UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS DO DEPARTAMENTO ACADÊMICO DE COMPUTAÇÃO

TAYLLAN BÚRIGO

ESTUDO COMPARATIVO DE ARCABOUÇOS PARA PROCESSAMENTO PARALELO DE GRAFOS

TRABALHO DE CONCLUSÃO DE CURSO

CORNÉLIO PROCÓPIO

2015

TAYLLAN BÚRIGO

ESTUDO COMPARATIVO DE ARCABOUÇOS PARA PROCESSAMENTO PARALELO DE GRAFOS

Trabalho de Conclusão de Curso apresentado ao Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas do Departamento Acadêmico de Computação da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador: Prof. Me. Francisco Pereira Junior

CORNÉLIO PROCÓPIO

2015

AGRADECIMENTOS

Primeiramente a mim, sem quem nada disso teria sido possível.

Ao meu orientator, Francisco Pereira Junior, pela paciência, confiança depositada e ajuda despendida. Demorei mas terminei!

A todos os professores e servidores da UTFPR Cornélio Procópio, pelos ensinamentos, dedicação e exemplos.

Aos meus pais, **porque né...**

Ao meu irmão Bruno, pelos ensinamentos, críticas, exemplo e monólogos. *And rule we shall*.

Aos meus amigos e colegas pelos bons e péssimos momentos compartilhados.

RESUMO

BÚRIGO, Tayllan. ESTUDO COMPARATIVO DE ARCABOUÇOS PARA PROCESSAMENTO PARALELO DE GRAFOS. 60 f. Trabalho de Conclusão de Curso – Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas do Departamento Acadêmico de Computação, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2015.

Diversas situações do mundo real podem ser modeladas em grafos, como mapas rodoviários, redes de abastecimento de energia elétrica e água, relações entre os usuários de redes sociais, entre outras. Com o alcance quase global da tecnologia, atualmente é gerada muito mais informação do que uma ou duas décadas atrás, o que faz com que o processamento e análise dessa se torne um problema computacional atual e bastante relevante. Diante desse fato, diversos arcabouços foram criados com o objetivo de processar de forma paralela e eficiente grafos com muitos vértices e arestas, com até bilhões e trilhões, respectivamente. O presente trabalho apresenta uma análise comparativa entre alguns dos muitos arcabouços de processamento paralelo de grafos existentes, utilizando os algoritmos do PageRank e do Caminho mais Curto. Ao final do mesmo também foram comentadas as qualidades, defeitos e possíveis melhorias a serem implementadas em cada arcabouço analisado.

Palavras-chave: Alto Desempenho, Big Data, Arcabouços, Processamento Paralelo, Grafos

ABSTRACT

BÚRIGO, Tayllan. COMPARATIVE STUDY OF FRAMEWORKS FOR PARALLEL PROCESSING OF GRAPHS. 60 f. Trabalho de Conclusão de Curso – Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas do Departamento Acadêmico de Computação, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2015.

Several real-world situations can be modeled in graphs, such as road maps, electricity and water supply networks, relationships among users of social networks, among others. With nearly global reach of technology, it is currently generated much more information than a decade or two ago, which makes the processing and analysis of it a very current and relevant computational problem. Given this fact, several frameworks have been created for the purpose of processing in a parallel and efficient way graphs with vertices and edges with up to billions and trillions, respectively. This work presents a comparative among some of the many frameworks of parallel processing of graphs existing, using the algorithms of PageRank and the Shortest Path. At the end of it are remarks on the qualities, defects and possible improvements to be implemented in the analyzed frameworks.

Keywords: High performance, Big Data, Frameworks, Parallel Processing, Graphs

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

API	*Application Programming Interface*
BSP	*Bulk-Synchronous Parallel*
CPU	*Central Processing Unit*
EC2	*Amazon Elastic Compute Cloud*
GB	*GigaByte*
GHz	*Gigahertz*
HDFS	*Hadoop Distributed File System*
MB	*MegaByte*
PSN	*Pocket Switched Network*
RAM	*Random-Access Memory*
RDD	*Resilient Distributed Dataset*
TCP	*Transfer Control Protocol*
URL	*Uniform Resource Locator*

API BSP CPU EC2 GB GHz HDFS MB PSN RAM RDD TCP URL

SUMÁRIO

1 INTRODUÇÃO

O aumento na quantidade de dados computacionais gerados todos os dias na web e em outras áreas da Ciência da Computação fez do processamento de dados um problema computacional extremamente relevante nos tópicos atuais.

Mais especificamente, dentro da área de processamento de dados, um dos itens que tem sido discutido atualmente na computação é o processamento de dados que são representados utilizando-se a estrutura de dados Grafo, ou seja, o processamento de grafos: utilizar um Grafo como fonte de dados de entrada, executar algum algoritmo sobre esses dados, e gerar alguma informação como dado de saída. Muitas informações, como a rede de amigos virtuais em redes sociais ou a estrutura de links da web, podem ser representadas utilizando a estrutura de dados Grafo.

Seguindo essa tendência, surgiram diversos arcabouços voltados especificamente para a execução de algoritmos baseados em grafos. Arcabouço é um programa que provê ao usuário funcionalidades genéricas, as quais podem ser modificadas e/ou aprimoradas, através de uma *Application Programming Interface* (API) bem definida; funcionalidades essas que podem ser utilizadas para facilitar e agilizar o desenvolvimento de novos programas (SOFTWARE..., 2015). Também houve a evolução de outros arcabouços mais genéricos para suportar, também, o processamento de grafos, como é o caso do Hadoop, arcabouço contruído para a execução de processamentos paralelos em geral, e não inteiramente voltado para o processamento de grafos (GUO et al., 2014).

Foram estudados, instalados e executados quatro arcabouços de processamento paralelo, sendo dois deles voltados especificamente ao processamento de grafos. Também foram estudados dois algoritmos de grafos, o PageRank e o Caminho mais Curto, os quais foram executados em todos os arcabouços como uma forma de realizar uma comparação de eficiência entre eles.

1.1 PROBLEMA

O mundo está vivenciando a "Era da Big Data" (quantidades massivas de informações) (LOHR, 2012). Em 2012, 2,5 Exabytes de informação eram gerados todos os dias, um número que tem dobrado a cada 40 meses desde então (MCAFEE; BRYNJOLFSSON et al., 2012). Considerando que boa parte dessa informação flui dentro da internet e pode ser representada utilizando grafos, o processamento de grandes grafos, com bilhões de vértices e trilhões de arestas tornou-se um problema computacional de extrema importância no contexto atual.

1.2 OBJETIVO GERAL

O objetivo deste Trabalho de Conclusão de Curto é realizar um estudo comparativo em relação à eficiência de processamento entre quatro arcabouços de processamento paralelo, Hadoop, Hama, Giraph e Spark, utilizando dois algoritmos de grafos, o PageRank e o Caminho mais Curto.

Para completar este trabalho foram realizadas as seguintes etapas:

- Estudo de algoritmos clássicos baseados em grafos, com ênfase nos algoritmos do PageRank e do Caminho mais Curto.
- Levantamento dos principais arcabouços de processamento paralelo de grafos.
- Análise, implementação, adaptação e execução de algoritmos baseados em grafos nos arcabouços de processamento paralelo levantados.
- Comparação dos resultados das execuções, levando em consideração a utilização de memória principal e tempo de execução em cada arcabouço.

1.3 JUSTIFICATIVA

O tamanho que os grafos atuais atingem torna impraticável executar algoritmos de grafos já assentados na computação em uma única máquina (KIM et al., 2013). Com essa necessidade de processar grandes grafos de forma paralela, surgiram diversos arcabouços que implementam o processamento paralelo. O que falta, entretanto, são comparações bem definidas entre os diversos arcabouços existentes, definindo os pontos fortes e fracos de cada um e quais tipos de algoritmos para grafos são melhor executados em quais arcabouços. Existem, contudo, poucas pesquisas e comparações realizadas nesse sentido, como é o caso de (GUO

et al., 2014) e (ELSER; MONTRESOR, 2013) que serão discutidas no capítulo de 2.12, sobre Trabalhos Relacionados.

1.4 ORGANIZAÇÃO DO TEXTO

A organização dos tópicos subsequentes desta monografia está dividida da seguinte forma: no capítulo 2 serão apresentados os principais conceitos que embasam este trabalho, assim como os trabalhos relacionados; no capítulo 3 será apresentado o processo de desenvolvimento seguido na realização da pesquisa; no capítulo 4 serão apresentados os resultados obtidos; e no capítulo 5 serão apresentadas as limitações que envolveram a presente pesquisa, assim como os trabalhos futuros que poderão dar continuidade a mesma.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos teóricos que embasaram a presente pesquisa.

2.1 GRAFOS

Um grafo consiste em um conjunto de nós (ou vértices) e arcos (ou arestas). Cada aresta em um grafo é especificada por um par de nós (TENENBAUM, 1990). As arestas podem ou não possuir um valor associado a elas: um grafo com arestas que possuam valor é um Grafo Ponderado, enquanto um grafo com arestas sem valor é um Grafo Não-Ponderado (CORMEN et al., 2009). As arestas podem, também, possuir direção, ou seja, uma aresta que vai de um nó a outro é dita uma aresta direcionada e, por conseguinte, o grafo se torna um Grafo Direcionado. Um grafo com arestas que não possuem direção é chamado de Grafo Não-Direcionado (CORMEN et al., 2009). Um Grafo Direcionado pode, também, ser chamado de Dígrafo (DIRECTED. . . , 2014).

Grafos podem, também, ser divididos em Grafos Esparsos e Grafos Densos. Grafos Esparsos são os grafos que possuem, em média, $|V|$ arestas, onde V representa o número de vértices do grafo, e Grafos Densos possuem $|V|^2$ arestas (SPARSE. . . , 2014).

Cada nó de um grafo possui duas medidas importantes, o grau de entrada e o grau de saída. O grau de entrada é definido como o número de arestas que terminam no nó do qual pretende-se medir o grau. Paralelamente, o grau de saída é o número de arestas saindo do nó a ser medido (FEOFILOFF, 2015).

2.2 FORMAS DE REPRESENTAÇÃO DE GRAFOS

Um grafo pode ser representado de duas formas padrão: uma Lista de Adjacência ou uma Matriz de Adjacência (ou de incidência). Os dois métodos podem ser aplicados tanto para Grafos Direcionados como para Grafos Não-Direcionados (CORMEN et al., 2009).

Considerando o Grafo Direcionado e Não-Ponderado da Figura 1, as Figuras 2 e 3 exemplificam a representação em forma de Matriz de Adjacência e Lista de Adjacência, respectivamente.

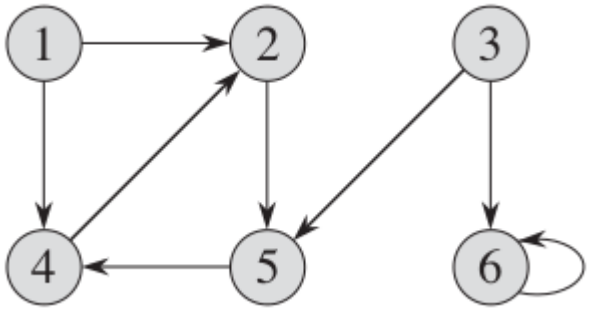

Figura 1: Grafo Direcionado e Não-Ponderado
Fonte: (CORMEN et al., 2009)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Figura 2: Matriz de Adjacência
Fonte: (CORMEN et al., 2009)

A Lista de Adjacência de um grafo $G = (V, E)$, onde V são os vértices e E as arestas, consiste de um vetor Ajd de $|V|$ listas. Para cada vértice u pertencente a V a lista $Adj[u]$ contém todos os vértices v que possuem conexão com u, ou seja, existe uma ou mais arestas ligando u e v diretamente (CORMEN et al., 2009).

A Matriz de Adjacência consiste de uma matriz de tamanho $|V|x|V|$ onde cada linha da matriz representa um nó. As posições (i, j) dentro da matriz podem assumir dois valores: 0 e 1, onde 0 indica que não existe uma conexão entre os nós i e j, e 1 indica que há um conexão entre esses dois nós. No caso de o grafo ser um Grafo Ponderado, o valor 1 pode ser substituído

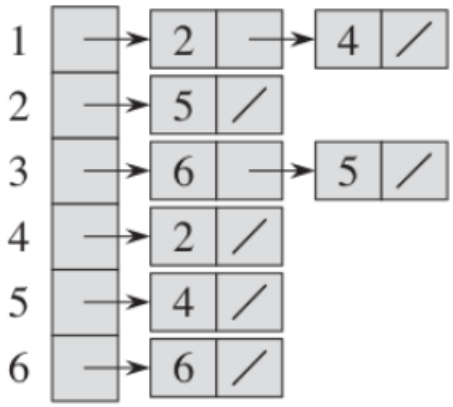

Figura 3: Lista de Adjacência
Fonte: (CORMEN et al., 2009)

pelo peso da aresta, enquanto o valor 0 pode ser substituído por qualquer valor que represente a inexistência de uma conexão, por exemplo *NULL* ou *NILL* (CORMEN et al., 2009).

A grande desvantagem de se utilizar a Lista de Adjacência para representar um grafo, e não a Matriz de Adjacência, está no custo da operação para descobrir se um par de nós (i, j) possui conexão, pois como deve-se percorrer toda a lista de nós destino a partir do nó *i*, chega-se a um tempo $\theta(n)$, onde *n* representa a quantidade de nós conectados a *i*, enquanto que na Matriz de Adjacência essa operação pode ser realizada em tempo constante $\theta(1)$, apenas acessando a posição (i, j) da matriz (CORMEN et al., 2009).

Já a maior desvantagem da Matriz de Adjacência é a quantidade de espaço necessário para armazená-la, na ordem de (V^2), enquanto que para a Lista de Adjacência esse espaço é apenas (V) para Grafos Esparsos (CORMEN et al., 2009).

Outro ponto a ser levado em consideração para decidir a melhor forma para representar um grafo é que os algoritmos para processamento de grafos em geral costumam percorrer todas as arestas, caso no qual a representação em forma de matriz é mais lenta, visto que para percorrer suas arestas é necessário percorrer toda a matriz. Um exemplo é o algoritmo de Dijkstra para encontrar o menor caminho em uma Grafo Ponderado com pesos positivos[1]. Sua implementação utilizando uma Lista de Adjacência e, como estrutura auxiliar, uma Fila de Prioridade tem um tempo de execução de (V^2) (CORMEN et al., 2009), enquanto a implementação utilizando uma Matriz de Adjacência tem um tempo de $(V^2 + E^2)$.

[1]DIJKSTRA, Edsger W. *A note on two problems in connection with graphs*. **Numerische mathematik**, v. 1, n. 1, p. 269-271, 1959.

2.3 APLICAÇÃO DE GRAFOS

Grafos podem ser utilizados para modelar qualquer tipo de relação entre duas ou mais entidades. Por exemplo, grafos podem modelar uma rede de rodovias, com cidades como os vértices e rodovias como os arcos (SKIENA, 2008). Muitos outros problemas atuais da computação também podem ser representados em grafos. Exemplos clássicos são os grafos da web e de muitas redes sociais. Algumas das áreas que utilizam grafos para solucionar problemas são: aplicações sociais, varejo online, inteligência de negócios e logística, e bioinformática (GUO et al., 2014).

Empresas estão utilizando grafos para modelar as relações entre seus consumidores (ou possíveis consumidores) e os serviços que elas disponibilizam, de forma que seja mais prático às empresas aferir o que seus consumidores necessitam:

- Grafos de Localização: esse setor de mercado está parcialmente dominado pela Foursquare. Ela utiliza o Venues Project (VENUES..., 2014) para estabelecer relações entre seus usuários e localidades que eles frequentam, e que tipo de relações são essas. Outras empresas que também planejam competir nesse setor são o Facebook, Google, Yelp e Gowalla (TROIA, 2014).

- Grafos de Saúde: é a representação em forma de grafos da saúde de uma pessoa, utilizando as medições de dados corporais, ações relacionadas à saúde e a evolução das informações com o tempo para construir um grafo que possa mapear facilmente a saúde da pessoa. A empresa dominante do setor é a RunKeeper, desenvolvedora do Health Graph (THE..., 2014) (TROIA, 2014).

- Grafos de Interesse: representação dos usuários e em que eles estão interessados. Por ser uma das bases da web atual, dificilmente uma empresa conseguirá dominar esse mercado sozinha, mas existem diversas empresas com foco na área: Facebook, Twitter, Google, e Quora (TROIA, 2014).

Outros exemplos de utilização de grafos são encontrados em (ARNOLD; PEETERS; THOMAS, 2004), onde os grafos são utilizados para modelar novos e melhores sistemas de transporte de cargas entre rodovias e trilhos, e em (HUI; CROWCROFT; YONEKI, 2011), onde são utilizados para representar a rede formada por dispositivos inteligentes, uma *Pocket Switched Network* (PSN) e também a interação entre esses dispositivos e seus usuários.

2.4 PROCESSAMENTO DE GRAFOS COM MUITOS VÉRTICES E ARESTAS

O volume dos conjuntos de dados que formam grafos se tornou grande que não é mais possível processá-los em uma única máquina em tempo hábil (KIM et al., 2013). A escala destes grafos - em muitos casos com bilhões de vértices e trilhões de arestas - implica desafios ao seu processamento de maneira eficiente (MALEWICZ et al., 2010).

Em abril de 2014 o Facebook contava com 1,28 bilhão de usuários (LUNDEN, 2014), logo, para competir no mercado de Grafos de Interesse ele deve ser capaz de processar os dados que tamanha quantidade de usuários geram diariamente. Nesse caso, processar os grafos implica criá-los, analisá-los e retirar informações úteis a partir da análise, e tudo deve ser feito em tempo adequado.

Grafos tão grandes violam a suposição de que eles podem ser armazenados em memória, ou sequer em discos rígidos, de um único computador, na forma como os algoritmos clássicos para grafos foram desenvolvidos. Desta forma, faz-se necessário repensar esses algoritmos e/ou desenvolver algoritmos paralelos e escaláveis para processar grafos que alcançam Terabytes ou mais de dados (KANG et al., 2011), e o desenvolvimento de arcabouços que implementem esses algoritmos e disponibilizem APIs facilmente utilizáveis pelos usuários/programadores.

2.5 BULK-SYNCHRONOUS PARALLEL

Um modelo para projetar algoritmos paralelos, o *Bulk-Synchronous Parallel* (BSP) é definido como a combinação de três atributos:

- Componentes: elementos do algoritmo que realizam o processamento;
- Roteador: elemento que realiza a comunicação entre pares de componentes;
- Sincronização: capacidade de sincronizar os componentes a cada intervalo de tempo definido pela aplicação.

O modelo BSP é executado em intervalos de tempo chamados superpassos. A cada superpasso um ou mais componentes realizam operações e podem enviar mensagens através do roteador a outros componentes. As mensagens são sincronizadas e transmitidas aos componentes ao final de cada superpasso. Após a sincronização de todas as mensagens, o próximo superpasso pode ser inicializado (VALIANT, 1990). A Figura 4 ilustra uma superpasso do modelo BSP.

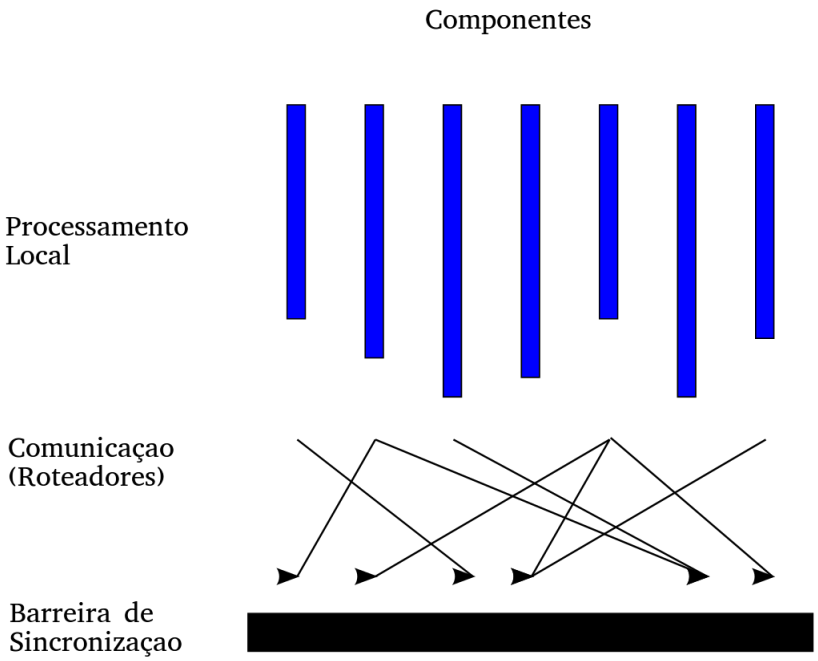

Figura 4: Ilustração de um Superpasso - BSP
Fonte: (BULK. . . , 2015)

Pode ser dito que a maior vantagem do modelo BSP é a simplicidade subjacente à sua implementação. Para adaptar um algoritmo ao modelo BSP basta implementá-lo em termos de componentes transmitindo mensagens a outros componentes, o que o torna adequado a algoritmos de grafos.

2.6 ARCABOUÇOS PARA PROCESSAMENTO DE GRAFOS COM MUITOS VÉRTICES E ARESTAS

Existem, atualmente, diversas iniciativas de arcabouços para processamento paralelo de alto desempenho de grafos. Alguns foram especificamente desenvolvidos com tal objetivo enquanto outros foram meramente adaptados para processar grafos. Neste trabalho foram estudados e executados apenas alguns deles, os quais foram escolhidos pela maior visibilidade que possuem na Web e no setor acadêmico, facilitando as consultas e referências.

- Hadoop: contruído para a execução de processamentos paralelos em geral, e não inteiramente voltado para o processamento de grafos, Hadoop tem se tornado o arcabouço *"de-facto"* para processamento de Big Data em paralelo (GUO et al., 2014).

- Hama: arcabouço construído em cima da arquitetura do Hadoop e que disponibiliza uma API para processamento de matrizes em geral e de grafos (SEO et al., 2010) (PEREIRA, 2013).

- Giraph: da mesma forma que o arcabouço Hama, Giraph também foi construído em cima do Hadoop, mas ao contrário destes dois (Hadoop e Hama), possui uma API voltada estritamente para o processamento de grafos (PEREIRA, 2013).

- Spark: um arcabouço de processamento paralelo de próposito geral, que disponibiliza APIs em Java, Scala e Python (SPARK. . . , 2015).

A seguir serão detalhados os arcabouços anteriormente listados.

2.6.1 HADOOP

Hadoop[2] é um arcabouço de código aberto, implementado em Java e utilizado para o processamento e armazenamento em larga escala, para alta demanda de dados, utilizando computadores de baixo custo.

- Arquitetura: Os elementos chave da arquitetura do Hadoop são o paradigma de programação MapReduce e o sistema de armazenamento distribuído *Hadoop Distributed File System* (HDFS). Para executá-los, Hadoop utiliza cinco processos Java: NameNode, DataNode, SecondaryNameNode, JobTracker e TaskTracker. Os três primeiros processos pertencem ao HDFS, enquanto os dois últimos pertencem ao paradigma MapReduce (GOLDMAN et al., 2012).

- MapReduce: MapReduce é um paradigma de programação e uma implementação para processar grandes conjuntos de dados. Usuários especificam duas funções: *map* e *reduce*. A função *map* recebe os dados de entrada e segmenta-os em tuplas de chave/valor, sobre as quais aplica cálculos especificados pelo usuário. Os valores gerados pelos cálculos intermediários são agrupados em tuplas chave/valor que possuam a mesma chave pela biblioteca que implemente o paradigma MapReduce (nesse caso, o arcabouço Hadoop). O valores agrupados são então passados à função *reduce*, a qual realiza outro conjunto de operações sobre todos os valores pertencentes a uma mesma chave (para todas as chaves existentes). A saída do programa é o conjunto de tuplas chave/valor gerado pela função *reduce* (DEAN; GHEMAWAT, 2008) (LÄMMEL, 2008). A Figure 5 demonstra o fluxo do processamento de dados realizado pelo MapReduce.

[2] Apache Hadoop. Disponível em: https://hadoop.apache.org/.

Dentro do Hadoop, as funções MapReduce são executadas pelo TaskTracker. E o JobTracker é responsável por gerenciar as execuções, escalonando-as às máquinas escravas do *cluster* e repetindo-as em caso de falha (GOLDMAN et al., 2012).

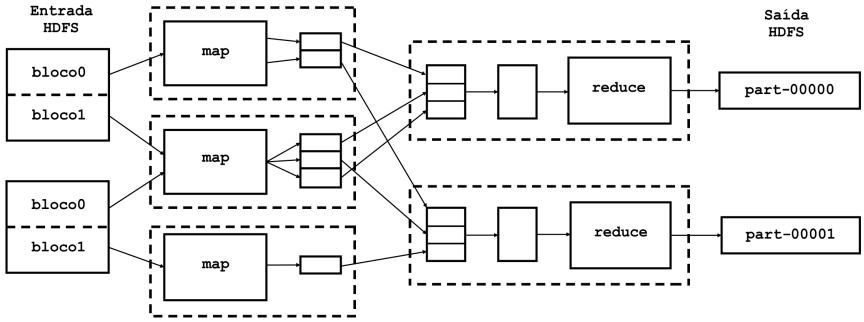

Figura 5: Fluxo de Execução do MapReduce
Fonte: (GOLDMAN et al., 2012)

- HDFS: sistema de arquivos distribuído do Hadoop. Ele foi criado para armazenar grandes conjuntos de dados de forma confiável, e para transmitir esses conjuntos de dados a uma grande velocidade às aplicações dos usuários.

  O HDFS armazena os metadados e os dados de aplicações separadamente. Os metadados são armazenados no NameNode, localizado na máquina mestre do *cluster*, e os dados das aplicações são armazenados em instâncias do DataNode, geralmente uma instância para cada máquina escrava do *cluster*. É comum, também, existir o processo SecondaryNameNode executando em outra máquina do *cluster*, o qual é responsável por realizar operações de checagem no NameNode e, em caso de falha deste último, possibilitar uma recuperação do *cluster* (WHITE, 2009) (GOLDMAN et al., 2012). Todos os servidores são conectados e se comunicam utilizando protocolos baseados no *Transfer Control Protocol* (TCP). A Figura 6 apresenta a arquitetura do HDFS.

- Execução: Para criar um programa a ser executado no arcabouço Hadoop o usuário deve implementar três classes Java (HOW. . . , 2015):

  - classe Driver: a classe que possui o método *main* Java (o primeiro método chamado na execução de um programa Java). Classe responsável por definir as configurações iniciais do programa, como: as classes Map e Reduce, arquivo(s) de entrada e de saída, formato do(s) arquivo(s) de entrada e de saída, entre outros.
  - classe Map: essa classe extende a classe Mapper encontrada na API do Hadoop[3]

[3]Neste trabalho foi utilizada a API do Hadoop versão 1.2.1

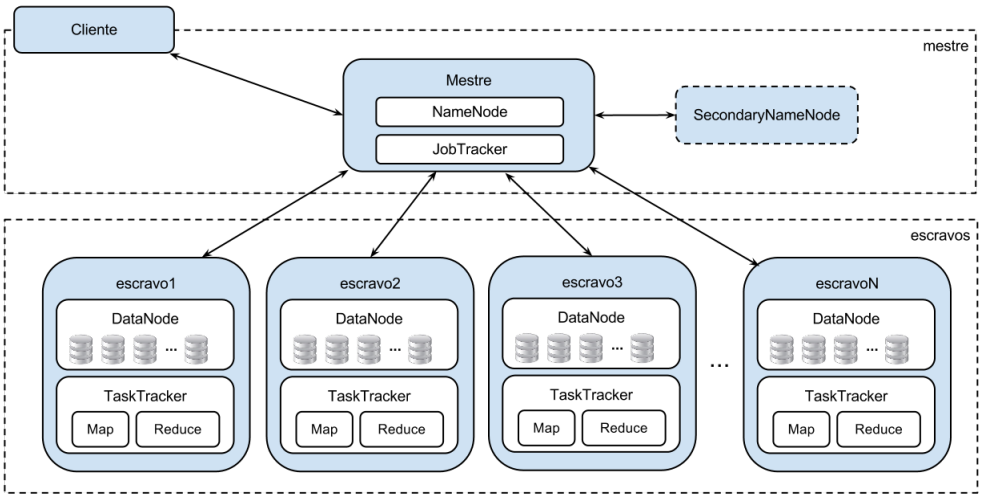

Figura 6: Arquitetura do HDFS

Fonte: (GOLDMAN et al., 2012)

e implementa o método *map*, o qual é responsável pelas mesmas funções anteriormente especificadas.

– classe Reduce: extende a classe Reducer encontrada na API do Hadoop e impleta o método *reduce*, responsável pelas mesmas funções acima especificadas.

2.6.2 HAMA

Hama[4] é um arcabouço distribuído construído em cima do Hadoop para processamento massivo de algoritmos de matrizes e grafos (SEO et al., 2010).

No início do projeto o Hama implementava em seu núcleo três mecanismos de processamento: o MapReduce do Hadoop, uma versão própria do paradigma BSP (VALIANT, 1990), e o mecanismo Dryad da Microsoft (ISARD et al., 2007). Atualmente o projeto Dryad da Microsoft foi descontinuado (FOLEY, 2014) e o arcabouço Hama está mais focado no processamento específico para grafos por meio do BSP, deixando de lado o MapReduce herdado do Hadoop.

- Arquitetura: o arcabouço Hama possui uma arquitetura de camadas com três componentes (SEO et al., 2010):
  - GroomServer: responsável por gerenciar a execução das tarefas escalonadas.
  - Zookeeper: responsável pelo gerencialmento das barreiras de sincronização entre os superpassos das tarefas e também pelo sistema de tolerância a falhas.

[4] Apache Hama. Disponível em: https://hama.apache.org/.

  - BSPMaster: nó mestre responsável por:
    * Monitorar o *cluster*;
    * Monitorar o estado do GroomServer;
    * Monitorar superpassos e outros contadores no *cluster*;
    * Monitorar *jobs* e tarefas;
    * Escalonar *jobs* a serem executados;
    * Fornecer aos usuários interfaces de controle do *cluster* (tanto em web como no terminal).

- Execução: Hama disponibiliza uma API para a implementação de algoritmos de grafos. Basta implementar uma classe java que extenda a classe Vertex e implemente o método *compute*, o qual é executado para todo nó ativo a cada superpasso do modelo BSP. Em um superpasso o método *compute* pode passar informações sobre um nó para os nós vizinhos ou um nó específico.

- Caso de Estudo: em (TING; LIN; WANG, 2011) é realizado uma comparação entre os arcabouços Hadoop e Hama na execução de um algoritmo de Web *crawler*: processo utilizado por ferramentas de busca para mapear páginas da Web (CASTILLO, 2005). É utilizado a implementação do MapReduce do Hadoop, e a implementação do BSP do Hama.

  Foram realizados testes com o programa *crawler* coletando e armazenando dados de 100, 1.000, 2.000 e 10.000 diferentes *Uniform Resource Locators* (URL). A Figura 7 mostra os resultados obtidos:

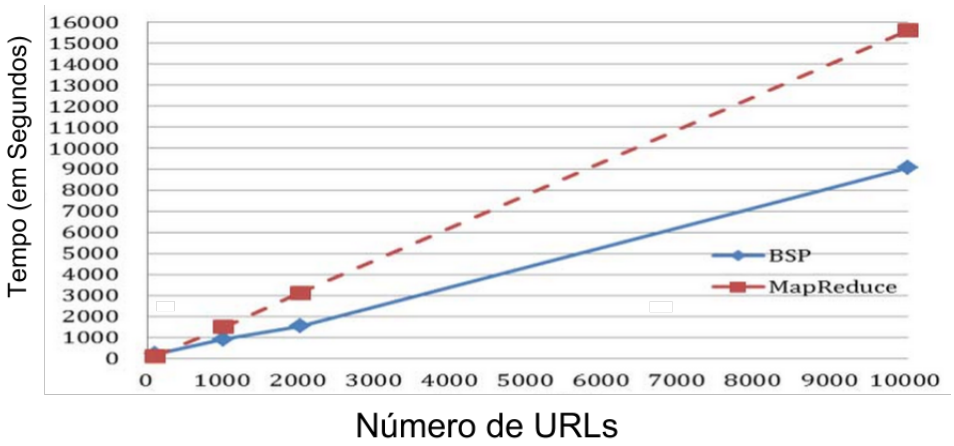

Figura 7: Resultados de Execução

Fonte: (TING; LIN; WANG, 2011)

Para a comparação foi utilizado um *cluster* com quatro máquinas com os seguintes processadores:

- Intel(TM) Core(TM) 2 Quad *Central Processing Unit* (CPU) Q8400 @2,66 *Gigahertz* (GHz)
- Intel(TM) Core(TM) 2 Duo CPU E4500 @2,20GHz
- Intel(TM) Core(TM) 2 Duo CPU E4500 @2,20GHz
- Intel(TM) Core(TM) 2 Duo CPU E8400 @3,00GHz

2.6.3 GIRAPH

Giraph[5] é um sistema de processamento iterativo de grafos construído para grande escalabilidade (APACHE..., 2014). É a versão de código aberto do arcabouço Pregel, da Google (MALEWICZ et al., 2010), e é inspirado pelo paradigma BSP.

- Arquitetura: construído em cima do Hadoop, o arcabouço Giraph segue a mesma arquitetura de Mestre/Escravo. O Mestre é responsável por monitorar os estados dos nós escravos, particionar os grafo anteriormente a cada superpasso e sincronizar o processamento ao final de cada superpasso. Os Escravos são responsáveis por executar o método *compute* sobre cada vértice designado e enviar mensagens, se necessário, a outros vértices (PEREIRA, 2013).

  É também utilizado o arcabouço Apache ZooKeeper[6] para coordenar o estado de toda a aplicação e garantir tolerância a falhas (PEREIRA, 2013).

  Os vértices de um grafo são os membros sobre os quais são realizadas as operações que afetam o grafo. As operações são realizadas em superpassos de acordo com o modelo do BSP. A cada superpasso os vértices realizam suas operações e comunicam outros vértices, se necessário, sobre seu estado atual pós-operações. Depois que todas as mensagens entre vértices forem sincronizadas um novo superpasso pode começar a ser executado (KAMBURUGAMUVE et al., 2013).

- Execução: Giraph disponibiliza uma API muito similar a do Hama. O usuário precisa implementar uma classe java que extenda a classe BasicComputation e implemente o método *compute*, o qual também é executado para todo nó ativo a cada superpasso do modelo BSP, a exemplo do Hama.

[5] Apache Giraph. Disponível em: https://giraph.apache.org.
[6] Apache ZooKeeper. Disponível em: https://zookeeper.apache.org/.

2.6.4 SPARK

Spark[7] é um arcabouço de processamento paralelo de propósito geral implementado em Scala, uma linguagem de programação funcional de alto nível estaticamente tipada que compila para Java *bytecode*[8].

- Arquitetura: O arcabouço Spark pode ser executado em modo independente, sem depender de nenhum outro arcabouço, ou em cima de outros arcabouços como Hadoop ou Mesos, e pode utilizar como entrada para processamento vários bancos de dados distribuídos, entre eles o HDFS, HBase e Cassandra (APACHE. . . , 2015).

  Em sua execução, Spark utiliza uma estrutura de dados abstraída chamada *Resilient Distributed Dataset* (RDD), a qual representa uma coleção apenas para leitura de objetos divididos através de um conjunto de máquinas, mas que podem ser reconstruídos caso alguma partição em alguma das máquinas seja perdida (ZAHARIA et al., 2010).

- Execução: Para utilizar o Spark basta implementar os algoritmos desejados em Java, Python ou Scala, utilizando a API funcional disponibilizada.

- Caso de Estudo: em 2014 Spark venceu a Gray Sort[9], competição que decide qual sistema dentre os concorrentes consegue ordenar 100 terabytes de informações (1 trilhão de registros) mais rapidamente. Foram utilizadas 206 máquinas *Amazon Elastic Compute Clound* (EC2), com as quais os 100 terabytes de informação foram ordenados em 23 minutos (SPARK. . . , 2014).

2.7 OUTROS ARCABOUÇOS

Na proposta do presente projeto de pesquisa foram apresentados dois outros arcabouços que seriam utilizados na realização da pesquisa: GBase (KANG et al., 2011) e Pegasus (KANG; TSOURAKAKIS; FALOUTSOS, 2009). Na etapa de pesquisa bibliográfica foi constatado que nenhum dos dois poderia ser utilizado.

GBase não foi utilizado por possuir apenas um artigo central propondo um novo arcabouço de processamento paralelo. Não foram encontradas outras referências na web, nenhum site pertecente ao GBase e muito menos um arcabouço instalável e executável, ponto crucial à presente pesquisa.

[7] Apache Spark. Disponível em: https://spark.apache.org/.

[8] Scala (programming language): https://en.wikipedia.org/wiki/Scala_(programming_language). Acesso em 23 de mar. de 2015

[9] Sort Benchmark Home Page: http://sortbenchmark.org/.

E Pegasus não foi utilizado porque um maior estudo revelou que este se trata de uma biblioteca de algoritmos de grafos implementada para ser executada no arcabouço Hadoop, e não um arcabouço completo como se acreditava quando da realização da proposta do projeto.

2.8 PAGERANK

PageRank é um método (algoritmo) para ranquear páginas web de maneira objetiva e mecânica, efetivamente medindo o interesse humano e atenção dados a elas (PAGE et al., 1999). O algoritmo foi desenvolvido por Sergey Brin e Larry Page em 1996 na Universidade de Stanford (PAGE. . . , 2014b).

Para o PageRank, uma página é popular se as páginas que apontam para ela também forem populares. Dessa forma, são cobertos tanto os casos em que uma página possui muitas páginas apontando para ela, ou apenas alguns poucas, porém importantes, páginas (PAGE et al., 1999). Diz-se que "uma página aponta para outra" quando aquela possui um *link* com o endereço web desta em sua constituição. Na Figura 8 está exemplificado o conceito aplicado pelo algoritmo do PageRank.

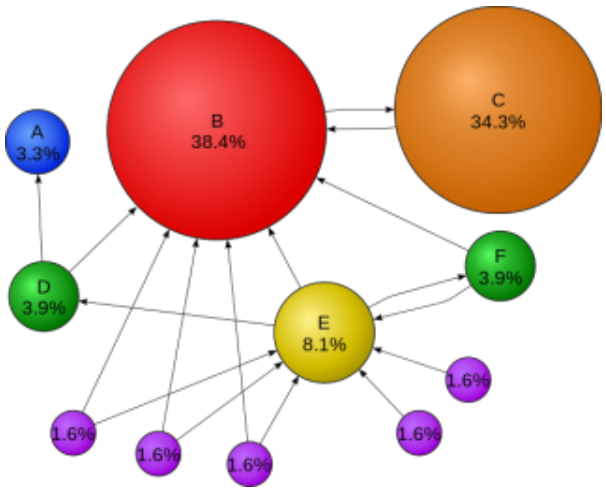

Figura 8: Exemplo do PageRank

Fonte: (PAGE. . . , 2014a)

O PageRank pode, também, ser utilizado em outras áreas da computação além da Web. Em (JING; BALUJA, 2008) o PageRank é utilizado para realizar pesquisas de imagens em larga escala; e em (MA; GUAN; ZHAO, 2008), ele é utilizado para mensurar a importância de artigos cietíficos.

2.9 O ALGORITMO DO PAGERANK

O algoritmo do PageRank considera em seu cálculo a probabilidade de se alcançar determinada página ao percorrer o grafo da web de maneira aleatória através dos links. É criada uma matriz de probabilidade M, a partir do grafo, de tamanho $|V| * |V|$, onde V são os vértices. Cada posição (i, j) da matriz possui um valor representando a probabilidade de se chegar à página j a partir da página i. Na Figura 9 tem-se um exemplo de um grafo da web com apenas 5 páginas.

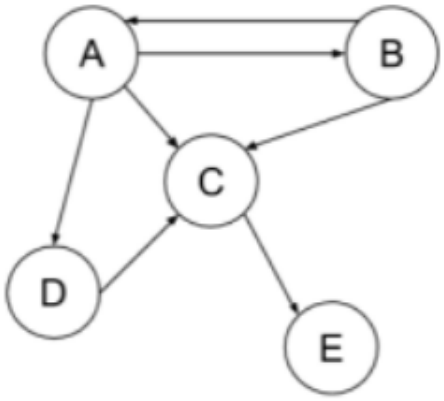

Figura 9: Exemplo de Grafo da web

E na Figura 10 tem-se a matriz M de probabilidade formada a partir desse grafo.

$$M = \begin{pmatrix} 0 & 1/3 & 1/3 & 1/3 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figura 10: Matriz de Probabilidade

Considerando u como uma página da web, F_u o conjunto de páginas para as quais u aponta, B_u o conjunto de páginas que apontam para u, N_u o número de links de u e c um fator para normalização, então uma definição formal e simplificada do algoritmo PageRank é dada pela equação 1 (PAGE et al., 1999):

$$R(u) = c \sum_{u \in B_u} \frac{R(u)}{N_u} \quad (1)$$

Utilizando o modelo da Cadeia de Markov (MARKOV..., 2014) é calculada a probabilidade de se alcançar determinada página no tempo T_i. Esse cálculo é realizado para todas as páginas repetidamente iterando-se sobre a matriz de probabilidades. A iteração sobre a matriz é interrompida quando percebe-se que as n-últimas iterações não a alteraram de maneira significativa.

O fator de normalização c representa a probabilidade de um usuário qualquer continuar seguindo os links das páginas web, enquanto $(1-c)$ representa a probabilidade do mesmo usuário simplesmente escolher uma nova página web aleatória (BRIN; PAGE, 1998).

c é incluído para prevenir que páginas web sem nenhum link de saída roubem o *pagerank* das outras páginas, visto que um usuário que apenas siga os links das páginas web (sem nunca se desviar para uma página aleatória) vai acabar em uma página sem nenhum link de saída, dado tempo suficiente (WHAT..., 2015). O valor mais comum utilizado para o fator de normalização c ainda é o mesmo sugerido inicialmente por Sergey Brin e Lawrence Page em (BRIN; PAGE, 1998) de $0,85$ (BOLDI; SANTINI; VIGNA, 2005).

Inicialmente o valor *pagerank* de cada nó do grafo é $1/|V|$. A cada iteração do PageRank, o valor do *pagerank* de cada nó é aprimorado com base nos nós vizinhos e seus respectivos *pageranks*. Quanto mais iterações forem realizadas, mais precisos se tornarão os *pageranks* dos nós do grafo.

2.10 CAMINHO MAIS CURTO

O Caminho mais Curto é o problema de encontrar o caminho a ser percorrido entre dois vértices de um grafo, de forma que a soma dos pesos das arestas pertencentes a esse caminho seja minimizada (CORMEN et al., 2009).

Dado o Grafo Ponderado da Figura 11, o caminho mais curto entre os vértices D e B é o caminho que passa pelo vértice E, com valor total de 5 ($D->E->B$). Embora exista outro caminho, passando pelos vértices E e A ($D->E->A->B$), tal caminho possui um valor total de 12, e por isso não é o caminho mais curto entre D e B.

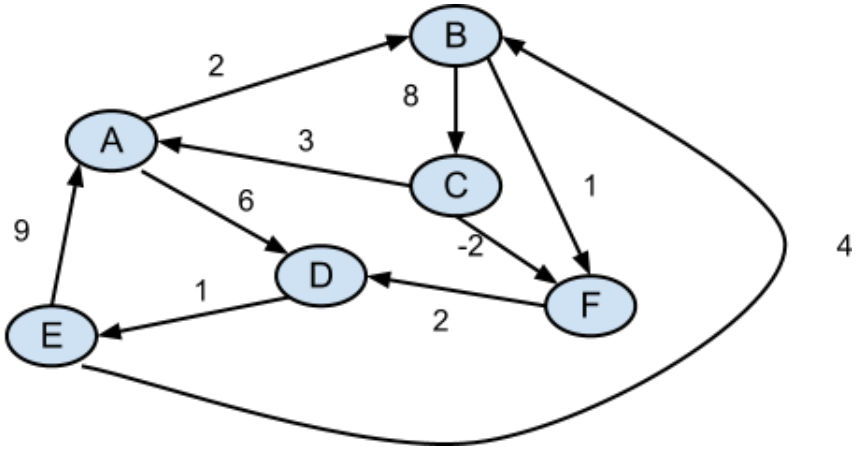

Figura 11: Grafo Ponderado

Fonte: (TWO. . . , 2015)

2.11 ALGORITMO DO CAMINHO MAIS CURTO

O algoritmo utilizado é uma adaptação do algoritmo de Dijkstra[10] para ser executado em arcabouços de processamento paralelo e busca resolver o problema do Caminho mais Curto em um Grafo Ponderado com pesos positivos a partir de um único vértice, o qual é considerado o vértice origem. Ao final da execução do algoritmo são coletados os valores dos caminhos mais curtos entre o vértice origem e todos os vértices do grafo.

Cada vértice possui um valor, definido inicialmente como $+\infty$ (positivo infinito). O vértice origem tem seu valor definido como 0 (porque o valor do caminho mais curto do vértice origem até ele mesmo é sempre 0 em um Grafo Ponderado com pesos positivos). As arestas do grafo são percorridas repetidamente. Para cada aresta conectando dois vértices *A* e *B*, é enviada uma mensagem ao vértice *B* com o valor do vértice *A* mais o peso da aresta. O vértice *B* aceita esse novo valor como seu próprio se ele for menor que o seu valor atual, do contrário a mensagem é ignorada. O processo é repetido enquanto houverem modificações nos valores dos vértices do grafo.

2.12 TRABALHOS RELACIONADOS

Em (GUO et al., 2014), os autores realizaram uma análise bastante significativa entre 6 arcabouços de processamento paralelo para execução de algoritmos de grafos, sendo eles: Hadoop, Hadoop versão 2x com YARN, Stratosphere, Giraph, GraphLab, e Neo4j. As

[10]DIJKSTRA, Edsger W. *A note on two problems in connection with graphs*. **Numerische mathematik**, v. 1, n. 1, p. 269-271, 1959.

comparações foram realizadas em ambientes computacionais controlados com o objetivo de não afetar qualquer resultado. As métricas para efeitos de comparação utilizadas foram: tempo de execução; utilização de CPU, memória e tráfego de rede; escalabilidade; e *overhead*.

Em (ELSER; MONTRESOR, 2013), os autores também realizaram uma execução comparativa entre arcabouços de processamento paralelo de grafos. Os arcabouços escolhidos foram: Hadoop, Stratosphere, Giraph, Hama, e GraphLab. Para realizar a comparação, foi implementado o algoritmo K-Core em todos os arcabouços. O algoritmo K-Core tem como objetivo identificar sub-grafos dentro de um grafo cujo grau de incidência em cada vértice (arestas conectadas ao vértice) seja ao menos *k* (DEGENERACY. . . , 2014).

Em (LIN; SCHATZ, 2010), os autores utilizaram o arcabouço Hadoop para processar o algoritmo do PageRank sobre um grafo com 1,4 bilhão de arestas e 1,53 Terabytes de dados não comprimidos. Na implementação mais básica o tempo de execução foi de 1.500 segundos, e na implementação mais otimizada o tempo foi de menos de 500 segundos. As execuções foram realizadas em um *cluster* da Universidade de Maryland com 10 nós, cada um com 2 processadores Intel Xeon 3,2 GHz, 4 *GigaByte* (GB) de *Random-Access Memory* (RAM), e 367 GB de disco local utilizado pelo HDFS. Cada nó estava executando o Hadoop versão 0.20.0 em um servidor Linux distribuição Red Hat Enterprise versão 5.3 e estavam conectados a um *switch* comum por meio de cabos Ethernet *gigabit*. Foram utilizados diversos padrões aperfeiçoados na implementação mais otimizada e cada um deles é explicado em profundidade.

Em (KANG; TSOURAKAKIS; FALOUTSOS, 2009), os autores utilizaram o *cluster* Hadoop M45 da Yahoo!, um supercomputador que figura entre os 50 mais rápidos supercomputadores do mundo, com 1,5 Petabytes de capacidade total de armazenamento e 3,5 Terabytes de memória. Foi executado o algoritmo da PageRank em um grafo com 1.977 milhões de arestas. O tempo de execução foi de 500 segundos. Os autores documentaram em profundidade os testes executados, com gráficos dos tempos de execução e códigos dos algoritmos utilizados.

Em (HAN; JIN; WANG, 2013), os autores utilizaram um *cluster* EC2 com 4 e 8 máquinas, cada uma com um processador Intel Xeon 1,7 GHz com dois núcleos, e 3,75 GBs de memória RAM. Foram comparados os arcabouços Giraph, GPS, e Mizan, executando os algoritmos do PageRank e do Caminho mais Curto em cada um, três bases de dados diferentes e coletando o tempo de execução.

A Tabela 1 resume os pontos principais da presente pesquisa e dos trabalhos relacionados, sendo eles:

- **Mais de um Arcabouço**: positivo em caso de a pesquisa em questão ter utilizado mais de um arcabouço para realizar o estudo comparativo;
- **Mais de um Algoritmo**: se a pesquisa utilizou mais de um algoritmo;
- **Tempo de Execução**: se a pesquisa utilizou o tempo de execução do(s) arcabouço(s) como métrica de comparação;
- **Utilização de Memória**: se a pesquisa utilizou a utilização de memória como métrica de comparação.

Tabela 1: Trabalhos Relacionados

	Mais de um Arcabouço	Mais de um Algoritmo	Tempo de Execução	Utilização de Memória
Estudo Comparativo de Arcabouços para Processamento Paralelo de Grafos	S	S	S	S
Constructing a Cloud Computing Based Social Networks Data Warehousing and Analysing System (TING; LIN; WANG, 2011)	S	N	S	N
How Well do Graph-Processing Plataforms Perform? An Empirical Performance Evaluation and Analysis (GUO et al., 2014)	S	S	S	S
An Evaluation Study of BigData Frameworks for Graph Processing (ELSER; MONTRESOR, 2013)	S	N	S	N
Design Patterns for Efficient Graph Algorithms in MapReduce (LIN; SCHATZ, 2010)	N	N	S	N
Pegasus: A Peta-Scale Graph Mining System Implementation and Observations (KANG; TSOURAKAKIS; FALOUTSOS, 2009)	N	N	S	N
Comparing Pregel-like Graph Processing Systems (HAN; JIN; WANG, 2013)	S	S	S	N

Dada a tabela 1, é possível verificar que das sete pesquisas descritas, apenas (GUO et al., 2014) e este trabalho utilizaram os principais pontos detalhados. Com isso, a presente pesquisa acrescenta informações consideráveis ao Estado da Arte das pesquisas sobre arcabouços de processamento paralelo.

3 DESENVOLVIMENTO DA PESQUISA

Para o desenvolvimento deste trabalho de pesquisa foi inicialmente definido um processo de desenvolvimento a ser seguido durante a realização da pesquisa, o qual pode ser visualizado na Figura 12:

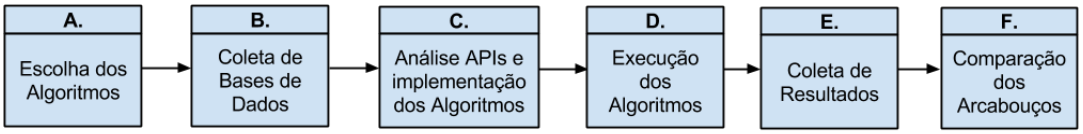

Figura 12: Processo de Desenvolvimento

3.1 BASES DE DADOS

Os dados utilizados para realizar as execuções dos arcabouços foram escolhidos de forma empírica. Foram utilizadas três bases de dados, duas retiradas da web[1] e uma gerada aleatoriamente: "web-Stanford.txt", "web-Google.txt" e "meu-grafo.txt", respectivamente. Para executar o algoritmo do Caminho mais Curto foram adicionados pesos aleatórios entre $[0, 100]$ para cada aresta das bases de dados. A Tabela 2 exibe alguns dos dados centrais das bases de dados.

Percebe-se que o desvio padrão da base de dados "web-Stanford.txt" é extremamente alto. Isso pode ser explicado pelo fato de alguns vértices serem referenciados apenas por si próprios, enquanto outros são referenciados por até 38.607 outros vértices, como é pode ser visto no valor do maior grau de entrada. O mesmo ocorre com a base "web-Google.txt", embora em menor tamanho. O fato da base de dados "meu-grafo.txt" possuir a mesma média para graus de entrada e saída que "web-Stanford.txt" é apenas coincidência, visto que o primeiro foi gerado aleatoriamente.

Embora as bases de dados "meu-grafo.txt" e "web-Stanford.txt" possuam quantidades similares de vértices e a mesma quantidade de arestas, aquela foi utilizada por possuir um menor

[1] Stanford Large Network Dataset Collection. Disponível em: http://snap.stanford.edu/data/index.html#web.

Tabela 2: Bases de Dados

		meu-grafo.txt	web-Stanford.txt	web-Google.txt
Quantidade de Vértices		282.000	281.903	875.713
Quantidade de Arestas		2.312.497	2.312.497	5.105.038
Tamanho (em *MegaBytes* (MB))		30,5	70,2	30,6
Grau de Entrada	**Maior**	25	38.607	6.327
	Menor	1	1	1
	Média	9,20	9,20	6,82
	Desvio Padrão	2,86	166,33	39,22
Grau de Saída	**Maior**	26	256	457
	Menor	1	1	1
	Média	9,20	9,20	6,82
	Desvio Padrão	2,86	11,31	6,59

desvio padrão quando comparada a esta, tanto para o grau de entrada quanto para o grau de saída, o que tornou as execuções menos tendensiosas para arcabouços que sejam possivelmente preparados para lidar com bases de dados com desvio padrão alto ou baixo.

- web-Stanford.txt: um grafo com vértices representando páginas da Stanford University. O grafo possui 281.903 vértices e 2.312.497 arestas direcionadas.
- meu-grafo.txt: um grafo próprio gerado aleatoriamente com 282.000 vértices e 2.312.497 arestas direcionadas.
- web-Google.txt: um grafo representando páginas web, com arestas direcionadas representando *hyperlinks* entre as páginas, liberado pela Google em 2002. O grafo possui 875.713 vértices e 5.105.039 arestas.

Para cada base de dados foi realizada uma etapa de pré-processamento. Nesta etapa, para cada vértice das bases de dados foi adicionada uma aresta tendo como origem e destino o próprio vértice. Isto foi feito porque o arcabouço Hama exige que todo vértice referenciado como destino de uma aresta esteja presente no arquivo de entrada também como origem de uma ou mais arestas, ou seja, deve haver uma lista de adjacência relacionada a todo vértice. Para a execução do algoritmo do Caminho mais Curto, que exige arestas com peso conectando dois vértices, foram utilizados pesos de valor 0 para conectar os vértices a eles mesmos, visto que tal não influencia o resultado final do algoritmo (o menor caminho de um vértice a ele mesmo é sempre 0). Em seguida, as bases de dados foram convertidas para listas de adjacência e representadas em três diferentes formatos: A, B e C.

- O formato A foi utilizado para executar o PageRank nos arcabouços Hadoop, Hama e Spark. O primeiro número de cada linha é o número de identificação de um vértice. Em

seguida, são os números de identificação dos vértices para os quais o primeiro vértice da linha aponta, separados por um espaço. Dada a Figura 13 de exemplo, a primeira linha representa o vértice de id 0, o qual possui arestas apontando para os vértices 0, 1 e 2.

```
0   0 1 2
1   1 3
2   2
3   3
4   2 3 4
5   2 5
6   8 9 6 7
7   7
8   8
9   0 9
```

Figura 13: Formato A

- O formato B foi utilizado para executar o Caminho mais Curto nos arcabouços Hadoop, Hama e Spark, e pode ser visualizado na Figura 14. A exemplo do formato A, o primeiro número de cada linha é o número de identificação de um vértice e em seguida são os números de identificação dos vértices para os quais o primeiro aponta. Ao lado de cada número de identificação encontra-se o caracter : (dois pontos) e um número, o qual representa o peso da aresta conectando os dois vértices. Dada a Figura 14 de exemplo, a primeira linha representa o vértice de id 0, conectado ao vértice 0 por uma aresta de peso 0, conectado ao vértice 1 por uma aresta de peso 68, e conectado ao vértice 2 por uma aresta de peso 54.

- E o formato C foi utilizado para executar tanto o PageRank quanto o Caminho mais Curto no arcabouço Giraph. O primeiro número de cada linha é o número de identificação de um vértice, seguido pelo valor do vértice, seguido por uma lista de vértices e o peso da aresta conectando o primeiro vértice a esses vértices. Dada a Figura 15 de exemplo, a primeira linha exibe o vértice de id 0, com valor 0, conectado aos vértices 0, 1 e 2, com arestas de peso 0, 31 e 5, respectivamente.

  Na execução do PageRank no arcabouço Giraph, o peso de cada aresta contido no arquivo foi desconsiderado.

```
0   0:0 1:68 2:54
1   1:0 3:87
2   2:0
3   3:0
4   2:48 3:30 4:0
5   2:20 5:0
6   8:25 9:73 6:0 7:76
7   7:0
8   8:0
9   0:8 9:0
```

Figura 14: Formato B

```
[0,0,[[0,0],[1,31],[2,5]]]
[1,0,[[1,0],[3,74]]]
[2,0,[[2,0]]]
[3,0,[[3,0]]]
[4,0,[[2,74],[3,84],[4,0]]]
[5,0,[[2,82],[5,0]]]
[6,0,[[8,37],[9,84],[6,0],[7,86]]]
[7,0,[[7,0]]]
[8,0,[[8,0]]]
[9,0,[[0,19],[9,0]]]
```

Figura 15: Formato C

3.2 ALGORITMOS UTILIZADOS

Para realizar esta pesquisa foram utilizados dois algoritmos de grafos como forma de comparação entre os diferentes arcabouços: o algoritmo do PageRank; e o algoritmo do Caminho mais Curto.

O algoritmo do PageRank foi utilizado como o principal algoritmo a ser executado em todos os arcabouços por ser o único algoritmo para grafos que possui uma implementação oficial em Java em três dos quatro arcabouços estudados (Giraph, Hama e Spark). E o algoritmo

do Caminho mais Curto foi utilizado com o intuito de coletar maiores informações sobre os arcabouços analisados e não tendenciar as comparações utilizando apenas um algoritmo.

3.2.1 PAGERANK

Para cada arcabouço foi implementada uma versão do algoritmo do PageRank na linguagem de programação Java seguindo a API do arcabouço, onde todas as implementações utilizaram a lógica de iterações definida em 2.9 e um fator de normalização c de $0,85$.

Por mais que os algoritmos utilizados em todos os arcabouços sigam a mesma lógica, devido às particularidades de cado arcabouço, é esperado que os resultados finais do algoritmo (os valores dos *pageranks* de cada vértice) sejam um tanto quanto dissimilares. Como exemplo, o menor *pagerank* de um vértice da base de dados "meu-grafo.txt", quando executada com 10 iterações, pertence ao vértice de número 77.667, cujo valor é $5,56e-07$, ao passo que o maior *pagerank* pertence ao vértice 217.402, com um valor de $3,43e-05$. É também neste último vértice que pode ser visualizada a maior discrepância entre os resultados dos arcabouços: nos arcabouços Hadoop, Hama e Spark este vértice recebeu o valor final de $3,43e-5$, enquanto no arcabouço Giraph o valor do vértice foi de $3,29e-5$, uma diferença de $1,37e-06$.

Abaixo são descritas as versões do algoritmo utilizadas:

- Giraph: foi utilizado o código-fonte oficial do Giraph[2], a partir do qual foi implementado o algoritmo utilizado pelo presente trabalho. O principal método de uma aplicação Giraph, o método *compute*, é exibido no Código 3.1.

  A primeira expressão condicional testa se o superpasso sendo executado é o de número 0, ou seja, o primeiro superpasso a ser executado. Caso seja, o valor do *pagerank* do vértice é definido como $1/|V|$, como explicado no Capítulo 2.9. E caso não seja o primeiro superpasso sendo executado, é calculado o *pagerank* atual do vértice baseado nos *pageranks* dos vértices que apontam para ele, os quais foram passados no superpasso anterior, seguindo o model BSP. O valor do *pagerank* atual do vértice é então enviado para todos os vértices para os quais ele aponta, os quais serão recebidos no próximo superpasso.

  Código 3.1: Método compute() - Giraph

```
1 @Override
```

[2]PageRank em Java oficial do Giraph. Disponível em: https://github.com/apache/giraph.

```
2         public void compute(Vertex<LongWritable, DoubleWritable,
              FloatWritable> vertex, Iterable<DoubleWritable> messages)
              throws
3 IOException     {
4                 if (getSuperstep() == 0) {
5                         vertex.setValue(new DoubleWritable(1.0d /
                              getTotalNumVertices()));
6                 }
7                 else {
8                         double pageRankSum = 0;
9                         for (DoubleWritable message : messages) {
10                                pageRankSum += message.get();
11                        }
12                        double alpha = (1.0 - 0.85) / this.
                              getTotalNumVertices();
13                        vertex.setValue(new DoubleWritable(
14                                alpha + (pageRankSum * 0.85)
15                        ));
16                }
17                long edges = vertex.getNumEdges();
18                this.sendMessageToAllEdges(vertex, new DoubleWritable(
                      vertex.getValue().get() / edges));
19        }
```

- Hama: a estrutura do código-fonte[3] foi mantida, principalmente o código de leitura do arquivo de entrada, embora a lógica tenha sido modificada para ler o formato específico utilizado. O principal método de uma aplicação Hama, a exemplo do Giraph, o método *compute*, pode ser visualizado no Código 3.2.

  A lógica aplicada aqui é a mesma do código utilizado no método *compute* utilizado no arcabouço Giraph, o qual poder ser visualizado no Código 3.1.

Código 3.2: Método compute() - Hama

```
1         @Override
2         public void compute(Iterable<DoubleWritable> messages) throws
              IOException {
3                 if (this.getSuperstepCount() == 0) {
4                         this.setValue(new DoubleWritable(
5                                 1.0 / this.getNumVertices()
6                         ));
```

[3]PageRank em Java oficial do Hama. Disponível em: https://github.com/apache/hama.

```
7                   }
8                   else {
9                           double pageRankSum = 0;
10                          for (DoubleWritable message : messages) {
11                                  pageRankSum += message.get();
12                          }
13                          double alpha = (1.0 - 0.85) / this.
                                getNumVertices();
14                          setValue(new DoubleWritable(alpha + (
                                pageRankSum * 0.85)));
15                  }
16                  long edges = this.getEdges().size();
17                  this.sendMessageToNeighbors(
18                          new DoubleWritable(this.getValue().get() /
                                edges)
19                  );
20          }
```

- Spark: o código-fonte[4] foi mantido praticamente sem modificações, exceto pelo particionamento do arquivo de entrada que foi modificado para atender ao formato utilizado.

- Hadoop: este foi o único código implementado sem ser baseado em algum código-fonte, visto que o arcabouço Hadoop não possui implementação oficial do algoritmo do PageRank. Entretanto, foram utilizados alguns códigos-fonte como auxílio[5]. Os métodos *map()* e *reduce()*, os principais métodos de uma aplicação Hadoop, podem ser visualizados nos Códigos 3.3 e 3.4, respectivamente.

  O método *map* envia o valor do *pagerank* de um vértice a todos os vértices para os quais ele aponta.

Código 3.3: Método map() - Hadoop

```
1           @Override
2           public void map (Text key, Text value, Context context) throws
                IOException, InterruptedException {
3                   String[] aux = value.toString().split(" ");
4                   int length = aux.length;
5                   double currentPageRank = Double.parseDouble(aux[0]);
6                   StringBuilder outerEdges = new StringBuilder();
```

[4]PageRank em Java oficial do Spark. Disponível em: http://github.com/apache/spark.
[5]PageRank em Java de terceiros. Disponível em: https://github.com/goranjovic/hadoop-twitter-pagerank.

```
7                 for (int i = 1; i < length; i++) {
8                         context.write(
9                                 new LongWritable(Long.parseLong(aux[i
                                      ])),
10                                new Text(String.valueOf(
                                      currentPageRank / (double)(length -
                                       1)))
11                        );
12                        outerEdges.append(aux[i]).append(' ');
13                }
14                context.write(
15                        new LongWritable(Long.parseLong(key.toString()
                              )),
16                        new Text(outerEdges.toString().trim())
17                );
18        }
```

O método *reduce* recebe os *pageranks* de todos os vértices que apontam para o vértice em questão, o qual, então, calcula seu próprio *pagerank* utilizando os valores recebidos e o escreve no HDFS para a próxima iteração do PageRank.

Código 3.4: Método reduce() - Hadoop

```
1         @Override
2         public void reduce(LongWritable key, Iterable<Text> values,
              Context context) throws IOException, InterruptedException {
3                 double sum = 0.0;
4                 String[] outerEdges = new String[0];
5                 for (Text value : values) {
6                         String v = value.toString().trim();
7                         String[] temp = v.split(" ");
8                         if (temp.length == 0 || temp.length > 1) {
9                                 outerEdges = temp;
10                        }
11                        else if (v.contains(".")) {
12                                sum += Double.parseDouble(v);
13                        }
14                        else {
15                                outerEdges = temp;
16                        }
17                }
18                int amountOfVertices = Integer.parseInt(context.
```

```
                getConfiguration().get("AMOUNT_OF_VERTICES"));
19              double vertexValue = (0.15 / amountOfVertices) + (0.85
                    * sum);
20              StringBuilder resultingValue = new StringBuilder();
21              resultingValue.append(vertexValue);
22              for (String outerEdge : outerEdges) {
23                      resultingValue.append(' ').append(outerEdge);
24              }
25              context.write(key, new Text(resultingValue.toString().
                    trim()));
26      }
```

3.2.2 CAMINHO MAIS CURTO

Os algoritmos executados nos arcabouços Hama e Giraph foram retirados diretamente de seus respectivos repositórios no GitHub[6], foram modificados apenas para ler os arquivos de entrada nos formatos desejados e foram testados para confirmar que agem de acordo com a lógica do Algoritmo de Dijkstra especificada em 2.11. Os algoritmos executados nos arcabouços Hadoop e Spark foram implementados em sua integridade seguindo a mesma lógica.

3.3 AMBIENTE DE EXECUÇÃO

Neste projeto foram utilizados dois ambientes de execução, Máquina Autônoma e *Cluster*, descritos a seguir:

3.3.1 MÁQUINA AUTÔNOMA

Foi utilizado um único computador com 7,7 GBs de memória RAM, processador Intel(TM) Core(TM) i5-2400M de segunda geração com 2 núcleos reais e 4 núcleos simulados de 2,1 GHz de capacidade cada, executando o sistema operacional GNU Linux distribuição Ubuntu 13.10 64-bit.

Foram instalados os arcabouços Hadoop versão 1.2.1, Hama versão 0.6.4, Giraph versão 1.1.0 e Spark versão 1.0.0. O arcabouço Hadoop foi instalado em modo Pseudo-Distribuído, onde cada processo é executado como um processo Java separado para simlular um ambiente distribuído em apenas uma máquina física.

[6]GitHub: https://github.com/.

3.3.2 CLUSTER

Foram utilizadas três Máquinas Virtuais, onde uma delas agiu como a máquina mestre do *cluster* e ao mesmo tempo também como uma máquina escravo, e as outras duas máquinas apenas como máquinas escravo. Cada uma das máquinas executa o sistema operacional GNU Linux distribuição Ubuntu 14.04 64-bit, possui 4 GBs de memória RAM e processador Intel(TM) Xeon(TM) E7-4830 com 1 núcleo real de 2,13 GHz de capacidade de processamento.

Foram instalados os arcabouços Hadoop versão 1.2.1, Hama versão 0.6.4, Giraph versão 1.1.0 e Spark versão 1.0.1 em cada uma das máquinas.

3.4 PROTOCOLO DE EXECUÇÃO

Foi definido um protocolo de execução para especificar os passos a serem seguidos para executar, recolher e analisar os dados desejados.

3.4.1 EXECUÇÃO

Para não prejudicar a execução dos arcabouços e a coleta dos resultados, apenas programas inicializados em plano de fundo pelo próprio Sistema Operacional estavam ativos durante as execuções dos arcabouços.

Para os dois algoritmos utilizados, cada arcabouço foi executado 3 vezes utilizando cada uma das bases de dados descritas em 3.1 e em cada um dos ambientes descritos em 3.3. Os resultados finais foram gerados calculando a média aritmética das 3 execuções (para tempo e consumo de memória). Vale ressaltar que o algoritmo do PageRank foi executado de duas formas: utilizando 10 iterações, e depois utilizando 20 iterações.

3.4.2 COLETA DE RESULTADOS

No ambiente de execução Máquina Autônoma foram recolhidas duas informações sobre cada arcabouço durante as execuções: tempo total de execução e consumo de memória principal. E para o ambiente *Cluster* foi recolhido apenas o tempo de execução.

Para obter o tempo de execução de cada arcabouço em segundos foi utilizado o seguinte pedaço de código em Java 3.5:

Código 3.5: Obter Tempo de Execução (em segundos)

```
1         long inicio = System.nanoTime();
```

```
2
3           // Codigo do arcabouco vai aqui!
4
5           double tempoTotalDeExecucao = (System.nanoTime() - start) * 1.0e-9;
```

Para obter o consumo de memória principal de cada arcabouço foi utilizado o comando Linux *free*[7] em conjunto com o comando *cron*[8]. O comando *free* exibe informações sobre a memória principal, em uso e disponível, no momento em que for executado e o comando *cron* é um programa executado em plano de fundo no Linux para executar tarefas em prazos e intervalos definidos.

O comando *cron* foi utilizado para executar o comando *free* a cada 60 segundos durante as execuções dos arcabouços e a saída do comando *free* foi direcionada para um arquivo de texto posteriormente utilizado para fazer a análise dos dados.

[7]Comando free: http://linux.about.com/library/cmd/blcmdl1_free.htm.
[8]Comando cron: http://linux.about.com/od/commands/l/blcmdl8_cron.htm.

4 RESULTADOS E DISCUSSÕES

Neste capítulo serão delineados os resultados alcançados atualmente com a presente pesquisa e apresentadas as conclusões retiradas a partir desses resultados.

4.1 COMPARATIVO DE EXECUÇÃO

Seguindo o protocolo de execução especificado em 3.4 foram executados os algoritmos do PageRank e do Caminho mais Curto nos dois ambientes descritos em 3.3 e em cada um dos quatro arcabouços analisados.

4.1.1 EXECUÇÃO DO PAGERANK

O tempo de execução de cada arcabouço na Máquina Autônoma descrita em 3.3 pode ser visualizado na Tabela 3. Cada linha da tabela exibe os tempos de execução, em segundos, do PageRank em cada um dos arcabouços. Pode-se visualizar que o arcabouço Giraph demorou aproximadamente $44,67$ segundos para executar o PageRank na base de dados "web-Stanford.txt" com 10 iterações, o menor tempo de todas as execuções e menos da metade do tempo utilizado pelo Hama para concluir a mesma execução, o qual precisou de $106,61$ segundos; Spark precisou de $1.353,69$ segundos para executar a base de dados "web-Google.txt" com 20 iterações, a execução mais demorada.

O arcabouço Giraph teve o melhor desempenho em todas as bases de dados e com os dois valores para iterações máximas do PageRank. O pior desempenho para as bases de dados "meu-grafo.txt" e "web-Stanford.txt" foram do arcabouço Hadoop, e para a base de dados "web-Google.txt" foi do arcabouço Spark.

As Figuras 16, 18 e 20 exibem a utilização média de memória, em MB, de cada um dos arcabouços durante a execução do PageRank na Máquina Autônoma utilizando 10 iterações. E as Figuras 17, 19 e 21 exibem a utilização para a execução com 20 iterações.

As Figuras 16 e 17 exibem a utilização média de memória de cada um dos arcabouços

Tabela 3: Tempo em segundos da execução do PageRank na Máquina Autônoma

Arcabouços	Número de Iterações	web-Stanford.txt	meu-grafo.txt	web-Google.txt
Hadoop	10	326,352	330,608	570,978
	20	638,019	647,312	1.129,263
Spark	10	269,221	283,653	729,459
	20	479,429	480,187	1.353,693
Giraph	10	44,679	45,140	62,430
	20	49,879	53,330	86,028
Hama	10	106,612	111,627	257,617
	20	189,532	190,560	476,689

executando a base de dados “web-Stanford.txt”, com 10 e 20 iterações, respectivamente.

Percebe-se que em ambos os gráficos, Hama é o arcabouço que possui o maior pico de utilização de memória. Na Figura 16 Hama apresenta um pico de utilização de memória de, aproximadamente, 3 GBs após 1 minuto de execução. Também no primeiro minuto, o Hadoop, arcabouço que apresentou o segundo maior pico de utilização de memória, utilizou aproximadamente 2,5 GBs de memória, representando $83,3\%$ da utilização do Hama. Entretanto, Hadoop precisou de mais que o triplo do tempo do Hama para terminar a execução, como pode ser visualizada na Tabela 3.

Tanto na Figura 16 quanto na Figura 17 o arcabouço Spark manteve um pico de utilização de memória de aproximadamente 2,5 GBs durante toda sua execução, exceto no primeiro minuto e no último minuto, nos quais a utilização de memória ficou abaixo dos 2 GBs.

Giraph utilizou pouco mais que 1,5 GB em suas duas execuções, e em ambas precisou de menos de 1 minuto para terminá-las, como pode ser visualizado na Tabela 3.

As Figuras 18 e 19 exibem a utilização de memória de cada um dos arcabouços executando a base de dados “meu-grafo.txt”, com 10 e 20 iterações, respectivamente. A exemplo das Figuras 16 e 17, Hama alcançou o maior pico de utilização de memória, chegando à 3,5 GBs na Figura 19 no terceiro minuto de execução. Dessa vez, o segundo arcabouço com maior pico de utilização de memória foi o Spark, o qual utilizou 2,5 GBs no terceiro minuto de execução, representando menos de 72% da memória utilizada pelo Hama.

O arcabouço Giraph utilizou menos de 2 GBs em suas duas execuções, as quais, a exemplo das execuções para a base de dados “web-Stanford.txt”, demoraram menos de 1 minuto, como poder ser visualizado na Tabela 3.

As Figuras 20 e 21 exibem a utilização de memória de cada um dos arcabouços executando a base de dados “web-Google.txt”, com 10 e 20 iterações, respectivamente.

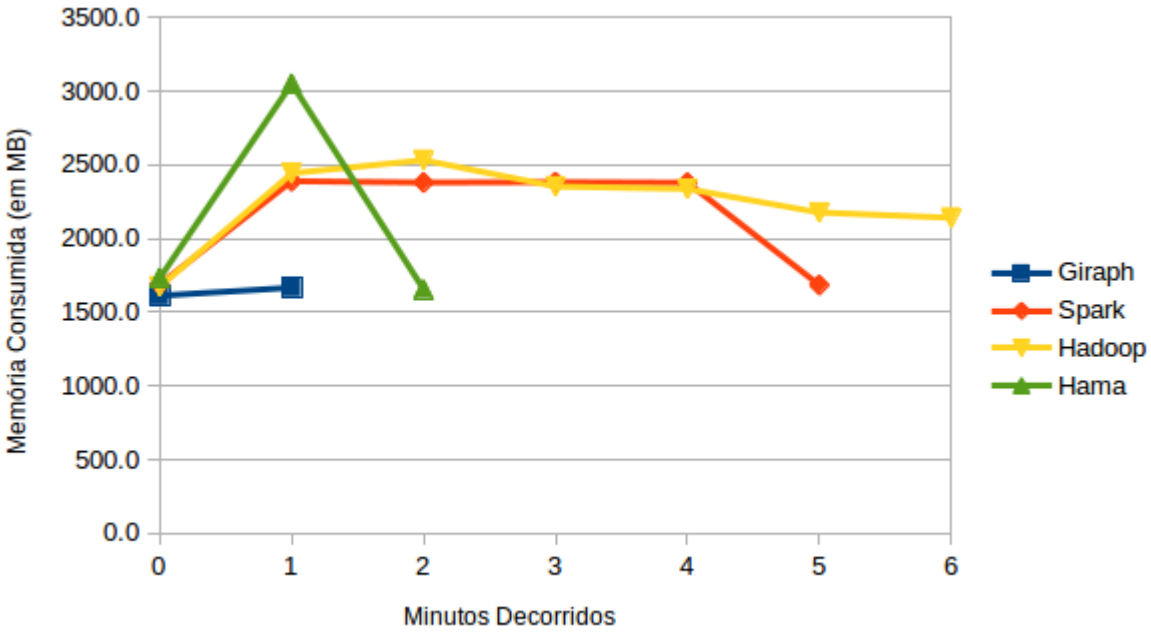

Figura 16: Utilização de Memória em "web-Stanford.txt" com 10 Iterações

Na Figura 20, Hama teve, novamente, o maior pico de utilização de memória, de pouco mais de 4 GBs no quarto minuto de execução. Nesse mesmo quarto minuto, Hadoop, arcabouço com o segundo maior pico de utilização de memória, utilizou pouco mais de 2 GBs em sua execução, embora tenha precisado de mais que o dobro do tempo do Hama para terminá-la, 570,98 e 257,61 segundos, respectivamente, como pode ser confirmado na Tabela 3. Spark manteve sua utilização de memória em 2,5 GBs, exceto no início e no último minuto, e demorou 729,45 segundos para terminar de executar, mas de 11 vezes o tempo utilizado pelo arcabouço Giraph, o qual demorou 62,43 segundos. Giraph manteve sua utilização de memória em 2 GBs ou menos durante seus menos de 2 minutos de execução.

Na Figura 21, Giraph teve o maior pico de utilização de memória, chegando à, aproximadamente, 4,7 GBs, 700 MBs a mais que o utilizado pelo Hama, arcabouço com o segundo maior pico. Giraph, por sua vez, precisou de apenas 86,02 segundos para concluir sua execução, enquanto Hama precisou de 476,68 segundos. Spark manteve-se linear em sua utilização de memória, utilizando 2,5 GBs durante toda sua execução, exceto no primeiro minuto, quando utilizou menos de 2 GBs, e nos últimos 3 minutos de execução, nos quais aumentou a utilização e teve um pico de, aproximadamente, 2,7 GBs.

A Tabela 4 apresenta o tempo de execução do algoritmo do PageRank de cada arcabouço no *Cluster* descrito em 3.3. A leitura desta tabela segue o mesmo padrão da Tabela 3.

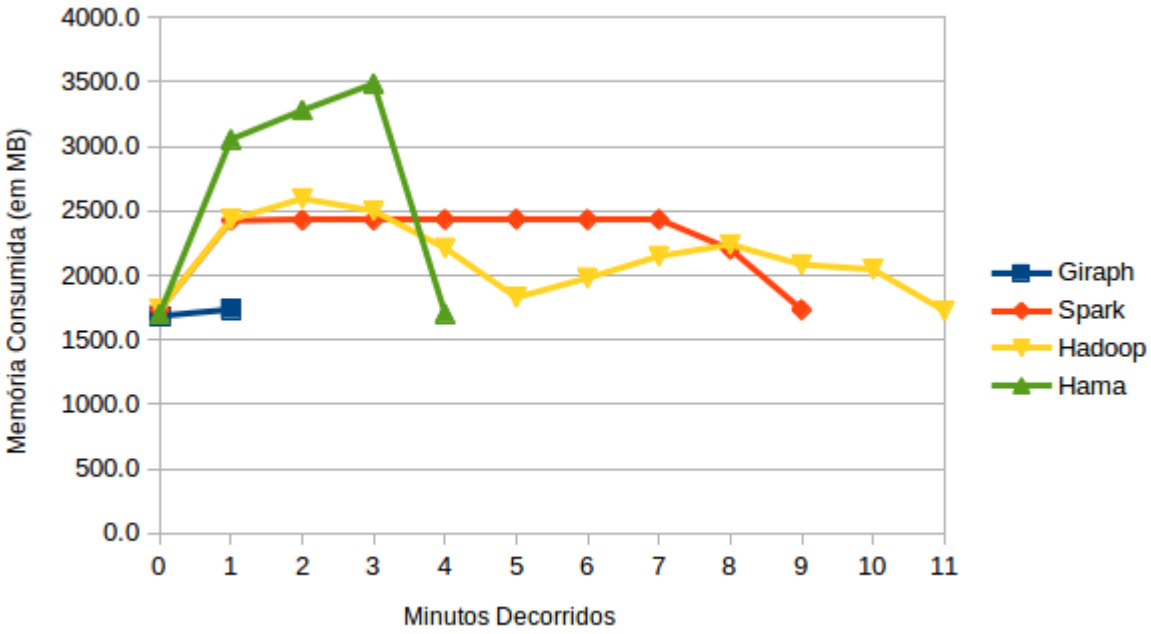

Figura 17: Utilização de Memória em "web-Stanford.txt" com 20 Iterações

Similar à execução na Máquina Autônoma, o arcabouço Giraph teve o melhor desempenho em todas as bases de dados e com os dois valores para iterações máximas. Ele precisou de 74,24 segundos para executar o PageRank com 10 iterações com a base de dados "web-Stanford.txt", o que representa aproximadamente 36,28% do tempo necessário para o arcabouço Hama concluir sua execução com as mesmas características, mesmo este sendo o segundo arcabouço mais veloz para essa execução. De maneira similar, Giraph demorou 197,9 segundos para concluir a execução com 20 iterações com a base de dados "web-Google.txt", enquanto o segundo arcabouço mais veloz nessa execução, Hama, precisou de 1.097,36 segundos, mais de 5 vezes o tempo do Giraph.

Ainda na Tabela 4, pode ser visto que o arcabouço Spark teve o pior desempenho na execução do PageRank com 20 iterações utilizando a base de dados "web-Google.txt", precisando de 2.496,86 segundos para concluí-la, 341,58 segundos a mais que o segundo arcabouço mais lento para essa execução, Hadoop, o qual demorou 2.155,28 segundos.

Percebe-se que as Tabelas 3 e 4 apresentam um comportamento bastante similar. Em ambas, o arcabouço Hadoop foi o mais lento, em tempo de execução, para as bases de dados "web-Stanford.txt" e "meu-grafo.txt", com os dois valores para iterações. De forma equivalente, o arcabouço Spark foi o mais lento para a base de dados "web-Google.txt" (com os dois valores para iterações) em ambas as tabelas. Entretato, a presente pesquisa não pode aferir o que causou este efeito ou se realmente há um padrão que seria mantido para comparativos utilizando outros

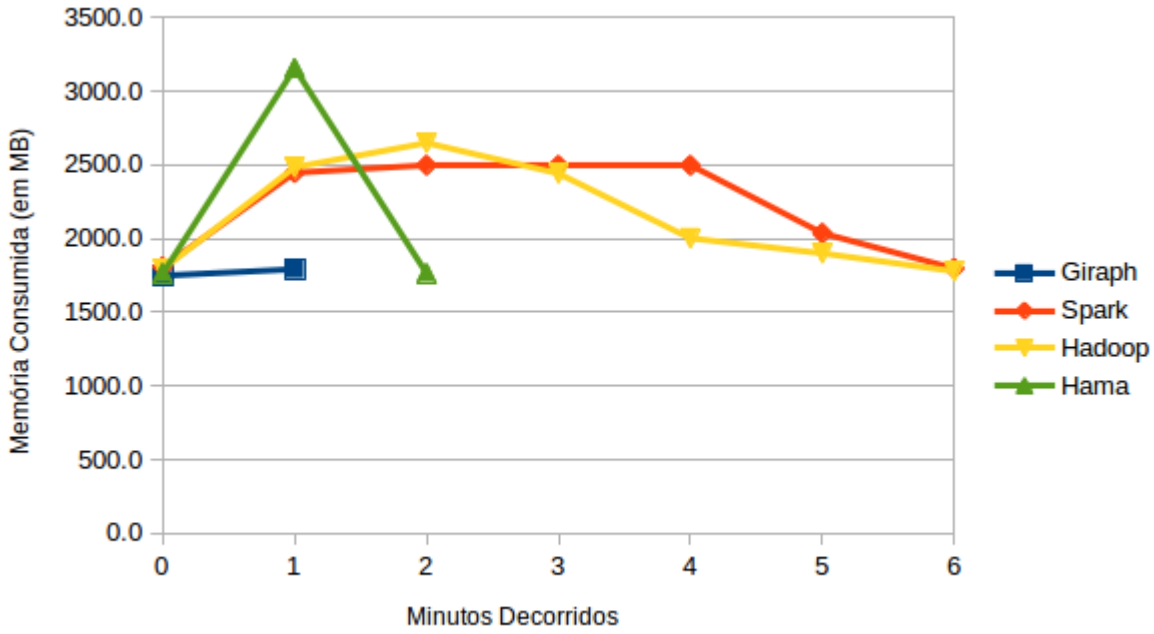

Figura 18: Utilização de Memória em "meu-grafo.txt" com 10 Iterações

valores para iterações do PageRank ou outras bases de dados.

Tabela 4: Tempo em segundos da execução do PageRank no *Cluster*

Arcabouços	Número de Iterações	web-Stanford.txt	meu-grafo.txt	web-Google.txt
Hadoop	10	626,837	638,616	1.063,856
	20	1.152,089	1.204,529	2.155,285
Spark	10	314,927	314,452	1.340,780
	20	515,130	522,765	2.496,869
Giraph	10	74,243	78,885	138,133
	20	87,959	106,934	197,900
Hama	10	205,626	221,529	558,984
	20	354,439	380,489	1.097,364

4.1.2 EXECUÇÃO DO CAMINHO MAIS CURTO

O tempo de execução do algoritmo do Caminho mais Curto em cada arcabouço na Máquina Autônoma descrita em 3.3 pode ser visualizado na Tabela 5. Cada linha da tabela exibe o tempo, em segundos, utilizado por um arcabouço para executar cada uma das bases de dados. Não foi possível executar o arcabouço Giraph com a base de dados "web-Stanford.txt" e nem o arcabouço Spark com a base de dados "web-Google.txt", logo, seus tempos de execução para tais bases encontram-se anulados.

O arcabouço Hadoop teve o pior tempo para todas as bases de dados. Em sua execução

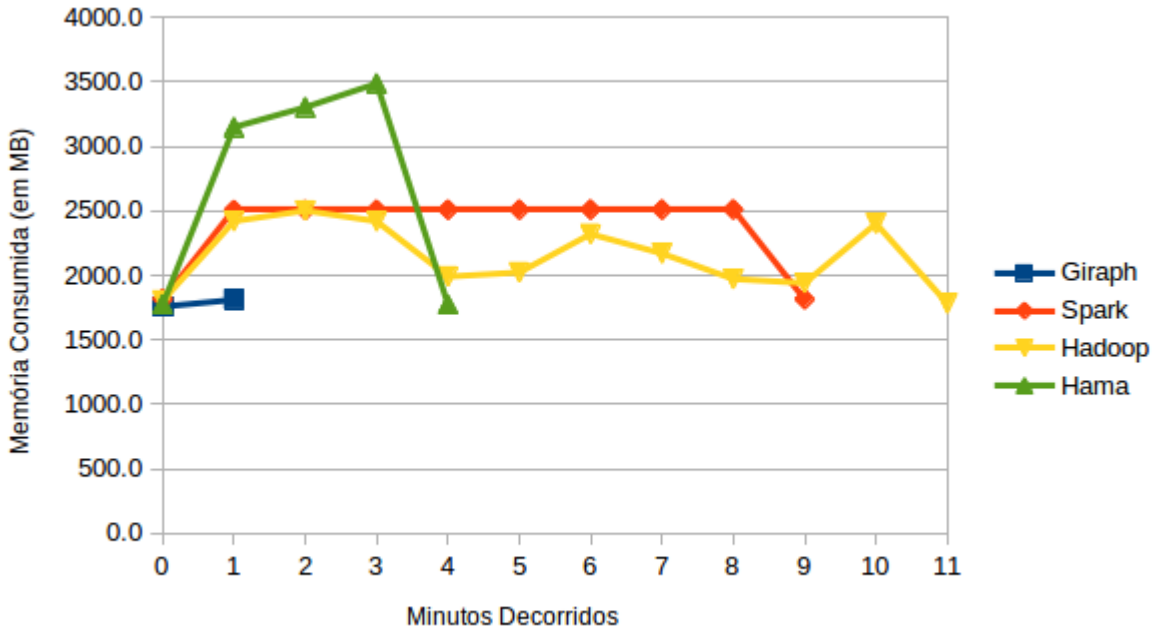

Figura 19: Utilização de Memória em "meu-grafo.txt" com 20 Iterações

para "web-Stanford.txt", ele precisou de 3.662,95 segundos, 2.596,44 segundos a mais que o segundo arcabouço mais lento para tal base de dados, o Spark, e mais de 15 vezes o tempo utilizado pelo Hama, arcabouço mais veloz para a mesma base de dados. Ao mesmo tempo, Hadoop terminou sua execução com a base de dados "web-Google.txt" mais rapidamente do que sua própria execução para "web-Stanford.txt", precisando de 2.825,37 e 3.662,95 segundos, respectivamente, comportamento contrário ao do arcabouço Hama, que precisou de mais tempo para processar a base de dados "web-Google.txt" (355,81 segundos) do que "web-Stanford.txt" (236,56 segundos).

Giraph foi o arcabouço com o menor tempo de execução global, com a devida ressalva de que ele não foi executado com a base de dados "web-Stanford.txt", precisando de 41,52 segundos para completar a execução com a base de dados "meu-grafo.txt", menos de $\frac{1}{20}$ do tempo necessário pelo Hadoop na mesma execução, e 50,16 segundos a menos que o Hama, segundo arcabouço mais veloz para a base de dados "meu-grafo.txt".

Tabela 5: Tempo em segundos da execução do Caminho mais Curto na Máquina Autônoma

Arcabouços	web-Stanford.txt	meu-grafo.txt	web-Google.txt
Hadoop	3.662,957	867,108	2.825,373
Spark	1.066,519	230,222	—
Giraph	—	41,526	74,133
Hama	236,567	91,686	355,818

As Figuras 22, 23 e 24 exibem a utilização média de memória em MB de cada um dos

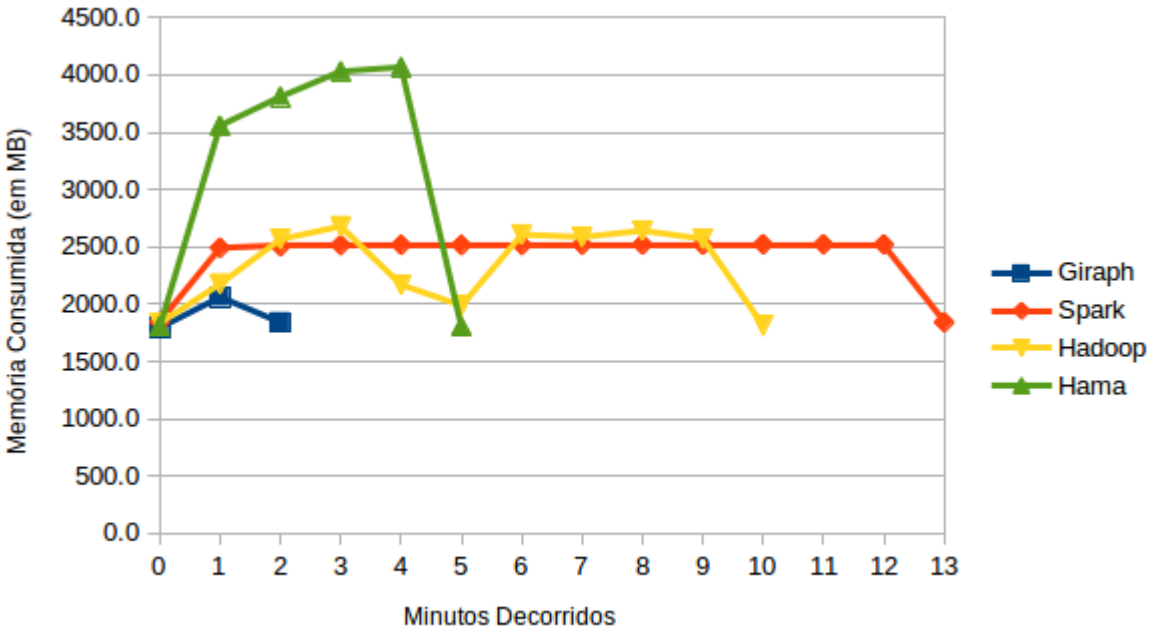

Figura 20: Utilização de Memória em "web-Google.txt" com 10 Iterações

arcabouços durante a execução do Caminho mais Curto na Máquina Autônoma.

A Figura 22 exibe a utilização média de memória de cada um dos arcabouços executando a base de dados "web-Stanford.txt". O arcabouço Hama alcançou o maior pico de utilização de memória, chegando a utilizar 2,5 GBs aos 4 minutos de execução. Nesse mesmo minuto de execução, Spark utilizou aproximadamente 2,1 GBs e Hadoop utilizou 1,7 GB. Entretanto, como pode ser visto na Tabela 5, Hadoop precisou de mais de 15 vezes o tempo utilizado pelo Hama para terminar sua execução, 3.662, 95 e 236, 56 segundos, respectivamente.

A Figura 23 exibe a utilização de memória de cada um dos arcabouços executando a base de dados "meu-grafo.txt". Com esta base de dados, o arcabouço que atingiu o maior pico de utilização de memória foi o Hadoop, chegando a utilizar mais de 4,5 GBs aos 10 minutos de execução. Ambos, Spark e Hama, também utilizaram uma quantidade similar de memória, de aproximadamente 4,5 GBs no primeiro minuto de execução. Giraph manteve sua utilização de memória abaixo dos 4 GBs durante seus 41, 52 segundos de execução.

A Figura 24 exibe a utilização de memória de cada um dos arcabouços executando a base de dados "web-Google.txt". Com esta base de dados o arcabouço Giraph alcançou o maior pico de utilização de memória, chegando a utilizar mais de 6 GBs em seu primeiro minuto de execução. Ao mesmo tempo, o arcabo com o segundo maior pico, Hama, utilizou pouco menos de 6 GBs. Entretanto, ao passo que Giraph precisou de 74, 13 segundos para completar sua execução, Hama precisou de quase 5 vezes esse valor: 355, 81 segundos.

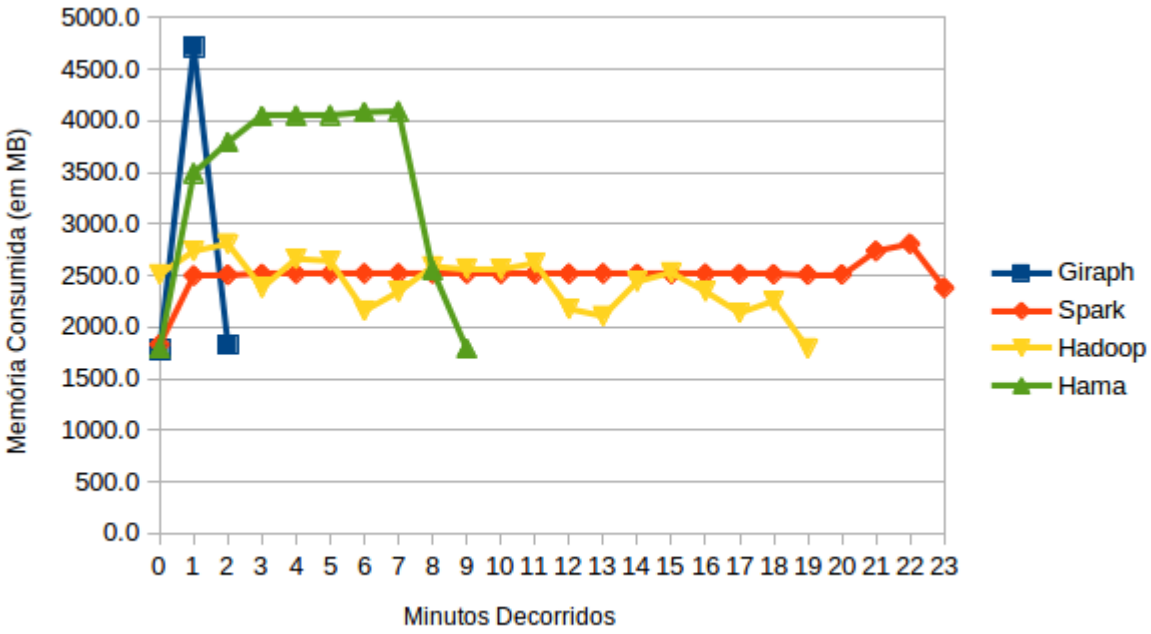

Figura 21: Utilização de Memória em "web-Google.txt" com 20 Iterações

A Tabela 6 apresenta o tempo de execução do algoritmo do Caminho mais Curto de cada arcabouço no *Cluster* descrito em 3.3. A leitura desta tabela segue o mesmo padrão da Tabela 5. A exemplo da execução do Caminho mais Curto na Máquina Autônoma, não foi possível executar o arcabouço Giraph com a base de dados "web-Stanford.txt" e nem o arcabouço Spark com a base de dados "web-Google.txt", por isso seus tempos de execução para tais bases de dados encontram-se anulados.

Também aqui Hadoop manteve-se como o arcabouço mais lento na execução de todas as bases de dados. Manteve, também, o comportamento contrário ao do Hama ao demorar menos tempo para processar a base de dados "web-Google.txt" do que "web-Stanford.txt". Para executar "web-Stanford.txt" Hadoop precisou de $6.482,97$ segundos, $4.424,74$ segundos a mais que o segundo arcabouço mais lento para esta base de dados, Spark, o qual precisou de $2.058,23$ segundos. Hama, o arcabouço mais veloz para a mesma base, precisou de $389,03$ segundos.

Giraph foi o arcabouço que precisou de menos tempo para processar a base de dados "meu-grafo.txt", $64,82$ segundos, menos de $1/20$ do tempo utlizado pelo Hadoop e menos da metade do tempo utilizado pelo Hama.

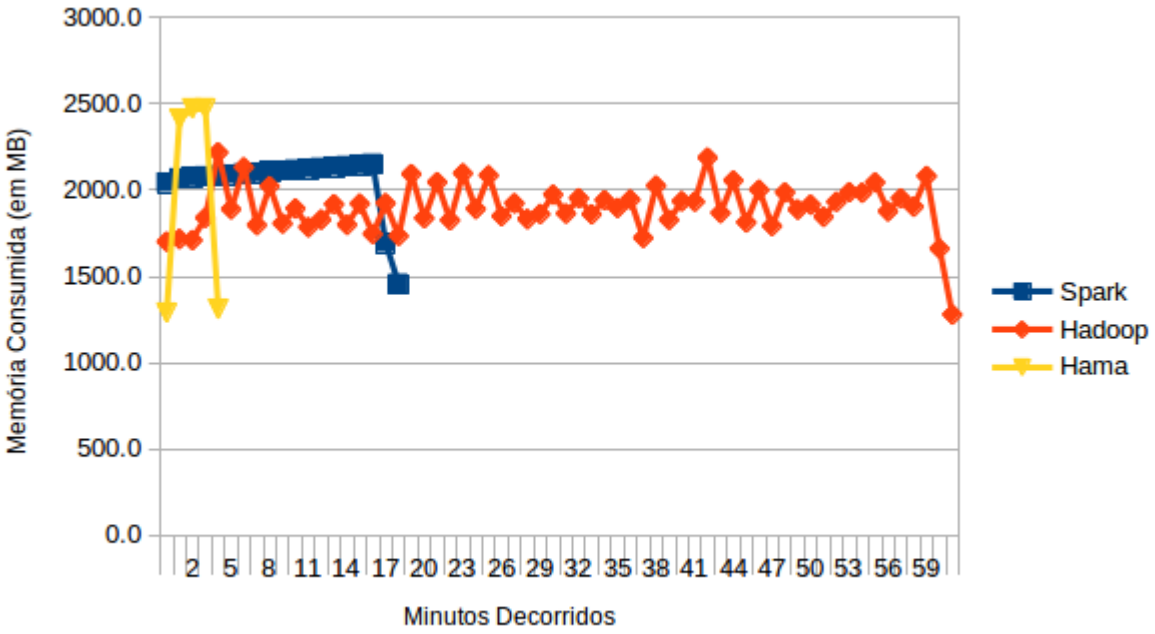

Figura 22: Utilização de Memória em "web-Stanford.txt" - Caminho mais Curto

Tabela 6: Tempo em segundos da execução do Caminho mais Curto no *Cluster*

Arcabouços	web-Stanford.txt	meu-grafo.txt	web-Google.txt
Hadoop	6.482,975	1.330,784	4.254,853
Spark	2.058,231	387,315	—
Giraph	—	64,826	139,430
Hama	389,035	170,952	707,703

4.2 DISCUSSÕES

A partir dos dados apresentados nas Tabelas 3, 4, 5 e 6, é possível aferir que, para os ambientes e métricas utilizados, o arcabouço Giraph é o mais veloz para a execução dos algoritmos do PageRank e do Caminho mais Curto. Também é possível verificar que nenhum arcabouço, em nenhuma execução, foi afetado positiva ou negativamente pela diferença entre o desvio padrão das bases de dados "meu-grafo.txt" e "web-Stanford.txt", pois os tempos de execução de todos os arcabouços para estas duas bases foram bastante similares. As variações nos tempos de execução do algoritmo do PageRank devem-se ao fato de que a base de dados "meu-grafo.txt" possui uma quantidade levemente maior de vértices; enquanto as variações para o algoritmo do Caminho mais Curto devem-se à diferente quantidade de iterações do algoritmo necessárias para se chegar ao resultado ótimo, de acordo com a lógica exposta em 2.11.

É possível visualizar nas Figuras 22, 23 e 24, que exibem a utilização de memória pelos arcabouços na execução do algoritmo do Caminho mais Curto na Máquina Autônoma,

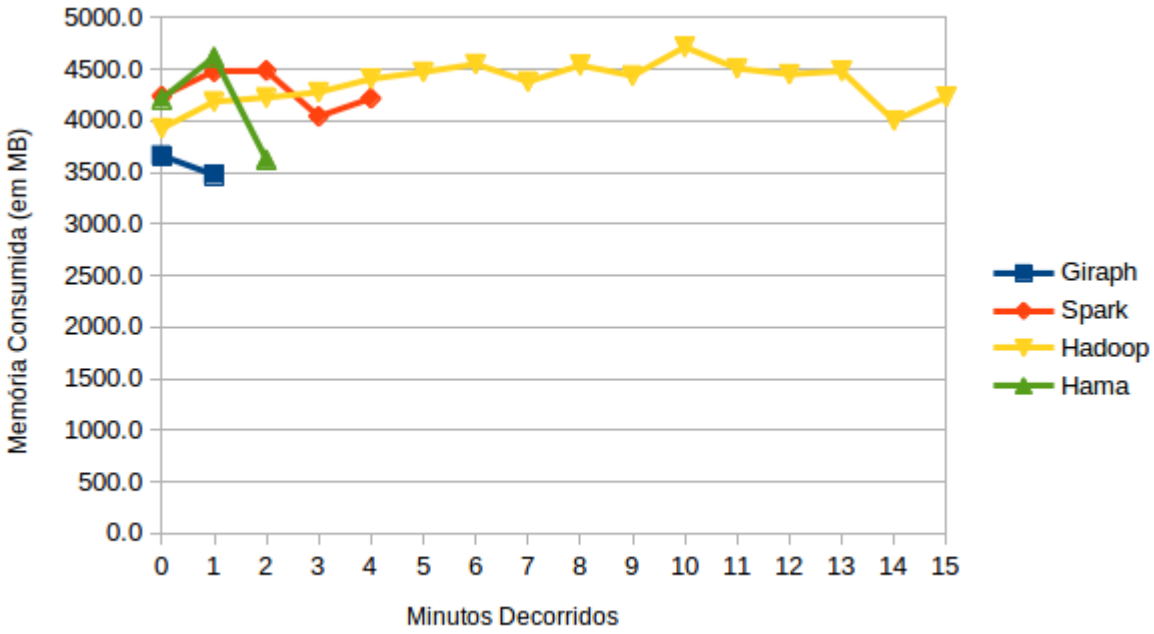

Figura 23: Utilização de Memória em “meu-grafo.txt” - Caminho mais Curto

que o arcabouço Hadoop apresenta um padrão de picos e vales de utilização de memória: em um minuto ele atinge um certo pico de utilização de memória (o maior valor local), e no minuto seguinte atinge um vale (o menor valor local). Embora não possa ser confirmada a fonte de tal padrão, ou se tal padrão sequer se manteria em outros ambientes/métricas, uma possível causa seria o fato de que para toda iteração do algoritmo o Hadoop lê os dados necessários do disco e os processa, e ao final da iteração ele escreve os novos dados em disco, o que representaria, respectivamente, os vales e picos de utilização de memória.

Também é possível visualizar um certo padrão por parte do arcabouço Spark nas Figuras 16, 17, 18, 19, 20 e 21, as quais representam a utilização de memória dos arcabouços na execução do algoritmo do PageRank com ambas 10 e 20 iterações. Spark aparenta, em tais execuções, manter uma utilização linear de memória. Embora nesse caso também não possa ser confirmada a causa ou existência do padrão, uma possível explicação seria o fato de que Spark, sempre que possível, não faz acessos a disco. Ele lê os dados necessários no início de sua execução e em seguida utiliza apenas a memória principal para realizar suas operações. Considerando o tamanho das bases de dados utilizadas, descritas em 3.1, elas podem facilmente ser armazenadas em memória principal durante toda a execução dos algoritmos, explicando a ausência de acesso a disco por parte do Spark. A tal soma-se o fato de que os algoritmos executados realizam operações por meio de iterações seguindo o modelo BSP: cada iteração depende apenas da última iteração, não sendo necessário armazenar grande quantidade de dados em memória principal.

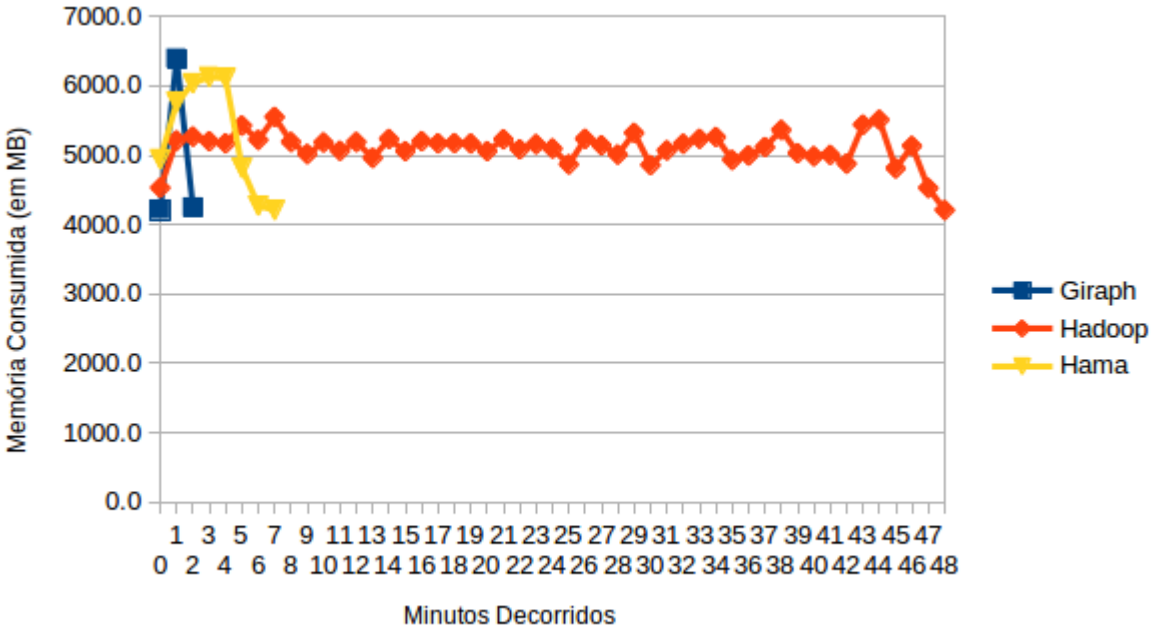

Figura 24: Utilização de Memória em “web-Google.txt” - Caminho mais Curto

5 CONSIDERAÇÕES FINAIS

O trabalho apresenta uma pesquisa comparativa entre arcabouços de processamento paralelo de grafos: Hadoop, Spark, Giraph e Hama. A comparação foi realizada executando os algoritmos do PageRank e do Caminho mais Curto. Como medidas de comparação para esta análise foram utilizados o tempo de execução e a utilização de memória pelos arcabouços.

Os arcabouços foram executados em dois ambientes, Máquina Autônoma e *Cluster* (este último constando de três máquinas virtuais), e com três bases de dados distintas, "web-Stanford.txt", "meu-grafo.txt" e "web-Google.txt". O algoritmo do PageRank foi executado utilizando dois valores para o número máximo de iterações, 10 e 20.

Pode-se concluir, dados os ambientes e métricas utilizados pela presente pesquisa, que o arcabouço mais veloz é o Giraph, pois este, em nehuma execução, passou a marca de 200 segundos, tendo alcançado seu maior tempo na execução do algoritmo do PageRank com 20 iterações, no *Cluster*, com a base de dados "web-Google.txt", na qual utilizou $197,90$ segundos. Nessa mesma execução, o segundo arcabouço mais veloz foi o Hama, com $1.097,36$ segundos. Entretanto, vale ressaltar que o arcabouço Giraph também alcançou os maiores picos de utilização de memória com a base de dados "web-Google.txt" nas execuções do PageRank com 20 iterações e do Caminho mais Curto (ambas as execuções na Máquina Autônoma), nas quais utilizou aproximadamente 4,7 e 6,3 GBs, respectivamente.

Nas execuções do PageRank na Máquina Autônoma, tanto com 10 como com 20 iterações, o arcabouço Spark se mostrou o mais estável em sua utilização de memória, mantendo-se sempre na faixa de 2,5 GBs.

O arcabouço Hadoop foi o mais lento na maioria das execuções, o que já era esperado, considerando que ele, ao contrário do Giraph e do Hama, não é um arcabouço próprio para a execução de algoritmos iterativos para grafos como os algoritmos do PageRank e do Caminho mais Curto. Como pode ser visualizado na Tabela 3, Hadoop precisou de $647,31$ segundos para terminar sua execução do algoritmo do PageRank com 20 iterações na Máquina Autônoma com a base de dados "meu-grafo.txt", $167,13$ a mais que o Spark, o qual precisou de $480,18$

segundos para a mesma execução, e mais de 12 vez o tempo utilizado pelo Giraph, 53,33 segundos.

Com tais resultados, pode-se perceber que organizações ou pessoas que precisem executar os algoritmos do PageRank e do Caminho mais Curto, ou outros algoritmos de grafos que sigam uma lógica parecida, em ambientes de execução similares aos expostos aqui deverão utilizar o arcabouço Giraph para uma execução mais veloz, ou o arcabouço Spark para uma menor utilização de memória principal.

5.1 DIFICULDADES ENCONTRADAS

Dentre outras, as principais dificuldades encontradas durante o desenvolvimento da presente pesquisa foram:

- **Imaturidade dos Arcabouços**: os arcabouços de processamento paralelo de grafos ainda são novos (Spark começou em 2009, Giraph em 2010 e Hama também em 2010), o que dificulta a consulta de materiais relacionados na web. Como exemplo, não foi encontrada nenhuma implementação do algoritmo do Caminho mais Curto em Java para o arcabouço Spark.
- **Implementação e execução dos algoritmos**: modificar os códigos-fonte originais do PageRank dos arcabouços Giraph, Hama e Spark para executá-los com os arquivos de entrada no formato desejado tomou parte considerável do tempo disponível.
- **Período para desenvolvimento do trabalho**: o tempo para desenvolver um trabalho como o presente é curto, considerando que devem ser realizados pequenos testes, a todo momento, para aferir se os resultados finais são válidos, e pequenas modificações (nem sempre modificações triviais) nos códigos dos algoritmos sendo executados para testar as variações nos tempos de execução e consumo de memória.

5.2 TRABALHOS FUTUROS

Futuramente poderiam ser realizados outros trabalhos de pesquisa que acrescentem algo ao presente. Exemplos:

- Utilizar outros algoritmos além do PageRank e do Caminho mais Curto;
- Utilizar maiores bases de dados, na escala de GigaBytes ou até mesmo TeraBytes;

- Analisar tráfego de rede de cada arcabouço sendo executado em um *cluster*;
- Recolher e analisar uso da CPU e mais detalhes da utilização de memória de cada arcabouço;
- Utilizar as técnicas disponíveis em cada arcabouço para aprimoramento de performance;
- Testar a escalabilidade dos arcabouços, tanto em relação ao tamanho das bases de dados como em relação à quantidade de máquinas de um *cluster*.

Este trabalho foi realizado com o auxílio financeiro de uma bolsa advinda do Programa de Bolsas de Fomento às Ações de Graduação da Universidade Tecnológica Federal do Paraná, câmpus Cornélio Procópio.

REFERÊNCIAS

APACHE Giraph. mai 2014. Disponível em: <https://giraph.apache.org/>. Acesso em: 13 de mai. 2014.

APACHE Spark. mar 2015. Disponível em: <http://spark.apache.org>. Acesso em: 17 de mar. de 2015.

ARNOLD, P.; PEETERS, D.; THOMAS, I. Modelling a rail/road intermodal transportation system. *Transportation Research Part E: Logistics and Transportation Review*, Elsevier, v. 40, n. 3, p. 255–270, maio 2004. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1366554503000723>.

BOLDI, P.; SANTINI, M.; VIGNA, S. Pagerank as a function of the damping factor. In: ACM. *Proceedings of the 14th international conference on World Wide Web*. [S.l.], 2005. p. 557–566.

BRIN, S.; PAGE, L. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, Elsevier, v. 30, n. 1, p. 107–117, 1998.

BULK synchronous parallel. mai 2015. Disponível em: <http://en.wikipedia.org/wiki/Bulk_synchronous_parallel>. Acesso em: 20 de mai. de 2015.

CASTILLO, C. Effective web crawling. In: ACM. *ACM SIGIR Forum*. New York, New York, USA, 2005. v. 39, n. 1, p. 55–56.

CORMEN, T. H. et al. *Introduction to Algorithms*. Cambridge, Massachusetts, USA: MIT press, 2009.

DEAN, J.; GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, ACM, New York, New York, USA, v. 51, n. 1, p. 107–113, jan. 2008. Disponível em: <http://dl.acm.org/citation.cfm?id=1327492>.

DEGENERACY (graph theory). mai 2014. Disponível em: <https://en.wikipedia.org/wiki/K-core#k-Cores>. Acesso em: 28 de mai. 2014.

DIRECTED Graph. mai 2014. Disponível em: <https://en.wikipedia.org/wiki/Directed_graph>. Acesso em: 19 de mai. 2014.

ELSER, B.; MONTRESOR, A. An evaluation study of bigdata frameworks for graph processing. In: IEEE. *Big Data, 2013 IEEE International Conference on*. Silicon Valley, California, USA, 2013. p. 60–67.

FEOFILOFF, P. *Graus e Cortes*. abr 2015. Disponível em: <http://www.ime.usp.br/ pf/algoritmos_em_grafos/aulas/graus.html>. Acesso em: 14 de abr. de 2015.

FOLEY, M. J. *Microsoft drops Dryad; puts its big-data bets on Hadoop*. mai 2014. Disponível em: <http://www.zdnet.com/blog/microsoft/microsoft-drops-dryad-puts-its-big-data-bets-on-hadoop/11226>. Acesso em: 13 de mai. 2014.

GOLDMAN, A. et al. Apache hadoop: Conceitos teóricos e práticos, evolução e novas possibilidades. *XXXI Jornadas de atualizações em informática*, jul 2012.

GUO, Y. et al. How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In: IEEE. *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. Phoenix, Arizona, USA, 2014. p. 395–404.

HAN, Y.; JIN, J.; WANG, J. Comparing pregel-like graph processing systems. 2013.

HOW to Write a MapReduce Program Using the Hadoop Framework and Java. abr 2015. Disponível em: <http://www.devx.com/Java/how-to-write-a-map-reduce-program-using-the-hadoop-framework-and-java.html>. Acesso em: 13 de abr. de 2015.

HUI, P.; CROWCROFT, J.; YONEKI, E. Bubble rap: Social-based forwarding in delay-tolerant networks. *Mobile Computing, IEEE Transactions on*, IEEE, v. 10, n. 11, p. 1576–1589, nov. 2011. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5677535>.

ISARD, M. et al. Dryad: Distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, ACM, New York, New York, USA, v. 41, n. 3, p. 59–72, 2007. Disponível em: <http://dl.acm.org/citation.cfm?id=1273005>.

JING, Y.; BALUJA, S. Visual rank: Applying pagerank to large-scale image search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, IEEE, v. 30, n. 11, p. 1877–1890, 2008.

KAMBURUGAMUVE, S. et al. *Survey of Apache Big Data Stack*. Tese (Doutorado) — Ph. D. Qualifying Exam, Dept. Inf. Comput., Indiana Univ., Bloomington, IN, 2013.

KANG, U. et al. Gbase: a acalable and general graph management system. In: ACM. *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, New York, USA, 2011. p. 1091–1099.

KANG, U.; TSOURAKAKIS, C. E.; FALOUTSOS, C. Pegasus: A peta-scale graph mining system implementation and observations. In: IEEE. *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. Miami, Florida, USA, 2009. p. 229–238.

KIM, S.-H. et al. Parallel processing of multiple graph queries using mapreduce. In: *DBKDA 2013, The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications*. Seville, Spain: Elsevier, 2013. p. 33–38.

LÄMMEL, R. Google's mapreduce programming model—revisited. *Science of computer programming*, Elsevier, v. 70, n. 1, p. 1–30, 2008.

LIN, J.; SCHATZ, M. Design patterns for efficient graph algorithms in mapreduce. In: ACM. *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*. New York, New York, USA, 2010. p. 78–85.

LOHR, S. *The Age of Big Data*. fev 2012. Disponível em: <http://www.nytimes.com/2012/02/12/sunday-review/big-datas-impact-in-the-world.html>. Acesso em: 21 de mai. 2014.

LUNDEN, I. *Facebook Passes 1B Mobile Users*. abr 2014. Disponível em: <http://techcrunch.com/2014/04/23/facebook-passes-1b-mobile-monthly-active-users-in-q1-as-mobile-ads-reach-59-of-all-ad-sales/?ncid=rss>. Acesso em: 21 de mai. 2014.

MA, N.; GUAN, J.; ZHAO, Y. Bringing pagerank to the citation analysis. *Information Processing & Management*, Elsevier, v. 44, n. 2, p. 800–810, 2008.

MALEWICZ, G. et al. Pregel: A system for large-scale graph processing. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. Indianapolis, Indiana, USA: ACM, 2010. (SIGMOD '10), p. 135–146. ISBN 978-1-4503-0032-2. Disponível em: <http://doi.acm.org/10.1145/1807167.1807184>.

MARKOV Chain. mai 2014. Disponível em: <https://en.wikipedia.org/wiki/Markov_chain>. Acesso em: 20 de mai. 2014.

MCAFEE, A.; BRYNJOLFSSON, E. et al. Big data: The management revolution. *Harvard business review*, v. 90, n. 10, p. 60–68, 2012. Disponível em: <http://automotivedigest.com/wp-content/uploads/2013/01/BigDataR1210Cf2.pdf>.

PAGE, L. et al. The pagerank citation ranking: Bringing order to the web. Stanford InfoLab, 1999.

PAGE Rank. mai 2014. Disponível em: <http://en.wikipedia.org/wiki/PageRank>. Acesso em: 16 de mai. 2014.

PAGE Rank History. mai 2014. Disponível em: <http://w3pagerank.com/history/>. Acesso em: 16 de mai. 2014.

PEREIRA, F. J. Using the bsp model on clouds. *Cloud Computing and Big Data*, IOS Press, v. 23, p. 123, 2013.

SEO, S. et al. Hama: An efficient matrix computation with the mapreduce framework. In: IEEE. *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. Indianapolis, Indiana, USA, 2010. p. 721–726.

SKIENA, S. S. *The Algorithm Design Manual*. Stony Brook, New York, USA: Springer, 2008. ISBN 1848000693. Disponível em: <http://researchbooks.org/1848000693>.

SOFTWARE framework. jun 2015. Disponível em: <http://en.wikipedia.org/wiki/Software_framework>. Acesso em: 15 de jun. de 2015.

SPARK officially sets a new record in large-scale sorting. nov 2014. Disponível em: <https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html>. Acesso em: 18 de mar. de 2015.

SPARK Overview. mar 2015. Disponível em: <http://spark.apache.org/docs/1.0.2/>. Acesso em: 17 de mar. de 2015.

SPARSE vs. Dense Graphs. mai 2014. Disponível em: <http://brpreiss.com/books/opus4/html/page534.html>. Acesso em: 16 de mai. 2014.

TENENBAUM, A. M. *Data Structures Using C*. [S.l.]: Pearson Education India, 1990.

THE Health Graph. mai 2014. Disponível em: <http://developer.runkeeper.com/healthgraph/introducing-the-health-graph>. Acesso em: 21 de mai. 2014.

TING, I.-H.; LIN, C.-H.; WANG, C.-S. Constructing a cloud computing based social networks data warehousing and analyzing system. In: IEEE. *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on*. Kaohsiung, Taiwan, 2011. p. 735–740.

TROIA, B. *The Business of Social Graphs (and Why Everyone Wants to Own One)*. mai 2014. Disponível em: <http://www.socialmediaplayground.com/social-media/the-business-of-social-graphs-and-why-everyone-wants-to-own-one/2011/09/15/>. Acesso em: 21 de mai. 2014.

TWO Shortest Path Algorithms. mai 2015. Disponível em: <http://cs.geneseo.edu/ baldwin/csci242/fall2012/1023short.html>. Acesso em: 10 de mai. 2015.

VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM*, ACM, New York, New York, USA, v. 33, n. 8, p. 103–111, ago. 1990. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/79173.79181>.

VENUES Platform. mai 2014. Disponível em: <https://developer.foursquare.com/overview/venues.html>. Acesso em: 21 de mai. 2014.

WHAT is the function of the damping factor in PageRank? mai 2015. Disponível em: <http://qr.ae/0dQ0X>. Acesso em: 10 de mai. de 2015.

WHITE, T. *Hadoop: The Definitive Guide*. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2009. ISBN 0596521979, 9780596521974.

ZAHARIA, M. et al. *Spark: Cluster Computing with Working Sets*. [S.l.], May 2010. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-53.html>.