Execuções utilizando apenas uma máquina separadas por máquina (pg. 118)

Máquina	5 *threads*	6 *threads*	7 *threads*	8 *threads*
1	15s 293ms	14s 865ms	14s 181ms	14s 95ms
2	15s 295ms	14s 942ms	14s 593ms	14s 465ms
3	15s 285ms	14s 984ms	14s 565ms	14s 348ms
4	15s 302ms	14s 807ms	14s 534ms	14s 239ms
5	15s 488ms	15s 23ms	14s 589ms	14s 284ms
6	41s 934ms	42s 290ms	41s 757ms	41s 489ms
7	25s 632ms	25s 394ms	25s 91ms	24s 762ms
8	25s 750ms	25s 455ms	25s 311ms	24s 892ms

Tabela 27: Desvios padrão dos dados de 1 a 4 *threads* do gráfico da Figura 60 - Execuções utilizando apenas uma máquina separadas por máquina (pg. 118)

Máquina	1 *thread*	2 *threads*	3 *threads*	4 *threads*
m1	205ms	221ms	172ms	217ms
m2	343ms	295ms	255ms	215ms
m3	298ms	382ms	283ms	311ms
m4	173ms	286ms	311ms	192ms
m5	227ms	402ms	132ms	171ms
m6	717ms	3s	1s	413ms
m7	277ms	396ms	264ms	254ms
m8	409ms	172ms	216ms	199ms

Tabela 28: Desvios padrão dos dados de 5 a 8 *threads* do gráfico da Figura 60 - Execuções utilizando apenas uma máquina separadas por máquina (pg. 118)

Máquina	5 *threads*	6 *threads*	7 *threads*	8 *threads*
m1	200ms	127ms	914ms	174ms
m2	179ms	231ms	148ms	273ms
m3	170ms	131ms	202ms	246ms
m4	179ms	124ms	148ms	182ms
m5	148ms	173ms	240ms	150ms
m6	381ms	374ms	407ms	253ms
m7	235ms	262ms	200ms	226ms
m8	171ms	242ms	201ms	358ms

Tabela 29: Dados de 1 a 4 *threads* utilizados no gráfico da Figura 61 - Tempo de execução em ambiente heterogêneo utilizando diferentes quantidades de *threads* (pg. 119)

Processos	1 *thread*	2 *threads*	3 *threads*	4 *threads*
1	21s 549ms	18s 661ms	17s 19ms	16s 325ms
2	25s 786ms	20s 595ms	19s 265ms	18s 678ms
3	26s 609ms	21s 112ms	20s 21ms	19s 567ms
4	28s 400ms	22s 51ms	21s 65ms	21s 229ms
5	28s 264ms	22s 178ms	21s 537ms	22s 274ms
6	41s 796ms	35s 585ms	36s 89ms	37s 84ms
7	45s 995ms	37s 921ms	39s 761ms	40s 738ms
8	90s 804ms	64s 693ms	67s 934ms	73s 403ms

Tabela 30: Dados de 5 a 8 *threads* utilizados no gráfico da Figura 61 - Tempo de execução em ambiente heterogêneo utilizando diferentes quantidades de *threads* (pg. 119)

Processos	5 *threads*	6 *threads*	7 *threads*	8 *threads*
1	15s 827ms	15s 548ms	15s 193ms	14s 855ms
2	18s 617ms	18s 932ms	18s 732ms	18s 424ms
3	19s 623ms	19s 252ms	19s 22ms	19s 187ms
4	21s 899ms	22s 406ms	22s 463ms	22s 71ms
5	22s 771ms	23s 177ms	23s 578ms	23s 402ms
6	39s 71ms	41s 76ms	41s 690ms	42s 382ms
7	43s 16ms	44s 349ms	46s 65ms	46s 612ms
8	74s 990ms	76s 896ms	78s 460ms	78s 962ms

Tabela 31: Desvios padrão dos dados de 1 a 4 *threads* utilizados no gráfico da Figura 61 - Tempo de execução em ambiente heterogêneo utilizando diferentes quantidades de *threads* (pg. 119)

Processos	1 *thread*	2 *threads*	3 *threads*	4 *threads*
1	145ms	312ms	233ms	179ms
2	729ms	216ms	197ms	369ms
3	2s 462ms	533ms	326ms	304ms
4	2s 413ms	375ms	221ms	286ms
5	1s 695ms	195ms	170ms	366ms
6	2s 521ms	610ms	504ms	471ms
7	2s 716ms	513ms	393ms	558ms
8	8s 951ms	5s 437ms	3s 421ms	2s 23ms

Tabela 32: Desvios padrão dos dados de 5 a 8 *threads* utilizados no gráfico da Figura 61 - Tempo de execução em ambiente heterogêneo utilizando diferentes quantidades de *threads* (pg. 119)

Processos	5 *threads*	6 *threads*	7 *threads*	8 *threads*
1	148ms	255ms	104ms	217ms
2	280ms	338ms	207ms	186ms
3	373ms	292ms	183ms	250ms
4	409ms	398ms	388ms	283ms
5	370ms	482ms	497ms	360ms
6	332ms	541ms	422ms	643ms
7	690ms	674ms	607ms	576ms
8	615ms	1s 401ms	1s 772ms	1s 731ms

Tabela 33: Dados utilizados no gráfico da Figura 62 - Tempo de execução em ambiente heterogêneo utilizando a mesma quantidade de dados de entrada (pg. 120)

Nro. de Máquinas	Sistema	Hadoop	Spark
5	47s 367ms	57s 850ms	29s 119ms
7	46s 715ms	57s 808ms	26s 270ms
8	73s 403ms	63s 75ms	29s 195ms

Tabela 34: Desvio padrão dos dados utilizados no gráfico da Figura 62 - Tempo de execução em ambiente heterogêneo utilizando a mesma quantidade de dados de entrada (pg. 120)

Nro. de Máquinas	Sistema	Hadoop	Spark
5	3s 435ms	799ms	1s 14ms
7	2s 496ms	1s 346ms	497ms
8	2s 23ms	1s 529ms	430ms

Tabela 35: Dados utilizado no gráfico da Figura 63 - Quantidade de bytes trafegados (pg. 121)

Tráfego	Sistema	Hadoop	Spark
Quantidade (bytes)	158506001	68156474	74618444

Tabela 36: Dados utilizado no gráfico da Figura 64 - Quantidade de pacotes trafegados (pg. 122)

Tráfego	Sistema	Hadoop	Spark
Número de pacotes trafegados	428612	7445	16650

Tabela 37: Dados utilizado no gráfico da Figura 65 - Quantidade de pacotes de diferentes tamanhos trafegados (pg. 122)

Tamanho (bytes)	Sistema	Hadoop	Spark
0 a 19	0	0	0
20 a 39	0	0	0
40 a 79	30774	2089	3688
80 a 159	147589	9	5
160 a 319	158023	12	8
320 a 639	52219	31	42
640 a 1279	9951	47	37
1280 a 2559	27442	289	7904
2560 a 5119	559	545	400
5119 em diante	0	0	0

Tabela 38: Dados utilizado no gráfico da Figura 66 - Tempo de execução com entrada utilizando blocos menores (pg. 123)

Nro. de processos	Sistema	Hadoop	Spark
1	8s 448ms	22s 800ms	13s 885ms
2	9s 515ms	24s 146ms	15s 591ms
3	9s 826ms	25s 13ms	15s 952ms
4	10s 518ms	26s 945ms	17s 240ms
5	10s 930ms	29s 409ms	17s 857ms
6	17s 512ms	31s 751ms	18s 831ms
7	19s 516ms	36s 687ms	19s 800ms
8	33s 874ms	39s 106ms	20s 757ms

Tabela 39: Desvio padrão dos dados utilizado no gráfico da Figura 66 - Tempo de execução com entrada utilizando blocos menores (pg. 123)

Nro. de processos	Sistema	Hadoop	Spark
1	45ms	561ms	411ms
2	89ms	527ms	266ms
3	102ms	600ms	450ms
4	118ms	489ms	432ms
5	83ms	477ms	471ms
6	249ms	271ms	423ms
7	99ms	510ms	454ms
8	390ms	695ms	530ms

Tabela 40: Dados utilizado no gráfico da Figura 67 - Tempo de execução utilizando diversos blocos por máquina (pg. 124)

	Tempo
Sistema	131s 993ms
Hadoop	157s 752ms
Spark	73s 510ms

Tabela 41: Dados utilizado no gráfico da Figura 67 - Tempo de execução utilizando diversos blocos por máquina (pg. 124)

	Tempo
Sistema	1s 352ms
Hadoop	1s 76ms
Spark	3s 182ms

REFERÊNCIAS

ALY, A. et al. M3: Stream Processing on Main-Memory MapReduce. In: *2012 IEEE 28th International Conference on Data Engineering (ICDE)*. Washington: IEEE, 2012. p. 1253–1256.

AMARASINGHE, S. et al. *ExaScale Software Study: Software Challenges in Extreme Scale Systems*. [S.l.: s.n.], 2009. Acesso em: 20 abr. 2018.

ANAGNOSTOPOULOS, I.; ZEADALLY, S.; EXPOSITO, E. Handling big data: research challenges and future directions. *The Journal of Supercomputing*, v. 72, n. 4, p. 1494–1516, abr. 2016. ISSN 0920-8542, 1573-0484. Disponível em: <https://link.springer.com/article/10.1007/s11227-016-1677-z>. Acesso em: 20 abr. 2018.

APACHE. *Apache Hadoop*. 2014. Disponível em: <http://hadoop.apache.org>. Acesso em: 01 jun. 2015.

APACHE. *Apache Hive TM*. 2014. Disponível em: <http://hive.apache.org/>. Acesso em: 28 out. 2015.

APACHE. *Apache Spark - Unified Analytics Engine for Big Data*. 2015. Disponível em: <http://spark.apache.org/>. Acesso em: 28 out. 2015.

APACHE. *Welcome to Apache Pig!* 2015. Disponível em: <https://pig.apache.org/>. Acesso em: 28 out. 2015.

APACHE. *Apache Tez - Welcome to Apache TEZ ®*. 2018. Disponível em: <https://tez.apache.org/>. Acesso em: 28 mar. 2018.

BHATOTIA, P. et al. Incoop: MapReduce for Incremental Computations. In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. New York: ACM, 2011. (SOCC '11), p. 7. Disponível em: <http://doi.acm.org/10.1145/2038916.2038923>. Acesso em: 20 abr. 2018.

BIN, S. et al. A Big Data Analytics Approach to Develop Industrial Symbioses in Large Cities. *Procedia CIRP*, v. 29, p. 450–455, 2015. ISSN 2212-8271. Disponível em: <http://www.sciencedirect.com/science/article/pii/S2212827115005016>. Acesso em: 20 abr. 2018.

BOLLIER, D.; FIRESTONE, C. M. *The promise and peril of big data*. Washington: The Aspen Institute, 2010.

BORKAR, V. R.; CAREY, M. J.; LI, C. Big Data Platforms: What's Next? *XRDS*, v. 19, n. 1, p. 44–49, set. 2012. ISSN 1528-4972. Disponível em: <http://doi.acm.org/10.1145/2331042.2331057>. Acesso em: 20 abr. 2018.

BORTHAKUR, D. *The Hadoop Distributed File System: Architecture and design*. 2008. Disponível em: <https://svn.apache.org/repos/asf/hadoop/common/tags/release-0.16.0/docs/hdfs_design.pdf>. Acesso em: 03 jul. 2018.

BULPIN, J. R. Multiprogramming Performance of the Pentium 4 with Hyper-Threading. *WDDD, 2004*, p. 53–64, 2004. Disponível em: <http://ci.nii.ac.jp/naid/10019399382/>. Acesso em: 20 abr. 2018.

BUYYA, R. *High Performance Cluster Computing: Architectures and Systems*. Melbourne: Prentice Hall, 1999. v. 1.

CAO, J. et al. Programming Platforms for Big Data Analysis. In: *Handbook of Big Data Technologies*. Springer International Publishing, 2017. p. 65–99. Disponível em: <http://link.springer.com/10.1007/978-3-319-49340-4_3>. Acesso em: 08 mar. 2018.

CHAMBERS, B.; ZAHARIA, M. *Spark: the definitive guide - Big Data processing made simple*. 1. ed. Sebastopol, CA, USA: O'Reilly Media, 2008. ISBN 1-4919-1221-9.

CHEN, M.; MAO, S.; LIU, Y. Big Data: A Survey. *Mobile Networks and Applications*, v. 19, n. 2, p. 171–209, abr. 2014. ISSN 1383-469X, 1572-8153. Disponível em: <http://link.springer.com/10.1007/s11036-013-0489-0>. Acesso em: 20 abr. 2018.

CHEN, R.; CHEN, H.; ZANG, B. Tiled-MapReduce: Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. New York, NY, USA: ACM, 2010. (PACT '10), p. 523–534. Disponível em: <http://doi.acm.org/10.1145/1854273.1854337>. Acesso em: 20 abr. 2018.

CONDIE, N. T. MapReduce Online. In: *Proceedings of 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*. San Jose: USENIX Association, 2010.

CORMEN, T. H. et al. *Introduction to Algorithms*. 2. ed. Cambridge, MA, EUA: The MIT Press, 2001.

DAGUM, L.; MENON, R. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science Engineering*, v. 5, n. 1, p. 46–55, jan. 1998. ISSN 1070-9924.

DEAN, J.; GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, v. 51, n. 1, p. 107–113, jan. 2008. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1327452.1327492>. Acesso em: 20 abr. 2018.

DIJKSTRA, E. W. Cooperating sequential processes. In: *The origin of concurrent programming*. New York, NY, USA: Springer, 1968. p. 65–138.

ÉCOLE POLYTECHNIQUE FÉDÉRALE. *The Scala Programming Language*. 2018. Disponível em: <https://www.scala-lang.org/>. Acesso em: 20 abr. 2018.

ELMASRI, R.; NAVATHE, S. *Sistemas de Banco de Dados*. 4. ed. São Paulo: Pearson, 2005.

ELNIKETY, E.; ELSAYED, T.; RAMADAN, H. iHadoop: Asynchronous Iterations for MapReduce. In: *2011 IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*. Athens: IEEE, 2011. p. 81–90.

EMANI, C. K.; CULLOT, N.; NICOLLE, C. Understandable Big Data: A survey. *Computer Science Review*, maio 2015. ISSN 1574-0137. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1574013715000064>. Acesso em: 20 abr. 2018.

FRAZIER, H. The 802.3z Gigabit Ethernet Standard. *IEEE Network*, v. 12, n. 3, p. 6–7, 1998. ISSN 0890-8044. Disponível em: <https://ieeexplore.ieee.org/document/690946>. Acesso em: 28 abr. 2018.

GEER, D. Chip makers turn to multicore processors. *Computer*, v. 38, n. 5, p. 11–13, maio 2005. ISSN 0018-9162.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The Google File System. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2003. (SOSP '03), p. 29–43. Disponível em: <http://doi.acm.org/10.1145/945445.945450>. Acesso em: 20 abr. 2018.

GIBSON, D. C.; WEBB, M. E. Data science in educational assessment. *Education and Information Technologies*, p. 1–17, jun. 2015. ISSN 1360-2357, 1573-7608. Disponível em: <http://link.springer.com/article/10.1007/s10639-015-9411-7>. Acesso em: 20 abr. 2018.

GNU. *GNU Bash*. 2018. Disponível em: <http://www.gnu.org/software/bash/>. Acesso em: 20 abr. 2018.

HEESCH, D. v. *Doxygen*. 2018. Disponível em: <http://www.stack.nl/~dimitri/doxygen/>. Acesso em: 28 mai. 2018.

HEINTZ, B.; CHANDRA, A.; WEISSMAN, J. Cross-Phase Optimization in MapReduce. In: LI, X.; QIU, J. (Ed.). *Cloud Computing for Data-Intensive Applications*. Springer New York, 2014. p. 277–302. Disponível em: <http://link.springer.com/chapter/10.1007/978-1-4939-1905-5_12>. Acesso em: 20 abr. 2018.

IBM; ZIKOPOULOS, P.; EATON, C. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. 1. ed. New York: McGraw-Hill Osborne Media, 2011.

IEEE. IEEE Standard for Prefixes for Binary Multiples. *IEEE Std 1541-2002 (R2008)*, p. c1–4, set. 2009. Disponível em: <https://doi.org/10.1109/IEEESTD.2009.5254933>. Acesso em: 31 mai. 2018.

IEEE; THE OPEN GROUP. *<dlfcn.h>*. 2001. Disponível em: <http://pubs.opengroup.org/onlinepubs/009695399/basedefs/dlfcn.h.html>. Acesso em: 20 abr. 2018.

INTEL. *Intel® Xeon® Processor E5335 (8M Cache, 2.00 GHz, 1333 MHz FSB) Product Specifications*. 2007. Disponível em: <https://ark.intel.com/products/28443/Intel-Xeon-Processor-E5335-8M-Cache-2_00-GHz-1333-MHz-FSB>. Acesso em: 09 ago. 2018.

INTEL. *Intel® Core™ i7-3770 Processor*. 2012. Disponível em: <https://ark.intel.com/pt-br/products/65719/Intel-Core-i7-3770-Processor-8M-Cache-up-to-3_90-GHz>. Acesso em: 28 abr. 2018.

INTEL. *Intel® Xeon® Processor E5-2620 (15M Cache, 2.00 GHz, 7.20 GT/s Intel® QPI) Product Specifications*. 2012. Disponível em: <https://ark.intel.com/products/64594/Intel-Xeon-Processor-E5-2620-15M-Cache-2_00-GHz-7_20-GTs-Intel-QPI>.

KAISLER, S. et al. Big Data: Issues and Challenges Moving Forward. In: *2013 46th Hawaii International Conference on System Sciences (HICSS)*. Wailea: IEEE, 2013. p. 995–1004. Acesso em: 20 abr. 2018.

KARAU, H. et al. *Learning Spark*. Sebastopol: O'Reilly Media, 2015. Disponível em: <http://shop.oreilly.com/product/0636920028512.do>. Acesso em: 20 abr. 2018.

KERRISK, M. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. San Francisco, CA, USA: No Starch Press, 2010.

KHOURY, M. J.; IOANNIDIS, J. P. A. Big data meets public health. *Science*, v. 346, n. 6213, p. 1054–1055, nov. 2014. ISSN 0036-8075, 1095-9203.

KUSIAK, A. Break through with BIG DATA. *Industrial Engineer: IE*, v. 47, n. 3, p. 38–42, mar. 2015. ISSN 1542894X.

LI, B. et al. A Platform for Scalable One-pass Analytics Using MapReduce. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2011. (SIGMOD '11), p. 985–996. Disponível em: <http://doi.acm.org/10.1145/1989323.1989426>. Acesso em: 20 abr. 2018.

LIN, J. Mapreduce is Good Enough?If All You Have is a Hammer, Throw Away Everything That's Not a Nail! *Big Data*, v. 1, n. 1, p. 28–37, nov. 2012. ISSN 2167-6461. Disponível em: <http://online.liebertpub.com/doi/abs/10.1089/big.2012.1501>. Acesso em: 20 abr. 2018.

LINUX QUESTIONS. *Is C malloc slow?* 2009. Disponível em: <https://www.linuxquestions.org/questions/programming-9/is-c-malloc-slow-703979/>. Acesso em: 20 abr. 2018.

LU, H.; HAI-SHAN, C.; TING-TING, H. Research on Hadoop Cloud Computing Model and its Applications. In: *Proceedings of Third International Conference on Networking and Distributed Computing*. Hong Kong, China: IEEE, 2012. p. 59–63. ISBN 978-1-4673-2858-6 978-0-7695-4832-6. Disponível em: <http://ieeexplore.ieee.org/document/6386653/>. Acesso em: 09 ago. 2018.

MAGALHÃES, J. L. et al. Big Data e a saúde negligenciada em dengue, zika e chicungunha: uma análise translacional da tríplice ameaça no século 21. *Cência da Informação*, v. 45, n. 3, 2018.

MAZUR, E. et al. Towards Scalable One-Pass Analytics Using MapReduce. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. Shangai: IEEE, 2011. p. 1102–1111.

MGI. *Big data: The next frontier for innovation, competition, and productivity*. [S.l.], 2011. 156 p. Disponível em: <http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation>. Acesso em: 20 abr. 2018.

MPI. *Message Passing Interface (MPI) Forum Home Page*. 2015. Disponível em: <http://www.mpi-forum.org/>. Acesso em: 04 dez. 2015.

MPI. *MPI Documents*. 2015. Disponível em: <http://www.mpi-forum.org/docs/docs.html>. Acesso em: 04 dez. 2015.

MPICH. *MPICH High-Performance Portable MPI*. 2015. Disponível em: <https://www.mpich.org/>. Acesso em: 04 dez. 2015.

MURTHY, A. C. et al. *Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2*. 1. ed. Upper Saddle River, NJ: Addison-Wesley Professional, 2014.

MUTHITACHAROEN, A. et al. Ivy: A Read/Write Peer-to-peer File System. *SIGOPS Oper. Syst. Rev.*, v. 36, n. SI, p. 31–44, dez. 2002. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/844128.844132>. Acesso em: 20 abr. 2018.

NICHOLS, B.; BUTTLAR, D.; FARRELL, J. *PThreads Programming: A POSIX Standard for Better Multiprocessing*. Sebastopol, CA, USA: O'Reilly Media, Inc., 1996.

OPENMPI. *Open MPI: Open Source High Performance Computing*. 2015. Disponível em: <http://www.open-mpi.org/>. Acesso em: 04 dez. 2015.

ORACLE. *Java*. 2018. Disponível em: <https://www.java.com/en/>. Acesso em: 20 abr. 2018.

PENG, D.; DABEK, F.; INC, G. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In: *9th USENIX Symposium on Operating Systems Design and Implementation*. Vancouver: USENIX Association, 2010. p. 4–6.

PYTHON SOFTWARE FOUNDATION. *Welcome to Python.org*. 2018. Disponível em: <https://www.python.org/>. Acesso em: 20 abr. 2018.

QUINN, M. J. *Parallel Programming in C with MPI and OpenMP*. [S.l.]: McGraw-Hill Education Group, 2004.

RAGUNATHAN, T. et al. Advertisement Posting Based on Consumer Behaviour. *Procedia Computer Science*, v. 50, p. 329–334, 2015. ISSN 1877-0509. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050915005414>. Acesso em: 20 abr. 2018.

RANGER, C. et al. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: *IEEE 13th International Symposium on High Performance Computer Architecture, 2007. HPCA 2007*. Scottsdale: IEEE, 2007. p. 13–24.

REED, D. A.; DONGARRA, J. Exascale computing and big data. *Communications of the ACM*, v. 58, n. 7, p. 56–68, jun. 2015. ISSN 00010782. Disponível em: <http://dl.acm.org/citation.cfm?doid=2797100.2699414>. Acesso em: 20 abr. 2018.

REHMANN, K.-T.; SCHOETTNER, M. An In-Memory Framework for Extended MapReduce. In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*. Tainan: IEEE, 2011. p. 17–24.

SAGIROGLU, S.; SINANC, D. Big data: A review. In: *2013 International Conference on Collaboration Technologies and Systems (CTS)*. San Diego: IEEE, 2013. p. 42–47.

SAHA, B. et al. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM Press, 2015. p. 1357–1369. ISBN 978-1-4503-2758-9. Disponível em: <https://doi.org/10.1145/2723372.2742790>. Acesso em: 01 jun. 2018.

SARKAR, V.; HARROD, W.; SNAVELY, A. E. Software challenges in extreme scale systems. *Journal of Physics: Conference Series*, v. 180, n. 1, p. 012045, 2009. ISSN 1742-6596. Disponível em: <http://stacks.iop.org/1742-6596/180/i=1/a=012045>. Acesso em: 20 abr. 2018.

SEDGEWICK, R.; WAYNE, K. *Algorithms*. 4. ed. Princeton, USA: Addison-Wesley, 2014. v. 1. Disponível em: <http://algs4.cs.princeton.edu/home/>. Acesso em: 20 abr. 2018.

SHEPLER, S. et al. *RFC 3530: Network File System (NFS) version 4 Protocol*. 2003. Disponível em: <https://tools.ietf.org/html/rfc3530>. Acesso em: 28 out. 2015.

SHNEIDERMAN, B. The Big Picture for Big Data: Visualization. *Science*, v. 343, n. 6172, p. 730–730, fev. 2014. ISSN 0036-8075, 1095-9203. Disponível em: <http://sciencemag-org.ez67.periodicos.capes.gov.br/content/343/6172/730.1>. Acesso em: 20 abr. 2018.

SHVACHKO, K. et al. The Hadoop Distributed File System. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. Incline Village: IEEE, 2010. p. 1–10.

SHVACHKO, K. V. Apache Hadoop: The scalability update. *;login: The Usenix Magazine*, v. 36, n. 3, p. 7–13, jun. 2011. Disponível em: <https://www.usenix.org/publications/login/june-2011-volume-36-number-3>. Acesso em: 08 ago 2018.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Fundamentos de sistemas operacionais*. 6. ed. Rio de Janeiro, RJ: LTC Livros técnicos e científicos, 2004.

SINGH, D.; REDDY, C. K. A survey on platforms for big data analytics. *Journal of Big Data*, v. 2, n. 1, p. 1–20, out. 2014. ISSN 2196-1115. Disponível em: <http://link.springer.com/article/10.1186/s40537-014-0008-6>. Acesso em: 20 abr. 2018.

SINGH, S. P.; JAISWAL, U. C. Machine Learning for Big Data: A New Perspective. *International Journal of Applied Engineering Research*, v. 13, n. 5, p. 2753–2762, 2018.

STALLINGS, W. *Arquitetura e Organização de Computadores: Projeto para o Desempenho*. 8. ed. São Paulo: Prentice Hall, 2010.

SUSE. *OpenSUSE - Linux OS*. 2018. Disponível em: <https://www.opensuse.org>. Acesso em: 27 abr. 2018.

TANENBAUM, A.; WOODHULL, A. *Sistemas operacionais: projeto e implementação*. 2. ed. Porto Alegre, RS: Bookman, 2000.

TANENBAUM, A. S.; STEEN, M. V. *Distributed Systems: Principles and Paradigms*. 2. ed. Upper Saddle River, NJ: Pearson, 2007.

TCPDUMP. *Tcpdump/Libpcap public repository*. 2018. Disponível em: <http://www.tcpdump.org>. Acesso em: 09 ago. 2018.

TERZI, D. S.; TERZI, R.; SAGIROGLU, S. Big data analytics for network anomaly detection from netflow data. In: *Computer Science and Engineering (UBMK), 2017 International Conference on*. Antalya, Turkey: IEEE, 2017. p. 592–597.

THE R FOUNDATION. *R: The R Project for Statistical Computing*. 2018. Disponível em: <https://www.r-project.org/>. Acesso em: 20 abr. 2018-04-20.

THUSOO, A. et al. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.*, v. 2, n. 2, p. 1626–1629, ago. 2009. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/1687553.1687609>. Acesso em: 20 abr. 2018.

ULFM. *User Level Failure Mitigation*. 2015. Disponível em: <http://fault-tolerance.org/ulfm/>. Acesso em: 09 out. 2015.

VAITSIS, C.; NILSSON, G.; ZARY, N. Visual analytics in healthcare education: exploring novel ways to analyze and represent big data in undergraduate medical education. *PeerJ*, v. 2, p. e683, nov. 2014. ISSN 2167-8359. Disponível em: <https://peerj.com/articles/683>. Acesso em: 20 abr. 2018.

VASATA, D.; SATO, L. M. Utilização de MPI com o sistema de arquivos distribuído HDFS. In: *Anais do V ERAD-SP*. São Bernardo do Campo: SBC, 2014.

VASATA, D.; SATO, L. M. Mapreduce com múltiplos blocos de execução atuando de forma coordenada utilizando entradas e saídas interligadas. In: *Anais do VI ERAD-SP*. São José do Rio Preto: SBC, 2015.

VASATA, D.; SATO, L. M. Cross Join Operation Over Large Quantities of Data Considering Data Locality and Data Replication. In: *Proceedings of 19th IEEE International Conference on Computational Science and Engineering*. Paris, France: IEEE, 2016. p. 522–525. ISBN 978-1-5090-3593-9. Disponível em: <https://ieeexplore.ieee.org/document/7982297/>. Acesso em: 28 mai. 2018.

VASATA, D.; SATO, L. M. Execução de aplicações Mapreduce utilizando paralelismo local e distribuído. In: *Anais do VII ERAD-SP*. São Paulo: SBC, 2016.

VASATA, D.; SATO, L. M. Estratégia para fase de Merge utilizando agrupamento local. In: *Anais do VIII ERAD-SP*. São Carlos: SBC, 2017.

W3C. *Extensible Markup Language (XML)*. 2018. Disponível em: <https://www.w3.org/XML/>. Acesso em: 20 abr. 2018.

WANG, K.; RAJENDRAN, A.; RAICU, I. MATRIX: MAny-Task computing execution fabRIc at eXascale. *Tech Report, IIT*, 2013.

WANG, K. et al. Optimizing load balancing and data-locality with data-aware scheduling. In: *Big Data (Big Data), 2014 IEEE International Conference on*. Washington, DC, USA: IEEE, 2014. p. 119–128.

WEST, D. M. Big Data for Education: Data Mining, Data Analytics, and Web Dashboards. Governance Studies at Brookings. *Brookings Institution*, set. 2012.

WIRESHARK. *Wireshark · Go Deep.* 2018. Disponível em: <https://www.wireshark.org/>. Acesso em: 09 ago. 2018.

ZAHARIA, M. et al. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2012. (NSDI'12). Disponível em: <http://dl.acm.org/citation.cfm?id=2228298.2228301>. Acesso em: 03 jul. 2018.

ZHANG, J.; ZHU, J. Research Intelligent Precision Marketing of E-commerce Based on the Big Data. *Journal of Management and Strategy*, v. 5, n. 1, fev. 2014. ISSN 1923-3973, 1923-3965. Disponível em: <http://www.sciedu.ca/journal/index.php/jms/article/view/4236>. Acesso em: 20 abr. 2018.

ZHAO, W.; MA, H.; HE, Q. Parallel K-Means Clustering Based on MapReduce. In: HUTCHISON, D. et al. (Ed.). *Cloud Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. v. 5931, p. 674–679. ISBN 978-3-642-10664-4 978-3-642-10665-1. Disponível em: <http://link.springer.com/10.1007/978-3-642-10665-1_71>.

GLOSSÁRIO

Ambiente de execução: Conjunto de *hardware* e *software* necessários para a execução de determinado conjunto de aplicações.

Árvore: Estrutura de dados organizada de maneira hierárquica. Os itens da árvore são denominados de nós da árvore, onde um nó superior é dito ser um nó pai e um nó inferior é chamado de nó filho. Ao primeiro nó da árvore denomina-se nó raiz.

Árvore rubro-negra: Estrutura de dados em árvore organizadas de acordo com regras de balanceamento, tendo os nós da árvore classificados em nós vermelhos ou nós negros.

Assíncrono: Comunicação em que o emissor não aguarda o recebimento da informação pelo receptor para continuar a sua execução.

Bash: Bourne-Again Shell. Interface de comunicação com o sistema operacional Linux.

Biblioteca: Conjunto de rotinas ou funções pré-definidas utilizadas para facilitar o desenvolvimento de aplicações.

Big Data: Tecnologias envolvendo a análise e processamento de grandes quantidades de dados.

Bit: Unidade básica de informação digital, podendo assumir valor 0 ou 1.

Bloco: Porção abstrata, parte de uma estrutura de um fluxo de execução.

Broadcast: Forma de transmissão coletiva de informações. Em um grupo de máquinas interconectadas, uma única máquina atua como emissor e todas as demais atuam como receptores.

Byte: Conjunto de oito (8) bits.

Cache: Região de memória com dados armazenados temporariamente, com o objetivo de acelerar a recuperação de dados em dispositivos de acessos de menor velocidade.

Ciclo: Repetição em um fluxo de execução. Durante a execução de aplicações em um fluxo de execução com ciclos, deve-se existir o controle da execução no ciclo, de modo a controlar a parada das repetições.

Cluster: Conjunto de computadores de propósito geral interconectados por uma rede de comunicação, com uma finalidade única, atuando de maneira coordenada.

Comutador: Elemento responsável pela comunicação de computadores. Opera na camada de enlace, encaminhando *frames* de acordo com o endereço físico de destino. Possui diversas portas, de forma que cada porta pertence a um domínio de colisão diferente. Também são conhecidos como *switches*.

CPU: Unidade central de processamento. Sigla do termo inglês *Central Processing Unit*.

Critério de parada: Condição necessária para saída de um ciclo.

Custo de comunicação: Esforço necessário para realizar a transferência de informações entre dispositivos computacionais.

Endereço IP: Código que identifica um computador em uma rede.

Exabyte: Conjunto de mil e vinte e quatro (1024) petabytes.

Fortran: Linguagem de programação de alto nível, desenvolvida para aplicações matemáticas e científicas.

Front Side Bus: Barramento frontal, responsável pela comunicação entre os processador e o *chipset*.

Gigabyte: Conjunto de mil e vinte e quatro (1024) megabytes.

Grafo: Conjunto de elementos compostos por vértices e arestas (também conhecidos como arcos). Grafos são amplamente utilizados para a representação abstrata de problemas, de maneira que o estudo do grafo específico pode levar à solução do problema. O estudo dos grafos é conhecido como teoria dos grafos.

Granularidade: Nível de subdivisão de uma determinada informação ou sistema. Diz-se que uma subdivisão em grande quantidade possui granularidade fina e uma subdivisão em menor quantidade possui granularidade baixa.

Hash: Porção de dados de tamanho fixo que representa unicamente uma mensagem. Um *hash* é obtido com a utilização de uma função de *hash*, onde a mensagem é a entrada para a função e o *hash* a saída da função. Caso descubra-se duas mensagens que possuam o mesmo valor de *hash*, diz-se que houve uma colisão de *hash*. Também é conhecido como resumo de uma mensagem.

ICMP: Protocolo utilizado para obter informações e relatórios de erros sobre dispositivos utilizando a rede de comunicação. É uma sigla do termo inglês *Intenet Control Message Protocol*.

Internet: Conjunto de computadores interligados globalmente, proporcionando a utilização de serviços.

Java: Linguagem de programação orientada a objetos, compilada e interpretada. Foi desenvolvida na década de 90, e é amplamente utilizada devido às suas características de portabilidade.

Kernel: Núcleo do Sistema Operacional.

Kilobyte: Conjunto de mil e vinte e quatro (1024) bytes.

K-Means: Algoritmo de agrupamento de dados, que realiza particionamentos com base em pontos de observação, de maneira que cada observação pertence ao grupo que mais se aproxima da média do grupo.

Lista: Estrutura de dados organizadas em itens de forma sequencial, de forma que cada item possui um item anterior e um item seguinte.

Mainframe: Computador de grande porte.

Map: Fase do modelo de programação MapReduce, em que a informação é organizada em chaves (k) e valores (v).

Mapreduce: Modelo de programação distribuído composto por diversas fases, amplamente utilizado em sistemas com manipulação de grandes quantidades de dados.

map: Tarefa executada na fase de Map do modelo Mapreduce.

Megabyte: Conjunto de mil e vinte e quatro (1024) kilobytes.

Multicore: Tecnologia de *hardware* processador, compostos de diversos núcleos de processamento.

Multinúcleo: *Hardware* processador, compostos de diversos núcleos de processamento encapsulados.

Multiprocessadores: Sistema comporto por múltiplos processadores, encapsulados separadamente.

Nó: Máquina componente de um sistema de computação distribuído.

Page Rank: Algoritmo de classificação de elementos, que consiste em aplicar pesos com base nas relações com os demais elementos. Foi amplamente utilizado para a definição da ordem em que as páginas web devem aparecer em sites de busca.

Petabyte: Conjunto de mil e vinte e quatro (1024) terabytes.

Ping: Aplicação utilizada para teste de comunicação entre dispositivos computacionais.

Ponto a ponto: Arquitetura de comunicação em que os nós comunicam-se diretamente, e atuam simultaneamente como cliente e como servidor, dispensando a utilização de um servidor central. Também pode ser representado como arquitetura P2P.

Python: Linguagem de programação interpretada de alto nível, tipagem dinâmica e forte.

Reduce: Fase do modelo MapReduce, em que o processamento é realizado sobre todos os valores que possuem a mesma chave.

reduce: Tarefa executada na fase de Reduce do modelo Mapreduce.

Registro: Porção de dados, dada como item de uma coleção de informações organizadas.

R :Linguagem e ambiente de desenvolvimento amplamente utilizada para cálculos estatísticos.

Scala: Linguagem de programação de propósito geral.

Script: Código de execução de uma linguagem de programação interpretada.

Shuffle: Fase do modelo MapReduce, em que ocorre a troca de dados entre as diversas máquinas do ambiente distribuído. A máquina destino é definida com base no *hash* da chave.

Software: Conjunto de intruções lógicas que definem o funcionamento de um sistema computacional.

SQL: Linguagem de consulta estruturada, utilizada como padrão em pesquisas em bancos de dados relacionais. Acrônimo do termo em inglês *Structured Query Language*.

Tarefa: Parte de um trabalho realizado.

Terabyte: Conjunto de mil e vinte e quatro (1024) gigabytes.

Theta-Junção: Operação da álgebra relacional, que pode ser realizado com a execução da operação de produto cartesiano e uma função de seleção.

Thread: Linha ativa de execução de um determinado código. *Threads* que executam o mesmo código compartilham a mesma região de código, diferente do que acontece com a execução de processos. Uma *thread* também é conhecida como um processo leve.

Workflow: Fluxo de trabalho. Sequência de passos necessária para a automatização de processos, utilizando um conjunto definido de regras.

YARN: Nome adotada para as versões do Hadoop a partir da versão 2.0. É um acrônimo para *Yet Another Resource Negociator*.