CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM MATEMÁTICA E COMPUTACIONAL

ADDIC - UM ALGORITMO DISTRIBUÍDO PARA A DESCOBERTA DE COMUNIDADES SOBREPOSTAS EM GRANDES GRAFOS

JOÃO PAULO BARBOSA NASCIMENTO

BELO HORIZONTE
SETEMBRO DE 2019

JOÃO PAULO BARBOSA NASCIMENTO

ADDIC - UM ALGORITMO DISTRIBUÍDO PARA A DESCOBERTA DE COMUNIDADES SOBREPOSTAS EM GRANDES GRAFOS

Tese apresentada ao Programa de Pós-graduação em Modelagem Matemática e Computacional do Centro Federal de Educação Tecnológica de Minas Gerais, como requisito parcial para a obtenção do título de Doutor em Modelagem Matemática e Computacional.

Área de concentração: Modelagem Matemática e Computacional

Linha de pesquisa: Métodos Matemáticos Aplicados

Orientador: Prof. Dr. Adriano César Machado Pereira

BELO HORIZONTE
SETEMBRO DE 2019

Nascimento, João Paulo Barbosa
N244a ADDIC – Algoritmo distribuído para a descoberta de comunidades sobrepostas em grandes grafos. / João Paulo Barbosa Nascimento. – – Belo Horizonte, 2019.
110 f. : il.

Tese (Doutorado) – Centro Federal de Educação Tecnológica de Minas Gerais, Programa de Pós-Graduação em Modelagem Matemática e Computacional, 2019.
Orientador: Prof. Dr. Adriano César Machado Pereira

Bibliografia

1. Modelos Matemáticos. 2. Algoritmos – Redes Complexas. 3. Teoria dos Grafos. I. Pereira, Adriano César Machado. II. Centro Federal de Educação Tecnológica de Minas Gerais. III. Título

CDD 511.8

Elaboração da ficha catalográfica pela Bibliotecária Elisângela Gonçalves Barbieri CRB-6: 2624 / CEFET-MG

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
COORDENAÇÃO DO PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM MATEMÁTICA E COMPUTACIONAL

ADDIC - UM ALGORITMO DISTRIBUÍDO PARA A DESCOBERTA DE COMUNIDADES SOBREPOSTAS EM GRANDES GRAFOS.

Tese de Doutorado apresentada por **João Paulo Barbosa Nascimento**, em 26 de setembro de 2019, ao Programa de Pós-Graduação em Modelagem Matemática e Computacional do CEFET-MG, e aprovada pela banca examinadora constituída pelos professores:

Prof. Dr. Adriano César Machado Pereira
Centro Federal de Educação Tecnológica de Minas Gerais

Prof. Dr. Henrique Cota de Freitas
Pontifícia Universidade Católica de Minas Gerais

Prof. Dr. Carlos Augusto Paiva da Silva Martins
Pontifícia Universidade Católica de Minas Gerais

Prof. Dr. Paulo Eduardo Maciel de Almeida
Centro Federal de Educação Tecnológica de Minas Gerais

Prof. Dr. Flávio Luis Cardeal Pádua
Centro Federal de Educação Tecnológica de Minas Gerais

Visto e permitida à impressão,

Prof. Dr. Thiago de Souza Rodrigues
Coordenador do Programa de Pós-Graduação Stricto Sensu em Modelagem Matemática e Computacional

Dedicado aos meus filhos Carlos Eduardo (Kadu), Gustavo (Guto) e minha esposa Pollyanna.

Agradecimentos

Agradeço primeiramente ao meu orientador Prof. Dr. Adriano César Machado Pereira, pelos ensinamentos e sabedoria transmitidos com maestria, pela dedicação às orientações e por ser um grande amigo.

Aos meus pais por terem me ensinado o caminho.

Ao amigo Daniel Oliveira Capanema pela amizade e parceria no desenvolvimento deste trabalho.

À minha esposa Polly por entender que esse projeto se trata especificamente de um sonho e por viver esse sonho junto.

Ao meu sobrinho Carlos Júnior e meu irmão Carlos Antônio pelo incentivo.

À Teca pelo companheirismo.

Aos professores e colegas do Programa de Modelagem Matemática e Computacional do CEFET-MG pelos ensinamentos e apoio.

A todos que direta ou indiretamente contribuíram com esse trabalho. Antes de reclamar e exigir é preciso trabalhar duro e dar o exemplo.

Nada resiste ao trabalho.

Resumo

Redes complexas são sistemas grandes e dinâmicos que podem ser modelados por grafos. Muitos sistemas sociais, biológicos ou tecnológicos são considerados redes complexas, dentre eles a Internet e a Web, as redes sociais reais ou virtuais e as redes de infraestrutura física, tais como redes de transporte e de comunicação. A análise destas redes, na forma de grandes grafos, provê informações importantes acerca das características dos sistemas modelados. No entanto, o processamento sequencial destes grafos pode requerer alto custo computacional ou mesmo ser inviável, dependendo de seu tamanho e complexidade. Para solucionar esse problema recorre-se ao processamento paralelo e distribuído. Este trabalho propõe o algoritmo paralelo e distribuído ADDIC, indicado para o processo de detecção de comunidades sobrepostas em uma rede complexa de grande porte. O algoritmo foi projetado a partir de um importante algoritmo da literatura denominado DEMON e também por meio de um estudo criterioso do *framework* de programação paralela e distribuída Apache Spark, aliado a um planejamento detalhado de experimentos. Os experimentos foram executados em vários tipos de grafos de diferentes tamanhos, reais e sintéticos. O tempo de execução, consumo de recursos (CPU e Memória RAM) e medidas de escalabilidade e eficiência do algoritmo foi avaliado. Os resultados dos experimentos indicam que o ADDIC alcança seu objetivo, que é detectar comunidades sobrepostas, além de apresentar resultados de tempo de execução melhores que o principal algoritmo sequencial da literatura. Além disso, o algoritmo proposto apresenta escalabilidade próxima à linearidade e boas taxas de eficiência.

Palavras-chave: Grafos. Detecção de Comunidades. Redes Sociais. Algoritmos Distribuídos. Apache Spark. Desempenho. Escalabilidade.

Abstract

Complex networks are large dynamic systems that can be modeled by graphs. Many social, biological or technological systems are considered complex networks, including the Internet and the Web, real or virtual social networks and physical infrastructure networks such as transport and communication. The analysis of these networks, in the form of large graphs, provides important information about the characteristics of modeled systems. However, sequential processing of these graphs may require high computational cost or may not be feasible, depending on their size and complexity. To solve this problem we resort to parallel and distributed processing. This work proposes the parallel and distributed algorithm ADDIC, indicated for the process of detecting overlapping communities in a large complex network. The algorithm was designed from an important literature algorithm and also through a careful study of the Apache Spark parallel and distributed programming framework, combined with a detailed design of experiments. The experiments were performed on various types of graphs of different sizes, real and synthetic. The runtime, resource consumption (CPU and RAM), scalability and efficiency measures of the algorithm were evaluated. The results of the experiments indicate that ADDIC achieves its goal, which is to detect overlapping communities, and to present better runtime results than the main sequential algorithm in the literature. In addition, the proposed algorithm presents scalability close to linearity and good efficiency rates.

Palavras-chave: Graph. Community Detection. Social Network. Distributed Algorithm. Apache Spark. Performance. Scalability.

Lista de Figuras

Figura 1 – Exemplo de uma pequena rede com três comunidades. As comunidades estão destacadas com os círculos pontilhados (NEWMAN, 2003). 18
Figura 2 – Exemplo de uma pequena rede com quatro comunidades sobrepostas por um vértice. 19
Figura 3 – Exemplo de curvas de *Speedup*, baseado em (BRESHEARS, 2009) . . 26
Figura 4 – Grafo com onze vértices e treze arestas 28
Figura 5 – Grafo de uma rede social particionado em comunidades (PONTOIZEAU, 2018) 29
Figura 6 – Exemplo de uma comunidade sobreposta com quatro cliques (PONTOIZEAU, 2018) 31
Figura 7 – Ego Network do Vértice 7 com dez vértices e dezessete arestas 33
Figura 8 – Exemplo de funcionamento do algoritmo Label Propagation (RAGHAVAN; ALBERT; TIRUPATIKUMARA, 2007) 33
Figura 9 – Funcionamento do Modelo MapReduce baseado em White (2009) . . . 36
Figura 10 – Armazenamento Distribuído no HDFS 37
Figura 11 – O Ecossistema Apache Spark 39
Figura 12 – Ano das referências utilizadas. 54
Figura 13 – Exemplo de um grafo com cinco vértices e sete arestas 59
Figura 14 – Exemplo de um grafo com quinze vértices e vinte e cinco arestas 60
Figura 15 – Primeira operação da função EgoMinusEgo - Vértice 1. 62
Figura 16 – Segunda operação da função EgoMinusEgo - Vértice 1. 62
Figura 17 – Resultado das três funções do algoritmo ADDIC aplicadas ao Vértice 1. 63
Figura 18 – Resultado das três funções do algoritmo ADDIC aplicadas ao Vértice 2. 63
Figura 19 – Resultado das três funções do algoritmo ADDIC aplicadas ao Vértice 6. 64
Figura 20 – Resultado das três funções do algoritmo ADDIC aplicadas ao Vértice 7. 64
Figura 21 – Resultado das três funções do algoritmo ADDIC aplicadas ao Vértice 3. 65
Figura 22 – Resultado das três funções do algoritmo ADDIC aplicadas ao Vértice 8. 65
Figura 23 – Resultado das três funções do algoritmo ADDIC aplicadas aos Vértices 4 e 5. 65
Figura 24 – Resultado das três funções do algoritmo ADDIC aplicada ao Vértice 9. . 66
Figura 25 – Resultado das três funções do algoritmo ADDIC aplicada ao Vértice 12. 66
Figura 26 – Resultado das três funções do algoritmo ADDIC aplicada ao Vértice 13. 67
Figura 27 – Resultado do algoritmo ADDIC - Comunidades encontradas. 67
Figura 28 – Problemas em aberto, baseado em (SOBIN; RAYCHOUDHURY; SAHA, 2017) 70
Figura 29 – Problemas em aberto, baseado em (HOLDER et al., 2016) 70

Figura 30 – Delineamento desta pesquisa baseado nos desafios apresentados por (SOBIN; RAYCHOUDHURY; SAHA, 2017) e (HOLDER et al., 2016) . . . 71
Figura 31 – Etapas da Pesquisa 72
Figura 32 – Visão em Grade do Ganglia - (MASSIE et al., 2012) 78
Figura 33 – Visão em Cluster do Ganglia - (MASSIE et al., 2012) 79
Figura 34 – Visão de um Nó no Ganglia - (MASSIE et al., 2012) 79
Figura 35 – Tempos do Algoritmo ADDIC - Grafos Reais. 82
Figura 36 – Tempos do Algoritmo ADDIC - grafos sintéticos 84
Figura 37 – Resultados do Algoritmo ADDIC - grafos reais - Consumo de CPU. . . . 86
Figura 38 – Resultados do Algoritmo ADDIC - grafos reais - Consumo de Memória RAM. 87
Figura 39 – Resultados do Algoritmo ADDIC - grafos sintéticos - Consumo de CPU. 88
Figura 40 – Resultados do Algoritmo ADDIC - grafos sintéticos - Consumo de Memória RAM. 89
Figura 41 – Resultado ADDIC - *Speedup* - grafos reais. 91
Figura 42 – Resultado ADDIC - *Speedup* - *grafos sintéticos*. 91
Figura 43 – Resultado ADDIC - *Eficiência* - *grafos reais*. 92
Figura 44 – Resultado ADDIC - *Eficiência* - *grafos sintéticos*. 93
Figura 45 – Comparação entre algoritmos ADDIC e sequencial - Grafos GR01 e GR02 94
Figura 46 – Comparação entre algoritmos ADDIC e sequencial - Grafos GR03 e GR04 94
Figura 47 – Comparação entre algoritmos ADDIC e sequencial - grafo sintético GS01 95

Lista de Tabelas

Tabela 1 – Descrição Detalhada do Arquivo de Entrada do Algoritmo ADDIC 60
Tabela 2 – Conjuntos de Dados - Grafos Reais 73
Tabela 3 – Conjuntos de Dados - Grafos Sintéticos 75
Tabela 4 – Grafos Reais 82
Tabela 5 – Grafos Sintéticos 83
Tabela 6 – Análise de tempos - Grafos Reais 84
Tabela 7 – Análise de tempos - grafos sintéticos 84
Tabela 8 – Comparação entre ADDIC e Algoritmo Sequencial. 94

Lista de Algoritmos

Algoritmo 1 – Algoritmo ADDIC - Fluxo Principal 56

Algoritmo 2 – Algoritmo ADDIC - Função EgoMinusEgo 57

Algoritmo 3 – Algoritmo *Label Propagation*, baseado em Raghavan, Albert e Tirupatikumara (2007) 59

Sumário

1 – Introdução 15
1.1 Motivação 18
1.2 Problema de Pesquisa 19
1.3 Objetivos 20
1.4 Contribuições 20
1.5 Organização da Tese 21

2 – Fundamentação Teórica 23
2.1 A Arquitetura *Multicore* 23
2.2 Comparação entre Soluções Paralelas e Sequenciais 25
2.3 Grandes Massas de Dados - *Big Data* 26
2.4 Redes Complexas e Grafos 27
2.5 O problema da Detecção de Comunidades 29
2.6 O problema da Detecção de Comunidades Sobrepostas 31
2.7 Rede Ego (*Ego Network*) 32
2.8 Algoritmo Label Propagation 33
2.9 O Modelo MapReduce e o Apache Hadoop 34
2.9.1 HDFS - *Hadoop Distributed File System* 36
2.10 Apache Spark 38
2.10.1 RDDs - *Resilient Distributed Datasets* 38
2.10.2 GraphX 40
2.11 Considerações Finais 40

3 – Trabalhos Relacionados 41
3.1 Modelo MapReduce e Apache Hadoop 41
3.2 Apache Spark 43
3.3 Grafos com Apache Hadoop e Spark 45
3.4 Hadoop x Spark 47
3.5 Algoritmos para Detecção de Comunidades 50
3.6 Detecção de Comunidades com Processamento Distribuído 51
3.7 Síntese e Considerações Finais 52

4 – O Algoritmo ADDIC 55
4.1 Ideia principal do algoritmo 55
4.2 Descrição do algoritmo 56
4.2.1 Fluxo principal 56

4.2.2 Fase 1: Função EgoMinusEgo . . . 57
4.2.3 Fase 2: Função Label Propagation . . . 58
4.2.4 Fase 3: Função *Merge* . . . 58
4.3 Formato dos dados de entrada . . . 59
4.4 Exemplo e fluxo de execução . . . 60
4.5 Considerações Finais . . . 68

5 – Metodologia . . . 69
5.1 Delineamento da Pesquisa . . . 69
5.2 Ambiente Computacional . . . 72
5.3 Conjunto de Dados . . . 73
5.3.1 Grafos Reais . . . 73
5.3.2 Grafos Sintéticos . . . 74
5.4 Planejamento dos experimentos . . . 75
5.4.1 Tempo de Execução . . . 76
5.4.2 Consumo de Recursos . . . 76
5.4.3 Escalabilidade . . . 77
5.4.4 Comparação com o algoritmo DEMON . . . 80
5.5 Considerações finais . . . 80

6 – Experimentos: Resultados e Análise . . . 81
6.1 Tempo de Execução . . . 81
6.1.1 Grafos Reais . . . 81
6.1.2 Grafos Sintéticos . . . 83
6.1.3 Análise dos Resultados . . . 83
6.2 Consumo de Recursos . . . 85
6.2.1 Grafos Reais . . . 85
6.2.2 Grafos Sintéticos . . . 85
6.2.3 Análise dos Resultados . . . 86
6.3 Escalabilidade . . . 90
6.4 Eficiência . . . 92
6.5 Comparação com o Algoritmo Sequencial . . . 93
6.6 Considerações Finais . . . 96

7 – Conclusão . . . 97
7.1 Trabalhos Futuros . . . 99
7.2 Considerações Finais . . . 99

Referências . . . 101

1 Introdução

Redes complexas, modeladas por grandes grafos, aparecem em diversas áreas do nosso cotidiano. Estudá-las tornou-se algo extremamente útil e importante. Redes complexas são modeladas para redes sociais (*Facebook, Twitter, Instagram*, etc), redes de distribuição (energia, água, telefone, mercadorias), redes de citações de artigos científicos, roteamento de veículos, serviços de urgência (bombeiros, ambulância, polícia), e a própria Internet (NASCIMENTO; MURTA, 2012).

Diversos fatores contribuem atualmente para que exista grande crescimento do volume dos dados gerados diariamente, o que ficou mais visível com a popularização da Internet, onde, a cada segundo, empresas, pessoas comuns, pesquisadores e órgãos governamentais geram uma avalanche de dados que passa a ser atualizada em tempo real por outras milhares de pessoas ao redor do mundo.

Outro motivo para o crescimento astronômico dos dados são os dispositivos móveis atualmente disponíveis, nos quais tornou-se muito comum o armazenamento de vídeos, fotos e músicas em meio digital. Além disso, a quantidade de arquivos que são produzidos tem aumentado consideravelmente devido à facilidade que as novas tecnologias proporcionam e também devido à redução do custo de armazenamento de dados. Por exemplo, hoje é produzido muito mais fotos que antigamente, obtém-se e visualiza-se vídeos e músicas de forma muito mais fácil, rápida e barata. Isto nos leva a crer que os números e os tamanhos dos arquivos digitais individuais estão aumentando rapidamente (ASANOVIC et al., 2009)

Aliado a tudo isso, ainda há a miniaturização do *hardware*, o aumento da capacidade dos processadores, a redução de custo das memórias, o desenvolvimento avançado do setor de telecomunicações e novas fontes e gerenciamento eficiente de energia (PACHECO; KLEIN; RIGHI, 2015).

A Internet das Coisas (IoT - *Internet of Things*) traz também uma grande mudança no cotidiano das pessoas e empresas ao permitir a presença de equipamentos e *software* inteligentes, tais como sensores dos mais diversos tipos, telefones, eletrodomésticos, automóveis, relógios e etiquetas RFID próximo às pessoas em diversos locais e prontos para interagir entre si nas mais diversificadas situações (ATZORI; IERA; MORABITO, 2010). Nesse imenso amontoado de dados, não figuram apenas os dados gerados por pessoas no seu dia-a-dia, mas também os gerados por grandes máquinas, sendo estes também responsáveis pelo crescimento da massa. Arquivos de logs, redes de sensores, arquivos de rastreamento de veículos através de GPS resultados de experimentos científicos figuram como os maiores produtores de dados (WHITE, 2015).

Por outro lado, ainda há o surgimento do conceito de Indústria 4.0 que, de fato, é tido

como outro grande gerador e consumidor da análise de todos esses dados. A Indústria 4.0 tem o objetivo de emergir o conceito de Fábrica Digital que é caracterizado pelos seguintes pontos (HOFER, 2018):

- Redes Inteligentes: sistemas e equipamentos automatizados.
- Uso de tecnologias emergentes: uso massivo de técnicas de Big Data, Inteligência Artificial e Internet das Coisas.
- Mobilidade: uso constante de dispositivos, como *smartphones* e *tablets*.
- Integração de clientes: permite a customização de produtos para a necessidade específica de um cliente.
- Modelo de negócio baseado em inovação: produtos modularizados e configurados, adaptando-se à requisitos específicos.

Uma vez que existem todos esses dados gerados, pode-se modelá-los por meio de grafos para posterior processamento em busca de informações importantes para as tomadas de decisões. Neste contexto, apresentam-se as técnicas de Mineração de Grafos, em busca de padrões de interesse em grande redes. Com o tamanho elevado dessas redes, o processo de mineração se torna muito custoso para os atuais algoritmos sequenciais conhecidos e, com isso, abrem-se algumas oportunidades de contribuição/trabalho em pontos específicos do assunto, como é o caso dos problemas em aberto sugeridos por Holder et al. (2016) e por Sobin, Raychoudhury e Saha (2017):

- Paralelização de algoritmos sequenciais;
- Aumento da eficiência da paralelização de algoritmos;
- Paralelização de algoritmos especificamente para o problema de Detecção de Comunidades e Detecção de Comunidades Sobrepostas em grandes grafos;
- Algoritmos inteligentes;
- Detecção de subgrafos;
- Algoritmos escaláveis para grafos utilizando modelos de programação emergentes.

Diante da necessidade de processar grandes grafos paralelamente e distribuidamente, surge a oportunidade de utilização de modelos de programação emergentes para esse fim. Para isso, é necessário explorar de forma eficiente todos os níveis de paralelismo e distribuição disponíveis, seja a partir dos processadores *multicore*, não deixando de explorar a arquitetura de redes integradas de alto desempenho em formato de *cluster*, até a utilização de grades de processamento, em redes como a Internet. A análise desse massivo volume de dados ajuda a descobrir muitas informações úteis que podem ser usadas como auxílio nas tomadas de decisões (SOBIN; RAYCHOUDHURY; SAHA, 2017).

A computação paralela, suportada pela arquitetura *multicore*, ressurgiu nos últimos anos como uma opção para a solução desse tipo de problema, entretanto ela requer algumas habilidades e conhecimentos que a maioria dos atuais desenvolvedores não está acostumada a lidar. Esses problemas englobam a criação, sincronização e gerenciamento

de *threads*, bloqueios (*locks*), gerenciamento de concorrência e mecanismos de tolerância a falhas (ASANOVIC et al., 2009). Para auxiliar os programadores a projetar programas paralelos considerando os aspectos da concorrência e, ao mesmo tempo, explorar a evolução contínua da capacidade dos processadores, por meio do aumento da quantidade de núcleos, surgiu o modelo MapReduce.

O MapReduce é um modelo baseado em linguagens funcionais e seu principal foco é o processamento de grandes massas de dados. Os programas projetados nesse modelo são inerentemente paralelos, permitindo o tratamento e o processamento de grandes massas de dados (*Big Data*) de forma paralela e distribuída, por desenvolvedores que não têm intimidade com programação paralela (RANGER et al., 2007), (DEAN; GHEMAWAT, 2004). O modelo MapReduce foi projetado para fornecer processamento paralelo e distribuído de uma maneira automática, escalável e tolerante a falhas (WHITE, 2015). Todo o gerenciamento do paralelismo fica a cargo do modelo e, com isso, o desenvolvedor pode focar seus esforços na solução do problema e não no controle da paralelização (DEAN; GHEMAWAT, 2004). O MapReduce modela o sistema como um *cluster* e o Apache Hadoop é a sua principal implementação livre (WHITE, 2015).

O Apache Hadoop ainda é amplamente utilizado para o processamento de grafos e de muitos outros tipos de algoritmos, entretanto pode-se encontrar problemas durante a iteratividade dos algoritmos, uma vez que a maioria das soluções para grafos tem essa característica. O Hadoop não é bom para processamento iterativo devido ao custo atribuído a leitura e gravação de todos os dados no disco que é feita a cada iteração do algoritmo (GU; LI, 2013).

Por outro lado existe a tecnologia Apache Spark que foi projetada com um eficiente mecanismo de memória *cache*, com o objetivo de alcançar melhor desempenho e tempo de resposta durante o processo iterativo. O grande diferencial do Spark é que ele mantém os dados em memória durante as iterações do algoritmo, o que evita o *overhead* durante os processos de leitura e gravação de dados no disco. Apesar de o Spark ser uma tecnologia nova quando comparada com o Hadoop, ainda assim existem muitos trabalhos na literatura que lidam com o processamento paralelo e distribuído de grandes grafos. Por meio do GraphX, o Apache Spark fornece uma API para manipulação de grafos que executa o processamento paralelamente (MALAK; EAST, 2016). A ferramenta combina as vantagens do processamento paralelo com uma forma de expressar os problemas em formato de grafos, tudo isso utilizando a plataforma Spark. A biblioteca GraphX fornece vários algoritmos embutidos, tais como PageRank, *Triangle Count*, Menor Caminho, Menor Caminho com Pesos, Componentes Conectados, Componentes Fortemente Conectados, Árvore Geradora Mínima, Caixeiro Viajante e *Label Propagation*. Além disso, o GraphX fornece classes onde o desenvolvedor pode representar os vértices e as arestas além de diversas operações que já estão embutidas nessas classes.

1.1 Motivação

Uma comunidade em grafos pode ser definida como um grupo de vértices densamente conectados os quais compartilham propriedades comuns e são esparsamente conectados ao resto dos vértices do grafo (NEWMAN; GIRVAN, 2004). Dentro de uma rede social, por exemplo, deseja-se encontrar um determinado grupo de pessoas que compartilham dos mesmos interesses. Uma vez encontrada uma comunidade, diversas ações podem ser realizadas sobre ela, como, por exemplo, uma campanha publicitária direcionada ou mesmo uma campanha de marketing com recomendação de compra de produtos. Os resultados dessas campanhas podem ser efetivos, uma vez que as pessoas que compõem a comunidade possuem um determinado interesse em comum. A Figura 1 apresenta um grafo onde três comunidades foram encontradas. Observar-se que, dentro de cada uma das comunidades, os vértices são densamente conectados entre si, entretanto eles são esparsamente conectados àqueles vértices que não fazem parte da comunidade.

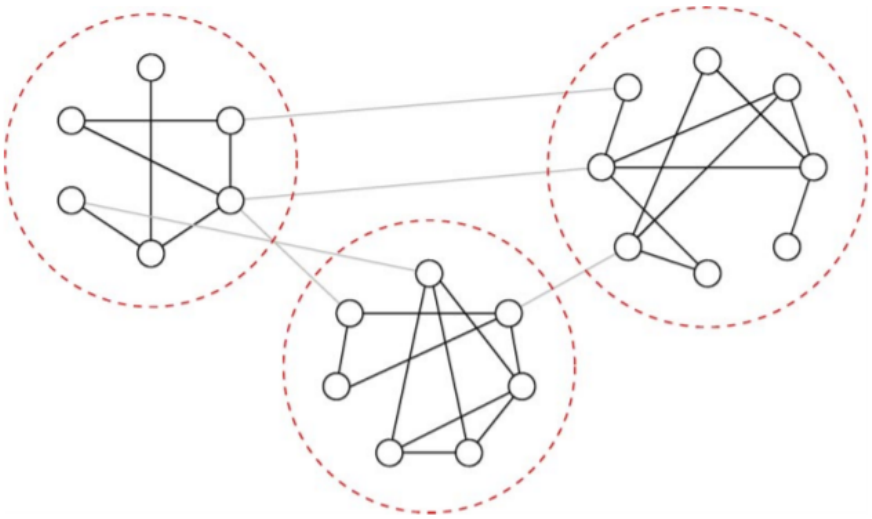


Figura 1 – Exemplo de uma pequena rede com três comunidades. As comunidades estão destacadas com os círculos pontilhados (NEWMAN, 2003).

A Detecção de Comunidades em grafos é um problema que está ligado a diversas aplicações do mundo real, tais como redes biológicas, redes sociais, melhoria de desempenho da Internet e serviços Web, recomendações de produtos em comércio eletrônico, redes *AD-HOC*, redes de citações entre autores de artigos científicos, dentre muitas outras aplicações (NASCIMENTO; SANDER; POUND, 2003).

Ainda dentro do problema de Detecção de Comunidades, existe a possibilidade e a necessidade de encontrar comunidades sobrepostas, ou seja, permitir que um determinado vértice possa fazer parte de mais de uma comunidade, pois é natural que pessoas ou objetos tenham relacionamentos com vários grupos de interesses dentro da mesma rede. A Figura 2 apresenta uma rede com quatro comunidades e dezesseis vértices, onde um desses

vértices faz parte de todas as quatro comunidades encontradas no grafo (PONTOIZEAU, 2018).

Existem vários métodos que podem ser aplicados no processo de Detecção de Comunidades, sobrepostas ou não. São alguns deles: Particionamento do Grafo (*Graph Partitioning*), Agrupamento Hierárquico (*Hierarchical Clustering*), Algoritmos Aglomerativos (*Agglomerative Algorithm*), Algoritmos Divisivos (*Divisive Algorithms*) e *Clustering Partitional* (SOBIN; RAYCHOUDHURY; SAHA, 2017).

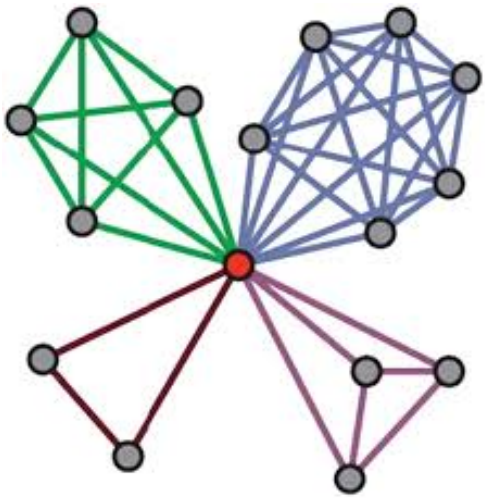


Figura 2 – Exemplo de uma pequena rede com quatro comunidades sobrepostas por um vértice.

1.2 Problema de Pesquisa

O uso de técnicas de Mineração de Grafos é uma linha de pesquisa que tem movimentado muitos pesquisadores dentro, principalmente, das áreas de Ciência da Computação, Ciências Biológicas e Ciências Sociais (BEI et al., 2016). Dentro da Mineração de Grafos, uma técnica que muito se destaca é a Detecção de Comunidades de redes complexas modeladas por meio de grafos. Esse é um problema que demanda muitos recursos computacionais, tanto de armazenamento, quanto de processamento (HOLDER et al., 2016).

Atualmente existem na literatura muitos algoritmos sequenciais para a Detecção de Comunidades e Detecção de Comunidades Sobrepostas, porém ainda existe uma grande carência da implementação desses algoritmos em um formato paralelo e distribuído, principalmente utilizando modelos emergentes de programação, para que eles possam escalar e manipular grandes quantidades de dados (SOBIN; RAYCHOUDHURY; SAHA, 2017).

Detectar comunidades em grandes estruturas de grafos é um problema *NP-Difícil* (KARP, 1972). A maioria dos algoritmos para a solução do problema de Descoberta de Comunidades é sequencial e não pode escalar em grandes proporções. Uma alternativa nesse caso é a aplicação de algoritmos de aproximação e não-exatos, porém essas técnicas se apresentam ineficientes para a demanda real existente.

Tal ineficiência dos algoritmos existentes motivam o desenvolvimento de algoritmos paralelos e distribuídos (SOBIN; RAYCHOUDHURY; SAHA, 2017). Diante disso, existe a necessidade de desenvolver e estudar modelos e ferramentas computacionais emergentes para o processamento de grandes massas de dados, tais como o MapReduce/Hadoop e o Apache Spark (HOLDER et al., 2016).

1.3 Objetivos

O objetivo central desta tese consiste em planejar, modelar e desenvolver um algoritmo paralelo e distribuído baseado no método de *Label Propagation*, no conceito de Redes Ego e no *framework* Apache Spark. Esse algoritmo, que visa a Detecção de Comunidades Sobrepostas em grandes grafos de redes complexas, deve ser escalável e eficiente. Com isso, pretende-se produzir resultados inéditos de pesquisa que levem à novas técnicas e/ou ferramentas de Detecção de Comunidades Sobrepostas.

A seguir são listados os objetivos específicos deste projeto:

- Comparar os resultados do algoritmo proposto com outra publicação da literatura, no que diz respeito a quantidade e tamanho das comunidades encontradas;
- Comparar os tempos de execução do algoritmo proposto com um algoritmo sequencial presente na literatura (COSCIA et al., 2012), utilizando vários conjuntos de dados em formato de grafos, reais e sintéticos;
- Verificar e avaliar a escalabilidade e eficiência do algoritmo proposto a medida que máquinas vão sendo adicionadas ao *cluster*, por meio da métrica de *Speedup*;
- Medir e avaliar o consumo de recursos computacionais do *cluster*, tais como: CPU e memória RAM.

1.4 Contribuições

Uma vez que é possível resolver o problema da Descoberta de Comunidades Sobrepostas em grandes grafos com o uso da computação paralela e distribuída e do modelo Apache Spark, a contribuição principal desta tese está na criação de um novo algoritmo paralelo, distribuído e escalável que tem a capacidade de encontrar essas comunidades em grandes grafos utilizando o Apache Spark.

Outras contribuições práticas proporcionadas por este trabalho para a área de mineração de dados de redes complexas utilizando grafos são:

- Nova perspectiva de aplicação de técnicas computacionais para a Descoberta de Comunidades Sobrepostas em grafos;
- Ampla revisão bibliográfica abordando o problema de Descoberta de Comunidades Sobrepostas em grafos usando o Apache Spark;
- Avaliação experimental do novo algoritmo proposto, denominado ADDIC (Algoritmo para Detecção Distribuída de Comunidades), considerando o tempo de execução;
- Comparação dos resultados de tempo de execução do algoritmo proposto (ADDIC) com outro algoritmo da literatura (DEMON).

O algoritmo ADDIC apresentou tempo de execução até 6,31 vezes melhor que o DEMON. Na grande maioria das configurações de *cluster*, o ADDIC foi melhor que o DEMON. Uma outra grande contribuição do algoritmo ADDIC é a capacidade de processar, em tempo aceitável, grafos maiores do que aqueles que o DEMON é capaz de processar. Quanto a escalabilidade, o ADDIC apresenta resultados de *Speedup* de até 97% da linearidade, o que reflete a sua boa taxa de eficiência.

1.5 Organização da Tese

Este trabalho encontra-se organizado como descrito a seguir. O Capítulo 2 apresenta a fundamentação teórica. Neste capítulo são abordados conceitos fundamentais para aplicações que realizam a Descoberta de Comunidades em grandes grafos. Na parte de programação paralela e distribuída é apresentada uma discussão sobre a revolução causada pela arquitetura *multicore* e as principais métricas utilizadas para a comparação entre soluções sequenciais e paralelas, o *Speedup* e a Eficiência. A respeito do modelo MapReduce e de sua implementação Hadoop, o Capítulo 2 discute o impacto das grandes massas de dados como motivação para o processamento paralelo e faz um apanhado sobre a estrutura do modelo MapReduce. Em seguida, o Apache Spark é apresentado, assim como sua principal API relacionada a este trabalho, que é a GraphX.

No Capítulo 3 são descritos os trabalhos relacionados. Neste capítulo, são abordados os trabalhos que utilizam o modelo MapReduce ou o Apache Spark para propor soluções em diversas áreas do conhecimento. O Capítulo 3 trata também de trabalhos referentes aos algoritmos paralelos e distribuídos para Descoberta de Comunidades Sobrepostas em grafos.

O Capítulo 4 apresenta o algoritmo ADDIC, que é o algoritmo proposto por esse trabalho para descobrir Comunidades Sobrepostas em Grandes Grafos de maneira distribuída usando o Apache Spark. O capítulo inicia-se com uma explicação detalhada sobre a estrutura do algoritmo, apresentando em seguida a ideia principal do ADDIC e também des-

crevendo como os resultados do algoritmo foram validados. O funcionamento do algoritmo é apresentar, detalhando o modo como o processamento é distribuído, aproveitando essa capacidade do Apache Spark. Em seguida um exemplo prático é utilizado para complementar a explicação sobre o funcionamento do ADDIC.

No Capítulo 5 são discutidos os experimentos. O capítulo descreve o ambiente computacional para a execução dos experimentos e, em seguida, os conjuntos de dados de entrada utilizados nos experimentos são detalhados. Os tipos de experimentos e a forma como estes serão conduzidos são descritos e justificados. O capítulo termina com as considerações finais sobre o projeto dos experimentos.

O Capítulo 6 descreve os resultados dos experimentos. O capítulo realiza o confronto entre os resultados dos algoritmos ADDIC e DEMON. Os resultados incluem análises referentes à escalabilidade ao aumentar do número de nós do *cluster* e eiciência.

Por fim, no Capítulo 6 são apresentadas as conclusões e os trabalhos futuros sugeridos.

2 Fundamentação Teórica

Neste capítulo são apresentados os fundamentos teóricos contemplados no desenvolvimento deste trabalho. Inicialmente na Seção 2.1 será realizada uma discussão acerca da arquitetura *multicore* e a revolução que ela fez na forma como as aplicações são aplicadas. Em seguida, na Seção 2.2 é apresentada a metodologia de comparação entre algoritmos paralelos e sequenciais. Após, o conceito de *Big Data* é apresentado na Seção 2.3. As redes complexas e os grafos são discutidos na Seção 2.4. Os Problemas de Detecção de Comunidades e Comunidades Sobrepostas são discutidos respectivamente nas seções 2.5 e 2.6.

Em seguida, as Seções 2.7 e 2.8 tratam respectivamente do conceito de Redes Ego e do Algoritmo Label Propagation. A Seção 2.9 trata do funcionamento do modelo MapReduce e da implementação Hadoop. A Seção 2.9.1 faz o mesmo com o sistema de arquivos distribuídos do Hadoop (HDFS). Os conceitos do Apache Spark, dos RDDs e da API GraphX são detalhadamente apresentados na Seção 2.10.

2.1 A Arquitetura *Multicore*

A arquitetura *multicore* surgiu nos últimos anos com o objetivo de proporcionar melhorias consideráveis no desempenho dos computadores. Durante os últimos vinte e cinco anos, arquitetos aproveitaram o rápido aumento da velocidade dos transistores, que a tecnologia do silício tornou possível, para duplicar a capacidade de processamento dos processadores a cada dezoito meses (ASANOVIC et al., 2009). A ideia é que enquanto fosse aumentando a quantidade de transistores e a capacidade do *hardware* de dissipar calor estivesse em níveis aceitáveis, seria mantida a programação sequencial. Essa ideia levou a diversas possibilidades de inovações que não foram tão eficientes em termos de transistores e capacidade, como por exemplo, execução fora de ordem, execução especulativa e *pipelines*, mas que aumentaram o desempenho preservando o modelo de programação sequencial. A indústria de processadores melhorou a relação custo/desempenho da computação sequencial por volta de cem bilhões de vezes nos últimos sessenta anos (ASANOVIC et al., 2009).

A programação sequencial e a ação de aumentar a quantidade e velocidade dos transistores funcionou bem até que o limite da capacidade de dissipação de calor foi atingido (ASANOVIC et al., 2009). Quando isso aconteceu e essa barreira física foi alcançada, os mesmos arquitetos se viram obrigados a encontrar uma nova maneira de sustentar a necessidade constante de aumento da capacidade. Com isso, a indústria produtora de *hardware* decidiu pelo caminho mais viável que seria a substituição de um único processador,

que já não tinha mais possibilidades de melhoria devido o alcance da barreira física, por um sistema de processamento com muitos núcleos, porém tudo no mesmo *chip*. A partir desse momento, deu-se o início a uma corrida de toda a indústria de *hardware* (microprocessadores) que declarou que o futuro seria a computação paralela, com o aumento contínuo da quantidade de núcleos dentro dos *chips*. Esse novo conceito de processador foi criado e passou a ser conhecido como processador *multicore* (BRESHEARS, 2009).

Em O'Hanlon (2006), John Hennessy, presidente da Universidade de Stanford, aponta o grande desafio do paralelismo. Segundo ele, quando se iniciou a discussão sobre o paralelismo e a facilidade de uso de computadores verdadeiramente paralelos, falava-se de um problema que era mais difícil do que qualquer outro que a Ciência da Computação já tinha enfrentado. O autor destaca e discute a grande preocupação que o paralelismo está gerando na indústria de tecnologia da informação.

Diante de todas essas mudanças, ainda há o enfrentamento de um novo problema, que é a forma como se constrói *software*. Grande parte dos programadores sempre foi acostumada à contínua melhoria da capacidade da arquitetura sequencial. Com o conceito do *multicore*, a perspectiva é outra. O maior e mais difícil objetivo da programação paralela não é apenas tornar fácil a escrita de programas eficientes, portáveis e corretos, mas sim que esses programas possam ser escaláveis de acordo com o número de núcleos por processador. Os programadores devem encontrar no ambiente paralelo a mesma facilidade que tinham ao escrever programas sequenciais (BRESHEARS, 2009), (ASANOVIC et al., 2009), (DEAN; GHEMAWAT, 2004), (DEAN; GHEMAWAT, 2010). A grande dificuldade da programação paralela está ligada diretamente à dificuldade dos programadores em lidar com tarefas típicas dos ambientes paralelos e distribuídos, tais como bloqueios (*locks*), gerenciamento de concorrência, balanceamento de carga e consistência de memória. Além disso ainda depende-se de evoluções nos sistemas operacionais e nos compiladores (ASANOVIC et al., 2009). Um grande desafio existente na área de computação paralela não é melhorar o desempenho de todo o *software* já existente, mas sim criar aplicações que realmente possam usufruir do aumento do número de núcleos e, ao mesmo tempo, executar as aplicações antigas com um desempenho aceitável. Para que a indústria de *hardware* possa continuar evoluindo satisfatoriamente, pesquisadores precisam demonstrar um grande valor para o usuário final ao aumentar o número de núcleos (ASANOVIC et al., 2009; BRESHEARS, 2009).

Nas últimas três décadas a indústria de computadores assistiu a uma grande melhoria na tecnologia de *hardware* e *software*. Mais recentemente, a demanda por mais realismo em jogos, a computação móvel e a indústria do entretenimento fomentou melhorias em *software* de tempo real, inteligência artificial, teoria de grafos e efeitos de renderização. As inovações e os avanços na comunidade científica foram alimentados pela constante

melhoria na capacidade do *hardware* nos últimos quarenta anos. Essas novas exigências atestaram a programação concorrente como a próxima grande revolução da computação.

2.2 Comparação entre Soluções Paralelas e Sequenciais

A comparação entre versões sequencial e paralela de um algoritmo deve ser realizada quando houver versões paralela e sequencial do mesmo algoritmo, o que nem sempre ocorre. Uma vez que um algoritmo foi codificado, deve-se investigar se a versão paralela do algoritmo é melhor que a versão do mesmo algoritmo implementado de forma sequencial. Uma forma trivial de realizar essa comparação é coletar o tempo gasto do início até o final da execução do algoritmo em um ambiente sequencial e em um ambiente paralelo, quando existir uma implementação do algoritmo para os dois ambientes. Se a versão paralela do algoritmo tem tempo menor que a versão sequencial, o objetivo dessa nova implementação foi alcançado. A questão é então saber o quão mais rápido é a versão paralela. Como representar essa melhoria à medida que recursos são adicionados ao processamento como, por exemplo, a troca de uma máquina por outra com mais núcleos ou mesmo a inclusão de um novo nó ao *cluster*?

Para responder a essa pergunta, deve-se aplicar a lei de Amdahl, que estabelece que o ganho de desempenho a ser obtido usando modos mais rápidos de execução é limitado pela fração de tempo em que esse modo mais rápido pode ser usado (PATTERSON; HENNESSY, 2007).

A lei de Amdahl define o *Speedup* que pode ser obtido usando uma característica particular. Suponha que seja possível fazer uma modificação em um computador que irá melhorar o desempenho quando ele é usado. O *Speedup* é a seguinte taxa:

$$S(p) = \frac{TempoSemMelhora}{TempoComMelhora}. \quad (1)$$

O *Speedup* diz o quão mais rápido uma tarefa irá executar usando um computador com alguma melhoria em relação ao computador original (PATTERSON; HENNESSY, 2007).

Suponha que uma melhoria seja a possibilidade de executar em paralelo. Uma formulação da lei de Amdahl é demonstrada abaixo, onde *pctPar* é a porcentagem de tempo de execução que irá executar em paralelo e *p* é o número de núcleos nos quais a aplicação irá executar em paralelo (BRESHEARS, 2009):

$$S(p) = \frac{1}{(1 - pctPar) + \frac{pctPar}{p}}. \quad (2)$$

Observa-se que o *Speedup* é proporcional à fração de tempo em que a melhoria pode ser utilizada e ao número de processadores que executarão essa melhoria.

A Figura 3 apresenta exemplos de curvas de *Speedup*. O *Speedup* perfeito ocorre quando o *Speedup* calculado é igual ao número de núcleos (linha sólida). Para uma aplicação ter uma boa escalabilidade, o *Speedup* deveria aumentar próximo ou na mesma proporção que novos núcleos são adicionados (linha tracejada), ou seja, se o número de núcleos é duplicado, o *Speedup* também deve aumentar na mesma proporção. Se o *Speedup* de uma aplicação não acompanha bem essa proporção (linha pontilhada), diz-se que a aplicação não escala bem (BRESHEARS, 2009).

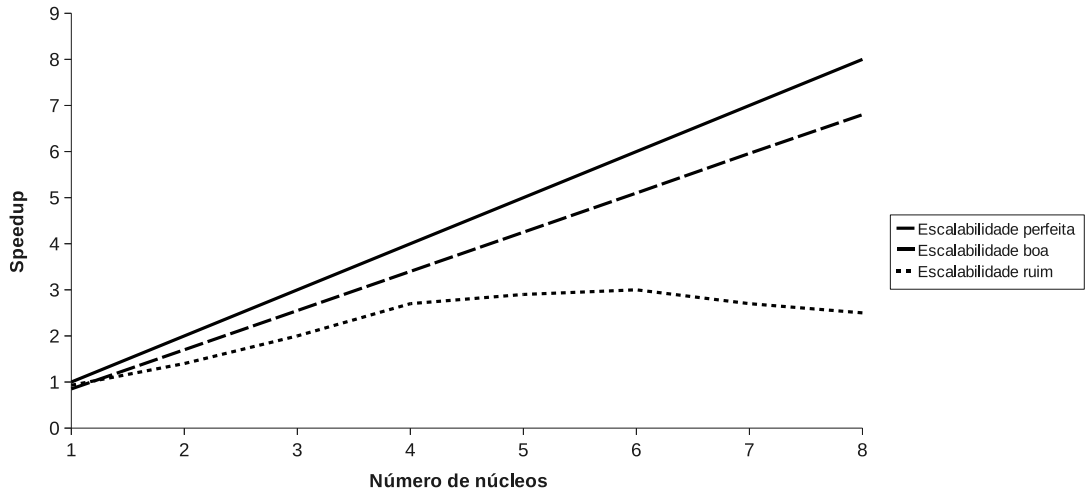


Figura 3 – Exemplo de curvas de *Speedup*, baseado em (BRESHEARS, 2009)

Outra métrica relacionada ao *Speedup* é a Eficiência. Enquanto o *Speedup* fornece uma medida para determinar quão mais rápido é uma aplicação paralela em comparação com sua implementação sequencial, a Eficiência nos diz quão bem os recursos computacionais estão sendo utilizados. Para calcular a Eficiência de uma execução paralela, deve-se dividir o *Speedup* pelo número de núcleos utilizado. Esse número é expresso em forma de porcentagem (BRESHEARS, 2009). Por exemplo se há um *Speedup* de valor 48 executando em uma máquina com 64 núcleos, a Eficiência alcançada é 75 por cento, ou seja, 48/64 = 0.75. Isso significa que, em média, durante o curso da execução, cada um dos núcleos está inativo durante 25 por cento do tempo. O cálculo da Eficiência é apresentado na Equação 3:

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)} \tag{3}$$

2.3 Grandes Massas de Dados - *Big Data*

A quantidade de dados que são produzidos atualmente tem aumentado em proporções nunca vistas e diversos fatores contribuem para que esse crescimento. O fator que

mais contribuiu foi a popularização da Internet, onde, a cada segundo, empresas, pessoas comuns, pesquisadores e órgãos governamentais geram uma grande quantidade de dados.

Outro fator que tem contribuído imensamente com o aumento da quantidade de dados gerados diariamente é a Internet das Coisas (*Internet of Things*). A Internet das Coisas é um novo paradigma que tem ganhado rapidamente o cenário das telecomunicações modernas. A ideia básica é a presença próxima a nós de uma variedade de coisas ou objetos, tais como: telefones móveis, sensores, sistemas de localização por satélite, eletrodomésticos, etc, todos eles conectados por meio de um mesmo esquema de endereçamento, com capacidade de interagir uns com os outros e cooperar com seus vizinhos para alcançar um objetivo comum, todos interligados à Internet (ATZORI; IERA; MORABITO, 2010) (GIUSTO et al., 2014).

Toda essa imensidão de dados gerada por diversos dispositivos ajuda a compreender melhor o conceito de *Big Data*. Esse termo (*Big Data*) foi definido por Katal, Wazid e Goudar (2013) como uma grande quantidade de dados que requerem novas tecnologias e arquiteturas, para que seja possível extrair valor no processo de análise desses dados. Os autores Ward e Barker (2013) conceituam *Big Data* como um termo muito usado atualmente para descrever o processo de aplicação de grande capacidade computacional - inclusive algoritmos de aprendizado de máquina e inteligência artificial - para oferecer conjuntos de informações altamente complexas.

Uma das primeiras e mais citadas definições de *Big Data* foi apresentada pelo artigo de Laney (2001). Os autores propuseram algumas propriedades do *Big Data*, que foram chamadas de Os Três V's: Volume, Variedade e Velocidade, definidos respectivamente como o aumento da quantidade dos dados gerados, os diversos tipos e formatos desses dados (texto, vídeo, fotos, etc) e o aumento da velocidade com que esses dados são gerados. No trabalho de Douglas (2012) foi incluído mais um V de Veracidade, característica que está ligada à confiança e certeza que os resultados das análises desses dados podem trazer.

Diante desta realidade, todos os dados produzidos e armazenados precisam ser, de alguma forma, um dia, processados e disponibilizados, possibilitando a extração de informações para os mais diversos tipos de usuários. Para isso, é necessário explorar de forma eficiente todos os níveis de paralelismo e distribuição disponíveis, seja através dos processadores *multicore*, não deixando de explorar a arquitetura de redes integradas de alto desempenho em formato de *cluster*, até a utilização de grades de processamento, em redes como a Internet.

2.4 Redes Complexas e Grafos

Um grafo é um conjunto de itens chamados vértices ou nós, com conexões entre eles chamadas de arestas (NEWMAN, 2003), conforme ilustra a Figura 4. Sistemas baseados

em redes complexas são modelados por meio de grafos. Alguns exemplos de redes são a Internet, as redes sociais ou alguma outra relação entre indivíduos, redes de distribuição (energia, telefone, entrega de mercadorias, etc), redes de citações entre artigos científicos e muitas outras.

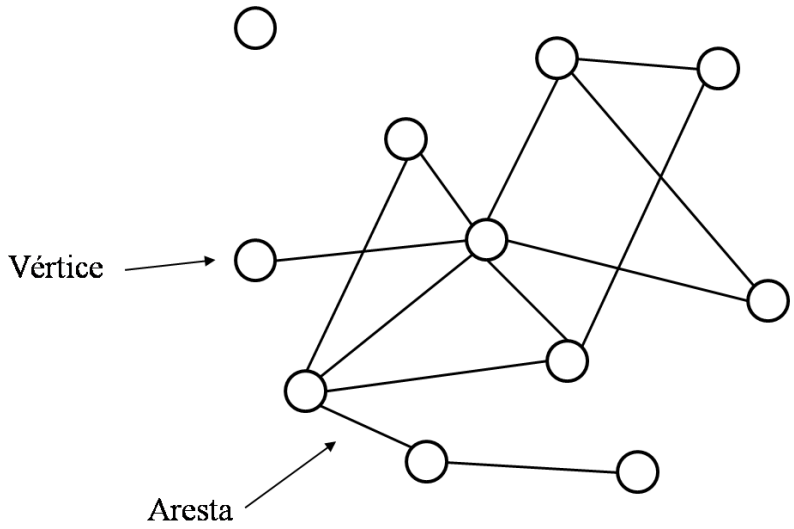


Figura 4 – Grafo com onze vértices e treze arestas

As redes foram e são amplamente estudadas ao longo dos anos dentro das ciências matemáticas e sociais, na disciplina de teoria de grafos. Modelos em rede permitem solucionar importantes problemas, tais como a centralidade de um vértice, que define se um vértice é mais conectado a outros e se possui maior influência dentro da rede, e a conectividade, que define se e como os vértices estão conectados uns aos outros por meio da rede.

Muitas situações do mundo real poderiam ser representadas através de um diagrama que consiste em um conjunto de pontos e linhas ligando pares desses pontos. Os pontos poderiam representar pessoas, com linhas ligando pares de amigos, ou os pontos poderiam ser centros de comunicação, onde as linhas representariam enlaces de comunicação (BONDY, 1976).

Estudos recentes em redes complexas têm voltado suas atenções para a análise estatística de propriedades de grandes grafos. Essa nova forma de abordar o estudo de grafos se deve à disponibilidade atual de computadores e de redes de comunicação de alta capacidade. Atualmente é possível coletar e analisar dados em escala maior, ao contrário de antigamente quando era comum o estudo de redes sociais com poucas centenas de vértices ou menos. Hoje em dia é possível encontrar redes com milhões e até bilhões de vértices.

Essa mudança na escala força uma nova abordagem analítica. A maioria dos problemas e questões que precisavam ser solucionados e respondidos em pequenas redes

não são mais tão úteis em grandes redes (NEWMAN, 2003). Uma análise em uma rede social precisa responder algumas importantes questões, por exemplo, qual vértice dessa rede traria mais impacto para a conectividade da rede se ele fosse removido. Essa remoção de vértice traria um impacto muito grande se ele estivesse em um pequeno grafo. Em contrapartida, uma questão razoável em uma grande rede com milhões e até bilhões de vértices seria que percentual de vértices precisa ser removido para afetar substancialmente a conectividade da rede.

Um grafo direcionado *G* é um par *(V, A)*, em que *V* é um conjunto finito de vértices e *A* é um conjunto de arestas com uma relação binária em *V* (ZIVIANI, 2007). Em um grafo direcionado, as arestas só podem ser percorridas em uma única direção. Em grafos direcionados podem existir arestas de um vértice para si mesmo. Essas arestas são chamadas de *self-loops*.

Um grafo não direcionado *G* é um par *(V, A)*, em que o conjunto de arestas *A* é constituído de pares de vértices não ordenados. As arestas *(u, v)* e *(v, u)* são consideradas como a mesma aresta. Essa aresta pode ser percorrida nas duas direções. Em grafos não direcionados, *self-loops* não são permitidos. Se *(u, v)* é uma aresta no grafo *G(V, A)*, o vértice *v* é adjacente ao vértice *u* e, se o grafo é não direcionado, a relação de adjacência é simétrica.

2.5 O problema da Detecção de Comunidades

A recente popularização das redes sociais (Facebook, Twitter, LinkedIn, Instagram, etc.) tem motivado fortemente a investigação de tais redes em busca de comunidades. A Figura 5 apresenta um grafo de uma rede social particionado em comunidades intuitivas. Um modelo de abstração para essas redes são os grafos, nos quais uma comunidade deveria intuitivamente descrever alguma coesão que oferece alguma densidade para um subgrafo (PONTOIZEAU, 2018).

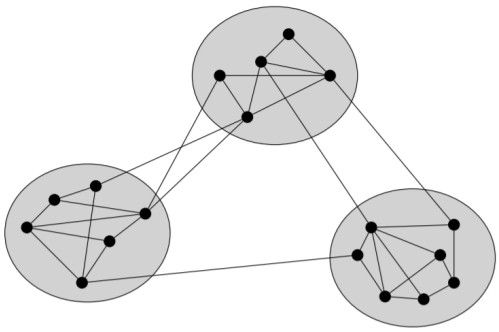


Figura 5 – Grafo de uma rede social particionado em comunidades (PONTOIZEAU, 2018)

A definição de comunidade em uma rede complexa é ambígua, pois leva a um

entendimento idêntico ao de agrupamento (*clustering*) em Estatística e Mineração de Dados e não é único. Em um ambiente complexo e semanticamente rico, como é o caso da Web moderna, uma comunidade é uma maneira de agrupar objetos de diferentes tipos por muitas diferentes razões (COSCIA et al., 2012).

De acordo com Pontoizeau (2018), existem muitas definições para comunidades em redes complexas e nenhuma pode ser considerada em absoluto. Ainda segundo o autor, a definição é escolhida dependendo de quais aspectos da comunidade desejam ser capturados.

Uma primeira forma de encontrar comunidades é procurar grupos de pessoas em que cada um dos membros conhece todos os outros membros. Na Teoria de Grafos, esse caso é conhecido como Clique. Dado um grafo $G = (V, A)$, então $G_1 = (V_1, A_1)$ denomina-se subgrafo de G se $V_1 \subseteq V e A_1 \subseteq A$, onde cada aresta de A_1 incide nos vértices de V_1 (GROSS; YELLEN, 1999). No Problema do Clique Máximo, deseja-se encontrar o subgrafo completo com o maior número de vértices em um grafo G. Esse é conhecido com um problema *NP-Difícil* (KARP, 1972).

Uma outra forma de encontrar comunidades é relaxar uma condição restritiva de um Clique, considerando os *s-cliques*. Dado um grafo $G = (V, A)$, um *s-clique* é um conjunto de vértices cujas distâncias entre cada par de vértices não é maior do que s (PONTOIZEAU, 2018).

Outra maneira de garantir a coesão entre um grupo de pessoas é verificar a quantidade de relacionamentos de cada membro do grupo. Pode-se avaliar se cada um dos vértices pertencente ao grupo possui, pelo menos, um certa quantidade de vizinhos no grupo (PONTOIZEAU, 2018). O autor Seidman (1983) criou a ideia de *k-Core*. Dado um grafo G, um subgrafo C de G é um *k-Core* se qualquer vértice de G tem no mínimo k vizinhos em C.

Pode-se expressar a coesão dentro de um subconjunto de vértices verificando a densidade dos relacionamentos dentro da comunidade, por exemplo, a densidade das arestas no subgrafo. Dado um grafo $G = (V, A)$ e um conjunto S de vértices V considerado como uma comunidade, define-se a função $f(S)$ que quantifica o quão boa é a conectividade dos vértices em S (PONTOIZEAU, 2018).

Existem diversas definições de Densidade na literatura, tais como: Densidade Interna (MANCORIDIS; MITCHELL; RORRES, 1998), Densidade Interna-Externa (MANCORIDIS; MITCHELL; RORRES, 1998), Edges Inside (RAMAN; SAURABH, 2006), Grau Médio (GOLDBERG, 1984), Fração Sobre Grau Médio (LESKOVEC; LANG; MAHONEY, 2010), Conductance (LESKOVEC et al., 2008), dentre outras.

Além das técnicas aqui apresentadas para a Detecção de Comunidades, a literatura apresenta muitas outras, tais como Lawler (1973), Delling et al. (2017), Kleinberg (2003),

Karp (1972), Ford, Jr. e Fulkerson (1956), Bazgan, Tuza e Vanderpooten (2006) e Heider (1946).

2.6 O problema da Detecção de Comunidades Sobrepostas

Em algumas situações durante o processo de Descoberta de Comunidades em grafos, pode ser que existam vértices ou pessoas que fazem parte de diversas comunidades. Classificar esse vértices como sendo de apenas uma comunidade, pode ser algo extremamente restritivo para o resultado final da análise. É natural que as pessoas tenham relacionamentos com diversos grupos de interesses. Assim, dado um grafo $G = (V, A)$, pode-se definir um conjunto $\{S_i\}_{i=1}^{p}$ como uma partição sobreposta de subconjuntos de V, tal que $\cup_{i=1}^{p} S_i = V$ (PONTOIZEAU, 2018).

Uma forma de definir uma partição sobreposta dentro de comunidades é procurar por partições sobrepostas dentro de cliques, técnica chamada de *Clique Cover*.

A Figura 6 apresenta um exemplo de uma comunidade sobreposta com quatro cliques.

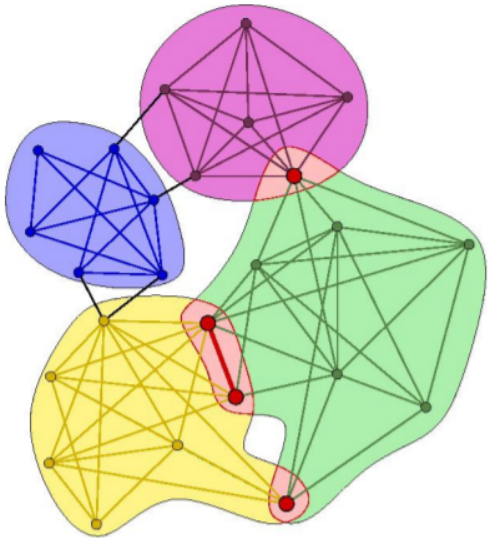


Figura 6 – Exemplo de uma comunidade sobreposta com quatro cliques (PONTOIZEAU, 2018)

Uma outra forma de descobrir comunidades sobrepostas é trabalhar com a Ego Network. Esse conceito será tratado no próximo tópico, porém é a forma como trabalha o principal o principal trabalho motivador dessa tese (COSCIA et al., 2012). Além desse trabalho, outros utilizam a mesma técnica, como é o caso do algoritmo EgoLP (BUZUN et al., 2014) e Epasto, Lattanzi e Leme (2017).

Existem muitas aplicações que podem se beneficiar da Descoberta de Comunidades Sobrepostas, tais como a identificação de comunidades envolvidas na metástase de casos

de câncer (JONSSON; BATES, 2006) e (JONSSON et al., 2006) e diversos estudos na área de redes sociais (DONG et al., 2018).

De acordo com Sobin, Raychoudhury e Saha (2017), existem diversos métodos para detecção de comunidades, conforme destacado abaixo:

- *Graph Partitioning*: processo de dividir os vértices do grafo em um número igual de partições de modo que a interconexão das arestas é minimizada. Trata-se de uma importante tarefa em algoritmos do tipo "dividir para conquistar", algoritmos paralelos, distribuídos, etc.
- *Hierarquical Clustering*: consiste em identificar grupos de vértices com alta similaridade e não requer um pré-conhecimento do número de grupos. Pode ser classificado como: *Agglomerative Algoritms* ou *Divisive Algorithms*.
- *Partitional Clustering*: o número de grupos é pré-determinado. São atribuídos como pontos e a distância entre os pontos determina a medida de dissimilaridade entre os grupos.

Embora existam na literatura muitos bons algoritmos utilizando o método de *Graph Partitioning*, apenas alguns poucos são paralelos e distribuídos para a tarefa de detecção de comunidades (SOBIN; RAYCHOUDHURY; SAHA, 2017).

2.7 Rede Ego (*Ego Network*)

Uma Rede Ego é uma sub-rede, cujo foco está em um ator individual (vértice) que é conhecido como o sujeito da rede. O ponto-focal da rede é chamado de ”ego” e os outros atores que o ego interage e que estão conectados são chamados de "alters". Em uma Rede Ego, apenas atores que são diretamente conectados ao ego formam a rede (FARRUGIA; HURLEY; QUIGLEY, 2011).

A Figura 7 apresenta um exemplo de um grafo com dez vértices e dezessete arestas. A Rede Ego exemplificada na figura diz respeito ao Vértice 7. Nesse caso, o Vértice 7 é o Ego e os vértices 1, 2, 3, 4, 5 e 6 são Alters. Os vértices 8, 9 e 10 não fazem parte da Rede Ego do Vértice 7.

As Redes Ego podem ser usadas para os mais diversos objetivos, tais como as relações entre indivíduos, acessos de pessoas a recursos, importância de vértices (indivíduos) dentro de uma rede, potencial de difusão, composição de comunidades (características demográficas), natureza de amizades em redes sociais, etc.

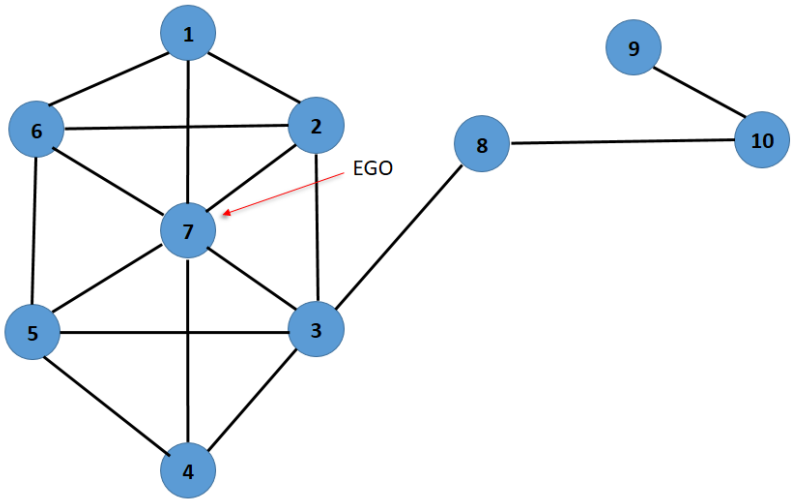


Figura 7 – Ego Network do Vértice 7 com dez vértices e dezessete arestas

2.8 Algoritmo Label Propagation

O Label Propagation é um algoritmo que fornece uma boa combinação de propriedades para o uso na Descoberta de Comunidades, que são: alta acurácia em comunidades disjuntas ou sobrepostas, baixa complexidade computacional, facilidade de implementação utilizando paradigmas modernos para processamento distribuído e escalabilidade (BUZUN et al., 2014).

A ideia por trás do algoritmo é a seguinte: suponha que um vértice v tem vizinhos $x_1, x_2, ..., x_n$ e que cada um desses vizinhos carrega um rótulo descrevendo a comunidade a que esse vértice pertence. Assim, x determinará a sua comunidade baseado nos rótulos de seus vizinhos. Cada vértice da rede escolhe se juntar a comunidade em que a maioria de seus vizinhos pertence (RAGHAVAN; ALBERT; TIRUPATIKUMARA, 2007).

A Figura 8 apresenta um exemplo de iterações do algoritmo *Label Propagation*. Observar-se que, a cada iteração, os vértices vão tendo os seus rótulos de comunidades modificados. Ao final todos possuem o mesmo rótulo a.

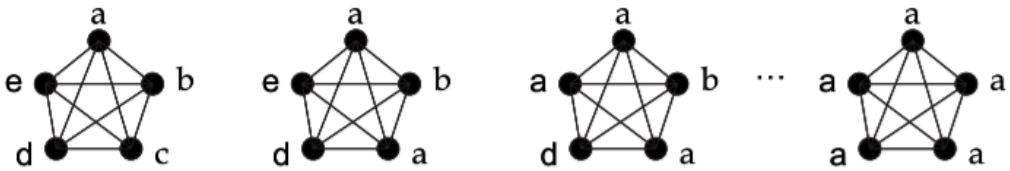


Figura 8 – Exemplo de funcionamento do algoritmo Label Propagation (RAGHAVAN; ALBERT; TIRUPATIKUMARA, 2007)

Esse processo vai acontecendo iterativamente e, a cada passo, cada um dos vértices atualiza seus rótulos baseado nos rótulos de seus vizinhos. O processo de atualização pode

ocorrer síncrona ou assincronamente. O vértice v na t^o iteração atualiza seu rótulo baseado nos rótulos de seus vizinhos na iteração $t-1$. Portanto, $C_v(t) = f(C_{v1}(t-1)), f(C_{v2}(t-1))..., f(C_{vk}(t-1))$, onde $C_v(t)$ é o rótulo do vértice v no tempo t. O processo continua até que a rede não apresente mais mudanças nos rótulos dos vértices ou até o número máximo de iterações ser atingido (RAGHAVAN; ALBERT; TIRUPATIKUMARA, 2007).

O algoritmo Label Propagation não requer informações de iterações anteriores, tais como o número ou o tamanho de comunidades encontradas. Trata-se de um algoritmo simples e de um processo iterativo eficiente (SOBIN; RAYCHOUDHURY; SAHA, 2017).

2.9 O Modelo MapReduce e o Apache Hadoop

O MapReduce é um modelo de programação paralela inspirado em linguagens funcionais e tem como alvo o processamento de grandes massas de dados. Os programas projetados nesse modelo são inerentemente paralelos, permitindo o tratamento e ou processamento de grandes massas de dados de forma paralela e distribuída. Os programas podem ser elaborados por desenvolvedores que não têm intimidade com programação paralela (DEAN; GHEMAWAT, 2004) (RANGER et al., 2007).

O modelo MapReduce trabalha primeiramente dividindo o processamento em duas fases: a primeira fase é a Map e a segunda é a Reduce. O formato dos dados de entrada é específico para cada aplicação e é definido pelo programador. O programador especifica um algoritmo usando duas funções: Map e Reduce. A função Map é aplicada aos dados de entrada e produz uma lista de pares intermediários <chave, valor>. A função Reduce é aplicada sobre todos os pares intermediários que possuem a mesma chave. Ela tipicamente executa uma operação para mesclar os dados que possuem a mesma chave e produzir os pares de saída. Finalmente, os pares de saída são ordenados com base em suas chaves. Na forma mais simples de criar programas MapReduce, o programador fornece apenas a função Map. Todas as outras funcionalidades, incluindo o agrupamento de pares intermediários que possuem a mesma chave e a ordenação final, são fornecidas pelo modelo.

O principal benefício desse modelo é a simplicidade. O modelo permite ao programador desenvolver um algoritmo focando na funcionalidade, e não na paralelização, que é a grande dificuldade dos programadores não familiarizados com a programação paralela. A parte de paralelização e os detalhes da concorrência são providos pelo *framework*.

Diversas implementações de MapReduce são conhecidas e novas implementações também podem ser feitas, ficando a escolha atrelada ao ambiente de execução e ao problema envolvido (WHITE, 2009). Uma implementação pode ser adequada para uma pequena máquina de memória compartilhada ou para uma grande máquina com multiprocessadores ou ainda uma grande rede de computadores.

As chamadas à função Map são distribuídas automaticamente em diversas máquinas particionando os dados de entrada em M partições. As partições de entrada podem ser processadas por diferentes máquinas em paralelo. As chamadas à função Reduce são distribuídas particionando-se as chaves intermediárias em R pedaços, usando uma função de particionamento (por exemplo, *hash(key) mod R*). O número de partições (R) e a função de particionamento são especificados pelo usuário.

A biblioteca MapReduce primeiramente divide os arquivos de entrada em M partes, frequentemente de tamanho 64 Megabytes (MB) por parte. Esse valor é controlado pelo usuário por meio de um parâmetro opcional e o valor *default* é 64 Megabytes. Esse processo inicia diversas cópias do programa em um *cluster* de máquinas. Uma das cópias do programa é especial - a *master*, que divide o trabalho em M tarefas Map e R tarefas Reduce. O resto das cópias são *workers* que recebem trabalhos atribuídos pelo *master*. O *master* escolhe *workers* inativos e atribui a cada um deles uma tarefa Map ou uma tarefa Reduce. O *worker*, a quem o *master* atribuiu uma tarefa Map, lê o conteúdo da partição de entrada, analisa o par chave/valor de cada entrada e passa cada par para a função Map que o usuário definiu. O par intermediário de chave/valor produzido pela função Map é armazenado em *buffer*. O tamanho desse *buffer* também é controlado pelo usuário por meio de um parâmetro opcional. Periodicamente, os pares armazenados em *buffer* são escritos no disco local, particionado em R regiões pela função de particionamento. A localização desses pares em *buffer* no disco local é passada de volta ao *master*, que é responsável por encaminhar essas localizações para os *workers* Reduce. Quando o *worker* Reduce é notificado pelo *master* sobre essas localizações, ele usa a chamada a um procedimento remoto para ler os dados em *buffer* do disco local dos *workers* Map. Quando o *worker* Reduce ler todos os dados intermediários, ele ordena esses dados pelas chaves intermediárias, de modo que as mesmas chaves são agrupadas. A ordenação é necessária porque frequentemente diferentes chaves Map têm a mesma tarefa Reduce. Caso a quantidade de dados intermediários seja grande para caber na memória, uma ordenação externa é usada. O *worker* Reduce interage sobre os dados intermediários ordenados. Para cada chave intermediária única encontrada, o *worker* passa para a função Reduce definida pelo usuário a chave intermediária e o conjunto de valores intermediários correspondentes à essa chave. Quando todas as tarefas Map e Reduce foram completadas, o *master* retorna ao programa que o chamou e que estava em modo de espera. Depois da execução com sucesso, a saída do MapReduce fica disponível em R arquivos de saída, um para cada tarefa Reduce, com nomes especificados pelo usuário. Os usuários não precisam combinar esses R arquivos dentro de um único arquivo, pois eles são frequentemente passados como entrada para outra chamada MapReduce e assim sucessivamente até que uma situação de parada, definida pelo usuário, seja alcançada.

A Figura 9 ilustra o funcionamento do modelo MapReduce.

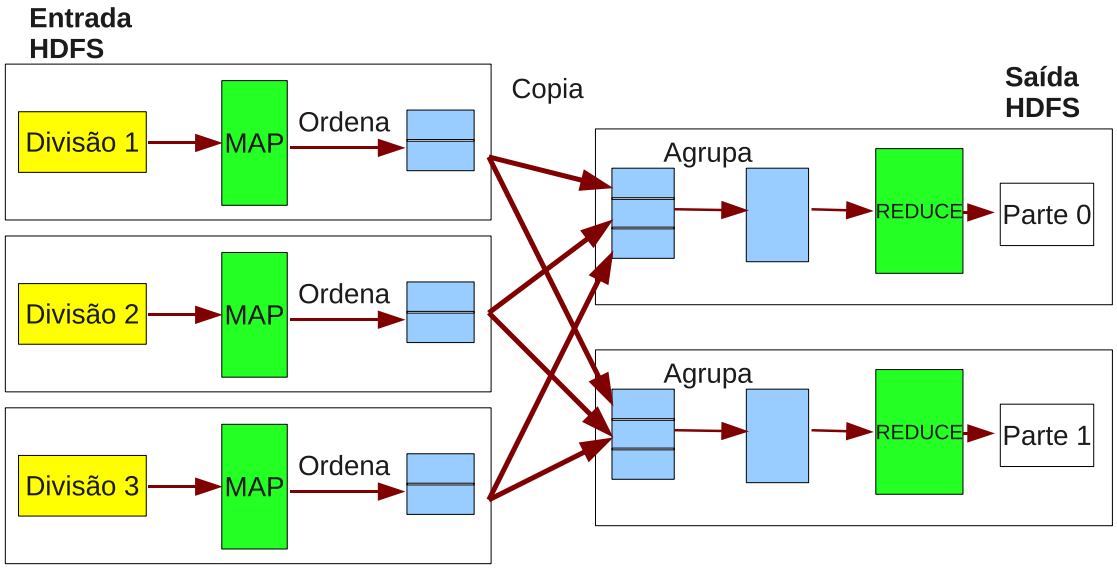


Figura 9 – Funcionamento do Modelo MapReduce baseado em White (2009)

O Apache Hadoop é a implementação do modelo MapReduce mais conhecida e foi criado por Doug Cutting, criador do Apache Lucene, que é uma biblioteca de busca textual amplamente usada (WHITE, 2009) (DEAN; GHEMAWAT, 2004). O Hadoop teve sua origem baseada no Apache Nutch, um mecanismo de busca na Internet com código fonte aberto.

2.9.1 HDFS - *Hadoop Distributed File System*

Quando uma massa de dados ultrapassa a quantidade disponível no disco de uma única máquina, esses dados precisam ser particionados e armazenados em outras máquinas, o sistema responsável por realizar este tipo de operação é chamado de sistema de arquivos distribuído. Esse sistema é mais complexo que o tradicional sistema de arquivos devido à necessidade de tratar complexidades da rede e possíveis falhas, sem a perda de dados.

O *Hadoop Distributed File System* (HDFS) é um sistema de arquivos distribuídos projetado para armazenar grandes quantidades de dados e possui acesso aos arquivos por meio de padrões de *streaming*, executando em grandes *clusters*. Os dados que são armazenados no sistema podem possuir centenas de gigabytes e até terabytes de tamanho e existem *clusters* Hadoop executando atualmente com petabytes de dados. O HDFS foi construído levando em consideração a ideia de que o padrão de processamento de dados mais eficiente é o "escreva uma vez e leia várias vezes"(*write-once, read-many-times*). Além disso, o HDFS não requer um *cluster* grande e caro, pois ele foi projetado para executar em diversas configurações de *clusters*, inclusive composto de nós com fabricantes ou configurações diferentes (BORTHAKUR, 2008), (WHITE, 2009).

Apesar de possuir algumas semelhanças com outros sistemas de arquivos distribuídos, o HDFS apresenta algumas características que o torna diferente dos demais, pois implementa um alto grau de tolerância a falhas. Ele também foi projetado para ser implantado em conjuntos de *hardware* de baixo custo. Além disso, o HDFS fornece acesso eficiente aos dados das aplicações e é adequado a trabalhar com aplicações que manipulam grandes massas de dados (*Big Data*) (KARUN; CHITHARANJAN, 2013).

O projeto do HDFS foi baseado na arquitetura mestre/escravo (*master/slave*), na qual um nó do *cluster* faz o papel de servidor mestre (*NameNode*), o qual gerencia o sistema de arquivos, regula o acesso que é feito pelos clientes (*slaves*) aos dados e armazena os metadados dos arquivos (nomes, localização, dados de *checksum*, etc). Os *slaves* são os *DataNodes* que normalmente são representados por um nó do *cluster* e são responsáveis por gerenciar os arquivos que lhe são confiados pelo *master*. Todos esses servidores são amplamente conectados e suas comunicações são baseadas no protocolo TCP (SHVACHKO et al., 2010).

Quando um arquivo é enviado para o HDFS, este poderá ser dividido em um ou mais blocos e estes serão armazenados em um conjunto de *DataNodes* (*slaves*). As operações básicas do HDFS, como abrir, fechar e renomear arquivos e diretórios, são realizadas pelo *NameNode*. Os *DataNodes* são responsáveis por atender as requisições de leitura e gravação que são solicitadas pelas aplicações cliente. As operações de exclusão, criação de blocos e replicação dos dados são dadas aos *DataNodes* pelo *Namenode*.

O HDFS é altamente tolerante a falhas e projetado para funcionar em diversas configurações de *clusters*, tanto homogêneos (nós idênticos) quanto heterogêneos (nós com configurações diferentes) e possui ainda grande capacidade de lidar com grandes quantidades de dados (*datasets* grandes). A granularidade dos arquivos é outra propriedade a qual o HDFS está adequado a trabalhar, ou seja, ele pode manipular grandes quantidades de pequenos arquivos (alta granularidade) ou pequenas quantidades de grandes arquivos (baixa granularidade).

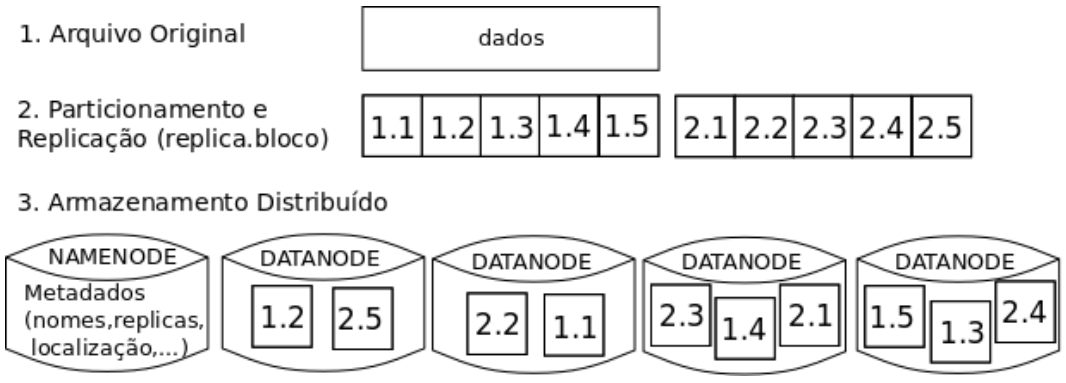


Figura 10 – Armazenamento Distribuído no HDFS

A Figura 10 apresenta o funcionamento do HDFS. No Passo 1 um arquivo é dado

como entrada para o armazenamento. O *Namenode* realizará o particionamento desse arquivo em diversos blocos (Passo 2) e esses blocos serão distribuídos para os *DataNodes* (Passo 3). É importante destacar que o *Namenode* armazenará informações relativas ao Metadados dos Arquivos, ou seja, em quais *DataNodes* determinados blocos de arquivos estão, seus nomes, sua replicação, etc.

O HDFS vem sendo amplamente utilizado pela ferramenta Hadoop desde o seu surgimento e em diversos tipos de processamento, entretanto, é importante destacar que outras ferramentas mais novas, tais como o Apache Spark, o Apache Flink e outras, também utilizam esse mesmo sistema de arquivos, provando que ele é uma ferramenta importante e já consolidada no ecossistema Apache e no mundo do processamento paralelo e distribuído de grandes quantidades de dados (*Big Data*).

2.10 Apache Spark

O Apache Spark é uma plataforma rápida e de propósito geral projetada para o processamento em *cluster*. No lado da velocidade, o Spark estende de maneira eficiente o modelo MapReduce, incluindo suporte para outros tipos de processamento, tais como consultas interativas e processamento usando *streaming*. Uma das principais características oferecidas pelo Spark, no que diz respeito a velocidade, é a habilidade da ferramenta de processar os dados em memória, entretanto, o sistema é também mais eficiente que o MapReduce para aplicações complexas executando no disco (KARAU et al., 2015).

O Spark foi projetado para ser altamente acessível, oferecendo API's simples em Python, Java, Scala e SQL, além de possuir várias bibliotecas embutidas. Além disso, o Spark pode executar em *clusters* Hadoop e acessar qualquer fonte de dados do Hadoop, inclusive o HDFS.

O Apache Spark possui em seu ecossistema múltiplos sistemas amplamente conectados e integrados. No seu núcleo encontra-se a sua *engine* computacional que é responsável pelo agendamento, distribuição e monitoramento das aplicações. Devido à sua capacidade de ser rápido e de propósito geral, o Spark possui alguns componentes especializados para processar diferentes cargas de trabalho, tais como aprendizado de máquina, grafos e SQL. Esses componentes foram projetados para interoperar de maneira próxima, combinando esses componentes como bibliotecas dentro do mesmo projeto de software, conforme apresentado na Figura 11.

2.10.1 RDDs - *Resilient Distributed Datasets*

Com o objetivo de melhorar a principal deficiência da modelo MapReduce, que é a gravação dos dados em disco após cada iteração do algoritmo, os desenvolvedores do Spark criaram o conceito de RDD *Resilient Distributed Dataset*. Um RDD é uma coleção

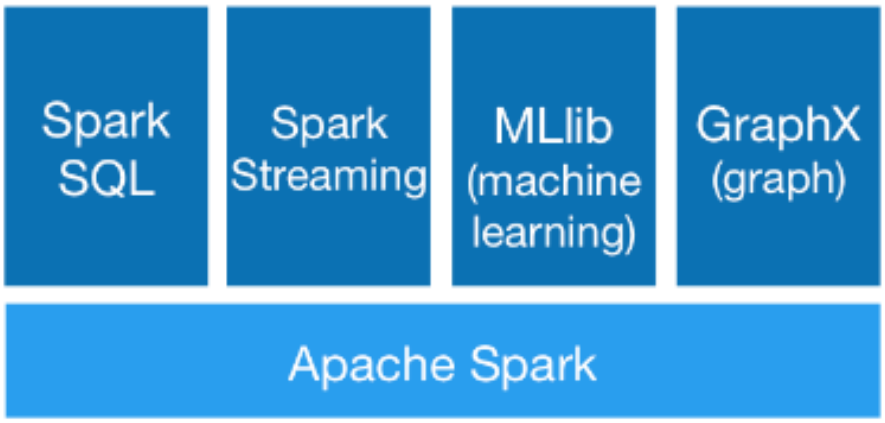


Figura 11 – O Ecossistema Apache Spark

de registros distribuídos, particionados e somente-leitura, que podem ser criados apenas por meio de determinadas operações: (i) dados em armazenamento estável ou (ii) outros RDDs. Essas operações são chamadas de Transformações. Alguns tipos de operações de transformação são: *Map, Filter e Join*. No Spark todo o trabalho é modelado criando novos RDDs, transformando RDDs existentes ou chamando operações dos RDDs para processar um resultado. Os dados contidos nos RDDs são automaticamente distribuídos pelo Spark através do *cluster* e as operações que serão executadas são automaticamente paralelizadas (ZAHARIA et al., 2012).

O RDD é uma coleção de objetos distribuídos e imutáveis. Cada RDD é dividido em mútiplas partições, que podem ser processadas por diferentes nós do *cluster*. Além disso, um RDD pode conter qualquer tipo de objeto do Python, Java ou Scala, incluindo tipos definidos pelo usuário. Os usuários podem criar seus RDDs de duas formas, a saber: (i) carregando um conjunto de dados externo ou (ii) distribuindo uma coleção de objetos em seu próprio programa (gerando os dados) (KARAU et al., 2015).

Uma vez criados, os RDDs oferecem as operações de Transformação e Ação. A Transformação cria um novo RDD a partir de um anterior. Por outro lado, a Ação processa um resultado baseado em um RDD e retorna os dados computados para o programa que realizou a chamada ou salva os dados em um armazenamento externo (Ex.: HDFS).

Os RDDs foi a forma que os desenvolvedores do Spark encontraram para manter os dados em memória (*cache*) durante as várias iterações de um algoritmo iterativo, evitando o *overhead* presente nas diversas leituras e gravações em disco e de uma maneira totalmente tolerante a falhas. Quando uma determinada partição do RDD é perdida em um determinado nó do *cluster*, o trabalho não é totalmente perdido, pois basta recuperar apenas aquela partição que estava no nó perdido e o trabalho pode ser retomado a partir do ponto onde ocorreu a falha. Essa recuperação pode acontecer em paralelo em diferentes nós (ZAHARIA et al., 2012).

2.10.2 GraphX

O GraphX é o Spark aplicado a grafos, ou seja, uma biblioteca para manipulação de grafos que executa o processamento de grafos paralelamente (MALAK; EAST, 2016). A ferramenta combina as vantagens do processamento paralelo com uma forma de expressar os problemas em formato de grafos, tudo isso utilizando a plataforma Spark. O Spark foi baseado em um *framework* chamado Pregel (MALEWICZ et al., 2010), que era uma alternativa à deficiência do modelo MapReduce em processar algoritmos iterativos, que é uma característica presente na maioria dos algoritmos para grafos.

A biblioteca GraphX fornece vários algoritmos embutidos, tais como PageRank, *Triangle Count*, Menor Caminho, Menor Caminho com Pesos, Componentes Conectados, Componentes Fortemente Conectados, Árvore Geradora Mínima, Caixeiro Viajante e *Label Propagation*. Além disso, o GraphX fornece classes nas quais o desenvolvedor pode representar os vértices e as arestas além de diversas operações que já estão embutidas nessas classes, para realizar as transformações nos grafos. A biblioteca fornece também geradores determinísticos e aleatórios.

2.11 Considerações Finais

Nesse capítulo foram apresentados vários conceitos sobre a arquitetura *multicore*, *Big Data*, ferramentas para processamento distribuído de grandes massas de dados, uma introdução à Teoria de Grafos e uma discussão sobre o problema de Detecção de Comunidades Sobrepostas.

O próximo capítulo apresentará detalhadamente os trabalhos relacionados e o estado da arte sobre o assunto de algoritmos paralelos e distribuídos para detecção de comunidades.

3 Trabalhos Relacionados

Neste capítulo são apresentados alguns dos principais trabalhos presentes na literatura e relacionados a esta tese. Esses trabalhos contribuíram diretamente na construção de algoritmos sequenciais e paralelos para a Detecção de Comunidades e Comunidades Sobrepostas e, durante a realização de um estudo sobre os resultados encontrados neles, pode-se justificar o desenvolvimento desse projeto de tese e apontar claramente suas contribuições no cenário atual.

Este capítulo está estruturado conforme descrito a seguir. A Seção 3.1 apresenta trabalhos recentes que utilizaram o modelo MapReduce e a tecnologia Apache Hadoop, assim como a Seção 3.2 apresenta os mais recentes trabalhos envolvendo a tecnologia Apache Spark.

A Seção 3.3 apresenta os trabalhos mais relevantes que utilizaram as tecnologias Hadoop e Spark para a implementação de algoritmos que manipulam grandes grafos e a Seção 3.4 aponta os principais e mais recentes trabalhos que realizaram comparações entre os dois *frameworks* (Hadoop e Spark).

A Seção 3.5 traz as principais técnicas, presentes em outros trabalhos da literatura, para Detecção de Comunidades e Comunidades Sobrepostas modeladas por grafos e, por fim, a Seção 3.6 apresenta os principais trabalhos que utilizaram *frameworks* paralelos e distribuídos para esse fim.

3.1 Modelo MapReduce e Apache Hadoop

Desde a primeira publicação do modelo MapReduce feita por Dean e Ghemawat (2004), diversos outros trabalhos vêm sendo encontrados na literatura aplicados às mais diversas áreas do conhecimento. Nesta seção será apresentado desses artigos.

Na área da bioinformática existem diversas referências bastante citadas, como é o caso do artigo de Taylor (2010) que fez um apanhado dos principais trabalhos que utilizaram o modelo MapReduce para a solução de problemas dessa área. Outros trabalhos seguem a mesma linha, como é o caso de Zou et al. (2014) que classifica o modelo MapReduce como "a nova geração de ferramentas para sequenciamento de domínios biológicos". Os autores O'Driscoll, Daugelaite e Sleator (2013) proporcionaram em seu artigo uma ampla discussão acerca dos novos desafios da bioinformática e as possíveis contribuições que o modelo MapReduce poderá fornecer. Outro trabalho da literatura muito importante e bastante referenciado é Schatz (2009) que criou o CloudBurst, um algoritmo otimizado para Hadoop que mapeia sequências de dados do genoma humano e outros genomas, para uso em uma

ampla variedade de análises biológicas. Ainda na área de bioinformática, pode-se encontrar trabalhos recentes, como é o caso de Siretskiy et al. (2015), que implementou algoritmos para solução de problemas que envolvem sequenciamento de DNA, mais especificamente NGS (*Next Generation Sequencing*) e alcançou bons resultados, no que diz respeito às medidas de escalabilidade, eficiência e tempo de processamento, quando comparado aos resultados de tecnologias tradicionais.

Outra área em que é possível encontrar muitos trabalhos utilizando o Hadoop é a área de *machine learning*. O recente trabalho de Landset et al. (2015) realizou um comparativo entre as diversas bibliotecas de *machine learning* presentes em *frameworks* paralelos e distribuídos e o Hadoop foi representado pela biblioteca Apache Mahout[1]. Foram utilizados algoritmos de classificação, clusterização, *Collaborative Filtering* e avaliados os critérios de facilidade de uso, extensibilidade e escalabilidade. Ao final os autores concluíram que cada uma das bibliotecas de *machine learning* tem a sua vantagem, dependendo do problema ao qual ela vai ser aplicada e a base de dados que será utilizada. O trabalho de Rosen et al. (2013) lida com uma questão muito importante do Apache Hadoop e muito relevante para essa tese, que é a implementação de algoritmos iterativos. Os autores propuseram uma extensão do modelo MapReduce para algoritmos iterativos de *machine learning*, chamado Iterative MapReduce.

O Hadoop também é amplamente utilizado para o processamento de grandes massas de dados científicos. Um exemplo disso é o trabalho de Zhang et al. (2015) que criou o ParSA *(Parallel Scientific Data Analysis)*, uma ferramenta para processamento de dados científicos escalável e que utiliza o Sistema de Arquivos Distribuídos do Hadoop (HDFS). O ParSA implementa algoritmos para acelerar as análises de modelos do clima e os resultados experimentais apresentaram alta eficiência e escalabilidade. Na mesma linha existe ainda o BioPig (NORDBERG et al., 2013), construído sobre o modelo MapReduce e sobre os *framewoks* Hadoop e Apache Pig[2]. O BioPig apresenta 3 vantagens sobre os algoritmos MPI: i) simplicidade na programação, devido às funcionalidades dos frameworks; ii) escalabilidade; iii) portabilidade sem modificações para qualquer infraestrutura Hadoop.

Mesmo com as várias novas tecnologias de *frameworks* para processamento paralelo e distribuído de grandes massas de dados disponíveis atualmente, como por exemplo o Apache Flink[3] e o Apache Spark[4], o Hadoop ainda tem sido muito utilizado como aliado à tomada de decisão em muitas outras áreas do conhecimento, conforme apresentam esses trabalhos: processamento de *Big Data* para análise de riscos (YAO; VARGHESE; RAU-CHAPLIN, 2013), sistemas de recomendação (XU et al., 2014), simulações em larga escala (LEE et al., 2016), sistemas de monitoramento e cidades inteligentes (XIA et al.,

[1] Apache Mahout - http://mahout.apache.org/
[2] Apache Pig - https://pig.apache.org/
[3] Apache Flink - https://flink.apache.org/
[4] Apache Spark - http://spark.apache.org/

2016) e (KHAN et al., 2015), análises de grandes massas de *logs* (MEHTA et al., 2015), análise de sentimento (KAUSHIK; MISHRA, 2014), coleta de experiências em redes sociais e processamento de massas de dados referentes a doenças e cuidados com a saúde (RASTEGAR-MOJARAD et al., 2015) e (MOHAMMED; FAR; NAUGLER, 2014), dentre muitos outros.

3.2 Apache Spark

O Apache Spark vem sendo utilizado atualmente como opção de solução para diversos problemas que envolvem *Big Data* e pode-se encontrar na literatura muitos trabalhos que utilizaram a ferramenta para a solução de variados problemas em muitas áreas do conhecimento.

O artigo de Meng et al. (2015) apresentou o MLLib, uma biblioteca Spark para *machine learning*. No artigo os autores descreveram detalhadamente a ferramenta e realizaram diversos experimentos comparando duas versões do MLLib (Versões 1.1 e 1.3) e do Mahout (biblioteca de *machine learning* para Hadoop). Nos experimentos foram utilizados os algoritmos de *Logistic Regression, Ridge Regression, Decision Trees, ALS e K-Means* para avaliar as bibliotecas e, em todos os resultados as duas versões do MLLib foram mais rápidas que o Mahout. É importante destacar que houve uma importante melhoria no tempo na comparação entre as versões 1.1 e 1.3 da MLLib, tendo a versão mais recente apresentado os melhores resultados.

Uma coleta de dados da rede social Twitter[5] foi realizada e processada no trabalho de Shoro e Soomro (2015). A coleta de dados e o processamento foram para simplesmente listar as palavras mais postadas na rede social, entretanto o objetivo real do trabalho foi mostrar os passos necessários para realizar essa tarefa e apresentar o comportamento da ferramenta Spark. Foram analisados aproximadamente 23.300 Tweets durante 60 segundos e apresentadas as 10 palavras mais postadas e também seu idioma de origem. O trabalho foi concluído com uma discussão acerca da aplicabilidade da ferramenta Spark nessas duas tarefas. Os autores concluíram que ela é adequada para as tarefas, entretanto, apresenta dificuldades de configuração e operação devido às dificuldades de depuração.

A ferramenta SparkCL, apresentada no trabalho de Segal et al. (2015), é um *framework* de programação unificada, *open-source* e baseado em Java, OpenCL (*Open Computing Language*) e no Apache Spark. O *framework* permite que um único código-base direcione e distribua os trabalhos para qualquer tipo de núcleo de computação que suporte OpenCL, possibilitando uma integração mais fácil entre novos tipos de núcleos dentro de um *cluster* Spark, aproveitando ao máximo os recursos de *clusters* heterogêneos.

[5] Twitter - https://twitter.com

O Apache Spark pode ser utilizado para a execução de algoritmos estocásticos, conforme foi apresentado por Zhang e Jordan (2015). Nesse trabalho os autores criaram o Splash, *framework* para a paralelização de algoritmos estocásticos em sistemas distribuídos. O Splash consite de uma interface de programação e uma base de execução. O *framework* permite que o usuário desenvolva algoritmos estocásticos sequenciais sem qualquer detalhe sobre a distribuição do processamento. O algoritmo é, então, automaticamente paralelizado por meio de uma base de execução otimizada. Os autores executaram experimentos utilizando algoritmos de *Logistic Regression e Collaborative Filtering*. As implementações utilizando o *framework* apresentaram bons tempos de execução e boa escalabilidade quando comparadas com outros trabalhos semelhantes da literatura que já foram criados originalmente de maneira distribuída.

Outra área do conhecimento que também já utilizou o Apache Spark para paralelizar e distribuir o processamento dos dados foi a área de análises científicas, mais especificamente dados coletados de equipamentos e experimentos astronômicos. O trabalho de Zhang et al. (2015) apresentou o Kira, um conjunto de ferramentas para processamento de imagens astronômicas que usa o Apache Spark. Os autores executaram experimentos para verificar a flexibilidade de programação, o fluxo dos dados e o desempenho. O Kira alcançou taxa de *speedup* de 2,5x quando comparado com outro algoritmo da literatura, processando 1TB de informações em um *cluster* de 512 núcleos. O trabalho foi concluído com uma discussão acerca da aplicabilidade do Apache Spark no processamento de grandes massas de dados científicos. Os autores apontaram que o Spark surge como uma boa alternativa para a solução desse tipo de problema.

Na área de bioinformática, é apresentado na Seção 3.1 diversos trabalhos relacionados ao Hadoop, entretanto, o Spark também é utilizado nessa área conforme o trabalho de Wiewiórka et al. (2014). Nesse artigo, foi apresentado o SparkSeq, uma biblioteca baseada no Apache Spark, flexível, extensível e que pode ser utilizada para processamento de dados de genoma para a computação em nuvem. No artigo, foram realizados experimentos com o SparkSeq e os autores concluíram, que a ferramenta apresenta boa escalabilidade e desempenho muito satisfatório ao processar conjuntos de dados de sequências de genoma. No trabalho, foram apresentados ainda alguns ajustes de parâmetros de desempenho, mesmo não sendo este o objetivo principal, e houve melhoria no tempo de resposta ao alterar o tamanho do bloco do HDFS.

Na área da neurociência o Spark também já foi utilizado e apresentou bons resultados, como é o caso do trabalho de Freeman et al. (2014). Nesse artigo, foi criada uma biblioteca de ferramentas analíticas para processamento de *big data* chamada Thunder. A biblioteca implementa um módulo para análise multivariada e univariada e os autores realizaram experimentos com dados de uma grande estrutura neural e concluíram que o Thunder alcançou bons resultados de tempo de resposta e apresenta-se como uma pro-

missora estrutura de código aberto para transformação de atividades cerebrais em *insights* biológicos.

O trabalho de Li et al. (2015) apresentou o SparkBench, um *benchmark* específico para a plataforma Spark que inclui uma série de aplicações cobrindo quatro categorias: *machine learning*, processamento de grafos, consultas SQL e aplicações de *streaming*. Os autores realizaram experimentos que apontaram a quantidade de recursos consumidos pela aplicação, informações de tempo e o impacto da alteração de valores de alguns parâmetros de configuração.

Alguns outros trabalhos da literatura são: Liu, Iftikhar e Xie (2014), Zaharia et al. (2013) e Akidau et al. (2013) realizaram experimentos de *streaming* em tempo real no Spark, Shi et al. (2016) utilizou o Spark para análises de falhas em equipamentos, Carlini et al. (2014) criou um algoritmo no Spark para o problema do *Balanced Graph Partitioning*, Maillo et al. (2016) criou uma solução exata para o algoritmo dos K-Vizinhos mais Próximos baseada no Spark, Alsheikh et al. (2016) e Kim et al. (2016) utilizaram *Deep Learning* no Spark, Armbrust et al. (2015) estudaram a usabilidade e desempenho do Spark e Ye et al. (2016) criou um *framework* para mineração de textos da área biomédica chamado SparkText.

Conforme foi apresentado nessa seção, o Apache Spark já é uma realidade e abrange soluções para diversas áreas do conhecimento, porém muitas outras áreas ainda poderão usufruir do benefício da ferramenta à medida que novas bibliotecas forem criadas e novos estudos realizados.

3.3 Grafos com Apache Hadoop e Spark

Utilizando o MapReduce para execução de algoritmos aplicados à Teoria de Grafos, é possível encontrar trabalhos importantes e com muitas citações, como é o caso do artigo de Lin e Schatz (2010) que criou padrões de projeto para a aplicação do modelo em algoritmos para grafos, Kang et al. (2011a) e Kang, Tsourakakis e Faloutsos (2009), que utilizaram o modelo para processar grandes grafos e encontrar medidas de centralidade de forma aproximada.

É possível encontrar também vários outros trabalhos, como é o caso de Nascimento e Murta (2012), que criou o algoritmo iterativo HEDA (*Hadoop-based Exact Diameter Algorithm*) para encontrar medidas de centralidade em grandes grafos, porém de maneira exata. Nesse trabalho, os autores realizaram experimentos com o algoritmo utilizando grafos reais e sintéticos e compararam seus resultados com o algoritmo HADI Kang, Tsourakakis e Faloutsos (2009) e também com um algoritmo sequencial presente na literatura (GONÇALVES, M. R. S.; MACIEL, J. N; MURTA, C. D., 2010). Ao final, foi confirmado que o HEDA possui boa escalabilidade e apresenta bons resultados no que diz respeito ao tempo de

execução. Além disso, os autores conseguiram melhorar o tempo de execução do algoritmo em aproximadamente 34% utilizando uma parametrização específica do Hadoop.

O trabalho de Cruz e Murta (2013) apresentou o algoritmo AHEAD (*Advanced Hadoop Exact Algorithm for Distances*), que é uma continuação do algoritmo HEDA, com várias melhorias, como a implementação das técnicas *In-Mapper*, *Combiner* e *Schimmy*, presentes no trabalho de Lin e Schatz (2010), que proporcionou um melhor balanceamento na distribuição dos dados. Ao final, os autores concluíram que o algoritmo AHEAD apresentou redução de aproximadamente 69% no uso de memória RAM do *cluster* e 35% do uso de espaço em disco e também melhorou o tempo de execução e as taxas de *speedup* e eficiência.

Mais recentemente o trabalho de Junghanns et al. (2015) criou o Gradoop (*Graph analytics on Hadoop*), projetado para ser um conjunto de operadores de alto nível que suportam o particionamento, a replicação, versionamento, escalabilidade e tolerância a falhas de grandes e heterogêneos grafos, utilizando a tecnologia Hadoop. Com esses operadores os autores projetaram uma linguagem de domínio específico e, em sua versão inicial, o Gradoop analisou vários grafos com dados de algoritmos de inteligência artificial e também de redes sociais.

Já o trabalho de Svendsen, Mukherjee e Tirthapura (2015) apresentou um novo algoritmo para a contagem paralela dos cliques de um grafo grande utilizando o *framework* Hadoop. O algoritmo PECO (*Parallel Enumeration of Cliques using Ordering*) enumera os cliques máximos não duplicados, partindo de todos os vértices ordenados. O algoritmo foi implementado em MapReduce/Hadoop e foi possível observar que ele processa uma grande variedade de grandes grafos do mundo real com milhões de vértices e escala bem à medida que novos recursos ou novos tamanhos de grafos são utilizados.

O Apache Hadoop ainda é amplamente utilizado para o processamento de grafos, entretanto pode-se encontrar problemas durante a iteratividade dos algoritmos, uma vez que a maioria dos algoritmos para grafos têm essa característica. O Hadoop não é bom para processamento iterativo devido ao custo atribuído à leitura de todos os dados no disco que é feita a cada iteração do algoritmo (GU; LI, 2013). Por outro lado, a tecnologia Apache Spark foi projetada com um mecanismo de *cache* com o objetivo de alcançar melhor desempenho e tempo de resposta durante o processo iterativo. Apesar de o Spark ser uma tecnologia nova quando comparada com o Hadoop, ainda assim existem muitos trabalhos que lidam com o processamento paralelo e distribuído de grandes grafos. O principal trabalho nessa linha é o de Xin et al. (2013), que introduziu o GraphX, uma biblioteca que combina as vantagens do processamento paralelo e os algoritmos para grafos dentro do mesmo *framework* (Apache Spark). O sistema explora o processamento em memória RAM, utiliza *cache* para diminuir o *overhead* de cada uma das iterações e possui ampla tolerância a falhas, fornecendo um poderoso conjunto de operações para construção e transformação

de grafos. O GraphX foi baseado em um conjunto de algoritmos para processamento de grafos criado em 2010, e era considerado uma alternativa ao modelo MapReduce/Hadoop na época, chamado Pregel (MALEWICZ et al., 2010). O GraphX já possui implementados alguns algoritmos clássicos, tais como: PageRank, Triangle Count, ShortestPaths, *Label Propagation, Connected Components* dentre outros.

O iGraph, proposto no trabalho de Ju et al. (2016), é um sistema incremental de processamento de grafos dinâmicos, ou seja, que permite atualizações constantes no conjunto de dados. As principais contribuições do iGraph são: uma estratégia de particionamento e refinamento de atualização dos grafos, processamento de grafos para suportar dados incrementais e um método de detecção e balanceamento para re-endereçar a carga de trabalho à medida que o grafo é incrementalmente atualizado. O iGraph pode implementar vários algoritmos, inclusive o PageRank. Segundo os autores, para grafos reais, o iGraph supera o GraphX tanto na atualização quanto no processamento do conjunto de dados.

O trabalho de Capota et al. (2015) criou um *benchmark* chamado Graphalytics, que pode ser utilizado para avaliar bibliotecas de processamento paralelo e distribuído aplicadas a grafos. O Graphalytics possui geração de dados (grafos sintéticos) e possui suporte a quatro plataformas: Hadoop/MapReduce, Giraph (baseado no Pregel), GraphX e Neo4j[6].

Por último, o trabalho de Lim et al. (2015) avaliou três plataformas de processamento de grafos: Pegasus, GraphX e Urika[7], por meio de três algoritmos PageRank, *Degree Distribution* e *Connected Components*, através de dados reais. Os resultados mostraram que cada plataforma tem força em um determinado tipo de operação em grafos. A plataforma Urika foi melhor para operações em grafos não-iterativas com o algoritmo *Degree Distribuition*. O GraphX foi melhor para operações iterativas aplicando os algoritmos *Connected Components* e PageRank. Ao final, os autores discutiram sobre opções de otimização que podem ser aplicadas a cada uma das plataformas para melhoria do desempenho.

Apesar de terem sido encontrados diversos trabalhos na literatura referentes à teoria de grafos, tanto para a ferramenta Hadoop quanto para a ferramenta Spark, é certo que ainda existem muitas oportunidades de trabalho, tanto na área de *benchmark* quanto na criação e adaptação de algoritmos clássicos para o processamento paralelo e distribuído, além do processamento de diversas bases reais.

3.4 Hadoop x Spark

Sempre quando uma nova tecnologia é criada surge o interesse de vários usuários e pesquisadores pela comparação desta com as tecnologias já existentes com o objetivo de encontrar quais são os pontos fortes e fracos desta nova forma de solução de problemas.

[6] Neo4j - https://neo4j.com/

[7] Urika Processing - http://www.cray.com/products/analytics/urika-gx

Com o Spark isso não é diferente, pois grande parte dos pesquisadores tem interesse nessa comparação e trabalhos referentes a esse assunto são amplamente encontrados na literatura.

O principal artigo nessa linha é o de Zaharia et al. (2010). Nesse trabalho, os autores criaram a ferramenta Spark para ser um novo *framework* que suporte a execução em memória de aplicações compostas por algoritmos iterativos, enquanto mantém a escalabilidade e a tolerância a falhas que já eram características presentes no *framework* Hadoop. No trabalho foi realizada a comparação do tempo de execução entre o Hadoop e o Spark e a segunda ferramenta alcançou um desempenho até 10 vezes melhor que a primeira. No Hadoop, o algoritmo *Logistic Regression* gastou 127 segundos para cada iteração e no Spark a primeira iteração gastou 174 segundos, porém as iterações posteriores gastaram apenas 6 segundos e isso foi justificado devido ao uso de dados em *cache* que é a principal inovação implementada no Spark. Os autores ainda utilizaram o Spark para processar iterativamente em memória, 39GB de dados da Wikipedia. A primeira iteração gastou 35 segundos e as subsequentes levaram entre 0,5 e 1 segundo.

O trabalho de Gu e Li (2013) também segue a linha de comparação entre Hadoop e Spark, porém com o foco totalmente voltado para o consumo de memória RAM. Os autores destacaram que o consumo de memória não foi considerado no trabalho de Zaharia et al. (2010) e por isso decidiram explorar mais profundamente esse critério. Foram conduzidos experimentos para algoritmos iterativos e comparados os tempos de resposta e os custos de memória RAM em ambos os *frameworks*. Os autores concluíram que, embora o Spark seja mais rápido que o Hadoop para operações iterativas, ele aumenta muito o consumo de memória. Outro destaque do trabalho é que a maior velocidade do Spark fica comprometida quando não se tem grandes quantidades de memória, pois nesse caso torna-se necessário realizar operações em disco, o que aumenta o tempo de processamento. Devido ao alto consumo de memória do Spark, os autores concluíram que, se a velocidade não é um requisito fundamental da aplicação e não se tem quantidades abundantes de memória, é melhor não escolher o Spark. Nesse caso, contando que se tenha espaço em disco suficiente para armazenar o conjunto de dados e os dados intermediários, o Hadoop é a melhor escolha. Obviamente se a aplicação é iterativa e sensível ao tempo o Spark deverá ser considerado. Nesse caso, é importante ressaltar que serão necessárias grandes quantidades de memória para extrair o melhor que a ferramenta Spark pode oferecer.

Um trabalho muito citado na literatura é o de Zaharia et al. (2012). Nesse trabalho, os autores focaram no RDD (*Resilient Distributed Dataset*), que é uma abstração de memoria distribuída que permite aos programadores executarem o processamento em memória em grandes *clusters* com tolerância a falhas. O trabalho mostrou que os RDDs são capazes de implementar uma ampla variedade de aplicações, principalmente as que utilizam algoritmos iterativos. Foram discutidas ainda as vantagens do modelo RDD, suas principais operações e

alguns exemplos de aplicações (*Logistic Regression, K-Means e PageRank*). Para avaliação foram realizados uma série de experimentos utilizando os *frameworks* Hadoop e Spark e *benchmarks*. Os resultados apontaram que o Spark supera o Hadoop em até 20 vezes para aplicações iterativas de *machine learning* e grafos. O Spark também apresentou boa escalabilidade e capacidade de se recuperar rapidamente de falhas, reconstruindo apenas as partições RDDs que foram perdidas.

Utilizando conjuntos de dados coletados de *logs* de aplicações, o trabalho de Lin e Wang (2013) analisou as características dos pares de ferramentas Hadoop/Spark e Hive[8]/Shark. Atualmente o Apache Shark passou a se chamar Spark SQL[9]. Foram implementados os algoritmos K-Means e PageRank ambos para o Hadoop e Spark. Como resultado, analisando o algoritmo K-Means, o Spark foi 40 vezes melhor que o Hadoop. A medida que a quantidade de dados foi aumentando essa vantagem do Spark teve uma tendência de diminuição, estabilizando em 8 vezes. No trabalho, os autores não apresentaram numericamente os resultados do algoritmo PageRank, entretanto, foi possível deduzir visualmente nos gráficos que a vantagem do Spark foi menor nesse algoritmo do que no *K-Means*, principalmente para conjuntos de dados grandes.

Os autores do trabalho Xu et al. (2014) desenvolveram e compararam sistemas de recomendação baseados no algoritmo *Collaborative Filtering*, aplicados ao Hadoop e Spark. No Hadoop a aplicação foi implementada utilizando a biblioteca de *machine learning* Mahout, a qual é um elemento do conjunto de aplicações Hadoop. É importante ressaltar que os autores não utilizaram biblioteca de *machine learning* do Spark, destacando que a linguagem escolhida foi o Scala. Conforme já relatado por outros trabalhos incluídos nessa revisão, os autores perceberam que o Spark tem uma grande vantagem sobre o Hadoop, porém essa vantagem tende a diminuir à medida que o tamanho do conjunto de dados cresce. Essa diminuição se dá devido a falta de memória para a alocação completa do conjunto de dados. No caso do Hadoop, a maior parte do tempo é gasta para instanciar os *jobs*, para o caso de algoritmos iterativos que demandam essa tarefa várias vezes durante sua execução.

Alguns outros artigos também realizaram a comparação entre as duas tecnologias, sendo eles: (GOPALANI; ARORA, 2015), (REYES-ORTIZA; ONETOB; ANGUITA, 2015) e (VERMA; MANSURI; JAIN, 2016).

Apesar de ser possível encontrar diversos artigos na literatura recente referentes à comparação entre o Hadoop e o Spark, ainda existem muitas oportunidades de trabalhos que tratam da comparação do desempenho das duas ferramentas, principalmente no que diz respeito à utilização de algoritmos iterativos aplicados à teoria de grafos. Ainda é preciso uma melhor definição sobre o momento e situação correta de utilizar cada uma das

[8] Apache Hive - https://hive.apache.org/
[9] Spark SQL - http://spark.apache.org/sql/

ferramentas.

3.5 Algoritmos para Detecção de Comunidades

Existem na literatura diversos trabalhos que propuseram a criação de algoritmos sequenciais para a Detecção de Comunidades Sobrepostas. Levando em consideração esse projeto, podemos classificar o trabalho de Raghavan, Albert e Tirupatikumara (2007) como um dos principais, pois nele foi apresentado a técnica para Detecção de Comunidades *Label Propagation*, na qual os algoritmos DEMON (base deste projeto) e ADDIC foram baseados. Os autores avaliaram a técnica criada, analisando a complexidade do algoritmo, realizando diversos experimentos e concluíram que o algoritmo criado apresenta resultados melhores que outros trabalhos da literatura até aquele momento.

Uma técnica hierárquica para Detecção de Comunidades é o Método *Agglomerative* o qual se encaixa no grupo dos algoritmos gulosos. Nesta abordagem, vértices individuais são encontrados progressivamente e mesclados em *clusters* de uma maneira *bottom-up*. Inicialmente, cada vértice forma sua própria comunidade e então o algoritmo combina a comunidade do vértice com outras, formando agrupamentos com alta Modularidade e com baixo custo computacional. A Modularidade determina a qualidade do agrupamento (*cluster*) produzido. Alguns algoritmos que detectam comunidades utilizando o método *Agglomerative* podem ser encontrados nos seguintes trabalhos: Riedy, Bader e Meyerhenke (2012), Staudt e Meyerhenke (2016), Bhowmick e Srinivasan (2013) e Cheong et al. (2013).

Os autores Rosvall e Bergstrom (2007) propuseram o método Infomap, o qual é uma técnica baseada na abordagem *Information-theoretic*. Usando esse conceito, a detecção de comunidades é mapeada para encontrar uma estrutura eficiente para a rede. O algoritmo é sequencial por natureza e não explora as vantagens do processamento em multiprocessadores. Segundo os autores, a técnica Infomap não é indicada para o processamento paralelo e distribuído, uma vez que existe uma alta dependência entre os dados.

O trabalho de Coscia et al. (2012) apresentou o DEMON, um algoritmo sequencial para Detecção de Comunidades Sobrepostas, baseado no conceito de Rede Ego e na técnica de *Label Propagation*. Nesse trabalho, os autores compararam o algoritmo criado com três outros, HLC (AHN; LEHMANN, 2010), Infomap (ROSVALL; BERGSTROM, 2008) e Walktrap (PONS; LATAPY, 2005). O DEMON apresentou melhores resultados na qualidade das comunidades encontradas em relação aos outros algoritmos utilizados na comparação.

Existe ainda uma versão paralela porém, não distribuída, do algoritmo DEMON, apresentada no trabalho de Amoretti Alberto Ferrari (2016). Os autores criaram o PaN-DEMON que apresenta melhores resultados em questão de escalabilidade e tempo de execução, quando comparado ao DEMON, entretanto esse trabalho se difere do nosso

projeto por executar, mesmo que paralelamente, em apenas uma máquina e não utilizar nenhum modelo emergente de programação.

O trabalho de Papadimitriou et al. (2008) criou o método CCT *(Context-specific Cluster Tree)* que encontra comunidades em múltiplos níveis, baseado no Princípio Teórico da Informação (*MDL - Information Theoretic Principles)*. Mais especificamente, o grafo é particionado em subgrafos progressivos, porém mais refinados, Esse é um método não-hierárquico. O algoritmo se mostrou próximo da escalabilidade linear no sentido do aumento do tamanho do grafo.

Outro trabalho que também utiliza técnicas de *Label Propagation* é o de Okamoto (2015). Os autores realizaram diversos experimentos com o algoritmo criado e concluíram que os resultados confirmam que as estruturas das comunidades detectadas são altamente consistentes.

3.6 Detecção de Comunidades com Processamento Distribuído

Existem diversos trabalhos na literatura que propõem soluções para o problema da Detecção de Comunidades e Comunidades Sobrepostas, utilizando para isso o processamento paralelo e/ou distribuído, seja com a criação de algoritmos nativos ou mesmo com a utilização de *frameworks*.

O trabalho de YANG YONG GUO e LI (2017) utilizou o algoritmo *RankClus*, proposto pelos autores Sun et al. (2009) para detecção de comunidades em grafos bipartidos, baseado na ideia de ranqueamento da qualidade dessas comunidades. Os autores utilizaram a implementação do *PageRank* presente na biblioteca *GraphX* do Apache Spark para substituir a função de ranqueamento do *RankClus*. Os resultados do trabalho apontaram conclusões promissoras, entretanto os autores não realizaram experimentos com bases realmente grandes e essa é uma proposta para trabalhos futuros.

Outro trabalho extremamente relevante da literatura é o que criou o *framework Ego-spliting* (EPASTO; LATTANZI; LEME, 2017). Nesse projeto os autores criaram um algoritmo distribuído que consegue analisar grandes grafos com até dezenas de milhões de vértices, alcançando boa escalabilidade. Esse trabalho difere-se desta tese, pois ele não utilizou técnicas de Redes Ego e nem o Apache Spark. A técnica de *Label Propagation* foi utilizada em uma parte do *framework* (Algoritmos de Clusterização).

No trabalho de Hung Miguel Araujo (2016), foi criada uma solução chamada TERA-COM, baseada no *framework* Apache Spark. O objetivo do algoritmo é, de maneira iterativa, reduzir o escopo do problema para realizar a detecção eficiente de comunidades. Os experimentos com o algoritmo foram concentrados em grupos de empresas de transporte aéreo, com o objetivo de distinguir aquelas linhas aéreas de baixo custo. Esse trabalho se difere

desta tese pois, apesar de utilizar o *framework* Apache Spark, ele não utiliza as técnicas de *Label Propagation* e Redes Ego.

Um dos trabalhos mais relevantes da literatura e que utiliza o processamento paralelo e distribuído para Detecção de Comunidades Sobrepostas foi apresentado por Buzun et al. (2014). Nesse trabalho, os autores criaram o EgoLP, um algoritmo baseado no Apache Spark, que utiliza técnicas de *Label Propagation* e também Redes Ego. O trabalho, que é tido como o primeiro a utilizar o Apache Spark e sua API GraphX para a solução do problema de Detecção de Comunidades, se assemelha com os algoritmos DEMON e ADDIC, entretanto, existem diferenças na forma como essas técnicas são aplicadas. O algoritmo DEMON e, consequentemente, o ADDIC, utilizam o conceito de Redes Ego, porém, em uma função EgoMinusEgo. Nesse caso, uma comunidade Ego não é extraída de uma Rede Ego de cada vértice, mas de uma rede chamada *EgoMinusEgo*, ou seja, uma rede sem o vértice Ego e todas as suas arestas. Essa técnica, de remover o vértice Ego, ajuda a evitar ruídos causados pela presença do vértice Ego conectado a todos os outros membros da Rede Ego. Além disso, a sequência de operações realizadas pelo ADDIC - *EgoMinusEgo*, Label Propagation e *Merge* - também se difere da sequência proposta pelo EgoLP, que possui os estágios: Pré-processamento, Recuperação de Comunidades Ego, *Label Propagation* e Pós-processamento.

O EgoLP pode ser classificado como o principal trabalho relacionado ao ADDIC e poderia até ser utilizado como base de comparação para esta tese, entretanto, diferentemente do DEMON, o EgoLP não teve o seu código disponibilizado por seus autores, o que dificulta a reprodução dos experimentos que foram realizados e de seus resultados. Além disso, o EgoLP foca mais na qualidade das comunidades encontradas, enquanto que esta tese está com o foco mais voltado para o tempo de processamento, consumo de recursos computacionais, Escalabilidade e Eficiência.

3.7 Síntese e Considerações Finais

Nesse capítulo foi realizado um estudo sobre o estado-da-arte do uso do *frameworks* Apache Spark e sua API GraphX, dos algoritmos para Detecção de Comunidades Sobrepostas e dos algoritmos paralelos para esse fim.

Antes do encerramento do capítulo é importante fazer um apanhado dos principais trabalhos listados e analisados e sua real relação com esta tese. Os principais trabalhos são:

1. **Holder et al. (2016):** Trata-se de um importante *survey* que apresentou desafios e problemas em aberto na área de mineração de grandes grafos. Por meio dele, pode-se definir alguns pontos a serem trabalhados nesse projeto, tais como: algoritmos para grafos escaláveis em modelos de programação emergentes e propagação em redes.

2. **Sobin, Raychoudhury e Saha (2017):** Esse trabalho é outro importante *survey* da literatura que trata especificamente de algoritmos paralelos para detecção de comunidades em grafos. Os autores do trabalho também apresentaram desafios que inspiraram o esta tese, tais como: paralelização de algoritmos sequenciais, detecção de comunidades sobrepostas e paralelização de algoritmos existentes para a descoberta de comunidades sobrepostas.
3. **Raghavan, Albert e Tirupatikumara (2007):** Uma vez definido que esta tese iria trabalhar com descoberta de comunidades sobrepostas, era preciso definir uma técnica a ser utilizada que fosse paralelizável e distribuível. Nesse ponto, surgiu a oportunidade de utilizarmos a técnica Label Propagation que tem a característica de ser mais indicada para a paralelização e distribuição do processamento, por não criar uma grande dependência entre os dados.
4. **Coscia et al. (2012):** Diante de um dos problemas em aberto apresentados por Sobin, Raychoudhury e Saha (2017), que foi a paralelização e distribuição de algoritmos existentes para detecção de comunidades, foi encontrado na literatura um algoritmo que não possuía essa versão, no caso o DEMON. A partir daí surgiu a oportunidade de desenvolver esse projeto com a criação da versão distribuída e paralela do DEMON.
5. **Xin et al. (2013):** Atendendo a sugestão de problemas em aberto em mineração de grafos apresentada por Holder et al. (2016), vislumbrou-se a necessidade de utilizar algum *framework*, paralelo e distribuído que atendesse a necessidade deste trabalho. A partir daí, o GraphX foi incorporado ao nosso projeto, por possibilitar a distribuição automática do processamento de algoritmos para grafos, como o *Label Propagation*, objeto do nosso trabalho.

Os trabalhos apresentados nesse capítulo indicam uma grande preocupação dos pesquisadores em produzir ferramentas, metodologias e técnicas que realmente possam resolver ou facilitar o processo de Detecção de Comunidades Sobrepostas, principalmente utilizando os benefícios da computação paralela e distribuída e dos frameworks que realizam a tarefa de paralelizar e distribuir o processamento e de maneira automática. Diante da revisão bibliográfica, pode-se observar que tanto o Hadoop MapReduce quanto o Apache Spark se apresentam como facilitadores para a solução do problema alvo deste projeto.

Essa tese surge com o objetivo de preencher a lacuna existente, ao propor um algoritmo para Detecção de Comunidades Sobrepostas, utilizando as técnicas de Label Propagation e Redes Ego e um modelo emergente de programação, no caso o Apache Spark e a API GraphX.

A Figura 12 apresenta uma compilação ano a ano dos trabalhos citados nesse projeto de tese. É importante destacar que houve um aumento considerável na quantidade de artigos nos últimos anos, o que indica que o assunto é atual, relevante e vem despertando o interesse dos pesquisadores.

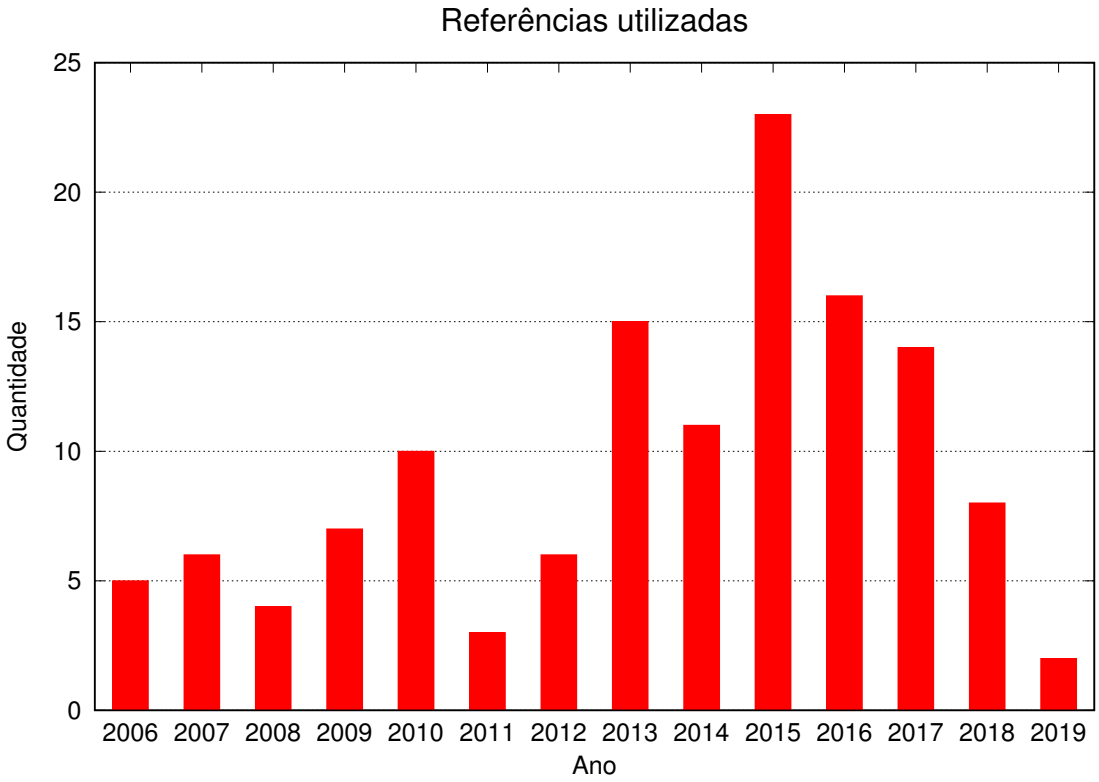


Figura 12 – Ano das referências utilizadas.

Em seguida, no Capítulo 4, será apresentado o foco deste trabalho, que é o algoritmo ADDIC, com todos os conceitos necessários para um bom entendimento desta proposta.

4 O Algoritmo ADDIC

Neste capítulo é apresentado e discutido os detalhes da implementação de um algoritmo distribuído para a descoberta de comunidades sobrepostas em grandes grafos proposto neste trabalho, utilizando a tecnologia Apache Spark.

O Apache Spark permite o processamento de grandes grafos, principalmente utilizando a biblioteca GraphX (XIN et al., 2013), que já traz implementada uma estrutura de dados específica para manipular grafos, além de algoritmos já implementados, como é o caso do Label Propagation.

Um trabalho que criou um algoritmo sequencial para descoberta de comunidades sobrepostas em grafos e que inspirou o desenvolvimento deste trabalho é o artigo de Coscia e Rosseti (COSCIA et al., 2012). O algoritmo sequencial DEMON é baseado na técnica Label Propagation e apresenta bons resultados para grafos médios. O problema é quando precisa-se processar grafos extremamente grandes, que é a realidade que vive-se atualmente, principalmente com a popularização das redes sociais.

Esse capítulo está organizado da seguinte forma. A Seção 4.1 descreve a ideia principal do ADDIC e apresenta a forma como a validação dos resultados foi realizada. A Seção 4.2 apresenta uma explicação detalhada do algoritmo, incluindo todas as suas fases. A Seção 4.3 detalha o formato dos dados de entrada suportado pelo ADDIC. Em seguida, a Seção 4.4 apresenta um exemplo prático para um melhor entendimento do algoritmo. Finalizando o capítulo, a Seção 4.5 apresenta as considerações finais.

4.1 Ideia principal do algoritmo

O algoritmo proposto foi denominado ADDIC, que significa Algoritmo para Detecção Distribuída de Comunidades, baseado no Apache Spark. O ADDIC é um algoritmo cuidadosamente desenvolvido e aperfeiçoado para descobrir comunidades sobrepostas em grafos médios e grandes. O ADDIC executa no ambiente Spark e apresenta boa escalabilidade à medida que novos recursos são adicionados ao *cluster*.

A ideia principal deste projeto é baseada no trabalho de Coscia et al. (2012), que criou o DEMON, um algoritmo sequencial para a descoberta de comunidades sobrepostas, baseada na técnica de *Label Propagation*. O algoritmo ADDIC foi inspirado no DEMON, porém a sua execução é paralela e distribuída. Até o momento não existe uma versão paralela e distribuída do algoritmo DEMON.

Durante a criação e desenvolvimento do algoritmo ADDIC, seus resultados de quantidade de comunidades encontradas e tamanho médio das comunidades foram validados

com os resultados produzidos pelo algoritmo sequencial DEMON, com o objetivo de comprovar sua exatidão. Durante esses testes preliminares, grafos de diversos tamanhos foram comparados. Todos os resultados comparados foram idênticos.

4.2 Descrição do algoritmo

A execução de busca por comunidades sobrepostas em grafos requer que todos os vértices pertencentes a essa estrutura sejam processados. Por isso, o algoritmo proposto por esse trabalho está dividido em fases, além de seu fluxo principal que será apresentado na Subseção 4.2.1. A Fase 1, descrita na Subseção 4.2.2, executa a função *EgoMinusEgo* sobre cada um dos vértices do grafo. Em seguida, na Fase 2, é executada a função Label Propagation e os detalhes são apresentados na Subseção 4.2.3. Por último, na Fase 3, os dados são mesclados por meio da Função Merge, descrita na Subseção 4.2.4.

4.2.1 Fluxo principal

O Algoritmo 1 apresenta o fluxo principal do algoritmo ADDIC.

Algoritmo 1: Algoritmo ADDIC - Fluxo Principal

Entrada: *O Grafo G(V, A)*; O Tamanho Mínimo da Comunidade T; O Percentual de Similaridade $\epsilon[0..1]$;
Saída: O Conjunto de Comunidades Encontradas C

```
1 G.vertices.collect.foreach {
2         v ← V
3         ego ← EgoMinusEgo(v, G)
4         cc[v] ← LabelPropagation(ego)
5         cc.collect.foreach{
6             c ← cc(v)
7         }
8 }
9 C ← Merge(C, c, ϵ)
```

O fluxo principal do algoritmo ADDIC recebe como parâmetros de entrada um grafo G(V, A), o tamanho mínimo que a comunidade encontrada deve ter e um percentual de similaridade entre grupos de vértices, usado na função Merge.

Logo no início, já na Linha 1, o ADDIC faz um laço em cada um dos vértices do grafo dado como entrada (variável G). Na Linha 2, cada vértice do grafo é atribuído para a variavel *v*. Na Linha 3, a função EgoMinusEgo é chamada, recebendo como parâmetro vértice por vértice e retornará a Rede Ego de um determinado vértice. A função *EgoMinusEgo* será detalhada ainda nesse capítulo. Em seguida, na Linha 4, será executada a função *LabelPropagation*, que retorna componentes conectados da Rede Ego do vértice v. A função *LabelPropagation* será detalhada ainda nesse capítulo. Em seguida, na Linha 5, é dado um

laço sobre os componentes conectados encontrados pela função LabelPropagation. Por fim, na linha 6, as comunidades são concatenadas pela função Merge, que também será detalhada ainda nesse capítulo.

4.2.2 Fase 1: Função EgoMinusEgo

Como apresentado na Linha 3 do Algoritmo 1, a função EgoMinusEgo é chamada para cada vértice v pertencente ao conjunto V do grafo original G.

A função EgoMinusEgo realiza duas operações principais. A primeira delas (EN) tem o objetivo de extrair a Rede Ego de um vértice. Dado um grafo G e um vértice $v \in V$, o resultado da função EgoMinusEgo $EN(v, G)$ é um subgrafo $G'(V', A')$, onde V é um conjunto contendo o vértice v e todos os seus vizinhos em A e A' é um subconjunto de A contendo todas as arestas (u, v) onde $u \in V' \wedge v \in V'$ (COSCIA et al., 2012).

A segunda operação tem o objetivo de realizar uma cópia do subgrafo sem a presença do vértice v e, consequentemente, sem as suas ligações com os vértices adjacentes: $-g(v, G)$. Essas duas operações formam a função $EgoMinusEgo(v, G) = -g(v, EN(v, G))$.

Para realizar a função EgoMinusEgo no Apache Spark, primeiramente foi filtrado no RDD G, todos os elementos onde o vértice v seja aresta, tanto na origem quanto no destino. Com isso, cumpriu-se a primeira operação da função EgoMinusEgo, que é encontrar a Rede Ego. Em seguida, o vértice v é extraído do resultado da primeira operação, bem como todas as arestas em que esse vértice se encontra.

Algoritmo 2: Algoritmo ADDIC - Função EgoMinusEgo

Entrada: O Grafo $G(V, A)$; O Vértice v;
Saída: A Rede Ego sem o vértice v

```
1 ego = G.fromEdges(Case Edge(src, dst) => src == v || dst == v).vertices.filter(
    case(id, name) => id != v).collect
2 minusEgo = G.subgraph(vpred - (case(id, name) =>
3             val resultado = ego.filter(case(identif, name) ==> identif == id)
4             val retorno = resultado > 0
5             retorno))
```

O Algoritmo 2 apresenta a função EgoMinusEgo. A Linha 1 tem o objetivo de extrair do grafo G os vértices que fazem parte da Rede Ego do vértice v. O RDD ego armazena a estrutura completa, ou seja, os vértices e as arestas que fazem parte da rede. Em seguida, na Linha 2, o vértice v é extraído da estrutura, assim como as arestas que pertencem a esse vértice. Com isso, tem-se a Rede Ego de um vértice, sem a sua presença.

4.2.3 Fase 2: Função Label Propagation

Ao encontrar o subgrafo com a função EgoMinusEgo, o próximo passo é efetivamente encontrar as comunidades. O algoritmo DEMON utiliza para essa tarefa um algoritmo da literatura (RAGHAVAN; ALBERT; TIRUPATIKUMARA, 2007), baseado na técnica de Label Propagation.

No caso desse trabalho, foi decidido utilizar uma implementação do algoritmo Label Propagation disponível na API GraphX do Apache Spark. O GraphX é uma biblioteca de algoritmos e estruturas de dados para manipulação distribuída de grafos.

A utilização da função LabelPropagation no GraphX é feita passando como parâmetro de entrada um grafo. No caso do nosso trabalho, a função LabelPropagation receberá a saída da função EgoMinusEgo, além de um parâmetro determinando o máximo de iterações.

Suponha que um determinado vértice v tem adjacentes $v_1, v_{,...v_k}$ e que cada vizinho carrega consigo um rótulo dizendo a qual comunidade ele pertence. Assim, v determina sua comunidade baseado nos rótulos de seus vizinhos. Cada vértice da rede escolhe juntar-se à comunidade na qual o número máximo de vizinhos pertence. Como os rótulos propagam, grupos de vértices densamente conectados rapidamente alcançam um consenso sobre um único rótulo. No fim do processo de propagação, os vértices com os mesmos rótulos são agrupados juntos como uma comunidade. No caso de um vértice com o mesmo número de vizinhos em mais de uma comunidade, poderá pertencer a ambas as comunidades, caracterizando assim comunidades sobrepostas.

O Algoritmo 3 apresenta os detalhes do *Label Propagation* e executa os seguintes passos (COSCIA et al., 2012):

1. Inicialize os rótulos de todos os vértices da rede. Para todos os vértices $v, C_v(0) = v$.
2. Atribua $t = 1$.
3. Organize os vértices da rede em um formato randômico e atribua para V.
4. Para cada $v_i \in V$, $C_{vi}(t) = f(C_{vi1}(t-1), ..., C_{vik}(t-1))$. A função f retorna o rótulo que possui a maior frequência entre os vizinhos.
5. Se cada vértice tem um rótulo idêntico ao rótulo da maioria de seus vizinhos, ou se t é o número máximo de iterações t_{max} então o algoritmo encerra, caso contrário $t = t + 1$ e vai para o passo 3.

4.2.4 Fase 3: Função *Merge*

O resultado da função *LabelPropagation* do algoritmo ADDIC é um conjunto de comunidades locais, de acordo com a perspectiva do vértice v. O objetivo seguinte é unir cada comunidade local para obter o resultado global.

Duas comunidades C e I são unidas se e somente se no mínimo $\epsilon\%$ da comunidade

Algoritmo 3: Algoritmo *Label Propagation*, baseado em Raghavan, Albert e Tirupatikumara (2007)

Entrada: *O Grafo G(V, A)*;
Saída: Comunidades rotuladas

```
1 t ← 1
2 V ← Randon( G(V,A) )
3 while CriterioDeParadaAusente do
4 |   i++
5 |   foreach v ∈ V do
6 |   |   C_vi(t) ← f(C_vi1(t − 1), ..., C_vik(t − 1))
7 return Vértices Rotulados C_vi(t) para cada v ∈ V, onde t é o último passo da
    execução.
```

menor está incluída na maior. Nesse caso, C e I são removidos da lista geral de comunidades e sua união é adicionada ao conjunto resultado. O ϵ é um percentual de elementos comuns fornecido por uma dupla de comunidades. Se $\epsilon = 1$, então duas comunidades são unidas, pois uma comunidade é subconjunto da outra. Se $\epsilon = 0$, então as duas comununidades não compartilham nenhum elemento. É importante destacar que a função *Merge* não é executada distribuidamente, pois existe dependência entre os dados, o que inviabiliza a distribuição do processamento.

4.3 Formato dos dados de entrada

Para padronizar a entrada de dados para o ADDIC, foi necessário criar um padrão de dados para a representação de um grafo. Inicialmente, para processar um grafo e encontrar as comunidades, um arquivo de dados representando o grafo é criado.

Para exemplificar o formato de dados criado, é apresentado o grafo da Figura 13.

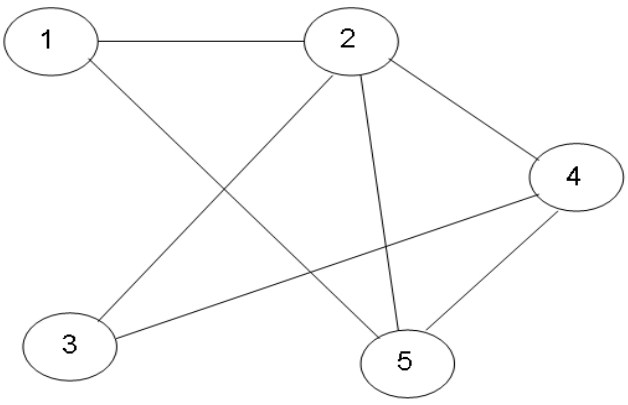


Figura 13 – Exemplo de um grafo com cinco vértices e sete arestas

Para esse grafo o arquivo terá o seguinte formato:

1TAB2
1TAB5
2TAB3
2TAB4
2TAB5
3TAB4
4TAB5

A Tabela 1 exibe a descrição detalhada das posições do arquivo, usando como exemplo a primeira linha do arquivo representado acima.

Posição	Exemplo	Significado
0 até TAB	1	Vértice de origem da aresta
Caracteres após o TAB	2	Vértice de destino da aresta

Tabela 1 – Descrição Detalhada do Arquivo de Entrada do Algoritmo ADDIC

4.4 Exemplo e fluxo de execução

Nesta seção, é apresentado um exemplo prático do algoritmo ADDIC para o grafo da Figura 14.

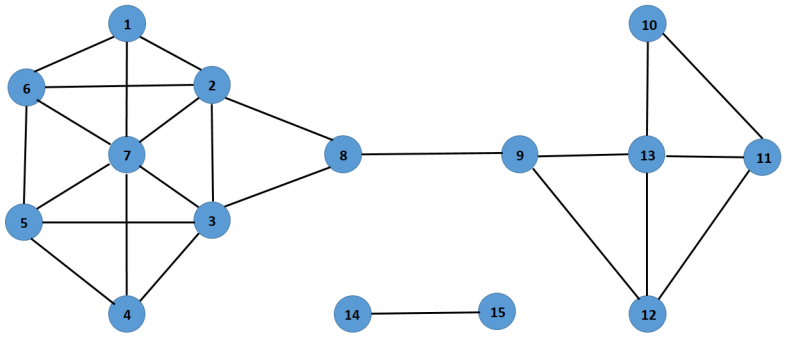


Figura 14 – Exemplo de um grafo com quinze vértices e vinte e cinco arestas

Para a representação do grafo da Figura 14, tem-se o arquivo de entrada com o seguinte formato:

1TAB2
1TAB6
1TAB7
2TAB3
2TAB6
2TAB7
2TAB8
3TAB4
3TAB5
3TAB7
3TAB8
4TAB5
4TAB7
5TAB6
5TAB7
6TAB7
8TAB9
9TAB12
9TAB13
10TAB11
10TAB13
11TAB12
11TAB13
12TAB13
14TAB15

O algoritmo inicia sua execução pelo fluxo principal (Algoritmo 1). Inicialmente foi realizado o carregamento de todo o arquivo de entrada para uma estrutura de dados específica do GraphX, que é o objeto Graph. Esse objeto permite o armazenamento da estrutura do grafo (vértices e arestas) em um RDD. Uma vez carregado esse objeto, já existem meios para realizar o processamento dos dados, pois a estrutura é um RDD e o armazenamento e processamento é distribuído.

Em seguida inicia-se um laço no objeto Graph, passando por cada um dos vértices pertencentes ao grafo. Nesse exemplo, esse laço executará 15 vezes. Para cada um dos vértices do grafo, será executado primeiramente a função EgoMinusEgo. O objeto Graph tem uma propriedade que armazena distintamente todos os vértices do grafo. Será executado um laço nessa propriedade e, nesse exemplo, o primeiro vértice a ser executado por EgoMinusEgo é o 1.

Como destacado na Seção 4.2.2, a função EgoMinusEgo executa duas operações. A primeira dela é encontrar a Rede Ego do vértice. Uma Rede Ego (Ego Network) é

uma rede composta por um vértice central do grafo (chamado EGO) e todos os vértices conectados a ele e também todos os relacionamentos existentes entre esses vértices (GONZALEZ-PARDO; JUNG; CAMACHO, 2016).

A Rede Ego do Vértice 1 é apresentada na Figura 17.

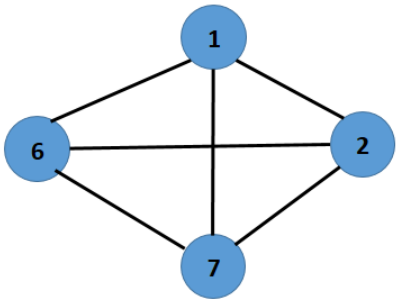


Figura 15 – Primeira operação da função EgoMinusEgo - Vértice 1.

Na segunda operação da função EgoMinusEgo, retira-se o vértice Ego da rede, nesse caso o Vértice 1. Por fim, a função EgoMinusEgo retorna o seguinte subgrafo, levando em considração o Vértice 1. A Figura 16 apresenta o grafo resultante.

Em seguida, será executada a função LabelPropagation (LP) e ela recebe como parâmetro de entrada o retorno da função EgoMinuEgo. O objetivo da função LP é rotular o subgrafo recebido e em seguida adicionar adicionar o vértice v. Como foi dado como entrada o conjunto de vértices formado por 2, 6 e 7, todos eles serão marcados com o mesmo rótulo, no caso dessa execução, o Vértice 2. Assim, ao final de LP haverá o seguinte conjunto: $LP = \{2 : [1, 2, 6, 7\ ]$.

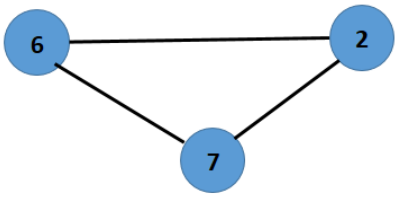


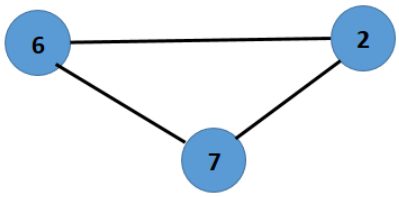
Figura 16 – Segunda operação da função EgoMinusEgo - Vértice 1.

Em seguida, haverá a execução da função Merge, que recebe os seguintes parâmetros de entrada:

1. c: comunidades já encontradas.
2. C: comunidades mescladas.
3. ϵ: fator de junção.

Para o Vértice 1, a função LP rotulou todos os vértices da mesma maneira (Rótulo 2), assim c terá o conteúdo vazio, pois como está na primeira iteração do algoritmo,

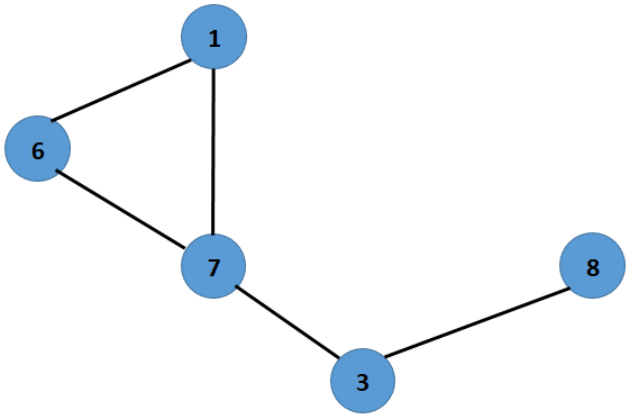
nenhuma outra comunidade foi encontrada. A variável C, que também é chamada de Comunidades Mescladas, receberá a comunidade que acabou de ser encontrada, ou seja: $C = \{0 : [1, 2, 6, 7]\}$. O número zero no início do conjunto C é simplesmente um identificador da comunidade encontrada. A Figura 17 apresenta o resultado final das 3 funções do algoritmo ADDIC (EgoMinusEgo, LP e Merge) aplicadas ao Vértice 1.



LP = {2: [1, 2, 6, 7]}
c = {}
C = {0: [1, 2, 6, 7]}

Figura 17 – Resultado das três funções do algoritmo ADDIC aplicadas ao Vértice 1.

A Figura 18 apresenta o resultado da execução das 3 funções do ADDIC para o Vértice 2. Pode-se observar que dois novos vértices foram mesclados na comunidade existente (Comunidade 0) e estes estão sublinhados na figura (Vértices 3 e 8). Pode-se observar também que, nessa iteração, a variável c veio preenchida com a comunidade encontrada na iteração anterior. A função LP encontrou a comunidade $[1, 2, 3, 6, 7, 8]$ e, ao compara-la com a comunidade já encontrada anteriormente $[1, 2, 6, 7]$, tem-se 4 elementos idênticos, ou seja, 66267% de similaridade, o que ultrapassa o valor do parâmetro ϵ, que nesse exemplo foi de 0,25 (25%).



LP = {6: [1, 2, 3, 6, 7, 8]
c = {0: [1, 2, 6, 7]}
C = {0: [1, 2, 3, 6, 7, 8]}

Figura 18 – Resultado das três funções do algoritmo ADDIC aplicadas ao Vértice 2.

Em seguida, será processado o Vértice 6. Não há uma sequência -seprocessamento

dos vértices, pois como o processamento é distribuído, os dados precisam ser independentes. A Figura 19 apresenta o resultado do processamento do Vértice 6.

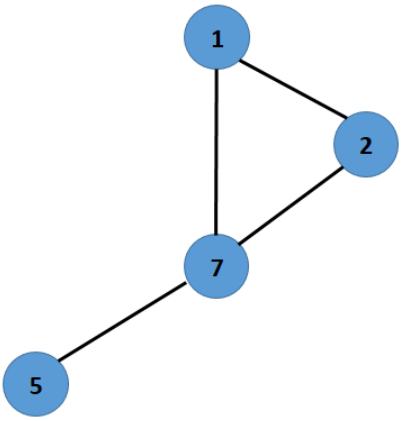


Figura 19 – Resultado das três funções do algoritmo ADDIC aplicadas ao Vértice 6.

O resultado do processamento do Vértice 7 é apresentado na Figura 20. Durante o processamento desse vértice, foi encontrado o Vértice 4.

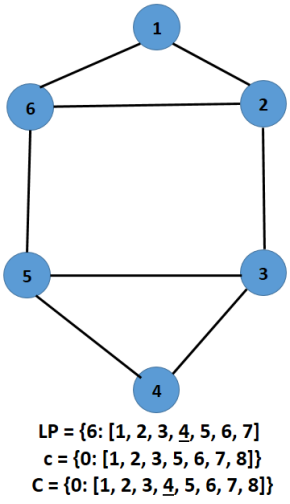


Figura 20 – Resultado das três funções do algoritmo ADDIC aplicadas ao Vértice 7.

O processamento do Vértice 3 é apresentado na Figura 21. Pode-se observar nesse caso que não houve a descoberta de nenhum vértice novo. Todos os vértices da Rede Ego do Vértice 3 já haviam sido descobertos em iterações anteriores.

O resultado do Vértice 8 é apresentado na Figura 22. No caso desse vértice, pode-se observar que foram descobertas duas comunidades. A primeira composta apenas pelo Vértice 9 e a segunda composta pelos vértices 2 e 8. Para esse exemplo foi utilizado o

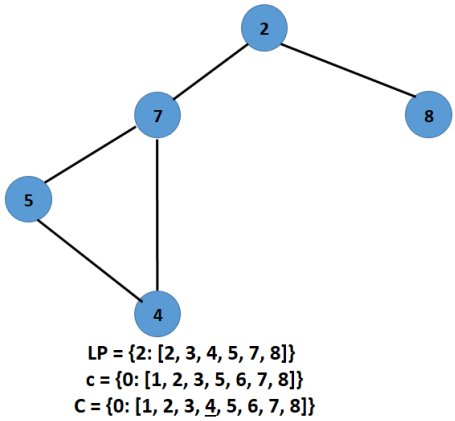


Figura 21 – Resultado das três funções do algoritmo ADDIC aplicadas ao Vértice 3.

parâmetro de tamanho mínimo de comunidade atribuído com 2, a primeira comunidade foi descartada pelo algoritmo. Nessa iteração nenhum novo vértice foi descoberto.

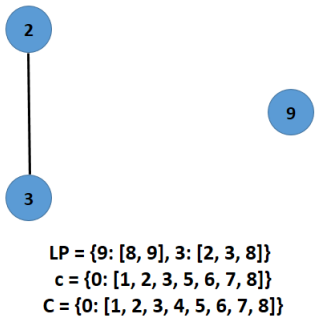


Figura 22 – Resultado das três funções do algoritmo ADDIC aplicadas ao Vértice 8.

A Figura 23 apresenta o resultado do processamento dos vértices 4 e 5, respectivamente. Nessas duas iterações do algoritmo, não há nenhum vértice novo para ser mesclado.

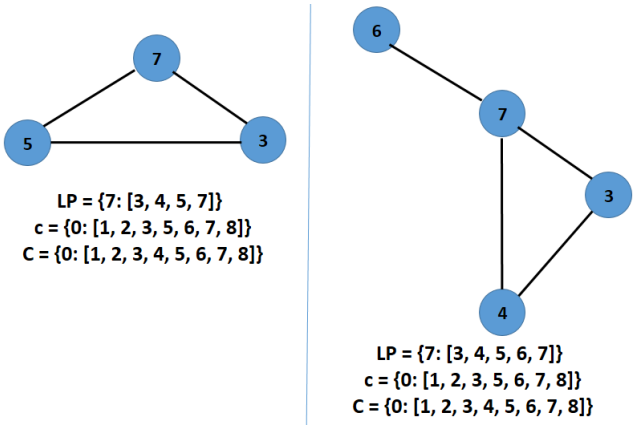


Figura 23 – Resultado das três funções do algoritmo ADDIC aplicadas aos Vértices 4 e 5.

A Figura 24 apresenta o resultado do processamento do Vértice 9. Pode-se observar

que nessa iteração que a função LP encontrou duas comunidades, entretanto somente uma será mesclada, respeitando o parâmetro definidor de tamanho mínimo. Pode-se notar uma situação diferente nesta iteração. No momento de realizar o *Merge*, o algoritmo calcula o percentual de similaridade entre a comunidade encontrada nessa iteração $([9, 12, 13])$ e a comunidade encontrada anteriormente $([1, 2, 3, 4, 5, 6, 7, 8])$. Pode-se perceber que o percentual de similaridade é zero e que o tamanho mínimo da comunidade foi atingido, portanto uma nova comunidade acabou de ser encontrada e seu identificador é 1.

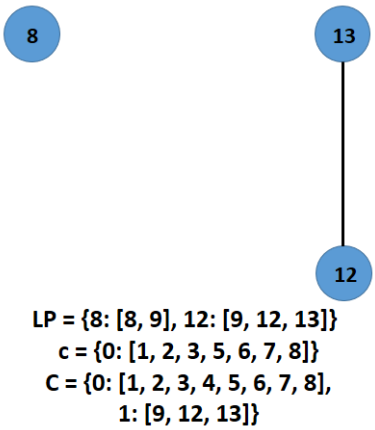


Figura 24 – Resultado das três funções do algoritmo ADDIC aplicada ao Vértice 9.

O próximo vértice a ser processado é o 12. A Figura 25 apresenta o resultado dessa iteração e pode-se observar que apenas uma comunidade foi descoberta e que existe um novo vértice que é o 11. A partir de agora as novas comunidades serão comparadas com as duas comunidades encontradas anteriormente, em busca de similaridade. Nesse caso, ao realizar uma comparação com a comunidade de identificador 0, percebe-se que não há nenhum vértice em comum, entretanto ao comparar com a comunidade de identificador 1, foram encontrados 3 vértices similares $[9, 12, 13]$.

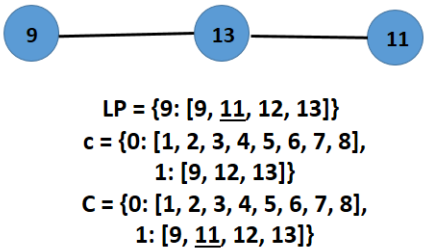


Figura 25 – Resultado das três funções do algoritmo ADDIC aplicada ao Vértice 12.

Na próxima iteração do algoritmo ADDIC, o vértice a ser processado será o 13. A Figura 26 apresenta o resultado. Percebe-se que foi encontrada a comunidade $[9, 10, 11, 12, 13]$ e apenas o vértice 10 é novo. Ao realizar a verificação de similaridade, percebe-se que essa comunidade tem maior similaridade com a comunidade de identificador 1, por isso a

comunidade descoberta será mesclada com ela. A Comunidade 0, nesse caso, não sofrerá nenhuma alteração.

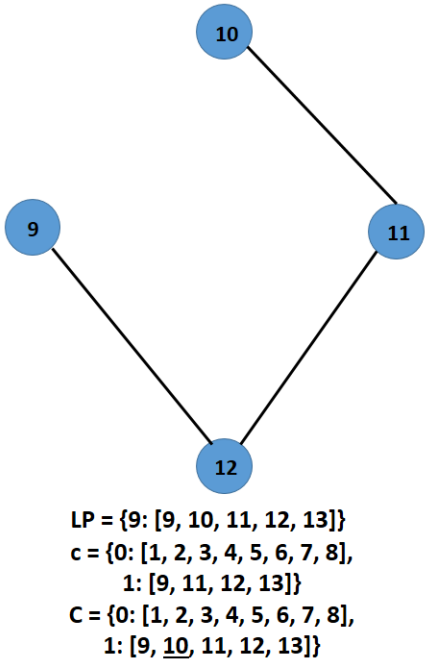


Figura 26 – Resultado das três funções do algoritmo ADDIC aplicada ao Vértice 13.

A próxima iteração do algoritmo processará o Vértice 10 e a função LP rotulou a seguinte comunidade: $[10, 11, 13]$. Todos esses vértices já se encontram na comunidade de identificador 1, portanto não há nenhuma alteração nas comunidades descobertas.

A iteração seguinte selecionou o Vértice 11 e a função LP rotulou a comunidade $[10, 11, 12, 13]$. Mais uma vez não haverá nenhuma mudança nas comunidades descobertas.

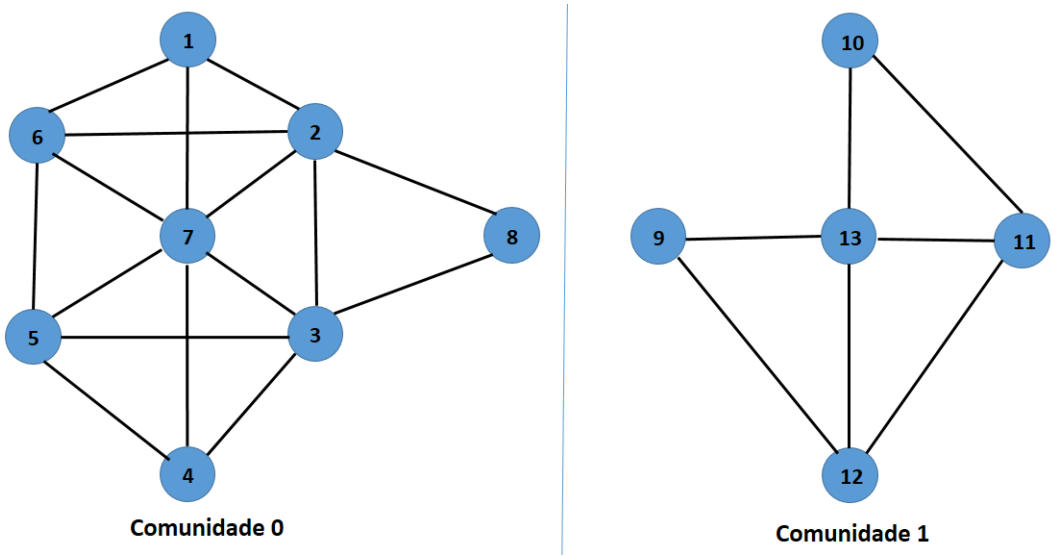


Figura 27 – Resultado do algoritmo ADDIC - Comunidades encontradas.

As duas próximas iterações fizeram a seleção dos vértices 14 e 15, respectivamente. Nenhuma das duas comunidades encontradas pela função LP atingiu a quantidade mínima de vértices, portanto foram descartadas.

Ao fim do algoritmo, duas comunidades foram encontradas e estão representadas na Figura 27.

4.5 Considerações Finais

Neste capítulo foi descrita a ideia principal e toda a estrutura do algoritmo ADDIC. A validação do algoritmo foi também discutida e, ao final, por meio de um exemplo foi apresentado o funcionamento do algoritmo passo a passo. O algoritmo ADDIC tem o objetivo de aproveitar o máximo possível da distribuição de processamento oferecida pelo *framework* Apache Spark.

Em seguida, no Capítulo 5, será apresentado a metodologia e o planejamento de experimentos que será utilizado para avaliar o algoritmo proposto. Em seguida, no Capítulo 6, os experimentos serão descritos e no capítulo seguinte os resultados dos experimentos serão apresentados.

5 Metodologia

Nesse capítulo será apresentada a metodologia e o planejamento dos experimentos para avaliação do algoritmo proposto neste trabalho. Na Seção 5.1, o delineamento da pesquisa é apresentado. Na Seção 5.2, é apresentado o ambiente computacional utilizado para executar os experimentos. A Seção 5.3 descreve os conjuntos de dados (grafos) que foram utilizados para execução dos experimentos e o formato adotado para esses dados. A Seção 5.4 apresenta a descrição do planejamento dos experimentos realizados nesse trabalho e a metodologia utilizada para comparar resultados do algoritmo ADDIC com o algoritmo DEMON (COSCIA et al., 2012), bem como uma discussão acerca da medição de escalabilidade do algoritmo.

5.1 Delineamento da Pesquisa

Um dos principais artigos que motivou o início das pesquisas para o desenvolvimento deste trabalho é um *survey* sobre algoritmos paralelos para detecção de comunidades (SOBIN; RAYCHOUDHURY; SAHA, 2017). Ao final do artigo, os autores fizeram uma discussão sobre os principais problemas em aberto na área e estes estão apresentados na Figura 28. Esses desafios foram fundamentais para o desenvolvimento deste trabalho. De acordo com os autores, o processo de detecção de comunidades em grandes grafos é um problema *NP-Difícil* e alguns algoritmos de aproximação são aplicados para essa tarefa, contudo esses algoritmos são, por natureza, sequenciais e não podem escalar para uma grande extensão. Tal ineficiência justifica a necessidade do desenvolvimento de algoritmos paralelos e distribuídos para detecção de comunidades.

Outro *survey* também muito importante para o delineamento desta pesquisa é o trabalho de Holder et. al. (HOLDER et al., 2016). Esse artigo trata dos atuais e futuros desafios para a mineração de grandes grafos. Os desafios e problemas em aberto, relatados pelos autores, estão destacados na Figura 29.

Diante dos problemas em aberto apresentados pelos dois artigos, primeiramente o foco será direcionado para o problema de Detecção de Comunidades Sobrepostas em Grandes Grafos. Com isso, parte-se para a possibilidade de incluir nesta pesquisa, novos desafios proposto pelos autores. Com isso, surgiu a possibilidade de trabalhar com algoritmos paralelos e distribuídos.

Nesse ponto, decidiu-se paralelizar um algoritmo sequencial presente na literatura, chamado DEMON (COSCIA et al., 2012). Foi decidido implementar então um algoritmo paralelo e distribuído para a a solução do problema de Detecção de Comunidades Sobrepostas

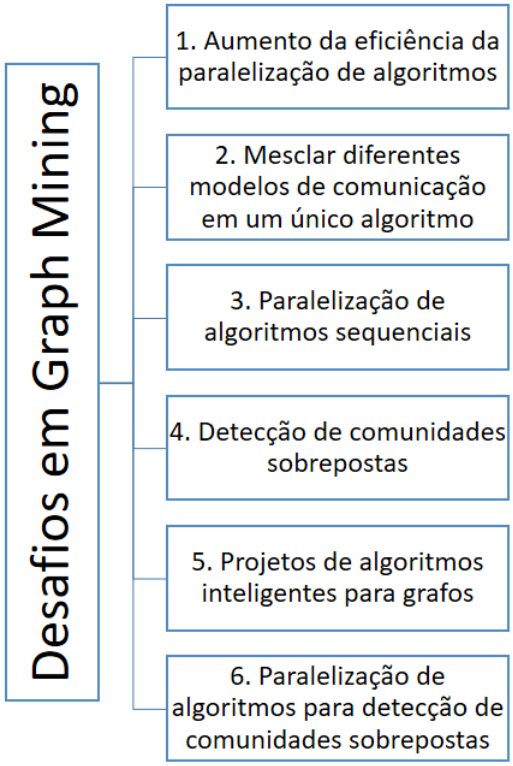


Figura 28 – Problemas em aberto, baseado em (SOBIN; RAYCHOUDHURY; SAHA, 2017)

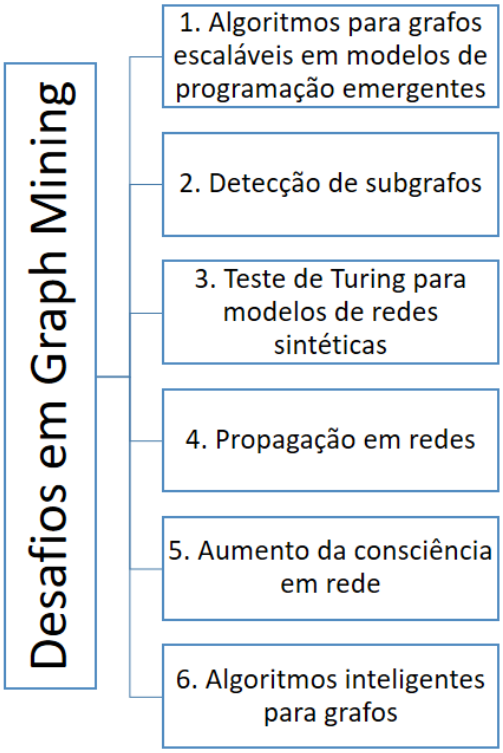


Figura 29 – Problemas em aberto, baseado em (HOLDER et al., 2016)

em Grandes Grafos, usando o modelo de programação emergente e escalável Apache Spark. Por último, decidiu-se implementar um algoritmo que fosse baseado na ideia de

Propagação em Redes, no caso o Label Propagation. A técnica de Label Propagation tem uma aplicabilidade melhor no processamento paralelo e distribuído, em comparação com outras técnicas, como o Infomap e Agglomerative, por exemplo (SOBIN; RAYCHOUDHURY; SAHA, 2017).

Surgiu então esse trabalho que é a implementação do ADDIC, um algoritmo paralelo e distribuído para detecção de comunidades sobrepostas, baseado na ideia de um algoritmo sequencial de mesmo propósito (DEMON), escalável, que utiliza um modelo de programação emergente (Apache Spark) e que implementa o conceito de propagação em redes (*Label Propagation*).

A Figura 30 apresenta o delineamento de desta pesquisa dividida em três colunas. Na da esquerda, estão os desafios apresentados pelo *survey* Sobin, Raychoudhury e Saha (2017) e incorporado aos objetivos deste projeto. Na coluna da direita estão os desafios do *survey* Holder et al. (2016) e que também foram incorporados nos objetivos deste projeto. Na coluna do meio encontram-se as características do algoritmo ADDIC e, cada uma delas, está ligado a um desafio de um dos *surveys*. Do trabalho de Sobin, Raychoudhury e Saha (2017), foi incorporado os desafios 3, 4 e 6. Já do trabalho de Holder et al. (2016), foi incorporado os desafios 1 e 4.

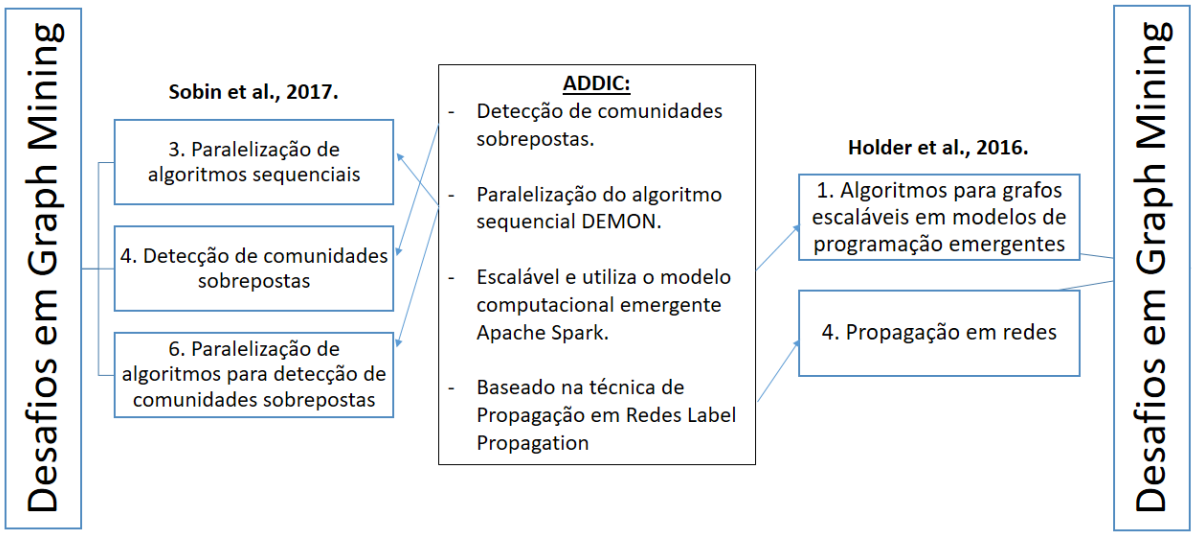


Figura 30 – Delineamento desta pesquisa baseado nos desafios apresentados por (SOBIN; RAYCHOUDHURY; SAHA, 2017) e (HOLDER et al., 2016)

Para o desenvolvimento deste trabalho no contexto de sistemas distribuídos, são utilizados os seguintes procedimentos metodológicos:

- Realização de uma ampla revisão bibliográfica com o objetivo de conhecer e analisar os principais trabalhos relacionados ao processamento de grandes grafos utilizando *frameworks* distribuídos (Hadoop, Spark, etc.);
- Avaliação dos algoritmos sequenciais para Deteccção de Comunidades Sobrepostas;

- Desenvolvimento de um algoritmo para Detecção de Comunidades Sobrepostas, baseado na técnica de Label Propagarion e em um algoritmo sequencial existente na literatura;
- Validação dos resultados do algoritmo criado baseada nos resultados de um algoritmo de mesmo propósito presente na literatura;
- Teste do novo algoritmo criado com diversas bases de grafos, reais e sintéticas, avaliando tempo de execução, consumo de recursos computacionais (CPU e memória RAM), escalabilidade e eficiência;
- Comparação dos resultados de tempo de execução com um algoritmo sequencial.

A Figura 31 apresenta um resumo de todos os passos que serão dados neste trabalho.

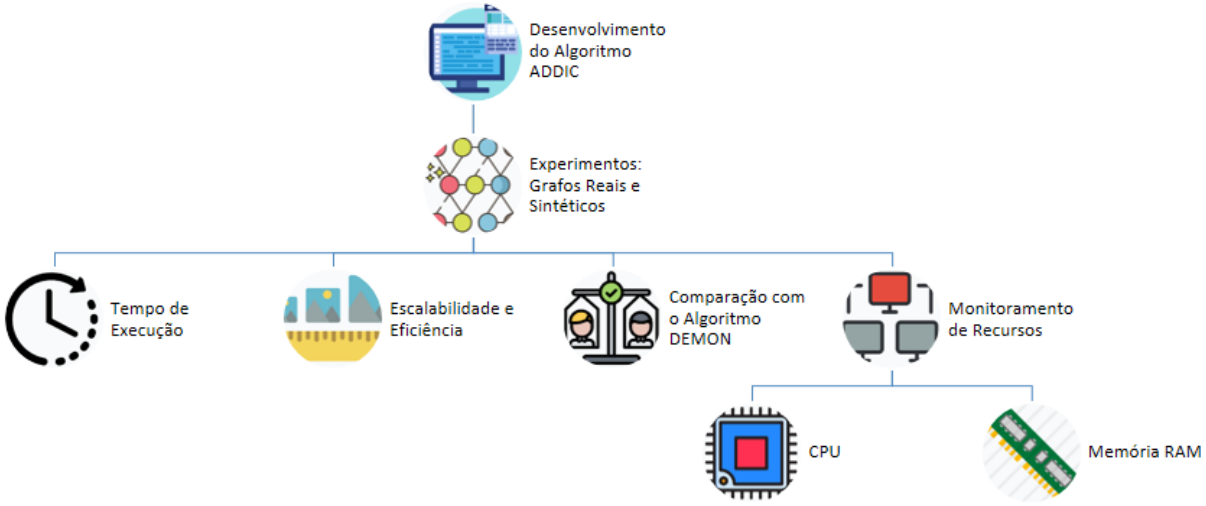


Figura 31 – Etapas da Pesquisa

Desde o início do projeto, deverão ser documentadas todas as etapas realizadas com o objetivo de possibilitar a reprodução exata dos resultados gerados.

5.2 Ambiente Computacional

Para os experimentos deste trabalho é utilizado um *cluster* composto por dezessete nós homogêneos, sendo um nó dedicado à coleta de dados referentes ao monitoramento do ambiente, um nó *master* e 15 nós *slaves* em um *dashboard* Open Stack[1].

O sistema operacional utilizado em cada um dos nós é o Linux Ubuntu 14.04.3LTS, com processador *Intel Westmere* (*Intel64 Family 6 Model 44 Stepping 1*) com 2.5GHz, 2 *threads*, 8GB de memória e 120GB de disco rígido. Todos os nós com o Hadoop em sua

[1] Open Stack 2016 - http://www.openstack.org

Id	Nome	Vértices	Arestas
GR01	Congress	526	14.198
GR02	ego-Facebook	4.039	88.234
GR03	IMDb	56.542	185.347
GR04	Amazon	410.236	2.439.437
GR05	soc-Pokec	1.632.803	30.622.564
GR06	soc-LiveJournal	4.847.571	68.993.773

Tabela 2 – Conjuntos de Dados - Grafos Reais

versão 3.1.0, o Apache Spark na versão 2.3.0 e a linguagem de programação Scala na versão 2.11.8.

5.3 Conjunto de Dados

Para a avaliação de desempenho do algoritmo ADDIC no ambiente Apache Spark foram utilizados diversos conjuntos de dados, de dois tipos básicos: grafos obtidos a partir de sistemas reais e grafos gerados sinteticamente. Os grafos sintéticos são grafos gerados aleatoriamente pelo modelo de Erdös-Rényi (ERDöS; RéNYI, 1959).

Os grafos reais nem sempre são facilmente encontrados com a quantidade de vértices e arestas e com as características necessárias para determinado tipo de experimento. Por isso, os grafos sintéticos foram utilizados, devido à flexibilidade que esse tipo de grafo fornece ao adicionar ou retirar vértices e arestas de acordo com a necessidade do experimento.

5.3.1 Grafos Reais

A Tabela 2 apresenta os grafos reais que foram utilizados durante a execução dos experimentos deste trabalho. Os grafos Congress, IMDb e Amazon também foram utilizados para testes do algoritmo sequencial DEMON, em seu artigo original (COSCIA et al., 2012).

Será feito agora uma descrição resumida de cada um dos conjuntos de grafos reais utilizados nos experimentos deste trabalho.

Congress: Representa uma rede de colaborações legislativas entre representantes da câmara e senado dos EUA durante a 111º legislatura (2009-2011). Foi obtido do site GovTrack[2] os dados sobre todos os projetos de lei discutidos durante essa legislatura. Os vértices podem representar projetos e também os políticos e as arestas a participação de determinado político em um projeto. Com isso pode-se ter comunidades formadas por legisladores que colaboraram com um mesmo projeto de lei.

[2] GovTrack - http:www.govtrack.us/developers/data.xpd

ego-Facebook: O Facebook é uma popular rede social com usuários em diversos locais do mundo e esse grafo fornece dados coletados pelo App dessa rede. No grafo, os vértices são usuários da rede e as arestas são os relacionamentos de amizade entre esses usuários. Os dados podem ser encontrados no site da Universidade de Stanford[3].

IMDb: O grafo IMDB (*Internet Movie Database*) é um grafo não dirigido e real que representa atores que tiveram participação em um determinado filme. Baseado na metodologia seguida pelo artigo original do DEMON (COSCIA et al., 2012), foi realizado o *download* da base inteira do IMDb em seu site oficial[4] em 15/06/2017. A partir daí, o foco foi nos atores que atuaram em, pelo menos, dois filmes no período de 01/01/2001 até 31/12/2010, descartando programas de TV e outros trabalhos.

Amazon: A Amazon é um dos maiores sites de comértio eletrônico no mundo. Os dados utilizados neste trabalho foram obtidos da Universidade de Stanford[5]. Trata-se de um grafo com as frequências de compras de produtos registrados no dia 05/05/2003. O grafo precisou ser transformado em um grafo não-direcionado. Se um produto i foi comprado em conjunto com o produto j, haverá uma aresta entre os vértices que representam esses dois produtos.

soc-Pokec: Pokec é a mais popular rede social da Eslováquia. A popularidade da rede social não mudou mesmo com a chegada do Facebook e conecta mais de 1.6 milhões de pessoas há mais de 10 anos. O conjunto de dados representa a rede toda e, nesse *dataset* utilizado, os dados das pessoas permanece no anonimato. Os dados podem ser encontrados no site da Universidade de Stanford[6].

soc-LiveJournal: LiveJournal é uma comunidade de blogs on-line e gratuita na qual os usuários realizam amizades entre si. O LiveJournal permite a criação de grupos onde vários usuários podem participar. Nesse grafo, os vértices caracterizam os usuários e as arestas as ligações existentes entre eles. Esses dados também poderão ser encontrados no site da Universidade de Stanford[7].

5.3.2 Grafos Sintéticos

Os geradores de grafos constroem grafos aleatoriamente. Partindo de um grafo vazio, o gerador vai adicionando progressivamente vértices e arestas até obter um grafo do tamanho que o usuário deseja.

Nesse trabalho foi utilizado um gerador de grafos (KANG et al., 2011a) disponível na Web[8], baseado no modelo de Erdös-Rényi (ERDöS; RéNYI, 1959). Foram gerados grafos

[3] Stanford University. Acesso em 14 de agosto de 2019. https://snap.stanford.edu/data/ego-Facebook.html
[4] IMDb - http://www.imdb.com/interfaces
[5] Stanford University. Acesso em 14 de agosto de 2019. https://snap.stanford.edu/data/amazon0505.html
[6] Stanford University. Acesso em 13 de agosto de 2019. https://snap.stanford.edu/data/soc-Pokec.html
[7] Stanford University. Acesso em 13 de agosto de 2019. https://snap.stanford.edu/data/com-LiveJournal.html
[8] http://www.cs.cmu.edu/ukang/HADI/

Id	Nome	Vértices	Arestas
GS01	Sint1	100.000	500.000
GS02	Sint2	500.000	2.500.000
GS03	Sint3	1.000.000	5.000.000
GS04	Sint4	5.000.000	25.000.000
GS05	Sint5	10.000.000	50.000.000

Tabela 3 – Conjuntos de Dados - Grafos Sintéticos

variando o número de vértices e arestas. Os vértices iniciaram com 100.000 e foram até 10.000.000. Para cada quantidade de vértices, haverá cinco vezes o número de arestas. A Tabela 3 apresenta os grafos sintéticos que foram utilizados.

Esse formato de grafo aleatório foi escolhido para possibilitar a análise do comportamento do algoritmo no que diz respeito ao tempo, consumo de recursos computacionais e escalabilidade, a medida que foi aumentada a quantidade de vértices e arestas. A quantidade de arestas (três vezes o número de vértices) foi escolhida para evitar vértices isolados no grafo, que não seria interessante para o objetivo deste trabalho, pois seriam itens que não fariam parte de comunidades.

Todos os vértices dos grafos sintéticos escolhidos possuem grau $g(v)$ maior que zero.

5.4 Planejamento dos experimentos

Experimentos são uma parte natural dos processos de tomada de decisão em engenharia e em ciência. Para um determinado problema, um experimento pode consistir em utilizar diversos métodos para a solução desse problema. Os dados desse experimento poderiam ser usados para determinar qual método deveria ser usado para fornecer a melhor solução dentre todas experimentadas. Um experimento pode ser somente um teste ou uma série de testes. Experimentos são feitos em todas as disciplinas científicas e de engenharia e constituem uma importante parte da maneira de aprender sobre como sistemas e processos funcionam. Consequentemente, o planejamento dos experimentos tem papel principal na solução futura do problema que inicialmente motivou o experimento (MONTGOMERY; RUNGER, 2007). O objetivo principal de um projeto de experimentos é obter o máximo de informação com um número mínimo de experimentos (JAIN, 1991).

A análise de experimentos ajuda na separação dos efeitos de vários fatores que afetam o desempenho, permitindo determinar se um fator teve um efeito significativo ou se a diferença observada é simplesmente devido a variações aleatórias causadas por medidas errôneas e parâmetros que não são controlados. Usando planejamento de experimentos, os engenheiros podem determinar que subconjunto das variáveis de processos tem a maior influência no desempenho do processo. Para este trabalho, o planejamento de experimentos

tem extrema importância, pois os resultados dos experimentos podem conduzir a um melhor rendimento do processo de avaliação, redução na variabilidade dos resultados e melhor obediência aos requisitos do projeto, assim como desenvolvimento, redução nos tempos do projeto e redução nos custos de execução (MONTGOMERY; RUNGER, 2007).

Vários tipos de projetos de experimentos são descritos na literatura. Os mais utilizados são projetos simples, projetos fatoriais completos e projetos fatoriais fracionados (MONTGOMERY; RUNGER, 2007). Devido ao grande número de combinações que haveria ao utilizar o projeto fatorial completo, optou-se pela utilização do projeto simples. Por meio de um projeto e análise de experimentos, espera-se obter resultados sobre o comportamento de um algoritmo paralelo e distribuído ao processar grandes massas de dados. O planejamento de experimentos tem importância fundamental sobre as seguintes etapas do trabalho: execução dos experimentos, análise dos resultados, análise de complexidade, comparação e conclusões.

A Seção 5.4.4 apresenta o planejamento de experimentos para comparação do algoritmo ADDIC com outro algoritmo existente na literatura. As Seções 5.4.1 e 5.4.2 apresentam os experimentos planejados para medir o tempo de execução e também do consumo de recursos computacionais (CPU e Memória RAM). A Seção 5.4.3 apresenta o experimento planejado para medir a escalabilidade do ADDIC.

5.4.1 Tempo de Execução

Um dos principais pontos de avaliação do algoritmo criado será o tempo de execução. Por isso, para todos os grafos propostos neste trabalho (reais e sintéticos), será monitorar o tempo de execução para todas as configurações de *cluster* propostas: 1, 2, 4, 8 e 16 máquinas. Esses experimentos auxiliarão também no cálculo de outras métricas propostas por este trabalho, tais como escalabilidade e eficiência. Inicialmente pretende-se executar cada um dos experimentos por três vezes e a média de tempo das execuções será calculada.

5.4.2 Consumo de Recursos

Ferramentas de monitoramento são usadas para medir o uso de recursos e o desempenho de sistemas e redes. Para a tarefa de monitoramento, sugere-se a escolha de uma ferramenta robusta, uma vez que essa tarefa, quando aplicada em um sofisticado ambiente de *cluster*, não é trivial (FATEMA et al., 2014). Realizar o monitoramento de máquinas é uma tarefa de suma importância quando se está realizando o teste de desempenho em *clusters*. Por meio dessas ferramentas pode-se gerenciar o uso do *hardware* e obter indicadores do nível de consumo de recursos computacionais, tais como CPU, memória, I/O (entrada e saída) e rede.

Para este trabalho, será utilizada a ferramenta Ganglia (MASSIE; CHUN; CULLER,

2004), que é um sistema de monitoramento distribuído e escalável desenvolvido para sistemas computacionais de alto desempenho, como *grids* e *clusters*. O Ganglia é um sistema de código aberto, desenvolvido na Universidade da Califórnia e baseia-se em um protocolo que faz uso da técnica de difusão *multicast* para comunicação entre os nós e utiliza uma estrutura de árvores para representar as conexões ponto-a-ponto. A escolha do Ganglia se deve ao fato de ser uma ferramenta que se integra muito bem aos *frameworks* Hadoop e Spark e por possuir gráficos e relatórios necessários para as análises dos recursos do *cluster* pretendidas nesse trabalho.

As análises serão feitas utilizando uma interface Web fornecida pela ferramenta Ganglia, que permite a visualização de diversas maneiras dos dados que foram registrados. É possível ter uma visão geral do *cluster* ou mesmo a visão de cada nó ou, em último nível, a visão dos recursos individuais do nó, tudo de forma muito detalhada. A ferramenta permite ainda que as análises sejam feitas filtrando um determinado período em que as coletas dos dados foram feitas e permite também a exportação desses dados em diversos formatos.

A Figura 32 apresenta uma visão em alto nível da interface Web da ferramenta, com gráficos em formato de grade resumindo em um simples processo de visão, os dados de todos os *clusters* conhecidos. A partir desse ponto é possível, em uma análise mais detalhada, monitorar cada um dos *clusters* individualmente e em um nível mais detalhado ainda é possível chegar ao monitoramento de um nó desse *cluster* e finalmente chegar até um determinado recurso desse nó.

A Figura 33 apresenta a visão em nível de *cluster* do Ganglia. Nessa visão, é possível monitorar o *cluster* como um todo, com o total de CPU's disponíveis (ativas e inativas) e ainda os recursos de maneira geral (Carga Geral, CPU, Rede, Memória, etc).

Por último, a Figura 34 apresenta uma visão em nível de um nó do *cluster*, onde é possível monitorar os recursos específicos de uma máquina, inclusive com filtros de datas.

Para a realização dos experimentos desta tese, foi designado um nó do *cluster* exclusivamente para realizar a coleta e armazenamento dos dados de monitoramento da ferramenta Ganglia.

Todos os experimentos realizados neste trabalho possuem os recursos computacionais consumidos monitorados pelo Ganglia e, no capítulo de resultados, será apresentado e detalhado esse consumo, principalmente no que diz respeito a CPU e Memória RAM que são dois recursos muito diretamente ligados ao Apache Spark.

5.4.3 Escalabilidade

A escalabilidade de um algoritmo paralelo e distribuído é a medida da capacidade da utilização efetiva dos recursos pelo algoritmo, ao aumentar o número de processadores (KUMAR; GUPTA, 1991). A análise da escalabilidade de uma combinação algoritmo-arquitetura

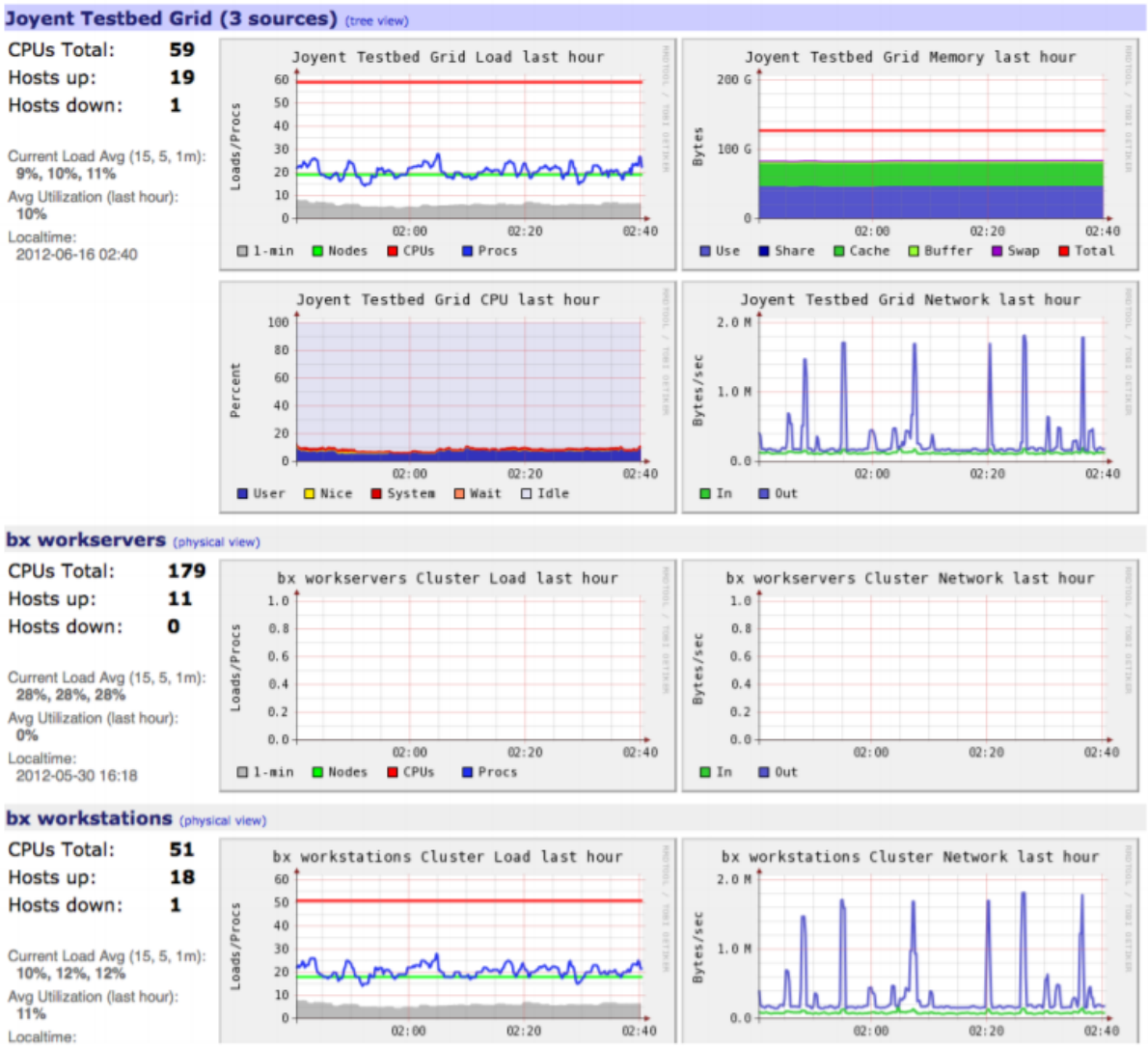


Figura 32 – Visão em Grade do Ganglia - (MASSIE et al., 2012)

pode ser usada para uma variedade de propósitos como, por exemplo, selecionar a melhor combinação entre o algoritmo e a arquitetura para a solução de um determinado problema que tenha várias restrições quanto ao aumento do tamanho desse problema e do número de processadores.

A escalabilidade pode ser usada também para predizer o desempenho de um algoritmo paralelo e uma arquitetura paralela para um grande número de processadores, por meio do desempenho conhecido em um ambiente com poucos processadores (KUMAR; GUPTA, 1991). Uma baixa escalabilidade pode resultar em um sistema com baixo desempenho, necessitando de uma reengenharia ou mesmo da duplicação dos sistemas (BONDI, 2000).

Os experimentos desta tese avaliaram a escalabilidade do algoritmo ADDIC à medida que máquinas são adicionadas ao *cluster*. Os testes foram iniciados partindo de

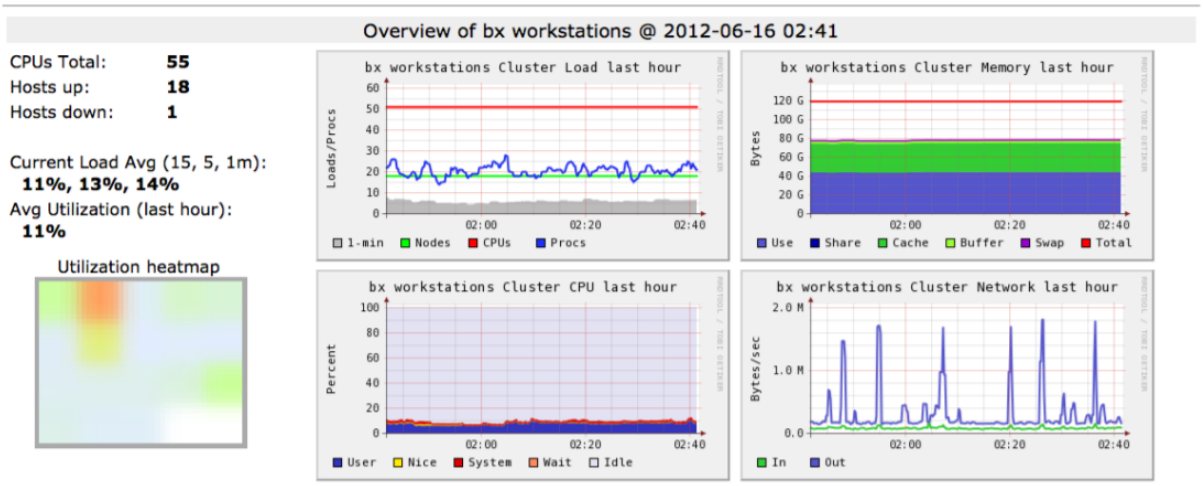


Figura 33 – Visão em Cluster do Ganglia - (MASSIE et al., 2012)

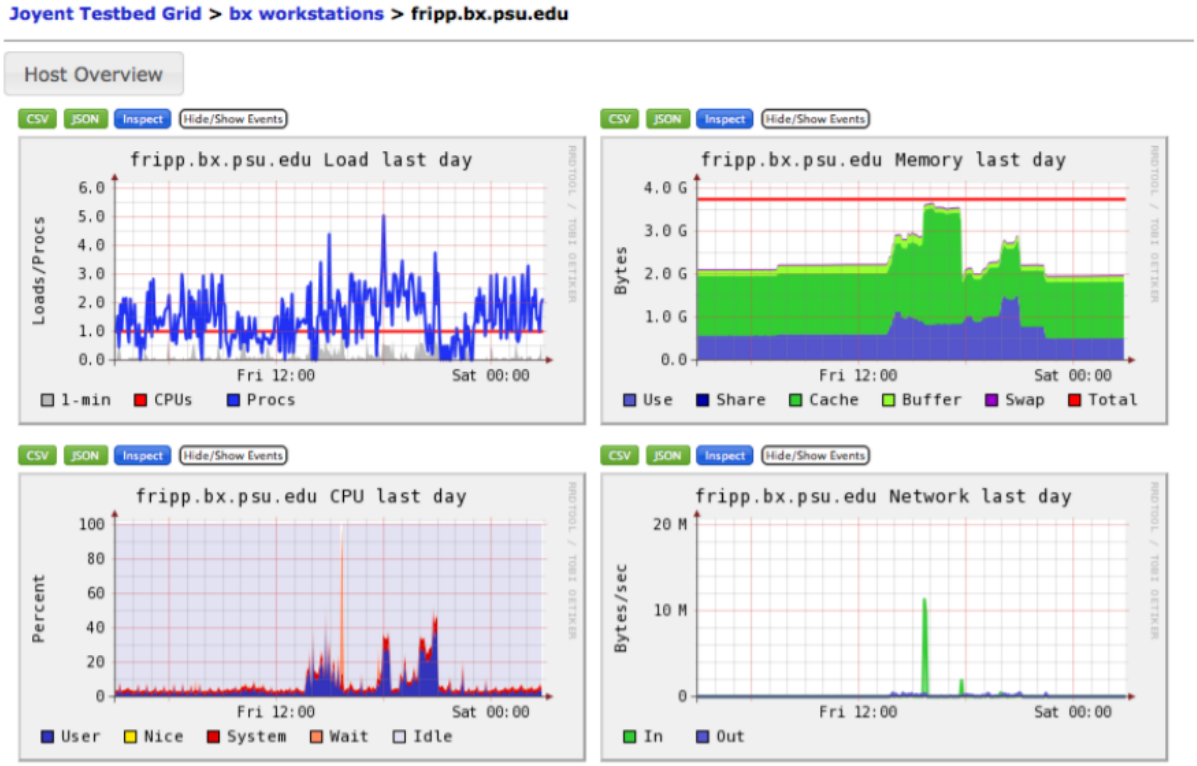


Figura 34 – Visão de um Nó no Ganglia - (MASSIE et al., 2012)

uma máquina e chegando até dezesseis máquinas, passando por 2, 4 e 8 máquinas. De posse dos tempos de execução avaliou-se a escalabilidade do algoritmo ADDIC verificando seus valores de *speedup* e eficiência. Isso foi feito para todos os grafos, reais e sintéticos.

5.4.4 Comparação com o algoritmo DEMON

Durante o desenvolvimento do algoritmo ADDIC, seus resultados de quantidade de comunidades e tamanho dessas comunidades foram comparados com o algoritmo sequencial DEMON, descrito no trabalho de Coscia et al. (2012), com o intuito de validar a exatidão dos resultados apresentados pelo algoritmo.

Nem todos os tamanhos de grafos propostos por esse trabalho puderam ser executados no algoritmo sequencial devido ao grande espaço ocupado por grandes grafos e pelo excessivo uso de memória RAM. Assim, apenas alguns grafos reais e sintéticos foram processados no algoritmo sequencial. Os detalhes destes experimentos serão apresentados no Capítulo 6.

Existe o planejamento de executar cada experimento por três vezes e fazer uma média dos resultados, entretanto, no Capítulo 6 (Resultados) a viabilidade desta estratégia será discutida.

Aqueles grafos que puderam ser executados no algoritmo DEMON e no algoritmo ADDIC, terão seus tempos de execução comparados.

5.5 Considerações finais

Neste capítulo foi discutida a metodologia dos experimentos executados. Foi apresentado o ambiente computacional para a execução dos experimentos, os conjuntos de dados utilizados, todo o planejamento de experimentos e a forma como os dados serão analisados.

No próximo capítulo, os resultados deste trabalho serão apresentados.

6 Experimentos: Resultados e Análise

Neste capítulo são apresentados os resultados dos experimentos utilizando o algoritmo ADDIC, executados em um *cluster* com o *framework* Apache Spark. Este capítulo está estruturado da seguinte forma. Na Seção 6.1, os resultados de tempo de execução do algoritmo são apresentados e discutidos. Esta seção engloba subseções que tratam especificamente de cada tipo de grafo utilizado nos experimentos reais e sintéticos, além de uma análise geral dos resultados obtidos.

Na Seção 6.2, os resultados referentes ao consumo de recursos do *cluster*, tais como o uso de CPU e Memória RAM, são apresentados e discutidos. Esta seção também engloba subseções que tratam especificamente de cada tipo de grafo utilizado nos experimentos, além de uma análise detalhada dos resultados obtidos. Nas Seções 6.3 e 6.4, são apresentados, respectivamente, estudos que analisam a Escalabilidade e Eficiência do algoritmo ADDIC. A Seção 6.5 apresenta a comparação entre o algoritmo distribuído ADDIC, proposto nesse trabalho, com o algoritmo sequencial DEMON. Ao final do capítulo, as considerações finais são apresentadas.

6.1 Tempo de Execução

Esta seção tem o objetivo de apresentar o comportamento do algoritmo ADDIC no que diz respeito ao tempo de execução dos experimentos realizados. Como pode ser visto nas subseções 6.1.1 e 6.1.2, foram utilizados todos os conjuntos de dados para a realização dos experimentos e, para cada um deles, foi computado o seu respectivo tempo de execução.

Cada um dos experimentos foi realizado variando a quantidade de nós do *cluster*, começando com apenas 1 máquina e, posteriormente, adicionando novas máquinas, chegando nas configurações de 2, 4, 8 e 16 nós. Além de medir o tempo de execução, o objetivo dessa variação na configuração do *cluster* é fornecer dados para a análise de escalabilidade e eficiência do algoritmo ADDIC. Essa análise será apresentada em seções posteriores desse capítulo.

6.1.1 Grafos Reais

Para alguns *datasets* de grafos reais, o algoritmo ADDIC foi executado três vezes e o tempo médio de execução calculado. Como o tempo de execução de cada *dataset* é elevado e também como foi observado que o tempo não variava de maneira considerável entre uma execução e suas repetições, decidiu-se por executar os experimentos uma vez

Id	Nome	Vértices	Arestas	Comunidades	Tam. Médio
GR01	Congress	526	14.198	425	63,36
GR02	ego-Facebook	4.039	88.234	3.719	75,32
GR03	IMDb	56.542	185.347	14.004	12,68
GR04	Amazon	410.236	2.439.437	151,887	67,14
GR05	soc-Pokec	1.632.803	30.622.564	787.030	55,29
GR06	soc-LiveJournal	4.847.571	68.993.773	2.042.215	49,30

Tabela 4 – Grafos Reais

apenas para cada um dos *datasets*, viabilizando a execução com todos os conjuntos de dados planejados.

A Tabela 4 apresenta os grafos reais e suas principais características, como número de vértices e arestas, quantidade de comunidades encontradas pelo ADDIC e o tamanho médio dessas comunidades.

Para melhor visualização dos tempos de execução, os grafos reais foram divididos em dois gráficos, conforme apresentado na Figura 35, com o tempo dado em minutos e a variação de configuração do *cluster* em 1, 2, 4, 8 e 16 nós.

Para esta classe de grafos, mais de 1.500 horas de experimentos foram executadas. Esses experimentos puderam ser executados em paralelo com várias configurações de *cluster* simultâneas. Exemplo: Cluster1 (1 máquina), Cluster2 (2 máquinas), Cluster3 (4 máquinas) e Cluster 4 (8 máquinas).

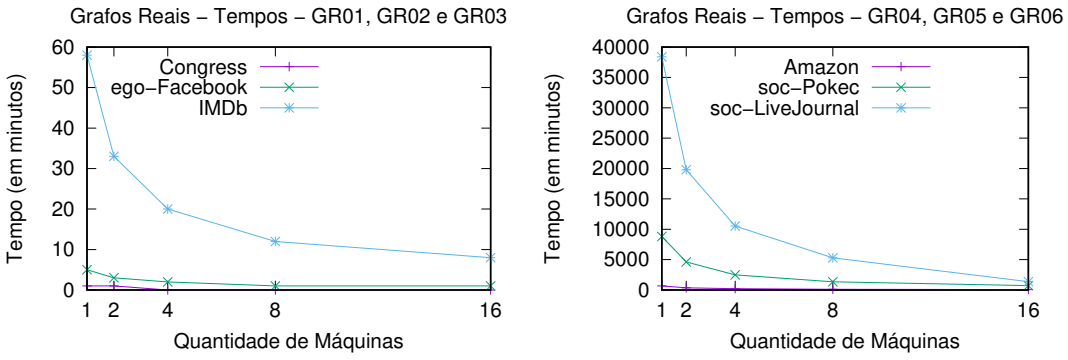


Figura 35 – Tempos do Algoritmo ADDIC - Grafos Reais.

Nos gráficos apresentados na Figura 35, pode-se perceber que, para todos os grafos utilizados, o tempo de execução dos experimentos vai diminuindo à medida que mais máquinas vão sendo adicionadas ao *cluster*. Observa-se que, aparentemente, o tempo pode continuar decrescente ao acrescentar mais recursos ao *cluster*, no entanto, os experimentos realizados foram limitados a dezesseis máquinas, conforme os recursos computacionais que estavam disponíveis para a execução dos experimentos.

Id	Nome	Vértices	Arestas	Comunidades	Tam. Médio
GS01	Sint1	100.000	500.000	112.020	69,20
GS02	Sint2	500.000	2.500.000	583.503	71,47
GS03	Sint3	1.000.000	5.000.000	1.287.807	70,14
GS04	Sint4	5.000.000	25.000.000	6.105.014	73,20
GS05	Sint5	10.000.000	50.000.000	11.988.032	74,13

Tabela 5 – Grafos Sintéticos

6.1.2 Grafos Sintéticos

Esta subseção apresenta os resultados dos experimentos realizados com os grafos sintéticos, que são bases de dados em formato de vértices e arestas criadas por Geradores de Grafos.

Neste trabalho foi utilizado o gerador de grafos criado por (KANG et al., 2011b) e disponível na Web[1], baseado no modelo de Erdös-Rényi (ERDöS; RéNYI, 1959). A Tabela 5 apresenta as características dos grafos sintéticos (número de vértices e arestas, quantidade de comunidades e tamanho médio dessas comunidades) que foram utilizados nesta categoria de experimentos.

Os experimentos com os grafos sintéticos foram inicialmente realizados com três repetições, porém isso foi feito apenas para os grafos GS01 e GS02. Essa estratégia foi abandonada devido ao elevado tempo que é necessário para a realização de cada uma das repetições e também porque foi observado variações muito pequenas no tempo de execução, em alguns casos de 0,001%.

Na categoria de grafos sintéticos, assim como foi feito com os grafos reais, os *datasets* foram divididos em dois gráficos para melhor visualização dos tempos de execução, conforme apresentado na Figura 36, com o tempo dado em minutos e a variação de configuração do *cluster* em 1, 2, 4, 8 e 16 nós.

Para esta classe de grafos, aproximadamente 1.700 horas de experimentos foram executados paralelamente em configurações de *cluster* simultâneas.

6.1.3 Análise dos Resultados

Nesta subseção os resultados de tempo de execução dos grafos reais e sintéticos são analisados mais detalhadamente. As Tabelas 6 e 7 apresentam, para cada um dos grafos reais e sintéticos respectivamente, o identificador do conjunto de dados (Id), a soma de vértices e arestas (V + A) e o tempo de execução em minutos realizado com o *cluster* configurado com o total de dezesseis nós. Além disso, a quarta e quinta colunas apresentam as variações do tamanho e tempo do grafo em relação ao primeiro grafo da tabela. Por

[1] http://www.cs.cmu.edu/ukang/HADI/

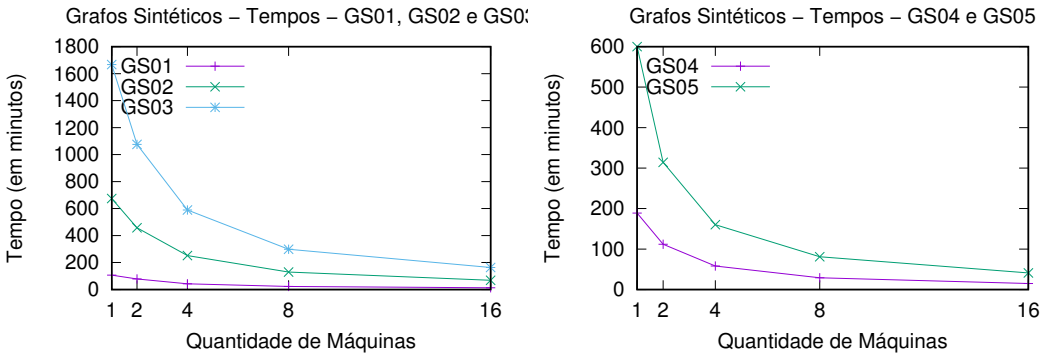


Figura 36 – Tempos do Algoritmo ADDIC - grafos sintéticos

Id	V+A	Tempo (minutos)	Variação Grafo	Variação Tempo
GR01	14724	0,52	-	-
GR02	92273	1,27	6,27X	2,44X
GR03	241889	8,18	16,43X	15,73X
GR04	2849673	70,68	193,54X	135,92X
GR05	32255367	729,08	2190,67X	1401,08X
GR06	73841344	1385,65	5015,03X	2663,71X

Tabela 6 – Análise de tempos - Grafos Reais

Id	V+A	Tempo (minutos)	Variação Grafo	Variação Tempo
GS01	600000	13,08	-	-
GS02	3000000	69,13	5X	5,28X
GS03	6000000	163,95	10X	12,53X
GS04	30000000	907,52	50X	69,38X
GS05	60000000	2466,35	100X	188,51X

Tabela 7 – Análise de tempos - grafos sintéticos

exemplo, o grafo GR02 possui aproximadamente 6,27 vezes o tamanho do grafo GR01 e seu tempo de execução foi 2,44 vezes o tempo de execução do grafo GR01.

Diante dos resultados apresentados para os grafos reais, pode-se observar que não existe uma relação de proporcionalidade entre a variação do tamanho do grafo e a variação do tempo. Em grafos reais isso é mais esperado, uma vez que, como está se detectando comunidades, não há garantias que a quantidade e tamanho das comunidades detectadas acompanhem o aumento do tamanho do grafo. O que chama mais a atenção na Tabela 6 é que a variação de tempo ficou sempre menor que a variação do tamanho do grafo, em todos os cinco casos.

No caso dos grafos sintéticos, pode-se observar que, assim como nos grafos reais, não existe uma relação clara de proporcionalidade na comparação entre a variação do tamanho do grafo e tempo. A variação do tempo sempre esteve acima da variação do tamanho do grafo e cresce a medida que o tamanho do grafo aumenta. No caso do GS02,

há um aumento de 5 vezes no tamanho do grafo e 5,28 vezes no tempo, o que resulta em 5,60% a mais na variação do tempo. Para os grafos GS03, GS04 e GS05, há 25,30%, 38,76% e 88,51% respectivamente.

Ao utilizar o gerador de grafos, é necessário informar o grau médio dos vértices e isso, nos grafos sintéticos, pode ter influenciado para a criação de grafos mais uniformes, no que diz respeito ao tamanho e quantidade de comunidades.

6.2 Consumo de Recursos

Nesta seção, será analisado o consumo de recursos do *cluster* (CPU e Memória RAM) durante a execução dos experimentos com os grafos reais e sintéticos. No monitoramento dos recursos foi utilizada a ferramenta Ganglia (MASSIE; CHUN; CULLER, 2004). Para essa atividade, foi utilizado um nó exclusivo do *cluster* que ficou responsável por coletar e armazenar as informações referentes ao uso dos recursos.

6.2.1 Grafos Reais

Para os grafos reais, será apresentado, inicialmente, o consumo de CPU. Para isso, foi feita uma média simples do consumo de cada uma das máquinas, em busca de um resultado geral para todo o *cluster*. A Figura 37 apresenta os resultados de consumo de CPU. Os grafos reais foram divididos em dois gráficos para melhor visualização dos resultados.

Por meio da ferramenta Ganglia, também foi coletado o consumo de memória RAM durante a execução de todos experimentos a apresentada a média de consumo na Figura 38. Nessa figura, os grafos reais também foram divididos em dois grupos, para melhor visualização dos resultados.

6.2.2 Grafos Sintéticos

Para os grafos sintéticos, o consumo de CPU também foi medido pela ferramenta Ganglia e os resultados estão apresentados na Figura 39. Para essa apresentação, foi realizada uma média simples do consumo de CPU de cada uma das máquinas, para que pudesse ser apresentado um resultado geral do consumo de todo o *cluster*. Os grafos sintéticos foram divididos em dois gráficos para uma melhor visualização dos resultados.

Foi realizado também, para os grafos sintéticos, o monitoramento do consumo de memória RAM em cada um dos nós do *cluster*. Posteriormente, foi feita uma média simples do total desse consumo e o resultado está presente na Figura 40. Nessa figura os grafos sintéticos também foram divididos em dois grupos, para melhor visualização dos resultados.

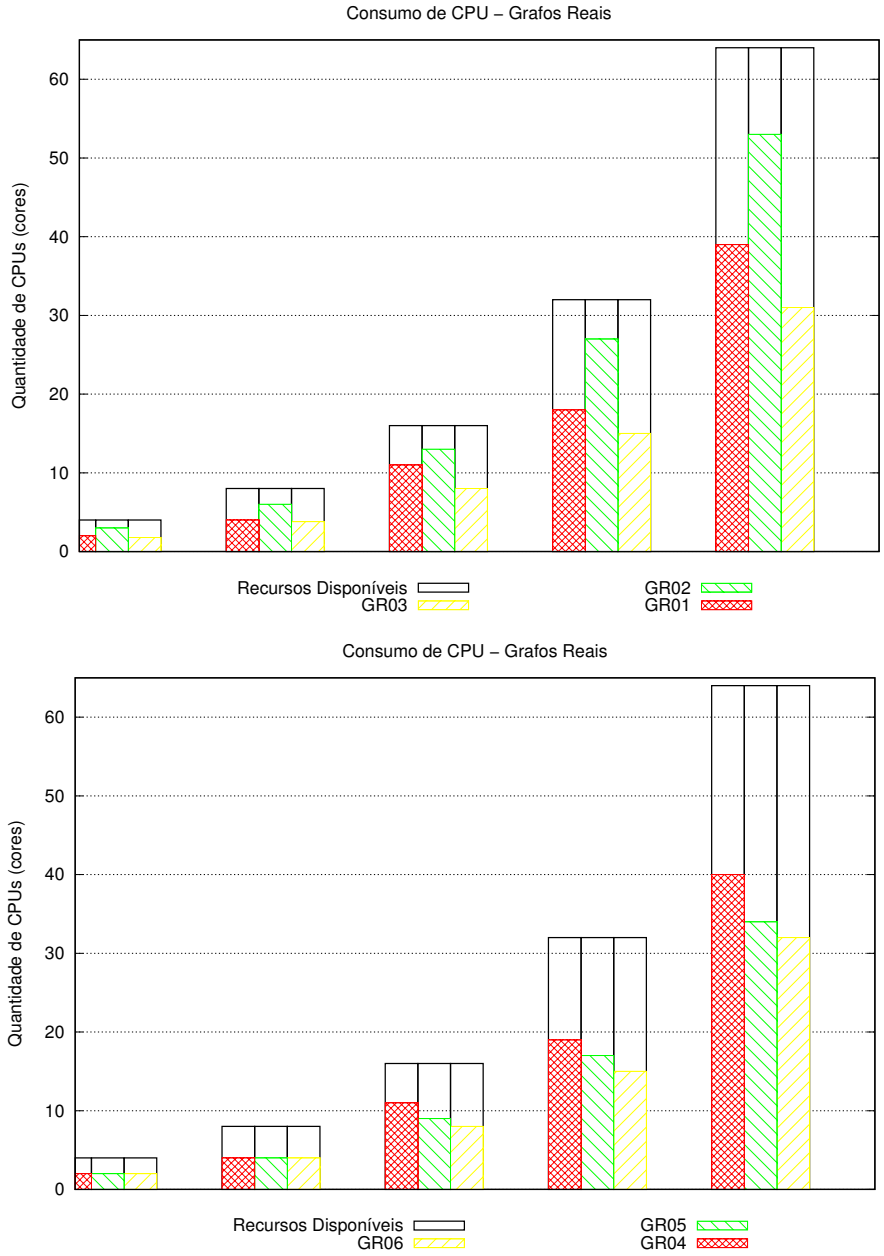


Figura 37 – Resultados do Algoritmo ADDIC - grafos reais - Consumo de CPU.

6.2.3 Análise dos Resultados

Será iniciada a análise dos resultados do consumo de recursos do *cluster*, destacando a CPU. A média de consumo de CPU nos grafos reais foi um pouco menor que nos grafos sintéticos. Percebe-se que existe um impacto do tamanho das comunidades encontradas no uso de CPU. Quanto maior a média de tamanho das comunidades que o algoritmo ADDIC encontra no grafo, mais a CPU é exigida.

Nos grafos reais, conforme apresentado na Tabela 4, há o grafo GR02 como aquele que possui o maior tamanho médio de comunidades (75,21) e, consequentemente, o consumo de CPU para esse grafo foi maior do que os outros. Isso se deve ao fato da função

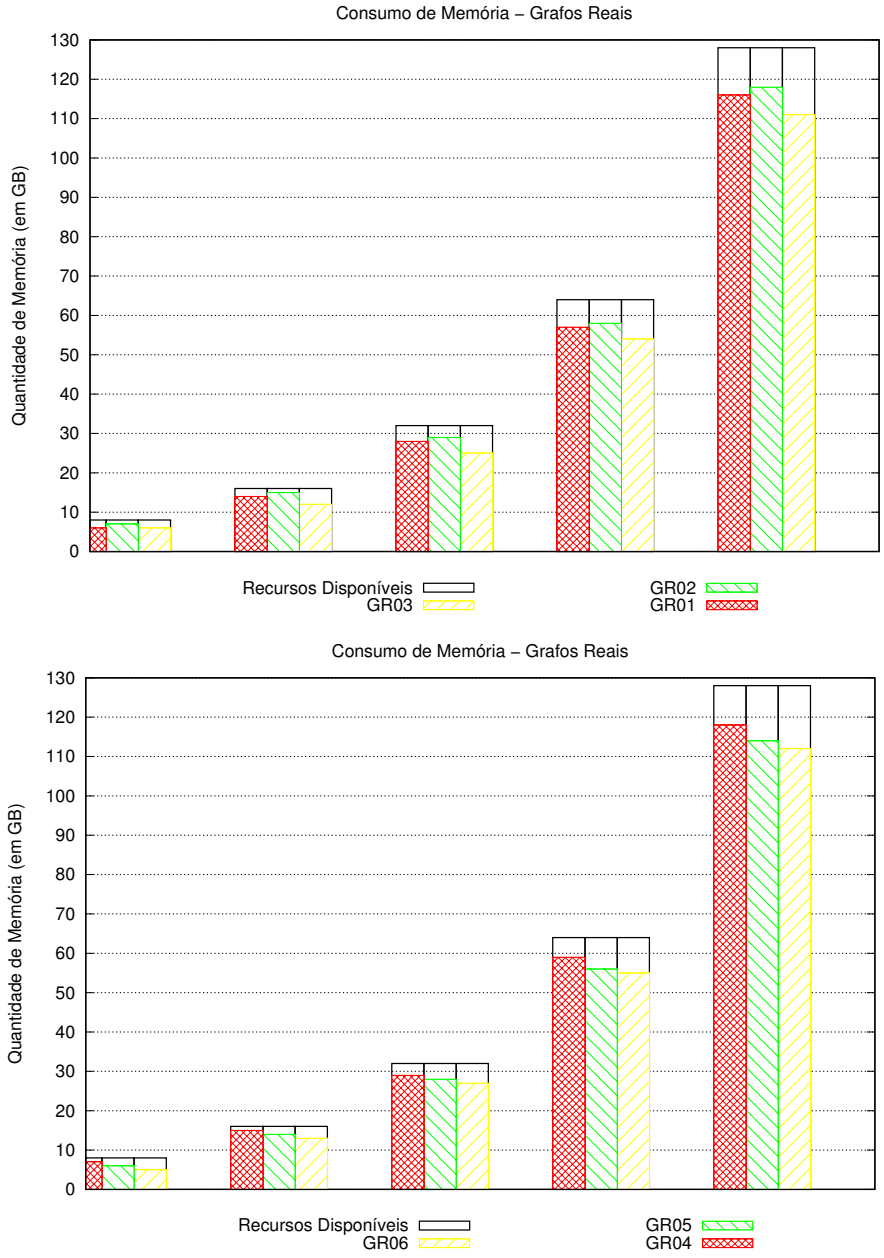


Figura 38 – Resultados do Algoritmo ADDIC - grafos reais - Consumo de Memória RAM.

Merge do algoritmo ADDIC ter sido mais exigida, pois com mais quantidade de elementos na comunidade, mais dados precisam ser mesclados. Por outro lado, corroborando com esta afirmação, há o grafo real GR03 como aquele que menos consumiu CPU. O GR03 é o grafo real com menor tamanho médio de comunidade: 12,68.

Ainda destacando o consumo de CPU, porém agora para os grafos sintéticos, observa-se que o consumo médio foi ainda maior. Isso é justificável, pois, como mostra a Tabela 5, a quantidade de comunidades e seus tamanhos médios são maiores que as comunidades dos grafos reais. Isso, como já dito anteriormente, está diretamente ligado ao gerador de grafos utilizado, pois os grafos sintéticos possuem um grau médio dos

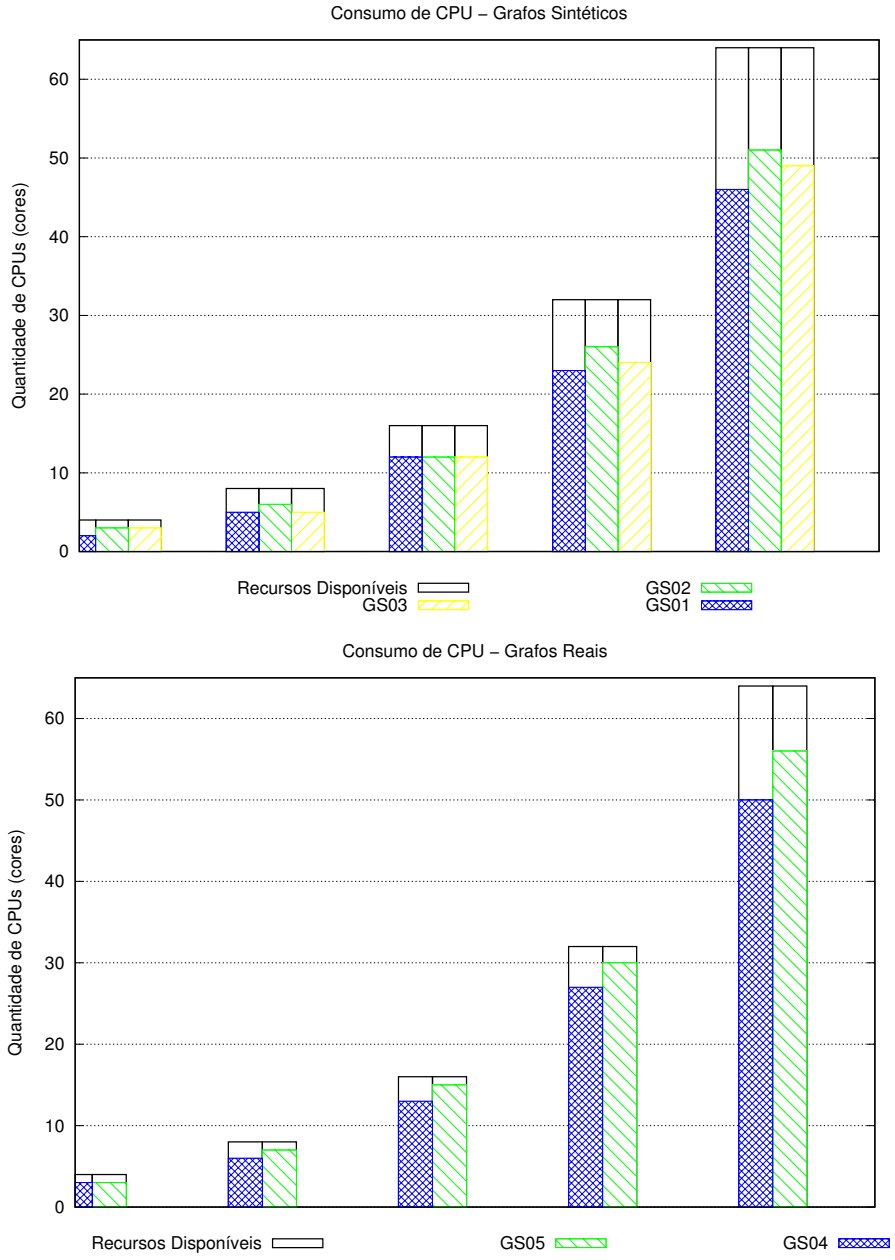


Figura 39 – Resultados do Algoritmo ADDIC - grafos sintéticos - Consumo de CPU.

vértices maior que os grafos reais. Como consequência, o algoritmo ADDIC encontra mais comunidades nessa classe de grafos. Observar-se que os grafos desta categoria que possuem tamanho médio das comunidades maior, também apresenta consumo de CPU maior, como é o caso do grafo GS05 (74,13). O grafo GS01 possui o menor tamanho médio de comunidade (68,20) e, consequentemente, o menor consumo de CPU da categoria.

No caso do uso de memória RAM. o consumo médio para os grafos reais e sintéticos foi ainda maior que o de CPU. Na maioria dos casos esse consumo ficou acima de 90%, o que aumenta a eficiência no uso dos recursos do *cluster*. Isso se deve ao fato do Apache Spark e seus RDDs priorizarem o uso de memória RAM em cada um dos nós do *cluster*,

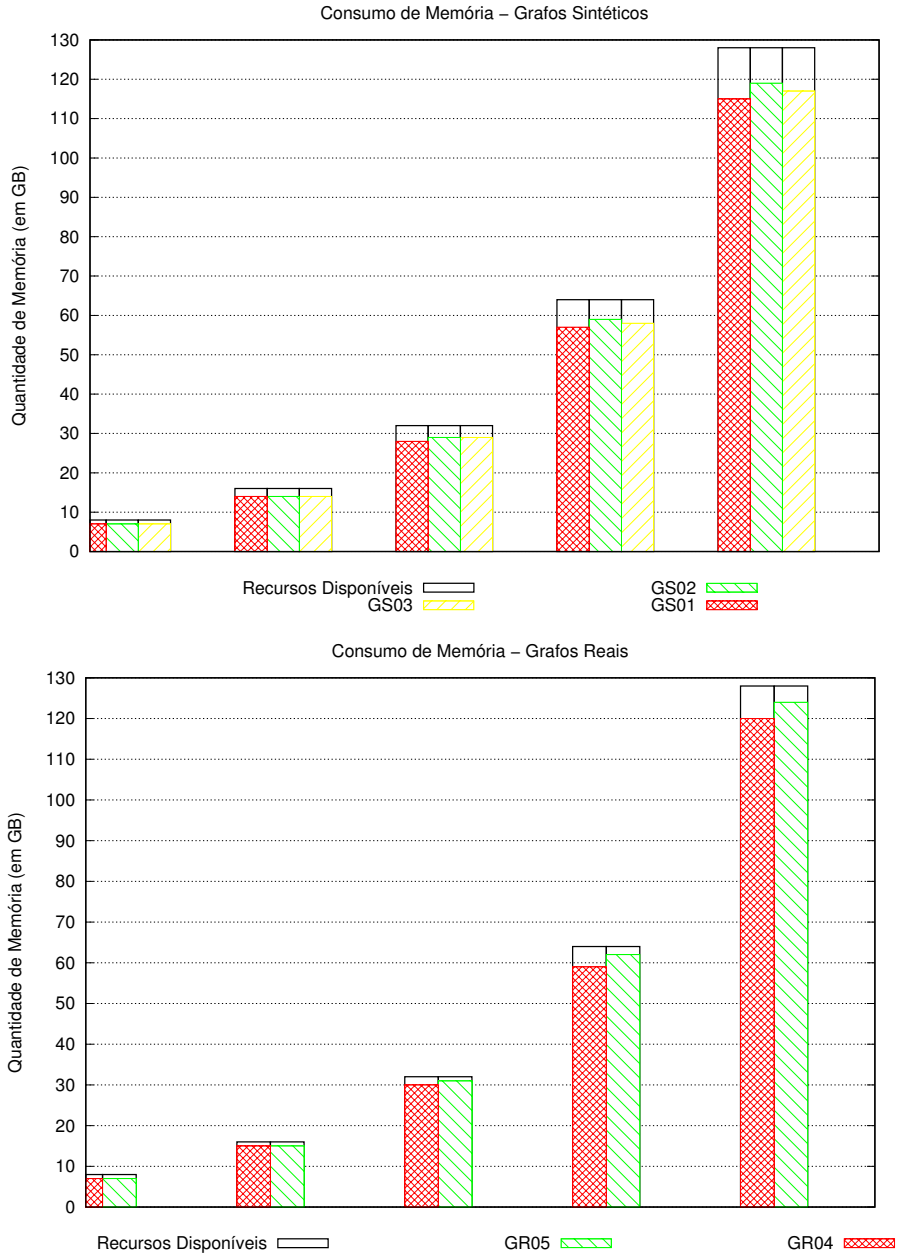


Figura 40 – Resultados do Algoritmo ADDIC - grafos sintéticos - Consumo de Memória RAM.

em detrimento ao uso de disco, como acontece em outros *frameworks*, como é o caso do Apache Hadoop MapReduce, por exemplo.

O consumo de memória RAM, em ambas as categorias de grafos, apresentou o mesmo comportamento do consumo de CPU. Aqueles grafos que possuem o tamanho médio de comunidades maior, também apresentaram maior consumo médio de memória RAM.

Por último, destaca-se que a quantidade de comunidades em conjunto com os tamanhos médios dessas comunidades têm impacto direto no consumo de recursos do

cluster. Diante das características do algoritmo ADDIC, isso é esperado, pois quanto maior as comunidades, mais custosa será a execução da etapa *Merge*, pois são mais vértices para serem concatenados. Não é o tamanho do grafo ou a quantidade de comunidades de maneira isolada que faz esse consumo aumentar.

6.3 Escalabilidade

A escalabilidade tornou-se uma medida importante e desejável em métodos que detectam comunidades, principalmente devido ao aumento da população que utiliza serviços de redes sociais (BUZUN et al., 2014). O algoritmo ADDIC beneficia-se do RDD (*(Resilient Distributed Dataset)* implementado na tecnologia Apache Spark, que contribui diretamente para a escalabilidade do algoritmo, ao processar os dados prioritariamente em memória.

Neste trabalho, para realizar a medida de escalabilidade, será utilizado o cálculo do *Speedup*. O *Speedup* é uma métrica que nos diz o quão mais rápido uma tarefa irá executar usando um computador com alguma melhoria em relação ao computador sem a melhoria (PATTERSON; HENNESSY, 2007). O *Speedup* é apresentado pela Equação 4.

$$S(p) = \frac{1}{(1 - pctPar) + \frac{pctPar}{p}} \tag{4}$$

A medida do *Speedup* do algoritmo ADDIC foi realizada para todos os grafos (reais e sintéticos). O cálculo do *Speedup* foi realizado executando-se cada grafo em cada configuração de *cluster* proposta no trabalho, sendo 1, 2, 4, 8, e 16 máquinas.

A Figura 41 apresenta a escalabilidade do algoritmo ADDIC para a categoria de grafos reais, além de uma linha representando a linearidade. Observa-se que o grafo GR06 alcançou 96,93% da linearidade para o *cluster* configurado com duas máquinas. Esse valor foi o mais próximo da linearidade que se conseguiu alcançar com os grafos reais, entretanto vários outros experimentos com essa categoria de grafo alcançaram bons resultados de *Speedup*, quando comparado com a linearidade. São os casos do grafo GR05 com 95,19% e GR04 com 89,87%. Vale destacar que os grafos que possuem os tamanhos maiores são também aqueles que apresentam melhores resultados de escalabilidade. Isso se deve ao fato de que essa característica do grafo (maior tamanho) faz com que o *cluster* permaneça com uma taxa de ocupação maior dos recursos, o que melhora a escalabilidade do algoritmo.

Os resultados de escalabilidade dos grafos sintéticos foram muito próximos aos resultados da categoria dos grafos reais, conforme apresentado na Figura 42. O melhor resultado apresentado foi com o grafo GS05 que apresentou 95,50% do percentual da linearidade. Vale destacar que outros grafos sintéticos também ficaram bem próximos da linearidade, como é o caso do grafo GS04 com 84,20% e GS03 com 77,57%.

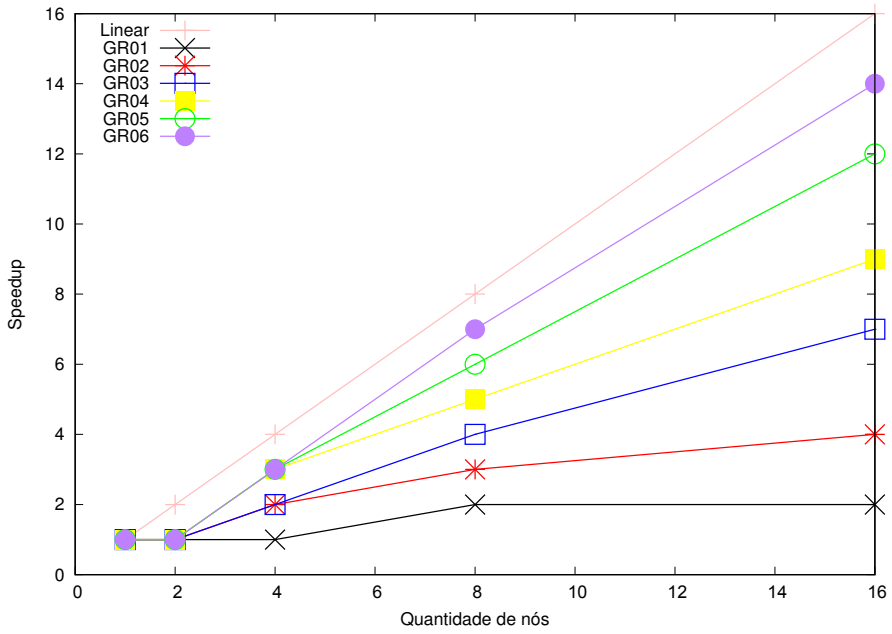


Figura 41 – Resultado ADDIC - *Speedup* - grafos reais.

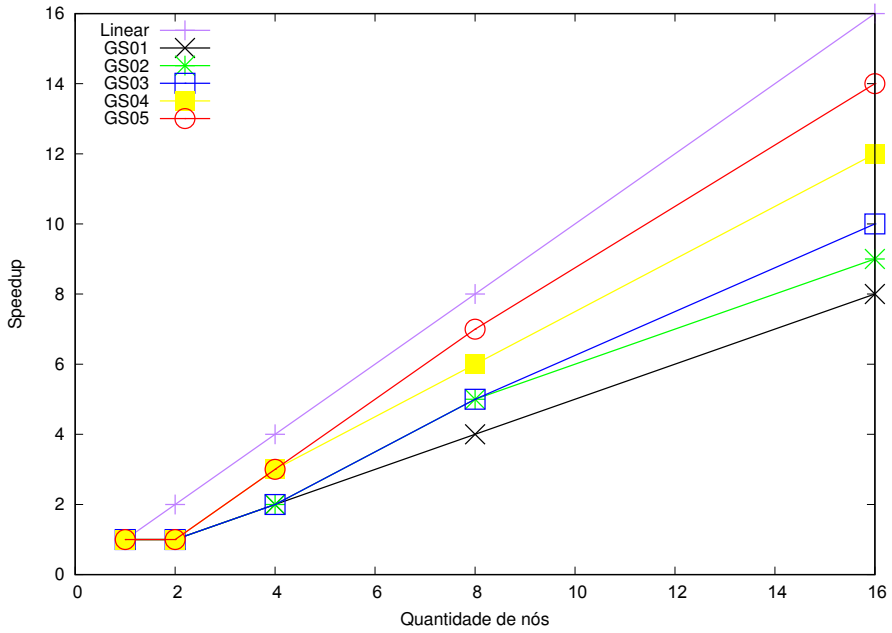


Figura 42 – Resultado ADDIC - *Speedup* - *grafos sintéticos*.

Por fim, na comparação da escalabilidade entre os grafos reais e grafos sintéticos, conclui-se que os resultados apresentados pelos grafos reais são mais eficientes, pois estão mais próximos da linearidade que é tida por alguns autores como a Linearidade Ótima. Mesmo assim, chama a atenção existirem alguns grafos sintéticos que são maiores que os grafos reais e mesmo assim possuem um *speedup* menor. O que explica esse acontecimento é o fato de as bases sintéticas poderem ter uma estrutura que, mesmo que haja o ajuste do gerador para que o resultado fique o mais próximo possível da realidade,

não condiz totalmente com a realidade, no que diz respeito aos graus dos vértices e também as comunidades geradas.

6.4 Eficiência

A eficiência é uma métrica diretamente relacionada ao *Speedup* e determina quão bem utilizados estão os recursos computacionais. Para calcular a eficiência de uma execução paralela, deve-se calcular o *Speedup* e dividí-lo pelo número de máquinas utilizadas. O número é expresso em percentagem. Seja $S(p)$ o *Speedup*, p o número de máquinas, $T(1)$ o tempo de execução do algoritmo em apenas uma máquina do *cluster* e $T(p)$ o tempo de execução em p máquinas, a eficiência $E(p)$ é dada pela Equação 5.

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)} \tag{5}$$

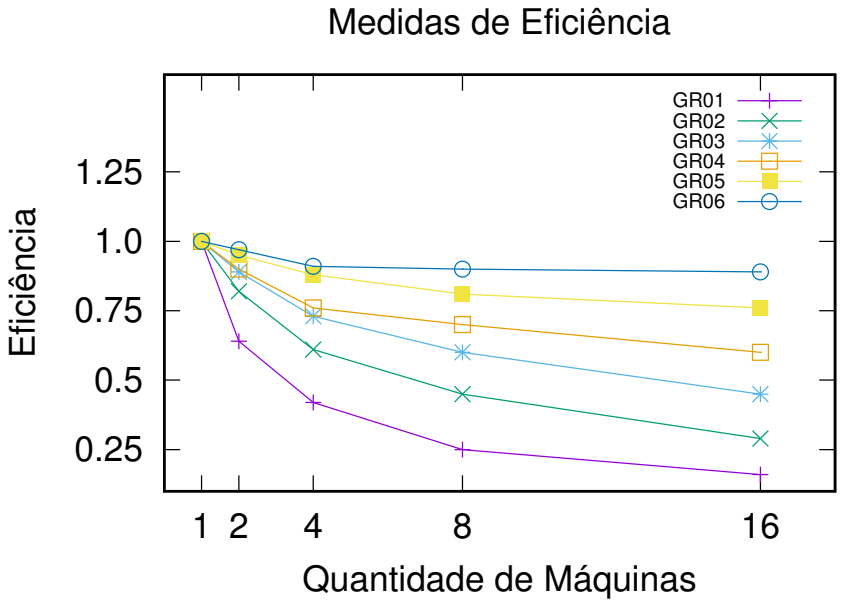


Figura 43 – Resultado ADDIC - *Eficiência - grafos reais*.

A Figura 43 apresenta os percentuais de eficiência para o conjunto de grafos reais. Grande parte dos grafos reais apresentam boas taxas de eficiência, principalmente GR04 e GR05 que apresentam taxas acima de 80% para todas as configurações de *cluster*. O GR05 apresentou taxas acima de 89% para todas as configurações de *cluster*.

A Figura 44 apresenta os resultados de eficiência para os grafos sintéticos. Para esse conjunto de grafos, obteve-se percentuais de eficiência ainda melhores. O Grafo GS05, por exemplo, apresentou taxas de eficiência acima de 90% para todos as configurações de *cluster*, o que caracteriza que o algoritmo ADDIC faz um bom uso dos recursos disponíveis.

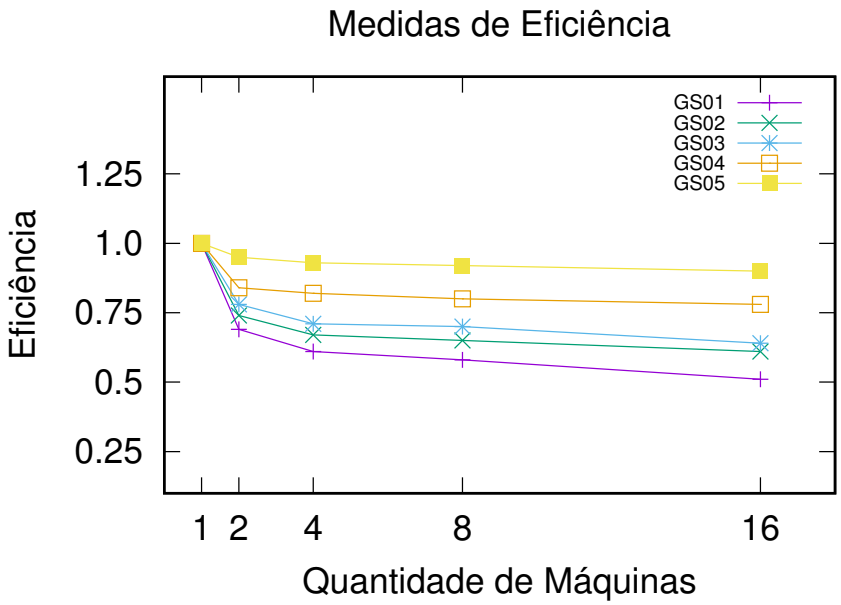


Figura 44 – Resultado ADDIC - *Eficiência - grafos sintéticos*.

Na Seção 6.3, foi apresentado que os grafos maiores apresentavam melhores resultados de *Speedup*, e isso impacta diretamente na Eficiência. A medida que obtem-se melhores taxas de *Speedup* os resultados de Eficiência também aumentam, pois isso significa que está utilizando mais os recursos computacionais disponíveis.

6.5 Comparação com o Algoritmo Sequencial

Esta seção tem o objetivo de comparar o algoritmo ADDIC com outro algoritmo da literatura que trabalha com detecção de comunidades sobrepostas em grandes grafos. O algoritmo escolhido foi o DEMON (COSCIA et al., 2012). Uma das limitações do DEMON é que ele não tem a capacidade de paralelização e distribuição dos dados ao longo de um *cluster* de máquinas. Isso impede que esse algoritmo encontre comunidades em bases de dados realmente grandes, como é o caso de bases de grandes redes sociais, por exemplo.

A ideia inicial era que o algoritmo sequencial DEMON fosse comparado com o algoritmo ADDIC usando todos os conjuntos de dados (reais e sintéticos). Durante os experimentos, alguns grafos simplesmente não executaram no DEMON, pois demandaram mais recursos computacionais do que a capacidade disponível, causando travamento na máquina e problemas com memória. Diante dessa situação, a Tabela 7 apresenta os grafos que finalizaram a execução no DEMON. Observa-se que foi possível executar o DEMON apenas para uma base de dados sintética, pois essas bases começam com tamanho de 600.000 (V+A).

A Figura 45 apresenta a comparação de tempo de execução do algoritmo ADDIC e

Id	Nome	Tempo Sequencial (hh:mm:ss)
GR01	Congress	00:00:53
GR02	ego-Facebook	00:04:12
GR03	IMDb	00:37:50
GR04	Amazon	08:36:45
GS01	Sint1	01:01:06

Tabela 8 – Comparação entre ADDIC e Algoritmo Sequencial.

o algoritmo sequencial, utilizando os grafos reais GR01 e GR02. Observa-se que o grafo GR01 apresenta um desempenho melhor com o algoritmo sequencial, porém ao configurar o *cluster* com 4 máquinas, o algoritmo ADDIC começa a apresentar um melhor desempenho e o tempo de resposta vai melhorando até a última configuração, com dezesseis máquinas. Já para o grafo GR02, o ADDIC é melhor já a partir da segunda máquina adicionada ao *cluster*.

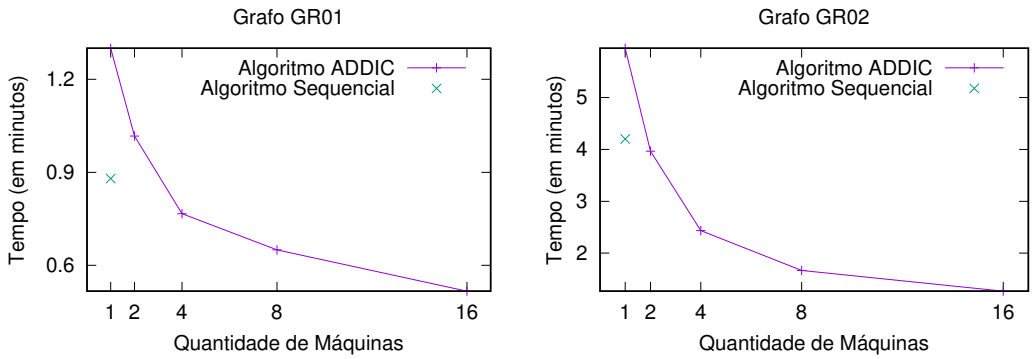


Figura 45 – Comparação entre algoritmos ADDIC e sequencial - Grafos GR01 e GR02

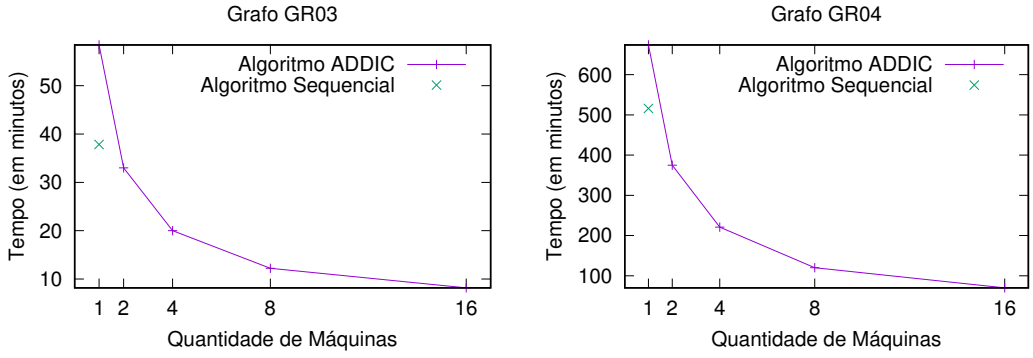


Figura 46 – Comparação entre algoritmos ADDIC e sequencial - Grafos GR03 e GR04

A Figura 46 apresenta a comparação entre o ADDIC e o algoritmo sequencial para os grafos reais GR03 e GR04. Observa-se que, para ambos os grafos, o algoritmo ADDIC apresentou melhor desempenho a partir da segunda máquina adicionada ao *cluster* e permaneceu melhorando o tempo até o número máximo de máquinas. Para o grafo GR03,

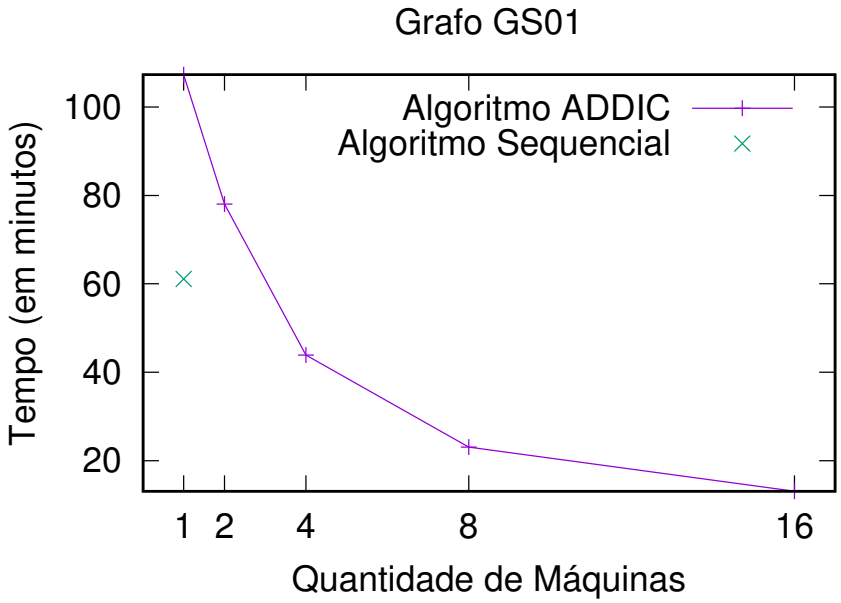


Figura 47 – Comparação entre algoritmos ADDIC e sequencial - grafo sintético GS01

executado com o *cluster* configurado com dezesseis máquinas, o ADDIC foi 3,62 vezes melhor que o algoritmo DEMON. Para o grafo GR04 essa taxa foi de 6,31 vezes. Essa diferença de tempo de execução entre DEMON e ADDIC tende a aumentar, à medida que o tamanho do grafo também aumenta. Infelizmente essa comparação é limitada, pois não foi possível realizá-la para grafos ainda maiores, como é o caso dos grafos reais GR05 e GR06.

Por último, a Figura 47 apresenta o resultado do único grafo sintético que executou até o fim no algoritmo sequencial. Assim como aconteceu com os grafos reais, o algoritmo ADDIC apresentou melhor desempenho a partir da inclusão da quarta máquina no *cluster*. Para esse grafo (GS01), a execução no *cluster* configurado com dezesseis máquinas foi aproximadamente 3,67 vezes mais rápido do que o algoritmo sequencial DEMON. Isso ajuda a destacar a importância do algoritmo ADDIC, pois além de proporcionar essa melhoria no tempo de execução, ele também pode ser usado para grafos realmente grandes, que são inviáveis de serem processados no DEMON.

A ferramenta Spark foi projetada para processar grandes massas de dados e para executar em um *cluster* com várias máquinas (WHITE, 2009). Por isso, o Spark apresenta *overhead* ao executar em poucas máquinas. Esse *overhead* é gerado devido ao particionamento dos RDDs que precisam ser processados e armazenados distribuidamente. Esse é o motivo do algoritmo ADDIC apresentar tempo de execução melhor que o algoritmo sequencial somente a partir de algumas máquinas adicionadas ao *cluster*.

6.6 Considerações Finais

Neste capítulo foram apresentados os resultados dos experimentos executados com o ADDIC para cada um dos tipos de grafos de teste, reais e sintéticos. Os resultados dos experimentos foram comparados com o algoritmo sequencial DEMON, presente na literatura.

Os resultados experimentais mostraram que:

1. O ADDIC executa distribuidamente e encontra comunidades sobrepostas em grandes grafos.
2. Os tempos de resposta do algoritmo ADDIC variam de acordo com a quantidade e tamanho das comunidades presentes no grafo, além do número de vértices e arestas.
3. O *Speedup* do algoritmo é próximo ao linear para alguns grafos experimentados, principalmente aqueles de maior tamanho.
4. A medida de eficiência do algoritmo ADDIC alcançou o valor máximo de 0,97.
5. O algoritmo ADDIC foi projetado para executar grafos de tamanho grande. Grafos pequenos podem apresentar tempos de execução piores que os tempos do algoritmo sequencial, pois existe o *overhead* do *framework* Apache Spark, que é um custo adicional para a instância das tarefas, gerenciamento dos nós do cluster e distribuição dos trabalhos. Esse *overhead* pode inviabilizar o processamento desses grafos.
6. O algoritmo ADDIC se apresenta como uma alternativa para processamento de grafos realmente grandes, pois um dos principais algoritmos da literatura utilizados para descoberta de comunidades sobrepostas (DEMON) não tem capacidade de processar esses grafos, devido sua característica sequencial.

O próximo capítulo apresenta as conclusões deste trabalho e os trabalhos futuros.

7 Conclusão

A programação paralela e distribuída esteve sempre presente nos estudos da computação. No entanto, a ampla disponibilidade de máquinas paralelas está sendo experimentada pela primeira vez na história da computação. Neste contexto, a programação paralela e distribuída tem um apelo sem precedentes. Os problemas modelados por meio da Teoria de Grafos também apresentam um grande avanço ao proporcionar métricas que permitem o avanço no estudo de Redes Complexas. Os *frameworks* de processamento paralelo e distribuído, tais como o Apache Hadoop/MapReduce e Apache Spark, surgiram como um auxílio aos desenvolvedores de *software*, principalmente ao proporcionarem automaticamente o controle da paralelização e distribuição do processamento, o gerenciamento do *cluster* e a tolerância a falhas. A combinação entre grafos grandes, programação paralela e distribuída e os *frameworks* inspirou a condução deste trabalho.

Neste momento de conclusão do trabalho, se torna muito importante revisitar todos os objetivos que foram traçados no início do Capítulo 1 deste projeto e confrontá-los com os resultados obtidos, apresentando as principais contribuições deste projeto.

O objetivo central traçado para este trabalho foi planejar, modelar e desenvolver um algoritmo paralelo e distribuído baseado no método de *Label Propagation*, no conceito de Redes Ego e no *framework* Apache Spark. Este objetivo foi alcançado pois, neste trabalho, foi desenvolvido o ADDIC (Algoritmo para Detecção Distribuída de Comunidades) que é um algoritmo paralelo e distribuído, criado para ser executado no ambiente Apache Spark.

O objetivo central ia além e propunha produzir resultados inéditos de pesquisa que levassem a novas técnicas e/ou ferramentas para Deteccção de Comunidades Sobrepostas. O algoritmo ADDIC é um algoritmo inédito que detecta comunidades sobrepostas em grafos grandes, que utiliza o Apache Spark como suporte de execução, é baseado em Redes Ego e na técnica de *Label Propagation*. O Capítulo 1 ainda destacava quatro objetivos específicos e cada um deles são discutidos em seguida.

O primeiro objetivo específico propôs validar os resultados do algoritmo criado com resultados de outra publicação de mesmo fim presente na literatura, no que diz respeito a quantidade e tamanho das comunidades encontradas. O ADDIC foi baseado na ideia do algoritmo DEMON e diversos experimentos foram executados utilizando os dois algoritmos. Em seguida, os resultados de quantidade de comunidades sobrepostas encontradas e tamanho dessas comunidades foram comparados. O algoritmo ADDIC apresentou resultados idênticos ao DEMON, o que atesta a correção dos resultados apresentados pelo novo algoritmo.

O próximo objetivo diz respeito a comparação do tempo de execução do algoritmo

ADDIC com um algoritmo sequencial presente na literatura, no caso o DEMON, utilizando todos os conjuntos de dados em formato de grafos reais e sintéticos. Não foi possível comparar todos os conjuntos de dados entre os dois algoritmos, pois o DEMON não suporta grafos muito grandes e apresentou travamento e estouro de memória em grande parte dos experimentos. Foram realizados mais de 3.200 horas de experimentos com o algoritmo ADDIC e, ao comparar aqueles grafos que foram possíveis de serem executados no algoritmo sequencial DEMON, observou-se que o ADDIC apresentou resultados até 6,31 vezes melhores, a partir do *cluster* configurado com duas ou mais máquinas. Sabe-se que o Apache Spark apresenta um *overhead* inicial devido ao particionamento dos RDDs que precisam ser processados e armazenados distribuidamente. Esse é o motivo do algoritmo ADDIC apresentar tempo melhor que o DEMON apenas após adicionar algumas máquinas ao *cluster*.

O terceiro objetivo específico deste trabalho propôs verificar e avaliar métricas de escalabilidade e eficiência do algoritmo ADDIC a medida que máquinas vão sendo adicionadas ao *cluster*, por meio da métrica de *Speedup*. Os experimentos foram realizados com o *cluster* configurado com 1, 2, 4, 8 e 16 máquinas. Com isso, realizou-se o cálculo do *Speedup* e da Eficiência do algoritmo ADDIC. Foi realizada uma comparação com a Escalabilidade Linear que é tida como a escalabilidade ótima por diversos autores na literatura e o ADDIC apresentou valores bem próximos a essa linearidade para diversos grafos experimentados, com destaques para o grafo real GR06 que alcançou 96,93% e o grafo sintético GS05, que alcançou 95,50% desta linearidade, ambos com o *cluster* configurado com duas máquinas. No que diz respeito a taxa de eficiência, o ADDIC apresentou resultados acima de 90% em todos os experimentos realizados com o grafo GS05, que foi o maior grafo utilizado, o que garante que o ADDIC faz bom uso dos recursos disponíveis no *cluster*, principalmente aumentando a taxa de eficiência a medida que o tamanho do grafo cresce.

O quarto objetivo específico trata da avaliação do consumo de recursos computacionais do *cluster*, tais como: CPU e Memória RAM. Durante a execução dos experimentos deste trabalho, os recursos computacionais do *cluster* foram monitorados utilizando uma máquina dedicada para esse fim e também a ferramenta Ganglia. No que diz respeito a CPU, o monitoramento demonstrou que o consumo dos grafos reais foram menores que os grafos sintéticos e que o tamanho médio das comunidades tem impacto nesse consumo, ou seja, quanto maior a média de tamanho das comunidades que o ADDIC encontra, mais a CPU é exigida. Uma explicação para isso é que, devido ao tamanho das comunidades encontradas, existe a execução da função *Merge* do algoritmo mais exigida e, com isso, mais consumo de CPU. O consumo de Memória RAM segue a mesma tendência e isso é esperado pois, o Apache Spark prioriza o armazenamento dos dados em memória primária, evitando ao máximo a manipulação de disco rígido.

Por fim, conclui-se que este trabalho alcançou plenamente os objetivos traçados

no início do projeto e que ele se torna bastante contribuitivo ao paralelizar e distribuir um algoritmo sequencial existente na literatura. O ADDIC foi criado para resolver um desafio real e presente em trabalhos importantes da literatura, ao surgir como uma opção de processamento paralelo e distribuído de grafos grandes, na busca por comunidades sobrepostas, utilizando as técnicas de Rede Ego e *Label Propagation*. O ADDIC se apresenta como uma ferramenta importante ao conseguir processar grandes grafos que não são possíveis de serem processados no DEMON que é sua versão sequencial, além de utilizar um modelo de programação emergente (Apache Spark) e, por meio de diversos experimentos, provou ser eficiente e escalável.

7.1 Trabalhos Futuros

Deve-se ressaltar que os experimentos realizados neste trabalho foram processados em um pequeno *cluster* em comparação aos experimentos realizados por outros trabalhos da literatura. É preciso futuramente executar os mesmos experimentos em um *cluster* maior, com o objetivo de verificar o comportamento do algoritmo ADDIC quanto ao tempo de execução, à medida que mais máquinas são adicionadas ao *cluster*.

Para processamento de grafos ainda maiores àqueles que foram processados nesse trabalho, são necessárias máquinas com mais espaço em memória RAM, pois como se sabe, os algoritmos para detecção de comunidades em grandes grafos e implementados em Apache Spark, utilizam grandes quantidades de memória.

Sugere-se ainda, como trabalho futuro, algumas possíveis melhorias no algoritmo ADDIC. A experiência da programação e execução do algoritmo permitiu descobrir que existem muitas formas de realizar essas melhorias. A principal delas é criar uma maneira de distribuir o processamento da função *Merge*. Isso não é uma tarefa simples, pois a junção dos dados é iterativa e todos os nós do *cluster* precisariam de acesso aos dados mesclados pelos outros nós, o que inviabiliza, pelo menos temporariamente, a paralelização e distribuição dessa tarefa. Outra questão diz respeito a otimização do uso dos RDDs. Como os RDDs se tratam de uma estrutura de dados imutável, muitos RDDs são criados com os resultados da função *Label Propagation* e isso ocasiona um maior uso de memória. Deve-se buscar uma maneira de otimizar ainda mais a criação dessas estruturas de dados.

7.2 Considerações Finais

Esse trabalho foi uma grande experiência com programação paralela e distribuída, principalmente com o ambiente Apache Spark. O ambiente de *cluster*, exigido pelo modelo, é um ambiente que tem uma configuração não trivial e que demanda muito tempo para esse fim. Em comparação com outros trabalhos da literatura já consolidados, tais como o DEMON,

EgoLP e Ego-Splitting, este trabalho contou com o ambiente computacional, a equipe de pesquisadores e o tempo de desenvolvimento bastante restritos. No entanto, mesmo frente à essas restrições, foi considerado que o objetivo foi alcançado e os resultados foram satisfatórios.

Referências

AHN, J. P. B. Y.-Y.; LEHMANN, S. Link communities reveal multiscale complexity in networks. **Nature**, Nature Publishing Group, v. 466, n. 7307, p. 761–764, 2010. ISSN 0028-0836. Citado na página 50.

AKIDAU, T. et al. Millwheel: Fault-tolerant stream processing at internet scale. **Proc. VLDB Endow.**, VLDB Endowment, v. 6, n. 11, p. 1033–1044, ago. 2013. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/2536222.2536229>. Citado na página 45.

ALSHEIKH, M. A. et al. Mobile big data analytics using deep learning and apache spark. **IEEE Network**, v. 30, n. 3, p. 22–29, 2016. Disponível em: <http://dx.doi.org/10.1109/MNET.2016.7474340>. Citado na página 45.

AMORETTI ALBERTO FERRARI, P. F. M. M. F. R. M. T. M. Local-first algorithms for community detection. In: . Cagliari, Italy: KDWEB - Knowledge Discovery on the WEB, 2016. Citado na página 50.

ARMBRUST, M. et al. Spark sql: Relational data processing in spark. In: **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2015. (SIGMOD '15), p. 1383–1394. ISBN 978-1-4503-2758-9. Disponível em: <http://doi.acm.org/10.1145/2723372.2742797>. Citado na página 45.

ASANOVIC, K. et al. A view of the parallel computing landscape. **Commun. ACM**, ACM, New York, NY, USA, v. 52, n. 10, p. 56–67, out. 2009. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1562764.1562783>. Citado 4 vezes nas páginas 15, 17, 23 e 24.

ATZORI, L.; IERA, A.; MORABITO, G. The internet of things: A survey. **Comput. Netw.**, Elsevier North-Holland, Inc., New York, NY, USA, v. 54, n. 15, p. 2787–2805, out. 2010. ISSN 1389-1286. Disponível em: <http://dx.doi.org/10.1016/j.comnet.2010.05.010>. Citado 2 vezes nas páginas 15 e 27.

BAZGAN, C.; TUZA, Z.; VANDERPOOTEN, D. The satisfactory partition problem. **Discrete Appl. Math.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 154, n. 8, p. 1236–1245, maio 2006. ISSN 0166-218X. Disponível em: <https://doi.org/10.1016/j.dam.2005.10.014>. Citado na página 31.

BEI, Z. et al. Rfhoc: A random-forest approach to auto-tuning hadoop's configuration. **IEEE Transactions on Parallel e Distributed Systems**, IEEE Computer Society, Los Alamitos, CA, USA, v. 27, n. 5, p. 1470–1483, 2016. ISSN 1045-9219. Citado na página 19.

BHOWMICK, S.; SRINIVASAN, S. A template for parallelizing the louvain method for modularity maximization. In: ____. **Modeling and Simulation in Science, Engineering and Technology**. [S.l.]: Springer Basel, 2013. (Modeling and Simulation in Science, Engineering and Technology, v. 55), p. 111–124. Citado na página 50.

BONDI, A. B. Characteristics of scalability and their impact on performance. In: **Proceedings of the 2nd International Workshop on Software and Performance**. New York, NY, USA: ACM, 2000. p. 195–203. Citado na página 78.

BONDY, J. A. **Graph Theory With Applications**. [S.l.]: Elsevier Science Ltd, 1976. Citado na página 28.

BORTHAKUR, D. Hdfs architecture guide. **HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf**, p. 39, 2008. Citado na página 36.

BRESHEARS, C. **The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications**. [S.l.]: O'Reilly Media, Inc., 2009. ISBN 0596521537, 9780596521530. Citado 4 vezes nas páginas 9, 24, 25 e 26.

BUZUN, N. et al. Egolp: Fast and distributed community detection in billion-node social networks. In: **2014 IEEE International Conference on Data Mining Workshops, ICDM Workshops 2014, Shenzhen, China, December 14, 2014**. [s.n.], 2014. p. 533–540. Disponível em: <https://doi.org/10.1109/ICDMW.2014.158>. Citado 4 vezes nas páginas 31, 33, 52 e 90.

CAPOTA, M. et al. Graphalytics: A big data benchmark for graph-processing platforms. In: **Proceedings of the GRADES'15**. New York, NY, USA: ACM, 2015. p. 7:1–7:6. ISBN 978-1-4503-3611-6. Disponível em: <http://doi.acm.org/10.1145/2764947.2764954>. Citado na página 47.

CARLINI, E. et al. Balanced graph partitioning with apache spark. In: **Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part I**. [s.n.], 2014. p. 129–140. Disponível em: <http://dx.doi.org/10.1007/978-3-319-14325-5_12>. Citado na página 45.

CHEONG, C. Y. et al. Hierarchical parallel algorithm for modularity-based community detection using gpus. In: **Proceedings of the 19th International Conference on Parallel Processing**. Berlin, Heidelberg: Springer-Verlag, 2013. (Euro-Par'13), p. 775–787. ISBN 978-3-642-40046-9. Disponível em: <http://dx.doi.org/10.1007/978-3-642-40047-6_77>. Citado na página 50.

COSCIA, M. et al. DEMON: a local-first discovery method for overlapping communities. **CoRR**, abs/1206.0629, 2012. Disponível em: <http://arxiv.org/abs/1206.0629>. Citado 13 vezes nas páginas 20, 30, 31, 50, 53, 55, 57, 58, 69, 73, 74, 80 e 93.

CRUZ, L. C.; MURTA, C. Uum algoritmo paralelo eficiente para cálculo de centralidade em grafos. In: **XIV Simpósio em Sistemas Computacionais de Alto Desempenho - WSCAD-SCC 2013**. [S.l.: s.n.], 2013. Citado na página 46.

DEAN, J.; GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In: **Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6**. Berkeley, CA, USA: USENIX Association, 2004. (OSDI'04), p. 10–10. Disponível em: <http://dl.acm.org/citation.cfm?id=1251254.1251264>. Citado 5 vezes nas páginas 17, 24, 34, 36 e 41.

DEAN, J.; GHEMAWAT, S. MapReduce: a Flexible Data Processing Tool. **Commun. ACM**, ACM, New York, NY, USA, v. 53, p. 72–77, Janeiro 2010. Citado na página 24.

DELLING, D. et al. Customizable route planning in road networks. **Transportation Science**, v. 51, n. 2, p. 566–591, 2017. Disponível em: <https://doi.org/10.1287/trsc.2014.0579>. Citado na página 30.

DONG, G. et al. Resilience of networks with community structure behaves as if under an external field. **Proceedings of the National Academy of Sciences**, National Academy of Sciences, v. 115, n. 27, p. 6911–6915, 2018. ISSN 0027-8424. Disponível em: <https://www.pnas.org/content/115/27/6911>. Citado na página 32.

DOUGLAS, L. The importance of'big data': A definition. **Gartner (June 2012)**, 2012. Citado na página 27.

EPASTO, A.; LATTANZI, S.; LEME, R. P. Ego-splitting framework: from non-overlapping to overlapping clusters. In: . [S.l.: s.n.], 2017. Citado 2 vezes nas páginas 31 e 51.

ERDöS, P.; RéNYI, A. On random graphs, I. **Publicationes Mathematicae (Debrecen)**, v. 6, 1959. Citado 3 vezes nas páginas 73, 74 e 83.

FARRUGIA, M.; HURLEY, N.; QUIGLEY, A. A.: Exploring temporal ego networks using small multiples and tree-ring layouts. In: **In Proceedings of the 4th International Conference on Advances in Computer-Human Interactions (2011), ACHI**. [S.l.: s.n.], 2011. Citado na página 32.

FATEMA, K. et al. A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. **Journal of Parallel and Distributed Computing**, v. 74, n. 10, p. 2918 – 2933, 2014. ISSN 0743-7315. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0743731514001099>. Citado na página 76.

Ford, Jr., L. R.; FULKERSON, D. R. Maximal flow through a network. v. 8, p. 399–404, 1956. ISSN 0008-414X (print), 1496-4279 (electronic). Citado na página 31.

FREEMAN, J. et al. Mapping brain activity at scale with cluster computing. **Nature Methods**, v. 11, p. 941–950, 2014. Citado na página 44.

GIUSTO, D. et al. **The Internet of Things: 20th Tyrrhenian Workshop on Digital Communications**. [S.l.]: Springer Publishing Company, Incorporated, 2014. ISBN 148998397X, 9781489983978. Citado na página 27.

GOLDBERG, A. V. **Finding a Maximum Density Subgraph**. [S.l.], 1984. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/1984/5956.html>. Citado na página 30.

GONZALEZ-PARDO, A.; JUNG, J.; CAMACHO, D. Aco-based clustering for ego network analysis. **Future Generation Computer Systems**, v. 66, 07 2016. Citado na página 62.

GONÇALVES, M. R. S.; MACIEL, J. N; MURTA, C. D. Geração de Topologias da Internet por Redução do Grafo Original. XXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), Gramado, Brasil, p. 959–972, 2010. Citado na página 45.

GOPALANI, S.; ARORA, R. Article: Comparing apache spark and map reduce with performance analysis using k-means. **International Journal of Computer Applications**, v. 113, n. 1, p. 8–11, March 2015. Full text available. Citado na página 49.

GROSS, J.; YELLEN, J. **Graph theory and its applications**. Boca Raton, FL, USA: CRC Press, Inc., 1999. Citado na página 30.

GU, L.; LI, H. Memory or time: Performance evaluation for iterative operation on hadoop and spark. In: **10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013**. [s.n.], 2013. p. 721–727. Disponível em: <http://dx.doi.org/10.1109/HPCC.and.EUC.2013.106>. Citado 3 vezes nas páginas 17, 46 e 48.

HEIDER, F. Attitudes and cognitive organization. **Journal of Psychology**, v. 21, p. 107–112, 1946. Citado na página 31.

HOFER, F. Architecture, technologies and challenges for cyber-physical systems in industry 4.0: A systematic mapping study. In: **Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement**. New York, NY, USA: ACM, 2018. (ESEM '18), p. 1:1–1:10. ISBN 978-1-4503-5823-1. Disponível em: <http://doi.acm.org/10.1145/3239235.3239242>. Citado na página 16.

HOLDER, L. B. et al. Current and future challenges in mining large networks: Report on the second sdm workshop on mining networks and graphs. **SIGKDD Explor. Newsl.**, ACM, New York, NY, USA, v. 18, n. 1, p. 39–45, ago. 2016. ISSN 1931-0145. Disponível em: <http://doi.acm.org/10.1145/2980765.2980770>. Citado 10 vezes nas páginas 9, 10, 16, 19, 20, 52, 53, 69, 70 e 71.

HUNG MIGUEL ARAUJO, C. F. S.-C. Distributed community detection on edge-labeled graphs using spark. In: . San Francisco, CA, USA: 12th International Workshop on Mining and Learning with Graphs, 2016. Citado na página 51.

JAIN, R. **The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling**. New York, USA: John Wiley and Sons, 1991. Citado na página 75.

JONSSON, P. F.; BATES, P. A. Global topological features of cancer proteins in the human interactome. **Bioinformatics**, v. 22, n. 18, p. 2291–2297, 07 2006. ISSN 1367-4803. Disponível em: <https://doi.org/10.1093/bioinformatics/btl390>. Citado na página 32.

JONSSON, P. F. et al. Cluster analysis of networks generated through homology: automatic identification of important protein communities involved in cancer metastasis. **BMC Bioinformatics**, v. 7, p. 2, 2006. Disponível em: <https://doi.org/10.1186/1471-2105-7-2>. Citado na página 32.

JU, W. et al. igraph: an incremental data processing system for dynamic graph. **Frontiers of Computer Science**, v. 10, n. 3, p. 462–476, 2016. Disponível em: <http://dx.doi.org/10.1007/s11704-016-5485-7>. Citado na página 47.

JUNGHANNS, M. et al. GRADOOP: scalable graph data management and analytics with hadoop. **CoRR**, abs/1506.00548, 2015. Disponível em: <http://arxiv.org/abs/1506.00548>. Citado na página 46.

KANG, U. et al. Hadi: Mining radii of large graphs. **ACM Trans. Knowl. Discov. Data**, ACM, New York, NY, USA, v. 5, n. 2, p. 8:1–8:24, fev. 2011. ISSN 1556-4681. Disponível em: <http://doi.acm.org/10.1145/1921632.1921634>. Citado 2 vezes nas páginas 45 e 74.

KANG, U. et al. HADI: Mining Radii of Large Graphs. **ACM Trans. Knowl. Discov. Data**, ACM, New York, NY, USA, v. 5, p. 8:1–8:24, February 2011. Citado na página 83.

KANG, U.; TSOURAKAKIS, C. E.; FALOUTSOS, C. Pegasus: A peta-scale graph mining system implementation and observations. In: **Proceedings of the 2009 Ninth IEEE International Conference on Data Mining**. Washington, DC, USA: IEEE Computer Society, 2009. (ICDM '09), p. 229–238. ISBN 978-0-7695-3895-2. Disponível em: <http://dx.doi.org/10.1109/ICDM.2009.14>. Citado na página 45.

KARAU, H. et al. **Learning Spark: Lightning-Fast Big Data Analytics**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2015. ISBN 1449358624, 9781449358624. Citado 2 vezes nas páginas 38 e 39.

KARP, R. Reducibility among combinatorial problems. In: MILLER, R.; THATCHER, J. (Ed.). **Complexity of Computer Computations**. [S.l.]: Plenum Press, 1972. p. 85–103. Citado 3 vezes nas páginas 20, 30 e 31.

KARUN, A. K.; CHITHARANJAN, K. A review on hadoop—hdfs infrastructure extensions. In: IEEE. **Information & Communication Technologies (ICT), 2013 IEEE Conference on**. [S.l.], 2013. p. 132–137. Citado na página 37.

KATAL, A.; WAZID, M.; GOUDAR, R. H. Big data: Issues, challenges, tools and Good practices. In: **Contemporary Computing (IC3), 2013 Sixth International Conference on**. IEEE, 2013. p. 404–409. ISBN 978-1-4799-0190-6. Disponível em: <http://dx.doi.org/10.1109/ic3.2013.6612229>. Citado na página 27.

KAUSHIK, C.; MISHRA, A. A scalable, lexicon based technique for sentiment analysis. **CoRR**, abs/1410.2265, 2014. Disponível em: <http://arxiv.org/abs/1410.2265>. Citado na página 43.

KHAN, Z. et al. Towards cloud based big data analytics for smart future cities. **Journal of Cloud Computing**, v. 4, n. 1, p. 2, 2015. ISSN 2192-113X. Disponível em: <http://dx.doi.org/10.1186/s13677-015-0026-8>. Citado na página 43.

KIM, H. et al. Deepspark: Spark-based deep learning supporting asynchronous updates and caffe compatibility. **CoRR**, abs/1602.08191, 2016. Disponível em: <http://arxiv.org/abs/1602.08191>. Citado na página 45.

KLEINBERG, J. M. An impossibility theorem for clustering. In: BECKER, S.; THRUN, S.; OBERMAYER, K. (Ed.). **Advances in Neural Information Processing Systems 15**. MIT Press, 2003. p. 463–470. Disponível em: <http://papers.nips.cc/paper/2340-an-impossibility-theorem-for-clustering.pdf>. Citado na página 30.

KUMAR, V.; GUPTA, A. Analysis of scalability of parallel algorithms and architectures: a survey. In: **Proceedings of the 5th Iternational Conference on Supercomputing**. New York, NY, USA: ACM, 1991. (ICS '91), p. 396–405. Citado 2 vezes nas páginas 77 e 78.

LANDSET, S. et al. A survey of open source tools for machine learning with big data in the hadoop ecosystem. **Journal of Big Data**, v. 2, n. 1, p. 24, 2015. ISSN 2196-1115. Disponível em: <http://dx.doi.org/10.1186/s40537-015-0032-1>. Citado na página 42.

LANEY, D. **3D Data Management: Controlling Data Volume, Velocity, and Variety**. [S.l.], 2001. Disponível em: <http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf>. Citado na página 27.

LAWLER, E. L. Cutsets and partitions of hypergraphs. **Networks**, v. 3, n. 3, p. 275–285, 1973. Disponível em: <https://doi.org/10.1002/net.3230030306>. Citado na página 30.

LEE, K. et al. ARLS: A mapreduce-based output analysis tool for large-scale simulations. **Advances in Engineering Software**, v. 95, p. 28–37, 2016. Disponível em: <http://dx.doi.org/10.1016/j.advengsoft.2016.01.025>. Citado na página 42.

LESKOVEC, J. et al. Statistical properties of community structure in large social and information networks. In: **Proceedings of the 17th International Conference on World Wide Web**. New York, NY, USA: ACM, 2008. (WWW '08), p. 695–704. ISBN 978-1-60558-085-2. Disponível em: <http://doi.acm.org/10.1145/1367497.1367591>. Citado na página 30.

LESKOVEC, J.; LANG, K. J.; MAHONEY, M. W. Empirical comparison of algorithms for network community detection. **CoRR**, abs/1004.3539, 2010. Disponível em: <http://arxiv.org/abs/1004.3539>. Citado na página 30.

LI, M. et al. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In: **Proceedings of the 12th ACM International Conference on Computing Frontiers**. New York, NY, USA: ACM, 2015. (CF '15), p. 53:1–53:8. ISBN 978-1-4503-3358-0. Disponível em: <http://doi.acm.org/10.1145/2742854.2747283>. Citado na página 45.

LIM, S. et al. Graph processing platforms at scale: Practices and experiences. In: **2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015**. [s.n.], 2015. p. 42–51. Disponível em: <http://dx.doi.org/10.1109/ISPASS.2015.7095783>. Citado na página 47.

LIN, J.; SCHATZ, M. Design patterns for efficient graph algorithms in mapreduce. In: **Proceedings of the Eighth Workshop on Mining and Learning with Graphs**. New York, NY, USA: ACM, 2010. (MLG '10), p. 78–85. ISBN 978-1-4503-0214-2. Disponível em: <http://doi.acm.org/10.1145/1830252.1830263>. Citado 2 vezes nas páginas 45 e 46.

LIN, X.; WANG, P. Log analysis in cloud computing environment with hadoop and spark. In: **5th IEEE International Conference on Broadband Network e Multimedia Technology (IC-BNMT 2013)**. Guilin, China: IEEE, 2013. p. 15–28. ISBN 978-1-4799-0095-4. Citado na página 49.

LIU, X.; IFTIKHAR, N.; XIE, X. Survey of real-time processing systems for big data. In: **Proceedings of the 18th International Database Engineering & Applications Symposium**. New York, NY, USA: ACM, 2014. (IDEAS '14), p. 356–361. ISBN 978-1-4503-2627-8. Disponível em: <http://doi.acm.org/10.1145/2628194.2628251>. Citado na página 45.

MAILLO, J. et al. kNN-IS: An Iterative Spark-based design of the k-Nearest Neighbors Classifier for Big Data. **Knowledge-Based Systems**, 2016. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950705116301757>. Citado na página 45.

MALAK, M. S.; EAST, R. Spark graphx in action. Manning Publ., 2016. Citado 2 vezes nas páginas 17 e 40.

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**. New York, NY, USA: ACM, 2010. (SIGMOD '10), p. 135–146. ISBN 978-1-4503-0032-2. Disponível em: <http://doi.acm.org/10.1145/1807167.1807184>. Citado 2 vezes nas páginas 40 e 47.

MANCORIDIS, S.; MITCHELL, B. S.; RORRES, C. Using automatic clustering to produce high-level system organizations of source code. In: **In Proc. 6th Intl. Workshop on Program Comprehension**. [S.l.: s.n.], 1998. p. 45–53. Citado na página 30.

MASSIE, M.; CHUN, B.; CULLER, D. The ganglia distributed monitoring system: design, implementation, and experience. **Parallel Computing**, v. 30, n. 5-6, p. 817–840, 2004. Disponível em: <http://dx.doi.org/10.1016/j.parco.2004.04.001>. Citado 2 vezes nas páginas 77 e 85.

MASSIE, M. et al. **Monitoring with Ganglia**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2012. ISBN 1449329705, 9781449329709. Citado 3 vezes nas páginas 10, 78 e 79.

MEHTA, J. et al. Trend analysis based on access pattern over web logs using hadoop. **International Journal of Computer Applications**, v. 115, n. 8, p. 34–37, April 2015. Full text available. Citado na página 43.

MENG, X. et al. **MLlib: Machine Learning in Apache Spark**. 2015. Disponível em: <http://arxiv.org/abs/1505.06807>. Citado na página 43.

MOHAMMED, E. A.; FAR, B. H.; NAUGLER, C. Applications of the mapreduce programming framework to clinical big data analysis: current landscape and future trends. **BioData Mining**, v. 7, n. 1, p. 22, 2014. ISSN 1756-0381. Disponível em: <http://dx.doi.org/10.1186/1756-0381-7-22>. Citado na página 43.

MONTGOMERY, D. C.; RUNGER, G. C. **Estatística Aplicada e Probabilidade para Engenheiros**. Rio de Janeiro, RJ: LTC Livros Técnicos e Científicos Editora S. A., 2007. Citado 2 vezes nas páginas 75 e 76.

NASCIMENTO, J. P.; MURTA, C. Um algoritmo paralelo em hadoop para cálculo de centralidade em grafos grandes. In: **XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. [S.l.: s.n.], 2012. Citado 2 vezes nas páginas 15 e 45.

NASCIMENTO, M. A.; SANDER, J.; POUND, J. Analysis of sigmod's co-authorship graph. **SIGMOD Record**, v. 32, n. 3, p. 8–10, 2003. Disponível em: <http://dblp.uni-trier.de/db/journals/sigmod/sigmod32.html#NascimentoSP03>. Citado na página 18.

NEWMAN, M. E. J. The Structure and Function of Complex Networks. **SIAM Review**, SIAM, v. 45, n. 2, p. 167–256, 2003. Citado 4 vezes nas páginas 9, 18, 27 e 29.

NEWMAN, M. E. J.; GIRVAN, M. Finding and evaluating community structure in networks. **Physical Review**, E 69, n. 026113, 2004. Citado na página 18.

NORDBERG, H. et al. Biopig: a hadoop-based analytic toolkit for large-scale sequence data. **Bioinformatics**, v. 29, n. 23, p. 3014–3019, 2013. Disponível em: <http://dblp.uni-trier.de/db/journals/bioinformatics/bioinformatics29.html#NordbergBWW13>. Citado na página 42.

O'DRISCOLL, A.; DAUGELAITE, J.; SLEATOR, R. D. 'big data', hadoop and cloud computing in genomics. **Journal of Biomedical Informatics**, v. 46, n. 5, p. 774–781, 2013. Disponível em: <http://dblp.uni-trier.de/db/journals/jbi/jbi46.html#ODriscollDS13>. Citado na página 41.

O'HANLON, C. A Conversation with John Hennessy and David Patterson. **Queue**, ACM, New York, NY, USA, v. 4, p. 14–22, Dezembro 2006. Citado na página 24.

OKAMOTO, H. Local detection of communities by attractor neural-network dynamics. In: KOPRINKOVA-HRISTOVA, P.; MLADENOV, V.; KASABOV, N. K. (Ed.). **Artificial Neural Networks**. Cham: Springer International Publishing, 2015. p. 115–125. ISBN 978-3-319-09903-3. Citado na página 51.

PACHECO, F.; KLEIN, A.; RIGHI, R. Modelos de negócio para produtos e serviços baseados em internet das coisas: uma revisão da literatura e oportunidades de pesquisas futuras. In: **Revista de Gestão**. São Paulo, SP, Brasil: Spell, 2015. p. 27–34. Citado na página 15.

PAPADIMITRIOU, S. et al. Hierarchical, parameter-free community discovery. In: DAELEMANS, W.; GOETHALS, B.; MORIK, K. (Ed.). **Machine Learning and Knowledge Discovery in Databases**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 170–187. ISBN 978-3-540-87481-2. Citado na página 51.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Architecture: a quantitative approach**. Fourth edition. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. Citado 2 vezes nas páginas 25 e 90.

PONS, P.; LATAPY, M. Computing communities in large networks using random walks. In: **Proceedings of the 20th International Conference on Computer and Information Sciences**. Berlin, Heidelberg: Springer-Verlag, 2005. (ISCIS'05), p. 284–293. ISBN 3-540-29414-7, 978-3-540-29414-6. Disponível em: <http://dx.doi.org/10.1007/11569596_31>. Citado na página 50.

PONTOIZEAU, T. **Community detection : computational complexity and approximation**. Tese (Theses) — PSL Research University, jun. 2018. Disponível em: <https://tel.archives-ouvertes.fr/tel-01825871>. Citado 5 vezes nas páginas 9, 19, 29, 30 e 31.

RAGHAVAN, U.; ALBERT, R.; TIRUPATIKUMARA, S. Near linear time algorithm to detect community structures in large-scale networks. **Physical Review E - Statistical, Nonlinear, and Soft Matter Physics**, v. 76, n. 3, 9 2007. ISSN 1539-3755. Citado 8 vezes nas páginas 9, 12, 33, 34, 50, 53, 58 e 59.

RAMAN, V.; SAURABH, S. Triangles, 4-cycles and parameterized (in-)tractability. In: **Algorithm Theory - SWAT 2006, 10th ScandinavianWorkshop on Algorithm Theory, Riga, Latvia, July 6-8, 2006, Proceedings**. [s.n.], 2006. p. 304–315. Disponível em: <https://doi.org/10.1007/11785293_29>. Citado na página 30.

RANGER, C. et al. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: **Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture**. Washington, DC, USA: IEEE Computer Society, 2007. p. 13–24. Citado 2 vezes nas páginas 17 e 34.

RASTEGAR-MOJARAD, M. et al. Collecting and analyzing patient experiences of health care from social media. **JMIR Res Protoc**, v. 4, n. 3, p. e78, Jul 2015. Disponível em: <http://www.researchprotocols.org/2015/3/e78/>. Citado na página 43.

REYES-ORTIZA, J. L.; ONETOB, L.; ANGUITA, D. Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. **INNS Conference on Big Data 2015**, San Francisco, CA, p. 121–130, 2015. Citado na página 49.

RIEDY, J.; BADER, D. A.; MEYERHENKE, H. Scalable multi-threaded community detection in social networks. In: **Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum**. Washington, DC, USA: IEEE Computer Society, 2012. (IPDPSW '12), p. 1619–1628. ISBN 978-0-7695-4676-6. Disponível em: <http://dx.doi.org/10.1109/IPDPSW.2012.203>. Citado na página 50.

ROSEN, J. et al. Iterative mapreduce for large scale machine learning. **CoRR**, abs/1303.3517, 2013. Disponível em: <http://arxiv.org/abs/1303.3517>. Citado na página 42.

ROSVALL, M.; BERGSTROM, C. An information-theoretic framework for resolving community structure in complex networks. **Proceedings of the National Academy of Sciences**, National Acad Sciences, v. 104, n. 18, p. 7327, 2007. Disponível em: <http://scholar.google.de/scholar.bib?q=info:AlbgSB2O_-AJ:scholar.google.com/&output=citation&hl=de&as_sdt=2000&as_vis=1&ct=citation&cd=0>. Citado na página 50.

ROSVALL, M.; BERGSTROM, C. T. Maps of random walks on complex networks reveal community structure. **Proceedings of the National Academy of Sciences**, v. 105, n. 4, p. 1118–1123, 2008. Disponível em: <http://www.pnas.org/content/105/4/1118.abstract>. Citado na página 50.

SCHATZ, M. C. Cloudburst: highly sensitive read mapping with mapreduce. **Bioinformatics**, v. 25, n. 11, p. 1363–1369, 2009. Disponível em: <http://dblp.uni-trier.de/db/journals/bioinformatics/bioinformatics25.html#Schatz09>. Citado na página 41.

SEGAL, O. et al. Sparkcl: A unified programming framework for accelerators on heterogeneous clusters. **CoRR**, abs/1505.01120, 2015. Disponível em: <http://arxiv.org/abs/1505.01120>. Citado na página 43.

SEIDMAN, S. B. Network structure and minimum degree. **Social Networks**, v. 5, n. 3, p. 269 – 287, 1983. ISSN 0378-8733. Disponível em: <http://www.sciencedirect.com/science/article/pii/037887338390028X>. Citado na página 30.

SHI, W. et al. An integrated data preprocessing framework based on apache spark for fault diagnosis of power grid equipment. **Journal of Signal Processing Systems**, p. 1–16, 2016. ISSN 1939-8115. Disponível em: <http://dx.doi.org/10.1007/s11265-016-1119-4>. Citado na página 45.

SHORO, A. G.; SOOMRO, T. R. Big data analysis: Ap spark perspective. **Global Journal of Computer Science and Technology Software and Data Engineering**, Global Journals Inc. (USA), Los Alamitos, CA, USA, v. 15, n. 1, p. 1470–1483, 2015. ISSN 0975-4172. Citado na página 43.

SHVACHKO, K. et al. The hadoop distributed file system. In: **Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)**. Washington, DC, USA: IEEE Computer Society, 2010. (MSST '10), p. 1–10. ISBN 978-1-4244-7152-2. Disponível em: <http://dx.doi.org/10.1109/MSST.2010.5496972>. Citado na página 37.

SIRETSKIY, A. et al. A quantitative assessment of the hadoop framework for analyzing massively parallel dna sequencing data. **GigaScience**, v. 4, n. 1, p. 26, 2015. ISSN 2047-217X. Disponível em: <http://dx.doi.org/10.1186/s13742-015-0058-5>. Citado na página 42.

SOBIN, C.; RAYCHOUDHURY, V.; SAHA, S. A survey of parallel community detection algorithms. **PHandbook of Research on Applied Cybernetics and Systems Science**, v. 1, n. 26, 2017. Citado 11 vezes nas páginas 9, 10, 16, 19, 20, 32, 34, 53, 69, 70 e 71.

STAUDT, C. L.; MEYERHENKE, H. Engineering parallel algorithms for community detection in massive networks. **IEEE Trans. Parallel Distrib. Syst.**, v. 27, n. 1, p. 171–184, 2016. Disponível em: <https://doi.org/10.1109/TPDS.2015.2390633>. Citado na página 50.

SUN, Y. et al. Rankclus: Integrating clustering with ranking for heterogeneous information network analysis. In: **Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology**. New York, NY, USA: ACM, 2009. (EDBT '09), p. 565–576. ISBN 978-1-60558-422-5. Disponível em: <http://doi.acm.org/10.1145/1516360.1516426>. Citado na página 51.

SVENDSEN, M.; MUKHERJEE, A. P.; TIRTHAPURA, S. Mining maximal cliques from a large graph using mapreduce: Tackling highly uneven subproblem sizes. **J. Parallel Distrib. Comput.**, v. 79-80, p. 104–114, 2015. Disponível em: <http://dx.doi.org/10.1016/j.jpdc.2014.08.011>. Citado na página 46.

TAYLOR, R. C. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. **BMC Bioinformatics**, v. 11, n. 12, p. s1, 2010. ISSN 1471-2105. Citado na página 41.

VERMA, A.; MANSURI, A. H.; JAIN, N. Big data management processing with hadoop mapreduce and spark technology: A comparison. In: **2016 Symposium on Colossal Data Analysis and Networking (CDAN)**. Guilin, China: IEEE, 2016. p. 15–28. ISBN 978-1-5090-0669-4. Citado na página 49.

WARD, J. S.; BARKER, A. Undefined by data: A survey of big data definitions. **CoRR**, abs/1309.5821, 2013. Disponível em: <http://arxiv.org/abs/1309.5821>. Citado na página 27.

WHITE, T. **Hadoop: The Definitive Guide**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2009. ISBN 0596521979, 9780596521974. Citado 4 vezes nas páginas 9, 34, 36 e 95.

WHITE, T. **Hadoop: The Definitive Guide**. 4st. ed. [S.l.]: OReilly Media, Inc., 2015. ISBN 9781491901632. Citado 2 vezes nas páginas 15 e 17.

WIEWIóRKA, M. S. et al. SparkSeq: fast, scalable, cloud-ready tool for the interactive genomic data analysis with nucleotide precision. **Bioinformatics**, Oxford University Press, p. btu343+, maio 2014. ISSN 1460-2059. Disponível em: <http://dx.doi.org/10.1093/bioinformatics/btu343>. Citado na página 44.

XIA, Y. et al. Big traffic data processing framework for intelligent monitoring and recording systems. **Neurocomputing**, v. 181, p. 139–146, 2016. Disponível em: <http://dx.doi.org/10.1016/j.neucom.2015.07.140>. Citado na página 43.

XIN, R. S. et al. Graphx: A resilient distributed graph system on spark. In: **First International Workshop on Graph Data Management Experiences and Systems**. New York, NY, USA: ACM, 2013. (GRADES '13), p. 2:1–2:6. ISBN 978-1-4503-2188-4. Disponível em: <http://doi.acm.org/10.1145/2484425.2484427>. Citado 3 vezes nas páginas 46, 53 e 55.

XU, R. et al. Distributed collaborative filtering with singular ratings for large scale recommendation. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 95, p. 231–241, set. 2014. ISSN 0164-1212. Disponível em: <http://dx.doi.org/10.1016/j.jss.2014.04.045>. Citado 2 vezes nas páginas 42 e 49.

YANG YONG GUO, B. D. Y.; LI, H. Community detection of heterogeneous networks with rankclus based on spark graphx. **2nd International Conference on Artificial Intelligence and Engineering Applications**, 2017. Citado na página 51.

YAO, Z.; VARGHESE, B.; RAU-CHAPLIN, A. High performance risk aggregation: Addressing the data processing challenge the hadoop mapreduce way. **CoRR**, abs/1311.5686, 2013. Disponível em: <http://dblp.uni-trier.de/db/journals/corr/corr1311.html#YaoVR13>. Citado na página 42.

YE, Z. et al. Sparktext: Biomedical text mining on big data framework. **PLOS ONE**, Public Library of Science, v. 11, n. 9, p. 1–15, 09 2016. Disponível em: <http://dx.doi.org/10.1371%2Fjournal.pone.0162721>. Citado na página 45.

ZAHARIA, M. et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: **Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)**. San Jose, CA: USENIX, 2012. p. 15–28. ISBN 978-931971-92-8. Disponível em: <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia>. Citado 2 vezes nas páginas 39 e 48.

ZAHARIA, M. et al. Spark: Cluster computing with working sets. In: **Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing**. Berkeley, CA, USA: USENIX Association, 2010. (HotCloud'10), p. 10–10. Disponível em: <http://dl.acm.org/citation.cfm?id=1863103.1863113>. Citado na página 48.

ZAHARIA, M. et al. Discretized streams: Fault-tolerant streaming computation at scale. In: **Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2013. (SOSP '13), p. 423–438. ISBN 978-1-4503-2388-8. Disponível em: <http://doi.acm.org/10.1145/2517349.2522737>. Citado na página 45.

ZHANG, T. et al. Parsa: High-throughput scientific data analysis framework with distributed file system. **Future Generation Comp. Syst.**, v. 51, p. 111–119, 2015. Disponível em: <http://dx.doi.org/10.1016/j.future.2014.10.015>. Citado na página 42.

ZHANG, Y.; JORDAN, M. I. Splash: User-friendly programming interface for parallelizing stochastic algorithms. **CoRR**, abs/1506.07552, 2015. Disponível em: <http://arxiv.org/abs/1506.07552>. Citado na página 44.

ZHANG, Z. et al. Scientific computing meets big data technology: An astronomy use case. **CoRR**, abs/1507.03325, 2015. Disponível em: <http://arxiv.org/abs/1507.03325>. Citado na página 44.

ZIVIANI, N. **Projeto de Algoritmos com Implementações em Java e C++**. Primeira edição. [S.l.]: Thomson, 2007. Citado na página 29.

ZOU, Q. et al. Survey of mapreduce frame operation in bioinformatics. **Briefings in Bioinformatics**, v. 15, n. 4, p. 637–647, 2014. Disponível em: <http://dx.doi.org/10.1093/bib/bbs088>. Citado na página 41.