CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
Mestrado em Modelagem Matemática e Computacional

Leonardo Carlos da Cruz

Um Algoritmo Paralelo Eficiente para Cálculo de Centralidades em Grafos

Belo Horizonte
Julho de 2013

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
Mestrado em Modelagem Matemática e Computacional

Leonardo Carlos da Cruz

Um Algoritmo Paralelo Eficiente para Cálculo de Centralidades em Grafos

Dissertação de mestrado submetida ao Programa de Pós-Graduação em Modelagem Matemática e Computacional, como requisito parcial à obtenção do título de Mestre em Modelagem Matemática e Computacional.

Orientadora: Professora Cristina Duarte Murta

Belo Horizonte
Julho de 2013

Dedico este trabalho a minha mãe Almira de Oliveira Campos da Cruz e ao meu pai Hirton Carlos da Cruz, por tudo que sou. Para minhas irmãs Camila Campos da Cruz e Renata Campos da Cruz, pelo apoio, incentivo, e por manterem-se fortes nos momentos difíceis. A uma pessoa muito especial em minha vida, Amanda Messeder de Oliveira, por todo amor, carinho e incentivo. Seu apoio e compreensão foram imprecindíveis para a realização desse trabalho. Dedico-lhe essa realização com todo meu amor.

Agradecimentos

Agradeço primeiramente a Professora Cristina Duarte Murta, pelas sugestões, pelo tempo dedicado, por ter me orientado no caminho que levou à realização dessa pesquisa.

Aos colegas e amigos do Laboratório de Computação Científica da Universidade Federal de Minas Gerais: Gessy Caetano Júnior, Edilson Costa de Oliveira, Antônio Carlos Fernandes e Renato Antônio Veneroso da Fonseca. Obrigado pelo apoio, incentivo e conselhos nos momentos em que surgiram obstáculos.

Aos professores, colegas e funcionários do curso de pós-graduação do CEFET-MG, que direta ou indiretamente contribuíram para a conclusão desse trabalho.

Resumo

O modelo MapReduce para processamento paralelo de grandes massas de dados tem chamado a atenção da comunidade acadêmica nos últimos anos. Diante da potencialidade de seu uso na resolução de problemas que envolvem instâncias de entrada em grande escala, o esforço para adaptar esse paradigma de programação no contexto científico é evidente. Os estudos em áreas tão diversas quanto sistemas de comunicação, Internet e Web, computação móvel, sistemas biológicos, redes sociais e redes de transporte, dentre outras, apresentam em comum o fato de que todos esses sistemas podem ser modelados por grafos. A quantidade de elementos participantes nessas redes complexas tem alcançado escalas cada vez maiores, o que requer o uso de processamento paralelo para a análise. Contudo, ainda há poucos trabalhos que envolvem a utilização do modelo MapReduce na resolução de problemas relacionados a grafos. Neste trabalho apresentamos um novo algoritmo paralelo para o cálculo exato de centralidades em grafos grandes, usando o paradigma de programação MapReduce/Hadoop. O objetivo é fazer o uso eficiente dos recursos computacionais de maneira a avançar nas escalas de tamanho de grafos em que é possível computar métricas exatas de centralidade. Para avaliar o algoritmo, foram processados diversos grafos e os resultados comparados com algoritmos sequenciais e paralelos. Os experimentos indicam melhorias no uso da memória principal e do espaço disponível em disco, quando comparado a outras implementações.

Palavras-chave: redes complexas, grafos, processamento paralelo, MapReduce.

Abstract

The MapReduce programming model for parallel processing of large data sets has attracted the attention of the academic community in recent years. Given the potential for its use in solving problems that involve instances of large-scale input, the effort to adapt that programming paradigm in the scientific context is evident. Studies in areas as diverse as communications systems, Internet and Web, mobile computing, biological systems, social networks and transportation networks, among others, have in common the fact that all of these systems can be modeled by graphs. The number of elements involved in these complex networks has achieved increasingly larger scales, which requires the use of parallel processing for analysis. However, there are still few studies involving the use of the MapReduce programming model to solve problems related to graphs. This work presents a new parallel algorithm for the computation of centrality in large graphs, using the MapReduce/Hadoop programming paradigm. The goal is to manage efficiently the use of computational resources in order to go further on the size scales of graphs in which it is possible to compute exact metrics of centrality. In order to evaluate the algorithm, several graphs were processed and the results compared with sequential and parallel algorithms. The experiments indicate improvements in the use of main memory and available disk space when compared to other implementations.

Keywords: parallel processing, graphs, complex networks, MapReduce.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A facilidade de aquisição de dados bem como a disponibilidade de dispositivos de alta capacidade para armazená-los possibilitou o surgimento de grandes massas de dados em diversos contextos. Essa realidade não é diferente para a comunidade científica - cada vez mais dados estão disponíveis em diversas áreas do conhecimento, partindo da coleta de amostras de moléculas de DNA, capaz de gerar milhões de fragmentos advindos de uma simples bactéria, até a aquisição de imagens e medidas de radiações capturadas por telescópios que varrem o universo [Bryant 2011]. Nesse ambiente, um modelo comumente usado para representar tais conjuntos de dados, de maneira relativamente intuitiva, são os grafos. Portanto, explorar modelos baseados em grafos é uma importante tarefa para o auxílio da descoberta científica. Contudo, desenvolver técnicas eficientes capazes de extrair propriedades em grafos é uma tarefa não trivial, e o desafio se torna ainda maior quando estamos trabalhando com grafos grandes, onde o número de vértices e arestas chegam às escalas de milhares ou milhões.

Na modelagem por grafos, os sistemas têm as suas entidades participantes representadas por vértices e as relações entre elas representadas por arestas. Uma maneira de extrair e analisar as propriedades de sistemas complexos por meio de grafos pode ser, por exemplo, pela identificação dos vértices (entidades) que possuem o maior número de relações (arestas). Esse tipo de informação poderia esclarecer quais entidades têm maior influência na dinâmica de funcionamento do sistema complexo. Contudo, as medidas de centralidade usadas para caracterizar individualmente os nós de um grafo, em termos de sua importância estrutural, como o *grau*, ou como as propri-

edades *excentricidade*, *betweenness* e *closeness* [Freeman 1979, Hage e Harary 1995], apresentam barreiras no que diz respeito à complexidade de sua computação. Essas medidas, que também são usadas para determinar características gerais do grafo, como diâmetro e raio, tornam-se difíceis, senão proibitivas, para cálculo em grafos de escala muito grande [Kang et al. 2011].

No contexto de processamento paralelo de grandes quantidades de dados, o modelo MapReduce tem ganhado espaço entre as empresas de tecnologia da informação e no meio acadêmico. Um dos fatores que permitiram a sua expansão foi o desenvolvimento da plataforma Hadoop, uma implementação de código aberto do modelo MapReduce. Com resultados que demonstraram grande êxito na computação de dados na escala de petabytes, tornou-se desejável conhecer o desempenho de algoritmos desenvolvidos sob este paradigma quando a aplicação destina-se ao processamento de grafos grandes. Algumas perguntas que gostaríamos de responder são: como um algoritmo desenvolvido sob este modelo pode escalar com a quantidade de vértices no grafo, ao se calcular os menores caminhos entre todos os pares de vértices? Qual é a eficiência de um algoritmo implementado à luz desse paradigma, quando utilizado para o processamento paralelo de grafos com quantidades de vértices e arestas em várias escalas? Para ajudar a responder essas perguntas, propomos o estudo e desenvolvimento de algoritmos paralelos para processamento de grafos grandes, implementados em MapReduce/Hadoop. Ao responder estas perguntas, estamos interessados em avançar nos limites que o modelo apresenta no processamento de grafos de grande escala.

1.1 Escopo

Esse projeto de dissertação trata de algoritmos paralelos para o cálculo de medidas de centralidade em grafos. O propósito é estudar o problema do cálculo de métricas de centralidade em grafos grandes e propor algoritmos paralelos para esse cálculo. Nosso objetivo é avaliar a possibilidade de se obter medidas exatas para grafos em várias escalas de tamanho, frente aos recursos computacionais disponíveis. A questão a ser avaliada é que tamanhos de grafos podemos analisar, por meio de algoritmos paralelos, para obter suas medidas exatas de centralidade. Para grafos

maiores, poderíamos ter apenas uma análise aproximada destas medidas.

Esse trabalho dá continuidade ao trabalho de dissertação desenvolvido em [Nascimento 2011], onde foi proposto o algoritmo HEDA (*Hadoop based Exact Diameter Algorithm*), que encontra medidas exatas de centralidade para grafos de tamanhos moderados. No entanto, o algoritmo tem limitações quanto ao armazenamento em disco, como é possível verificar nos resultados apresentados em [Nascimento 2011] para o processamento do grafo IMDB (*Internet Movie Data Base*). A limitação do uso de espaço no disco tem relação com o processamento eficiente dos dados, uma vez que os algoritmos executados na plataforma Hadoop armazenam frequentemente os resultados parciais do processamento no sistema de arquivos. Neste texto apresentamos resultados que mostram avanços no uso do modelo MapReduce/Hadoop para o cálculo exato de centralidades em grafos grandes.

O caráter interdisciplinar deste trabalho se manifesta pelo vasto contexto em que se aplica a modelagem matemática por grafos. Tal modelo matemático é usado na biologia, notadamente para a genômica estrutural, onde é feito o mapeamento das estruturas de moléculas de DNA. Em engenharia elétrica/eletrônica, aparece na construção de circuitos que requer a otimização do uso de espaço para a colocação de dispositivos eletrônicos e suas conexões. Nos estudos de redes sociais, é usado para levantar hipóteses acerca do comportamento em grupo de seres vivos. Também surge nos diversos problemas de otimização abordados na pesquisa operacional e em muitas modelagens que envolvem redes complexas. Portanto, os resultados deste projeto podem contribuir para pesquisas científicas em diversas áreas do conhecimento.

1.2 Objetivos do Estudo

Diante do contexto introduzido, o uso do modelo MapReduce para o cálculo de centralidades em grafos grandes merece atenção e deve ser avaliado. Para tanto, é necessário mapear os espaços de atuação dos algoritmos disponíveis, que dependem da quantidade de arestas e vértices presentes no grafo. Estamos interessados em explorar essas regiões, identificando possíveis aprimoramentos no uso do modelo MapReduce/Hadoop para cálculo exato de centralidades em grafos grandes. Neste trabalho, esperamos expandir as possibilidades do processamento exato de centrali-

dades em grafos de escalas mais amplas, aumentando o espaço de atuação do modelo MapReduce/Hadoop. O objetivo específico do trabalho é fazer um estudo do modelo MapReduce aplicado ao problema de calcular os menores caminhos em um grafo, investigando as possibilidades de melhoria quanto ao uso de vários recursos tais como tempo de execução, uso da memória, troca de dados em comunicação na rede e uso de espaço em disco.

1.3 Resultados e contribuições

A principal contribuição do trabalho é o aprimoramento do processamento paralelo para o cálculo exato de diâmetro e raio de um grafo grande, usando o modelo MapReduce/Hadoop. O aprimoramento é alcançado através da implementação de técnicas observadas na bibliografia citada, além da identificação das propriedades do modelo MapReduce que favorecem a adaptação desse paradigma de programação na solução do problema do cálculo de diâmetro e raio em grafos grandes. O uso do espaço em disco durante o processamento do grafo é fator limitante da escala de tamanhos de grafos em que é possível fazer cálculos exatos de centralidades. Com os aprimoramentos propostos, foi possível reduzir consideravelmente o uso desse recurso. Adicionalmente, o uso eficiente da memória principal e do espaço em disco promoveram a redução do tempo de execução em relação ao algoritmo HEDA [Nascimento 2011]. Também foram observados valores de *speedup* superlinear durante o processamento dos maiores grafos do conjunto experimentado.

1.4 Organização do trabalho

O presente trabalho, apresentado em forma de dissertação, está organizado como segue. O Capítulo 2 descreve o modelo MapReduce, objeto de interesse no que diz respeito ao desenvolvimento de algoritmos paralelos para processamento massivo de dados. Também são apresentadas as medidas de centralidades em grafos e os desafios do desenvolvimento de algoritmos para o cálculo dessas medidas. Além disso, é apresentado nesse capítulo uma discussão sobre o processamento de grafos grandes usando o modelo MapReduce. Adicionalmente, são apresentados alguns algoritmos que usam esse modelo e que serão foco de estudo. No Capítulo 3 é feita

a descrição de um novo algoritmo para o cálculo de centralidades, usando o modelo MapReduce/Hadoop. No Capítulo 4 encontra-se o planejamento dos experimentos para análise do algoritmo proposto, com os resultados apresentados no Capítulo 5. Finalmente, no Capítulo 6, apresentamos as conclusões obtidas por meio desse trabalho.

Capítulo 2

Trabalhos Relacionados

Neste capítulo apresentamos e discutimos conceitos encontrados na literatura que contemplam o escopo desta proposta. Nas Seções 2.1 e 2.2 descrevemos o modelo MapReduce e sua implementação Hadoop. Na Seção 2.3 apresentamos conceitos relacionados a grafos e centralidades em grafos, bem como os desafios relacionados ao cálculo de métricas de centralidade. Na Seção 2.4 apresentamos uma discussão sobre o uso do modelo MapReduce/Hadoop no processamento paralelo de grafos grandes. Finalmente, na Seção 2.5, apresentamos as medidas de desempenho para algoritmos paralelos.

2.1 Introdução ao Modelo MapReduce

O modelo de programação MapReduce tem por objetivo descrever uma maneira de processar grandes quantidades de dados de forma paralela [Dean e Ghemawat 2004]. Nesse modelo considera-se que o volume de dados a ser processado não é compatível com a capacidade de armazenamento de um único nó computacional. Portanto, em seu projeto, assume-se o uso de vários desses nós. Neste contexto é natural que exista no MapReduce o detalhamento de uma dinâmica de conexões entre os recursos computacionais, cujo objetivo é distribuir de forma coerente os dados e o processamento. Adicionalmente, o MapReduce não foi concebido ponderando-se que as máquinas sejam necessariamente robustas e, portanto, dispendiosas [Dean e Ghemawat 2004]. Essa condição permite flexibilidade na redução de custos com as mesmas - uma qualidade que é especialmente interessante quando a expectativa é realizar a computação empregando-se

milhares de nós computacionais. Diante dessa perspectiva o modelo também prevê a manutenção da confiabilidade no processamento, mesmo que a probabilidade de falhas aumente não apenas devido à menor robustez das máquinas, mas também devido ao número de máquinas presentes.

Do ponto de vista de programação paralela, o MapReduce especifica apenas duas funções que devem ser escritas pelo usuário do modelo: a função *Map* e a função *Reduce*. A programação necessária para a sincronização dos processamentos, distribuição dos dados e o uso eficiente dos recursos computacionais está presente em camadas do modelo que não são visíveis ao programador, sendo acessíveis apenas pela configuração de parâmetros do modelo MapReduce. Esse paradigma de programação, portanto, procura reduzir a complexidade do desenvolvimento de programas que processam em paralelo grandes volumes de dados, tipicamente na escala de petabytes, provendo confiabilidade no processamento através do gerenciamento de falhas nas máquinas constituintes do sistema distribuído [Dean e Ghemawat 2008].

2.1.1 Um Exemplo do Modelo MapReduce

Nessa seção apresentamos um exemplo de uso do modelo MapReduce para o processamento de um conjunto grande de texto. A partir do exemplo exibido na Figura 2.1, introduzimos o esquema de funcionamento do modelo MapReduce. A ideia básica do modelo é processar paralelamente vários arquivos de entrada usando uma função *Map* e uma função *Reduce*. No exemplo da Figura 2.1, o objetivo é contar as palavras contidas nos documentos armazenados em um sistema de arquivos distribuídos. Inicialmente, a função *Map* recebe como entrada o par arquivo/conteúdo, processando uma linha do arquivo por vez. A saída do processamento é o par constituído por uma palavra, tomada como *chave*, e pelo *valor* númerico 1. Após todos os arquivos de entrada terem sido processados paralelamente, as saídas das funções *Map* são redistribuídas pelo sistema, em um procedimento chamado *shuffle* [Karloff, Suri e Vassilvitskii 2010].

Nesse processo, as saídas das funções *Map* com palavras ou *chaves* idênticas são agrupadas no mesmo bloco de dados, sendo estes armazenados no sistema de arquivo distribuído. Ao fim da redistribuição, a fase seguinte é realizada pela função *Reduce*,

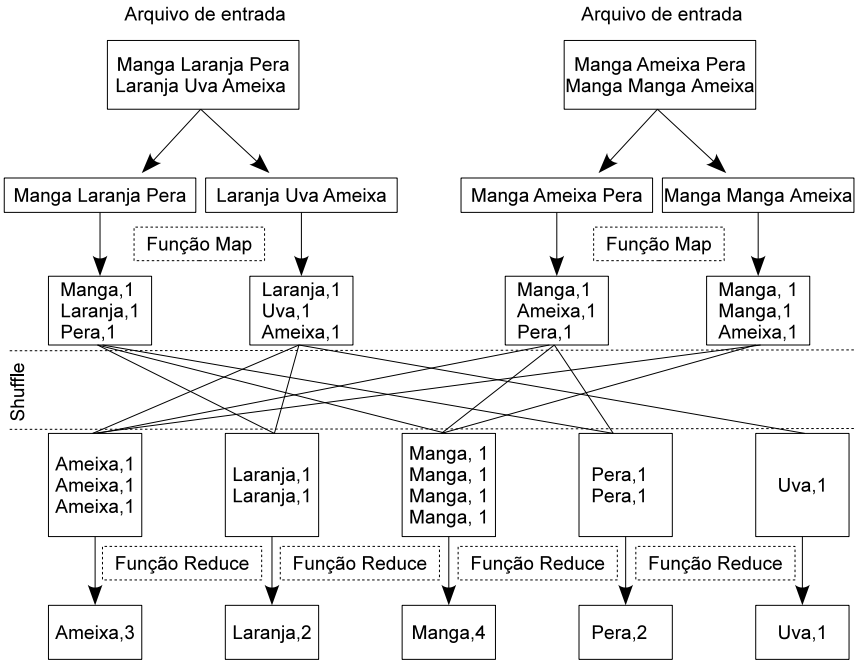


Figura 2.1: Exemplo de um contador de palavras usando o modelo MapReduce.

aplicada paralelamente em cada bloco de dados. No exemplo da Figura 2.1, a função *Reduce* soma todos os valores associados à mesma chave, resultando na contagem das palavras contidas nos arquivos de entrada.

2.1.2 Fluxo de execução de um programa no modelo MapReduce

No paradigma MapReduce os dados de entrada e de saída são organizados em multiconjuntos de pares chave/valor, ou seja, conjuntos que permitem a existência de elementos repetidos, nesse caso, de pares chave/valor repetidos. A Figura 2.2 mostra o fluxo de execução de um programa escrito nesse paradigma. O multiconjunto de entrada é dividido em M subconjuntos, chamados de blocos na figura, e esses são distribuídos entre os nós computacionais, de forma que todo *nó mapper* terá uma parcela dos dados a ser processada localmente por uma cópia da função *Map*. Especificada pelo usuário, essa função recebe cada par do subconjunto local de entrada, que então é processado e gera como saída um multiconjunto de novos pares chave/valor. Todos os elementos de todos os multiconjuntos de saída produzidos, também chamados de pares chave/valor intermediários, são armazenados localmente em R partições. Os domínios dessas partições locais, representadas na figura pelos

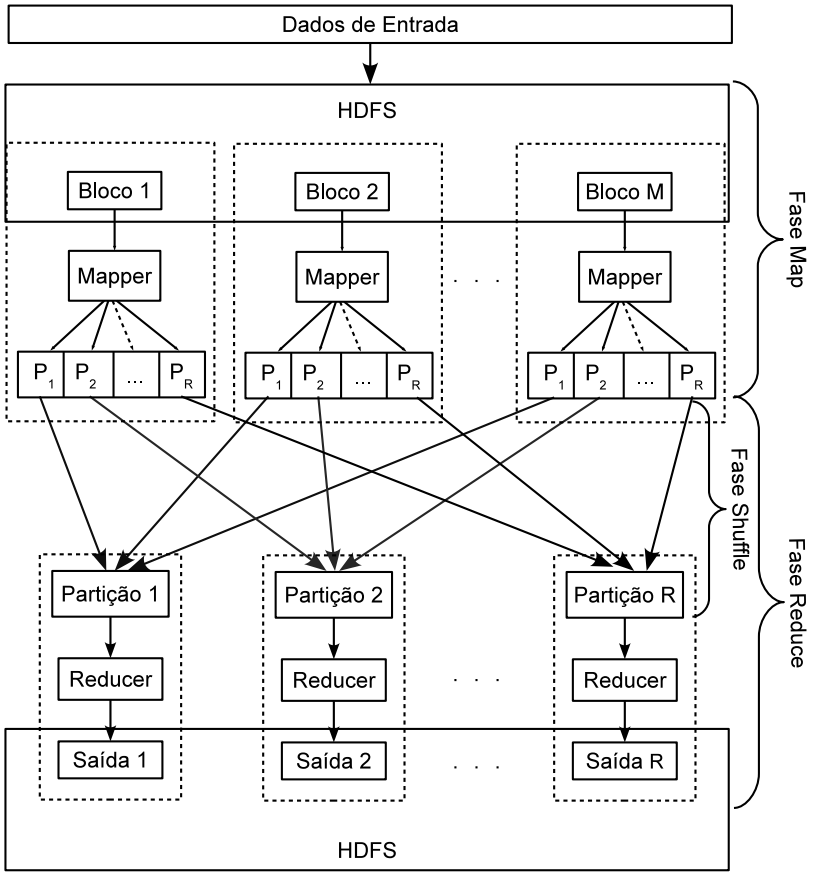


Figura 2.2: Fluxo de execução do ciclo *Map-Shufle-Reduce* no modelo MapReduce.

nomes $P_1, P_2 \ldots P_R$, são determinados segundo uma *função de partição* definida opcionalmente pelo programador, e as informações sobre as localizações das mesmas são compartilhadas globalmente. Quando todos os M blocos de entrada tiverem sido processados e os respectivos resultados armazenados nas R partições locais, é dito que foi encerrada a *fase Map* do processamento paralelo.

Após a finalização da fase *Map*, as R partições locais contendo os pares chave/valor intermediários são realocadas entre os nós computacionais responsáveis pelo processamento paralelo das mesmas. Tal redistribuição, conhecida como *fase Shuffle*, segue a relação um para um entre partição e *nó reducer*, este último encarregado de extrair as partições remotas criadas pelos *mappers* e executar localmente nas mesmas uma cópia da função *Reduce*, definida pelo usuário. Portanto, a fase conhecida como *Reduce* é constituída por duas partes: a parte referente à fase

Shuffle, em que são copiadas e unidas as partições de mesmo nome no lado do *nó reducer*, e a parte em que essas partições são processadas pelo *nó reducer*.

Na fase *Reduce*, uma vez que o *reducer* termina de buscar a sua respectiva partição, esta deverá ter seus elementos ordenados por uma chave intermediária, o que permite o agrupamento dos valores cujas chaves são idênticas, isto é, tornando possível a reorganização dos elementos da partição em pares chave/*lista de valores*. A ordenação se faz necessária porque o *nó reducer* pode receber como entrada uma partição que não contém necessariamente pares com chaves iguais [Dean e Ghemawat 2008]. A uniformidade das chaves intermediárias dentro de uma mesma partição vai depender da função de partição usada na fase *Map*.

Finalmente, o nó *reducer* passa cada par intermediário chave/lista de valores de sua partição para a função *Reduce*, que o processa e emite na saída um multiconjunto de pares chave/valor, associados à mesma chave intermediária. A conclusão de tal procedimento por parte de todos os nós *reducers* caracteriza o fim da fase *Reduce*, constituída por R partições de saída. Por outro lado, o término da sequência de etapas *Map*, *Shuffle* e *Reduce* não caracteriza necessariamente o fim da execução de um programa implementado sob este modelo, podendo o mesmo ser constituído de vários ciclos com essa sequência, onde as funções *Map* e *Reduce* são possivelmente distintas entre os ciclos [Karloff, Suri e Vassilvitskii 2010].

2.2 MapReduce implementado: Hadoop

Hadoop[1] é uma plataforma de código livre, implementada em Java, cujo objetivo é executar programas desenvolvidos de acordo com o modelo MapReduce. A plataforma conta com o sistema de arquivos HDFS (*Hadoop Distributed File System*), que provê os recursos para distribuição e redundância dos dados. Esse sistema de arquivos é baseado no conceito de metadados, uma vez que em seu projeto é utilizado o próprio sistema de arquivos local dos nós para criar o sistema de arquivos global do conjunto de nós. A distribuição é feita em blocos de dados de tamanho configurável, porém fixo, criados e alocados pelo HDFS nos nós do sistema, responsáveis por armazená-los e posteriormente processá-los: são esses nós que executam o ser-

[1]Disponível em: hadoop.apache.org

viço chamado *DataNode*. A redundância é executada da seguinte forma: para cada bloco alocado são criadas três[2] réplicas distribuídas no sistema, o que potencializa a possibilidade de recuperação de dados em caso de falhas nos nós *DataNode*.

As funcionalidades fornecidas pelo HDFS no Hadoop são controladas em dois nós computacionais no sistema: eles executam os serviços *NameNode* e *Secondary NameNode*. O serviço *Secondary NameNode* tem a função de auxílio ao *NameNode* e não de substituição em caso de falhas deste último, fazendo do nó *NameNode* uma máquina cujo funcionamento deve ser muito seguro. Isso é justificado pelo fato do nó *NameNode* ser encarregado pelo armazenamento de todas informações relativas às localizações dos blocos de dados espalhados entres os nós *DataNode*, tornando-o portanto um ponto único de falha do sistema.[3]

Outro nó computacional que tem participação essencial na plataforma Hadoop é aquele que executa o serviço *JobTracker*, cuja função é gerenciar o processamento dos dados nos nós *DataNode*. O serviço *JobTracker* é responsável por designar as tarefas *mapper* ou *reducer* aos nós *DataNode*, baseado nas informações que ele extrai do *NameNode*. Além disso, é de responsabilidade do *JobTracker* identificar falhas nos nós *DataNode* de forma a possibilitá-lo realocar as tarefas em outros nós *DataNode* que estejam funcionais. A comunicação entre o *JobTracker* e os nós *DataNode* é feita pelo serviço *TaskTracker*, que deve ser executado em todos os nós computacionais que mantêm o serviço *DataNode* em funcionamento.

De maneira geral, a plataforma Hadoop foi projetada com sua arquitetura baseada em dois princípios: a distribuição dos dados com confiabilidade, feita pelo HDFS (*NameNode*, *Secondary NameNode*, *DataNodes*) e a distribuição do processamento com alta disponibilidade (*JobTracker*, *TaskTrackers*) [Lee et al. 2011].

[2]O número de réplicas é configurável, mas deve ser sempre maior ou igual a um [White 2012].

[3]Uma nova versão do Hadoop chamada YARN (*Yet Another Resource Negotiator*) ou MRv2 (*MapReduce 2*) possibilita o uso de redundância no *NameNode*.

2.3 Centralidade em grafos

2.3.1 Definições preliminares

Nessa subseção apresentamos algumas definições matemáticas acerca de grafos, afim de prover referências suficientes para a discussão que será feita sobre medidas de centralidade na subseção seguinte. Um grafo $G = (V, A)$, também chamado de *grafo não direcionado*, é definido por um conjunto finito de vértices V e um multi-conjunto finito de arestas A. Os elementos pertencentes ao conjunto A são pares não ordenados de elementos do conjunto V. A Figura 2.3 mostra um grafo $G = (V, A)$, onde $V = \{1, 2, 3, 4, 5, 6\}$ e $A = \{(1, 5), (1, 2), (2, 5), (5, 4), (2, 3), (4, 3), (4, 6)\}$. Esse grafo não possui arestas múltiplas e nem laços sendo, portanto, chamado de *grafo simples*.

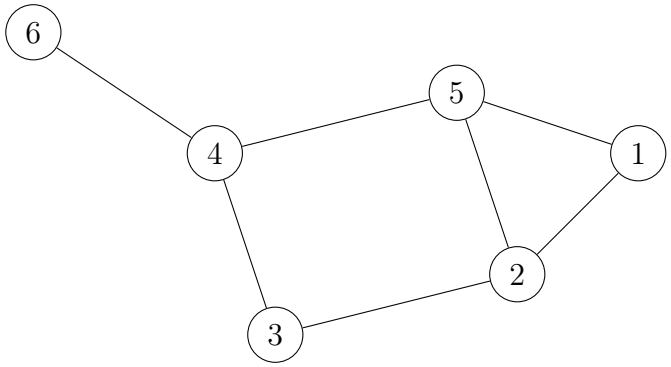


Figura 2.3: Um grafo simples.

A literatura apresenta diversos tipos de grafos e diversas formas de definí-los [Harary 1969, Townsend 1987, Buckley e Harary 1990, Gross e Yellen 2006]. Porém, neste texto, levaremos em consideração apenas as definições relativas aos grafos simples. Em um grafo simples, vértices adjacentes são aqueles que determinam uma aresta. Por exemplo, na Figura 2.3, os vértices $1, 2$ e 5 são adjacentes entre si e o vértice 4 não é adjacente ao vértice 2. O *grau* de um vértice, denotado por $g(v)$, é o número de arestas incidentes no mesmo. No caso da Figura 2.3 tem-se $g(3) = 2$ e $g(2) = 3$.

Um *caminho* em um grafo simples $G = (V, A)$, partindo de um vértice v_0 até um vértice v_n, é uma sequência de arestas $C = \langle a_0, a_1, \ldots a_n \rangle$ tal que $v_0 \in a_0$,

$v_n \in a_n$ e $(a_{i-1} \cap a_i) \neq \emptyset$ para $i = 1 \ldots i = n$. No grafo da Figura 2.3, a sequência $C_1 = \langle (1,2), (2,3), (3,4), (4,6) \rangle$ forma um caminho que começa no vértice 1 e termina no vértice 6. O *comprimento* de um caminho C é o número de arestas que compõem a sua sequência, por exemplo, o comprimento do caminho C_1 é 4. A *distância* $d(s,t)$ do vértice s até o vértice t em um grafo simples é o comprimento do menor caminho entre s e t, se tal caminho existir; caso contrário, define-se $d(s,t) = \infty$. Dizemos que um grafo $G = (V, A)$ é *conectado* se existe um caminho entre todos os pares de vértices no grafo, ou seja, $\forall\ s, t \in V$ tem-se $0 \leq d(s,t) < \infty$, onde $d(s,s) = 0\ \forall\ s \in V$. Assim, o grafo da Figura 2.3 é conectado e o menor caminho entre os vértices 1 e 3 é a sequência $C = \langle (1,2), (2,3) \rangle$, com $d(1,3) = 2$.

Por fim, denotamos por σ_{st} o número de menores caminhos que partem de $s \in V$ e terminam em $t \in V$, tomando a notação $\sigma(v)_{st}$ como o número de menores caminhos entre s e t que passam por $v \in V$.

2.3.2 Medidas de Centralidade

Em diversos modelos matemáticos que utilizam grafos, a noção de *centro* de uma rede é bastante explorada, onde muitas vezes deseja-se determinar localização de elementos no grafo que atendam algum critério. Por exemplo, pode-se querer determinar um ponto na rede em que o tempo de acesso aos demais pontos seja o menor possível. Em outros casos, deseja-se que esse tempo tenha um limite superior, ou seja, deseja-se que o tempo de resposta entre tal ponto e os demais não ultrapasse um valor pré-determinado.

O estudo aprofundado de centralidade em grafos teve seu início na década de cinquenta, no contexto de redes sociais [Bavelas 1948]. A ideia de centralidade é ancorada no interesse em se estimar o quão importante é um elemento participante de uma rede. A partir dessa motivação, várias medidas foram propostas com o objetivo de avaliar quantitativamente as propriedades que possam transmitir tal importância [Freeman 1979]. A medida *betweenness* foi uma das primeiras sugeridas, cujo cálculo envolve a quantidade de menores caminhos, dentre todos existentes, que passam em um determinado nó [Freeman 1977, Gago, Hurajova e Madaras 2012].

Segundo esse conceito, o cálculo da medida de *betweenness* é dado por:

$$\text{Betweenness}(v) = \sum_{(s \neq v \neq t) \in V} \frac{\sigma(v)_{st}}{\sigma_{st}}. \tag{2.1}$$

Nota-se que essa medida de centralidade pode ter um custo computacional relevante dependendo do tamanho da entrada, uma vez que a quantidade de menores caminhos presentes em um grafo de $|V|$ vértices pode ser estimada superiormente pelo cálculo de $\binom{|V|}{2} = \frac{|V|^2 - |V|}{2}$. Além disso, deve-se levar em consideração o custo computacional necessário para encontrar um único menor caminho.

A métrica *betweenness* é talvez a mais frequentemente utilizada, mas não é a única baseada no cálculo de menores caminhos. Outra medida interessante é aquela relacionada à propriedade *closeness* [Sabidussi 1966]. Essa métrica é calculada por:

$$\text{Closeness}(v) = \frac{1}{\sum_{t \in V} d(v,t)}. \tag{2.2}$$

Nesse caso são somadas todas as distâncias entre o nó em que se deseja mensurar a propriedade e os demais. O inverso dessa soma fornece a noção de *closeness*, ou seja, uma medida da proximidade desse vértice em relação aos outros: quanto maior o valor do *closeness* mais central é o vértice no grafo. Obviamente essa propriedade só é mensurável ou faz sentido em grafos conectados, uma vez que em grafos não conectados existe pelo menos um vértice u tal que $d(v,u) = \infty$ para algum v, e portanto a soma $\sum_{t \in V} d(v,t)$ não é definida quando $t = u$. De fato, nesse caso não existe um caminho entre os vértices v e u e portanto o valor calculado para a métrica *closeness* no vértice v não representaria a sua real relevância no grafo. O mesmo tipo de problema ocorre na Equação 2.1, caso o vértice avaliado seja desconectado (isolado).

Semelhante à medida de *closeness*, encontramos na literatura a medida de *excentricidade* de um nó em uma rede [Hage e Harary 1995]. A medida de excentricidade de um vértice é o valor máximo do conjunto de valores referentes ao cálculo do menor caminho entre o vértice em questão e todos os outros no grafo. Assim, denotamos a excentricidade de um vértice por $e(v) = \max_{t \in V}\{d(v,t)\}$. Os *índices* excentricidade e *closeness* são semelhantes no sentido que ambos transmitem a noção de centralidade em relação à posição de um vértice no grafo, enquanto o *betweenness* e o grau

de um vértice sugerem uma ideia de centralidade que diz respeito à comunicação ou capacidade de vazão de comunicação do vértice.

A maior das excentricidades entre os vértices é definida como o *diâmetro* do grafo, ou seja, é a maior distância entre dois vértices presentes no grafo. Matematicamente, expressamos o diâmetro de um grafo $G = (V, A)$ por $D(G) = \max_{v \in V}\{e(v)\}$. O *raio* do grafo também é baseado na medida de excentricidade, e é dado pela menor excentricidade entre os vértices do grafo. Sendo assim, o raio é denotado por $R(G) = \min_{v \in V}\{e(v)\}$. Os vértices com a menor excentricidade são ditos *vértices centrais*, ou seja, são os vértices $v \in V$ tal que $e(v) = R(G)$. Ao analisar a excentricidade, pode-se verificar que vértices com valores baixos de excentricidade são mais *centrais* no grafo, enquanto aqueles com valores mais altos são mais *periféricos* no grafo.

A métrica menos custosa computacionalmente é aquela cujo conceito de centralidade está relacionado ao grau de um vértice e diz respeito à quantidade de vértices adjacentes que o mesmo possui, ou a quantidade de vértices alcançáveis por um caminho de comprimento igual a 1. Assim, o valor dessa métrica é $g(v)$, e pode ser calculado por:

$$\text{Grau(v)} = \sum_{u \in V} \delta_{u,v}, \text{ onde } \delta_{u,v} = \begin{cases} 1 & \text{se } u \text{ e } v \text{ são adjacentes,} \\ 0 & \text{caso contrário.} \end{cases} \tag{2.3}$$

Quando o grafo é representado por uma matriz quadrada, para calcular a expressão 2.3 basta percorrer as linhas (ou colunas) da mesma, o que é computacionalmente rápido. No caso da representação por lista de vizinhos, se o grafo não for completamente conectado, a computação é otimizada, pois será necessário percorrer apenas os nós adjacentes contidos na lista de adjacência e não todas as linhas (ou colunas) de uma matriz quadrada. O valor calculado para essa métrica expressa a capacidade de comunicação local de um nó na rede.

2.3.3 Medidas de Centralidade em grafos de grande escala

Como vimos na seção anterior, as medidas de centralidade em grafos fornecem importantes informações sobre a relevância de um vértice na topologia do grafo. Além disso, através das medidas de excentricidade dos vértices, podemos extrair uma medida global do grafo por meio do cálculo de seu raio e diâmetro. No entanto,

o cálculo de medidas de centralidade envolve um alto custo computacional. Os algoritmos implementados para o cálculo de *betweenness* têm complexidade $O(n^3)$ no tempo de computação e $O(n^2)$ para o consumo de espaço em memória, onde n é o número de nós no grafo. Tal ordem de complexidade deve-se principalmente à avaliação da soma expressa na Equação 2.1. O algoritmo mais rápido conhecido para o cálculo de *betweenness* tem complexidade $O(n + m)$ para espaço e $O(nm)$ para tempo de execução, onde n é o número de vértices e m é o número de arestas em um grafo não ponderado [Brandes 2001]. Nesse estudo, a melhoria na complexidade é alcançada pelos autores pela identificação e eliminação de somas de termos já avaliados previamente.

A mesma dificuldade ocorre quando estamos interessados em medidas que não envolvem o somatório de uma grande quantidade de termos. Por exemplo, na medida de excentricidade, ainda é necessário o cálculo de todos os menores caminhos referentes a todos os pares de vértices conectados em um grafo. Nesse caso a escolha mais eficiente para o cálculo de menor caminho considerando um vértice de origem é o algoritmo sequencial elaborado por Dijkstra [Dijkstra 1959]. Esse algoritmo tem custo $O(m + n \log n)$ quando implementado usando a estrutura *Fibonacci Heap* nas estruturas de filas de prioridade [Crauser et al. 1998]. Ainda assim, diante da complexidade computacional apresentada, o processamento de grafos com poucos milhares de arestas e vértices torna-se proibitivo em termos de tempo de execução. Portanto, faz-se necessário o uso de novas estratégias para a exploração de grafos grandes.

Uma abordagem que aparece de forma natural é a tentativa de particionar o problema do cálculo dos menores caminhos. No estudo feito em [Möhring et al. 2007] os autores propõem o pré-processamento do grafo de forma a criar partições retangulares no mesmo. Para criar as partições retangulares, é definido um *grid* com coordenadas em 2D que delimitam o seu tamanho e o tamanho das regiões retangulares internas. Posteriormente, uma função associa cada vértice do grafo a uma região interna do *grid*. Também são verificadas quais arestas do grafo fazem parte dos menores caminhos dentro de uma dada partição. Essa informação é coletada para todas as arestas, em relação a cada partição. Tal conhecimento sobre as arestas

é então usado na execução do algoritmo de Dijkstra, de forma a evitar o processamento de arestas desnecessárias.

Na mesma estratégia, que procura dividir o problema em partes menores, é proposto o uso de algoritmos paralelos. Nesse contexto, considerações que dizem respeito ao modelo de arquitetura computacional devem ser avaliadas. Sistemas com memória compartilhada e acessível por alta largura de banda demonstraram ser mais eficientes no processamento paralelo de grafos [Bader, Cong e Feo 2005, Hong, Oguntebi e Olukotun 2011]. Adicionalmente, é importante ter em mente que nem sempre é viável a paralelização de um algoritmo sequencial já conhecido. Nem mesmo é trivial a elaboração de um novo algoritmo paralelo. Contudo, é possível encontrar na literatura esforços para paralelizar algoritmos sequenciais consolidados [Crauser et al. 1998, Bader e Madduri 2006] e para desenvolver bibliotecas de programação que facilitem o desenvolvimento de novos algoritmos paralelos [Buluç e Gilbert 2011].

Outra abordagem, concomitante com o desenvolvimento de algoritmos paralelos, foi apresentada por autores que propõem diferentes medidas de centralidade [Kang et al. 2011]. Tais propostas têm como objetivo permitir a escalabilidade de algoritmos que processam grafos com milhões e até bilhões de vértices e arestas. Por exemplo, no trabalho realizado em [Tan, Tu e Sun 2009], o custo da métrica *betweenness* tem a complexidade de tempo de execução alterada de $O(nm)$, obtida pelo algoritmo sequencial em [Brandes 2001], para a complexidade $O(\frac{nm}{p})$, onde p é o número de processadores que executam a versão paralela. Com m e n na escala de milhões e até bilhões, o número de processadores usados faz pouca ou nenhuma diferença na escalabilidade do algoritmo executado em paralelo. Dessa maneira, os autores propõem novas medidas de centralidade que impliquem em funções de complexidade menos custosas em termos de tempo de execução e de espaço consumido na memória. Essas novas medidas de centralidade, utilizadas pelos autores em [Kang et al. 2011], são medidas *aproximadas* para o cálculo do diâmetro de um grafo. Segundos os autores, a justificativa para usar valores aproximados pode ser demonstrada pelo seguinte procedimento, que calcula a excentricidade do vértice v_i em um grafo $G(V, A)$:

1. Para cada vértice v_i do grafo, considere um conjunto C_i que contenha inicialmente o próprio vértice v_i;

2. Para cada vértice v_i do grafo, atualize seu conjunto C_i, adicionando a este os vértices adjacentes;

3. Para cada vértice v_i, repita a atualização da vizinhança do vértice, adicionando os vértices $t \in V$ tal que a distância seja $d(v_i, t) = 2, 3, \ldots$ e assim sucessivamente. Quando o tamanho do conjunto C_i não aumentar com a atualização, então a maior distância possível entre o vértice v_i e um vértice $t \in V$ do grafo $G(V, A)$ foi encontrada.

Segundo o algoritmo acima, o uso de espaço cresce com complexidade $O(n^2)$, sendo n o número de vértices no grafo. A análise feita é que, para cada vértice, deve-se adicionar n vértices ao conjunto, no pior caso. Com isso, deve existir uma escala de tamanho de grafo onde a computação da excentricidade dos vértices se torna inviável. Os autores propõem contornar tal limitação pela *aproximação* da quantidade de elementos vizinhos do vértice, em detrimento da contagem exata. De fato, dado que o cálculo de todos os menores caminhos em um grafo é um problema combinatorial, é essencial desenvolver projetos de algoritmos pensando desde o início em sua escalabilidade e no uso eficiente de recursos computacionais. Tal necessidade é ainda mais evidente quando estamos trabalhando com grafos com milhões de vértices e arestas.

2.4 Processamento paralelo de grafos

Considere o problema de encontrar o menor caminho entre um vértice de origem e todos os outros vértices em um grafo $G(V, A)$. A Figura 2.4 representa o problema de forma generalizada. O conjunto U é o conjunto de vértices do grafo dos quais são conhecidas as distâncias para o vértice de origem $v \in V$. É imediato concluir que $v \in U$, pois sabe-se que $d(v, v) = 0$. Lembramos que, segundo a definição feita na seção 2.3, a distância entre dois vértices é o comprimento do menor caminho que há entre ambos. O conjunto W é o conjunto dos vértices os quais não se sabe a distância para o vértice de origem. Contudo, para este conjunto, vale a propriedade que todos

os vértices em W são alcançáveis a partir de vértices em U por um caminho de comprimento igual a um. Nota-se que ambos conjuntos são subconjuntos de V, em um grafo $G(V, A)$.

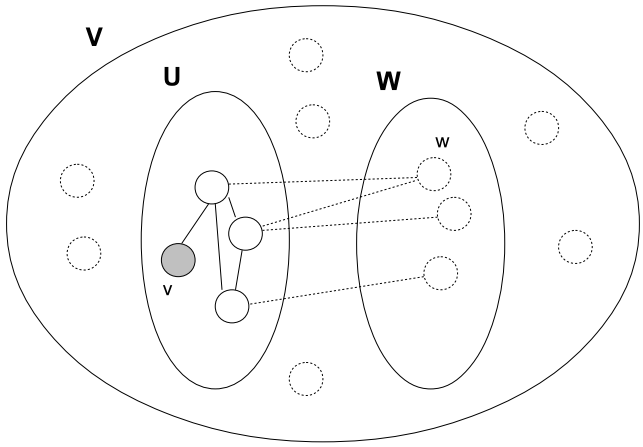


Figura 2.4: Expansão do conhecimento da estrutura de um grafo.

Para calcular os menores caminhos, queremos acrescentar em U os vértices retirados de W, atualizando o conhecimento dos menores caminhos da seguinte forma:

$$d(v, w \in W) = \text{mínimo}\{d(v, u \in U)\} + 1 \tag{2.4}$$

isto é, a distância do vértice de origem v para um vértice w em W é a menor das distâncias entre v e os vértices em U que alcançam w, acrescida de 1. Com essa informação, w passa pertencer a U.

À medida que aumenta-se o conjunto U com os vértices retirados do conjunto W, este último também é renovado com o acréscimo dos vértices alcançáveis a partir dos novos elementos de U. Assim, W faz o papel do conjunto de vértices recentemente descobertos, enquanto U faz o papel do conjunto de vértices que já são conhecidos. Quando $U = V$, o que também implica em $W = \emptyset$, todos os menores caminhos em relação ao vértice de origem foram calculados.

Em algoritmos sequenciais como *Breadth-first search* e *Depth-first search*, estruturas de dados como filas FIFO (*First-In, First-Out*) e pilhas são usadas

para o controle da atualização do conjunto W descrito acima [Ziviani 2004, Cormen et al. 2009]. No caso de algoritmos paralelos, o objetivo é executar paralelamente o processo de atualizar o conjunto W. Essa atualização pode ser realizada com a passagem de mensagens entre processos paralelos, como feito nos trabalhos [Buluç e Madduri 2011, Malewicz et al. 2010]. Deseja-se não só acrescentar paralelamente diversos vértices ao conjunto U, pela atualização também paralela de W, mas adicionalmente executar este processo simultâneamente em todos os vértices considerando os mesmos como vértices de origem.

2.4.1 Processamento paralelo de grafos usando MapReduce

O modelo MapReduce apresentou resultados que demonstraram sua capacidade em processar paralelamente grandes quantidades de dados de forma eficiente e escalável [Dean e Ghemawat 2004]. Considerando os desafios expostos na seção anterior, o modelo MapReduce é um forte candidato à plataforma de escolha para o desenvolvimento de algoritmos paralelos aplicados ao processamento de grafos grandes. Contudo, para se fazer o uso do modelo, é necessário fazer uma análise acerca da redutibilidade da aplicação alvo ao paradigma de programação MapReduce. Alguns algoritmos mais complexos não são de fácil adaptação ao modelo, enquanto outros são inerentemente paralelizáveis, com o desempenho do MapReduce sendo mais evidente neste último grupo.

No estudo realizado em [Srirama, Jakovits e Vainikko 2012] os autores classificam algumas aplicações científicas em relação ao seu desempenho no modelo MapReduce. Um dos resultados obtidos mostra que o modelo MapReduce tem desempenho prejudicado quando a saída da execução de um ciclo *Map-Shuffle-Reduce* equivale a uma iteração do algoritmo que está sendo adaptado ao modelo, sendo ainda necessário algum tipo de sincronização entre os ciclos. Porém, possivelmente, isso é uma consequência de não se repensar a forma de desenvolver algoritmos paralelos ao usar o paradigma MapReduce. Por exemplo, no estudo realizado em [Cohen 2009], o autor se concentra em elaborar algoritmos para operações em grafos usando uma abordagem que não é focada na tentativa de adaptar ao modelo os algoritmos sequenciais conhecidos. Ao contrário, o raciocínio de construção do algoritmo é com-

patível com a forma que o MapReduce funciona, conseguindo assim desenvolver implementações que usam poucos ciclos do modelo para chegar ao resultado final. O autor conclui que é possível implementar operações em grafos de forma simples e eficiente, desde que essas operações possam ser caracterizadas por permitir processamento local. Em contrapartida a essa caracterização, encontram-se as operações que se baseiam na técnica de busca em profundidade. Tais operações, segundo sugere o autor, possuem pouca adaptabilidade ao modelo, formando portanto uma classe de operações em grafos que não é apropriada para execução na plataforma MapReduce/Hadoop.

Outro estudo que segue a mesma linha de pensamento é aquele feito em [Lin e Schatz 2010]. Nesse trabalho, os autores mostram um conjunto de boas práticas para o desenvolvimento de projeto de algoritmos para grafos usando o modelo MapReduce. Essas práticas são compostas por três técnicas, chamadas *In-Mapper Combiner*, *Schimmy* e *Range Partitioning*. Todas elas foram inspiradas na dinâmica de funcionamento do modelo MapReduce. A técnica *In-Mapper Combiner* consiste em pré-processar a saída da fase *Map*, reduzindo assim a quantidade de dados transferidos pela fase *Shuffle* e diminuindo o trabalho realizado pelos nós *reducers*. Em outras palavras, a técnica objetiva reduzir o tempo gasto na fase *Reduce* com a diminuição do tamanho da saída da fase *Map*. Quando se usa técnica *Schimmy*, a intenção é transmitir da maneira mais eficiente possível, entre um ciclo e outro, informações sobre a estrutura do grafo. Por fim, a técnica *Range Partitioning* consiste em projetar uma função de partição que aproveita informações sobre a topologia do grafo. Por exemplo, é mais interessante criar uma função de partição que prioritariamente coloca vértices vizinhos na mesma partição de saída da fase *Map* (veja Figura 2.2), pois assim qualquer processamento que envolva ambos é feito localmente, ou seja, na memória do mesmo nó *reducer*. Combinando essas três práticas de projeto, os autores conseguiram ganhos de desempenho de até 70% em relação a implementações básicas.

De uma maneira geral, as ideias dos artigos citados nessa seção convergem para a noção de que um esforço deve ser feito no sentido de buscar o processamento local ao se usar o modelo MapReduce na exploração de grafos. Isso significa que devemos

concentrar em técnicas que privilegiam o processamento local dentro de um mesmo ciclo *Map-Shuffle-Reduce* (*In-Mapper combine, Range Partitioning*) bem como nas técnicas que favorecem o processamento local entre ciclos (*Schimmy*, técnicas de busca em largura em detrimento de busca em profundidade).

2.4.2 Algoritmo HADI

O algoritmo HADI *(HAdoop based DIameter estimator)* [Kang et al. 2011] é um algoritmo implementado em MapReduce/Hadoop para estimar o diâmetro de grafos com tamanhos na ordem de petabytes. Para fazer o cálculo aproximado do diâmetro, o algoritmo faz uso do conceito de *diâmetro efetivo*, cuja definição é o comprimento mínimo em que 90% dos pares de vértices conectados no grafo são alcançáveis entre si. A principal operação do algoritmo é contar, paralelamente, o número de pares que se alcançam dentro de um caminho de comprimento com tamanho h. Com essa operação, denotada por $N(h)$ e que usa o *algoritmo de Flajolet-Martin* [Flajolet e Martin 1985] para estimar a contagem de pares conectados, é possível fazer o cálculo aproximado do diâmetro do grafo. Como o cálculo não é exato, o algoritmo alcança bons resultados em termos de escalabilidade com o número de máquinas e em termos de tempo de execução com o número de vértices. Tendo em vista a impossibilidade atual de se calcular o valor exato de métricas de centralidade em grafos muito grandes, a pergunta que surge é: para que tamanhos de grafos (nº de vértices e arestas) é possível fazer o cálculo exato do raio e do diâmetro? Essa pergunta se faz necessária pois, para aqueles interessados no cálculo do diâmetro de grafos grandes, cabe escolher qual resultado - exato ou aproximado - melhor atende as suas necessidades.

2.4.3 Algoritmo HEDA

Também desenvolvido para a plataforma MapReduce/Hadoop, o algoritmo HEDA *(Hadoop based Exact Diameter Algorithm)* [Nascimento e Murta 2012] é um algoritmo baseado na expansão de fronteiras para descoberta de vértices vizinhos, técnica chamada de busca em largura. O algoritmo faz a expansão paralelamente, calculando a distância entre um nó origem e todos os outros de forma simultânea e exata, na medida que avança na descoberta de novos vértices. Um laço verifica se

a expansão foi feita a partir de todos os vértices presentes no grafo, considerando em cada interação a existência de nós cuja distância para os outros ainda não foi processada. Dessa forma, o HEDA calcula a distância entre todos os vértices com exatidão.

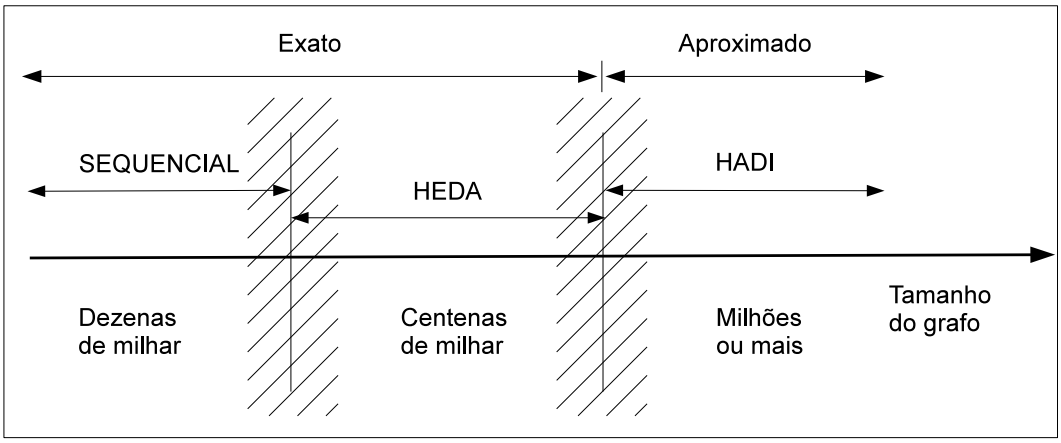


Figura 2.5: Faixas de atuação dos algoritmos HEDA, HADI e sequencial no cálculo de diâmetro em grafos de diversos tamanhos.

A Figura 2.5 exibe as escalas de tamanhos de grafos onde os algoritmos HEDA, HADI e sequencial atuam. A região hachurada representa uma flexibilidade na transição entre a utilização de um algoritmo e outro, dependendo do tamanho do grafo e dos recursos computacionais utilizados no processamento. Por exemplo, um algoritmo sequencial pode processar grafos maiores ou menores dependendo do ambiente de execução. No entanto, os limites podem ser facilmente alcançáveis frente à quantidade de dados que observamos atualmente. Há um espaço para atuação do algoritmo HEDA que é explorado através do paralelismo, afim de obter resultados exatos para grafos que são impossíveis de serem processados por soluções sequenciais. Os resultados apresentados em [Nascimento e Murta 2012] mostram que o algoritmo HEDA obteve bom desempenho em relação a escalabilidade, eficiência e *speedup* em sua escala de atuação.

2.5 Medidas de desempenho de algoritmos paralelos

Definir uma medida de desempenho de algoritmos paralelos pode ser uma tarefa difícil. Em um ambiente de programação paralela, há diversos parâmetros que estão envolvidos na execução de um algoritmo desenvolvido para este ambiente. Assim, métricas para medição de desempenho podem passar pela *eficiência* do algoritmo paralelo, taxa de entrada/saída de dados na memória, taxa de entrada/saída de dados nos discos, taxa de transmissão da rede e muitos outros. A importância de se medir cada uma dessas métricas deve ser baseada nos requisitos de sistema, que variam de aplicação para aplicação.

Sabemos que o modelo MapReduce/Hadoop foi desenvolvido de forma a proporcionar ao programador a concentração de seu esforço apenas no desenvolvimento das funções que serão aplicadas paralelamente, deixando as preocupações em relação ao desempenho de outros componentes do ambiente paralelo para camadas inferiores da plataforma. Diante dessas considerações, podemos inicialmente reduzir para duas a quantidade de métricas que estaremos interessados: *speedup* e *eficiência* - medidas que estão relacionadas ao tempo de execução de um algoritmo e ao uso eficiente dos recursos computacionais. Ambas serão definidas formalmente nas seções seguintes.

Adicionalmente, estamos interessados na *escalabilidade* de um algoritmo paralelo com o incremento do número de processadores que o executa. Na execução de um algoritmo, se há diminuição no número de tarefas executadas em cada processador quando aumenta-se o número de processadores paralelos, sem que haja a necessidade de alteração no algoritmo, dizemos que o mesmo é escalável [Foster 1995]. Essa propriedade pode ser inferida medindo-se o tempo de execução de um algoritmo paralelo com diferentes quantidades de processadores.

2.5.1 Lei de Amdahl

A Lei de Amdahl afirma que o acréscimo de desempenho provido por algum método de aprimoramento é limitado pela extensão do uso desse método [Patterson e Hennessy 2008]. Em outras palavras, não se espera o aumento de desempenho na mesma proporção do tamanho do aprimoramento implementado, a não ser que este último seja completamente efetivo. Deve-se levar em consideração

o quão esse aprimoramento de fato é usado, independente da sua dimensão. Considerando como parâmetro de desempenho de um computador o tempo de execução de um dado programa, a Lei de Amdahl pode ser expressa pela seguinte equação:

$$T_M = \frac{T_A}{K} + T_{NA} \tag{2.5}$$

onde T_M é o tempo de execução após a implementação da melhoria, T_A é a porção de tempo que é afetada pela melhoria, K é a proporção da melhoria e T_{NA} é a porção de tempo que não é afetada pela melhoria. A Lei de Amdhal também pode ser expressa pelo conceito de *speedup*, que é a relação entre o desempenho de um computador antes do uso de alguma melhoria e o desempenho após uso da melhoria:

$$\text{Speedup} = \frac{T_{SM}}{T_M} \tag{2.6}$$

onde T_{SM} é o tempo de execução antes da implementação do aprimoramento. Substituindo a equação 2.5 na equação 2.6 obtemos:

$$\text{Speedup} = \frac{T_{SM}}{\frac{T_A}{K} + T_{NA}} = \frac{T_{SM}}{\frac{T_A}{K} + (T_{SM} - T_A)} \tag{2.7}$$

Sabendo que o tempo T_A é uma porção do tempo T_{SM}, podemos denominar de $\alpha = \frac{T_A}{T_{SM}}$ a porcentagem de tempo em que a melhoria de fato é usada. Assim, dividindo o numerador e denominador da equação 2.7 por T_{SM} e considerando que o tamanho da melhoria é a quantidade $K = P$ de processadores em paralelo, finalmente obtemos:

$$\text{Speedup} = \frac{1}{\frac{\alpha}{P} + (1 - \alpha)} \tag{2.8}$$

2.5.2 Eficiência

A eficência de um algoritmo que está sendo executado em paralelo é a métrica que representa o quão bem os recursos computacionais estão sendo usados paralelamente [Foster 1995]. Essa medida pode ser calculada dividindo-se o Speedup pelo número de processadores paralelos usados, obtendo assim:

$$\text{Eficiência} = \frac{T_1}{T_P} \cdot \frac{1}{P} = \frac{\text{Speedup}}{p} \tag{2.9}$$

onde T_1 é o tempo de execução usando um processador e T_P é o tempo de execução usando P processadores em paralelo.

2.6 Considerações finais

Nas Seções 2.1 e 2.2 deste capítulo descrevemos em detalhes o modelo MapReduce/Hadoop, discutindo na Seção 2.4 as possibilidades de utilização desse modelo para o processamento de grafos grandes. A motivação para tal discussão emerge da análise feita na Seção 2.3, onde são apresentadas as medidas de centralidade em grafos e os desafios de calcular essas medidas. Nas Seções 2.4.2 e 2.4.3 apresentamos dois algoritmos paralelos para o cálculo de centralidade em grafos grandes, implementados no modelo MapReduce/Hadoop. Esses são os únicos algoritmos encontrados na literatura até o momento da escrita dessa proposta. Finalmente, na Seção 2.5 foram apresentadas e discutidas medidas de desempenho para algoritmos paralelos, com o objetivo de definir os termos *speedup, eficiência* e *escalabilidade*.

Nesse capítulo, podemos observar que o problema de computar o valor exato do diâmetro e raio de um grafo grande passa pelo crivo da quantidade de vértices e arestas que estão presentes no grafo. Porém, diante das técnicas e observações acerca do processamento de grafos pelo modelo MapReduce/Hadoop, verifica-se que há espaço para a exploração da flexibilidade de atuação do algoritmo HEDA observada na Figura 2.5.

Capítulo 3

Algoritmo AHEAD

Neste Capítulo apresentamos o algoritmo AHEAD, um acrônimo para *Advanced Hadoop Exact Algorithm for Distances*. O algoritmo AHEAD foi desenvolvido sob o modelo Hadoop/MapReduce e implementado na linguagem Java, para o cálculo de raio e diâmetro de grafos grandes.

3.1 Histórico

No ano de 2007 a empresa Google em conjunto com a Universidade de Washington promoveram um curso sobre o modelo *MapReduce* e sua aplicações. Em particular, durante uma aula intitulada *Problem Solving on Large Scale Clusters*, foi proposto o uso do modelo *MapReduce* para a implementação de um algoritmo de busca em largura. Essa aula está disponível na Internet em formato de vídeo[1], e a referência acerca de sua existência ocorre pela primeira vez em [Kimball, Michels-Slettvet e Bisciglia 2008].

Uma menção a respeito das aulas descritas acima também é feita em [Lin e Schatz 2010]. Nesse trabalho, como já citado na Seção 2.4 do Capítulo 2, os autores propõem padrões de projeto para o processamento de grafos usando a plataforma Hadoop. À época da publicação do artigo, os autores chamaram a atenção para o fato de haver poucas referências na literatura que abordassem o processamento de grafos usando o modelo *MapReduce*. Tal escassez de referências foi motivação para que um dos autores escrevesse um livro com análise sistemática a

[1]Disponível em: *Google Developers, Cluster Computing and MapReduce Lecture 5, August 28, 2007* - `http://www.youtube.com/watch?v=BT-piFBP4fE` - Último acesso em maio de 2013.

respeito das potencialidades do uso do modelo *MapReduce*, incluindo processamento de grafos [Lin e Dyer 2010].

O algoritmo AHEAD é fundamentado nas ideias e técnicas apresentadas em [Lin e Schatz 2010] e [Lin e Dyer 2010] que, em nosso entendimento, são complementares. Em [Lin e Schatz 2010], há a descrição de técnicas que favorecem o uso da plataforma Hadoop para processamento de grafos direcionados, considerando que em toda iteração as seguintes características são observadas:

1. O processamento de cada vértice ocorre em função do estado do vértice (ex. visitado, não visitado) bem como de sua localização na estrutura do grafo (ex. probabilidade de ser visitado);

2. Resultados parciais, em formato de mensagens arbitrárias, são passadas por meio de arestas direcionadas a todos os vértices vizinhos do vértice em processamento;

3. O processamento de cada vértice baseia-se nos resultados parciais recebidos, possivelmente alterando o estado do vértice em processamento.

A partir dessas três premissas, os autores propõem técnicas de projetos para algoritmos baseados na plataforma Hadoop/MapReduce, a saber, as técnicas *In-Mapper Combiner*, *Range Partitioning* e *Schimmy*. Para demonstrar a eficiência dessas técnicas, os autores utilizam o *PageRank* [Page et al. 1999], um conhecido algoritmo para cálculo da importância de um vértice em função da topologia do grafo. Contudo, os autores não fazem referências a algoritmos baseados na busca em largura.

O trabalho [Lin e Dyer 2010], por sua vez, apresenta um estudo do algoritmo de busca em largura considerando o uso da plataforma Hadoop. Esse estudo discorre acerca do algoritmo de busca em largura que considera como ponto de partida um vértice de origem e como destino todos os outros vértices do grafo. Nesse trabalho, os autores não mencionam as técnicas *Range Partitioning* e *Schimmy*.

Segundo [Nascimento 2011], os *slides* produzidos nas palestras do curso descrito em [Kimball, Michels-Slettvet e Bisciglia 2008] formam a base do desenvolvimento

do algoritmo HEDA. O autor implementa os conceitos de busca em largura, considerando várias origens, por meio de um arquivo auxiliar o qual as expansões na fronteira entre os vértices com distâncias conhecidas e não conhecidas são registradas.

O algoritmo AHEAD, utilizando as ideias expostas nos trabalhos [Lin e Schatz 2010] e [Lin e Dyer 2010], implementa o uso das técnicas *In-Mapper Combiner* e *Schimmy* para o caso de busca em largura considerando cada vértice como fonte e todos os restantes como destino. A técnica *Range Partitioning* não é implementada por ser efetiva apenas para grafos que representam domínios da *Web*. O objetivo é encontrar todas as distâncias existentes entre todos os vértices do grafo, afim de calcular o raio e diâmetro de grafos grandes de maneira exata. A implementação das técnicas acima mencionadas permite que o algoritmo AHEAD processe grafos maiores, de acordo com o sentido explicado na Figura 2.5.

3.2 Visão Geral do Algoritmo AHEAD

O modelo *MapReduce* foi projetado para processamento paralelo de grandes quantidades de dados. No contexto deste trabalho, grande quantidade de dados significa que os mesmos não cabem na memória de uma única máquina. Assim, mesmo que um grafo com bilhões de elementos esteja contido em um único arquivo, digamos, de um *terabyte* (10^{12} bytes) no disco rígido, atualmente é impossível que tenhamos essa mesma quantidade de memória RAM em apenas uma máquina. Por exemplo, para a execução do algoritmo de Dijkstra [Dijkstra 1959], o mais eficiente conhecido para cálculo de distâncias, é necessário carregar todo o grafo na memória, ou seja, não é possível usar o algoritmo de Dijkstra nas condições descritas acima. A justificativa do uso de sistema distribuído nasce desse fato e, particularmente, o modelo *MapReduce* utiliza o artifício de dividir em blocos os dados que serão processados separadamente. Assim, cada máquina do *cluster* processa o conjunto de dados que é viável em termos de processamento local.

Nesse aspecto, podemos destacar uma diferença importante em relação ao algoritmo HEDA. Quando o HEDA é executado na plataforma Hadoop, todo o grafo é carregado na memória local de cada um dos nós computacionais. O que o HEDA faz

de forma distribuída é o processamento local de uma parte do grafo, que é completamente carregado na memória. Portanto ocorre a distribuição do processamento, e não dos dados do grafo. Para resolver esse problema, no AHEAD foram implementadas as técnicas *Hash Partitioning* e *Schimmy*, esta última proposta pelos autores em [Lin e Schatz 2010]. Como veremos nesse Capítulo, o algoritmo AHEAD faz a distribuição do processamento e dos dados referentes ao grafo.

3.2.1 Divisão do grafo entre os *Mappers*

No caso do processamento paralelo de grafos usando o modelo *MapReduce*, a divisão dos dados de entrada significa a divisão da estrutura do grafo no sistema distribuído. Isso quer dizer que, dado um certo número de entidades computacionais, ou seja, aquelas que de fato irão fazer o cálculo desejado no grafo, a estrutura do grafo deve ser distribuída entre essas entidades computacionais na forma de subconjuntos excludentes. Na plataforma Hadoop, essas entidades são máquinas virtuais Java executando a função *Map* em um determinado bloco de dados, ou seja, em uma determinada parte do grafo. Referimos a essas máquinas virtuais Java como *mappers*.

Consequências importantes derivam do fato de se dividir a estrutura do grafo entre os *mappers* instanciados e gerenciados pela plataforma Hadoop. O modelo Hadoop/MapReduce é fortemente baseado no processamento local dos nós do sistema distribuído, o que implica na inexistência do conceito de compartilhamento global de endereços de memória. Isso significa que um determinado *mapper*, executando um cálculo em uma determinada parte do grafo, não compartilha os dados calculados em memória com os demais *mappers* do sistema distribuído. Como estamos interessados em calcular distâncias entre vértices que certamente estarão em blocos diferentes, e possivelmente em máquinas diferentes, é necessário atualizar os resultados de cálculos em partes do grafo que estão em outros nós de processamento, bem como receber informações de outras partes do grafo para a atualização local. Tal dinâmica exige, portanto, que resultados parciais sejam armazenados localmente de maneira que possam ser repassados devidamente para as iterações seguintes do processamento.

Essa característica é coerente com as premissas descritas na Seção 3.1. No contexto da plataforma Hadoop, esses resultados parciais são materializados em pares chave/valor armazenados no disco da máquina local, isto é, no disco da máquina que processa determinada parte do grafo. A recepção e a síntese dos resultados parciais deverão ser feitos por entidades computacionais as quais denominamos *reducers*. Assim como os *mappers*, os *reducers* são máquinas virtuais Java instanciadas e gerenciadas pela plataforma Hadoop.

3.2.2 Recepção e síntese dos resultados parciais nos *Reducers*

Os resultados parciais gerados pelos *mappers* contêm informações a respeito da partição local do grafo processado. Essas informações devem ser utilizadas para atualizar as outras partições do grafo, bem como a partição local. A síntese diz respeito a necessidade de, dentre as informações recebidas pelos *reducers*, selecionar aquelas que atendem algum critério de satisfação relacionado ao problema proposto. Por exemplo, um critério poderia ser definido como sendo a menor distância entre dois vértices calculada até o momento.

Uma consequência direta da síntese e posterior atualização das partições do grafo é a necessidade de escrever as informações no sistema de arquivo distribuído do Hadoop, o *Hadoop Distributed File System*. Portanto, os *reducers* terão que realizar operações de I/O sobre arquivos distribuídos pré-existentes no HDFS, isto é, na partição do grafo sob a responsabilidade do *reducer*, o que implica na função *reduce* ter um nível de complexidade maior.

3.2.3 Tipos de grafos processados pelo algoritmo AHEAD

O algoritmo AHEAD processa grafos direcionados ou não direcionados, sempre não ponderados. Por grafos não ponderados entende-se que a distância entre dois vértices vizinhos é sempre um. Pela definição dada no Capítulo 2, Seção 2.3, de raio e diâmetro de um grafo, tais medidas de centralidade são válidas apenas para grafos conectados. Contudo, mesmo que o grafo seja desconectado, ainda é possível interpretar o valor de diâmetro e raio retornado pelo algoritmo AHEAD. Em casos de grafos desconectados, o algoritmo retornará um valor de diâmetro que deve ser entendido como o diâmetro do maior componente conectado e, da mesma forma, o

raio retornado deverá ser entendido como o raio do menor componente conectado.

3.3 Funcionamento do Algoritmo AHEAD

Para o cálculo do raio e diâmetro de um grafo, o algoritmo AHEAD executa as fases listadas a seguir:

1. Particionamento da matriz de distâncias do grafo em R partes no HDFS, onde R é o número de reducers especificados. As R partições são criadas em paralelo;

2. Cálculo paralelo das distâncias entre todos os vértices e atualização paralela no HDFS das partições matriciais do grafo;

3. Cálculo paralelo da excentricidade de todos os vértices. As excentricidades são usadas para o cálculo de raio e diâmetro do grafo.

A representação matricial do grafo informa que a posição de linha i e coluna j na matriz é ocupada pelo valor calculado da distância entre o vértice destino i e o vértice origem j. A matriz é criada a partir da leitura de cada linha da lista de adjacências do grafo, que é o arquivo de entrada do algoritmo. A lista de adjacências contém em cada registro um vértice do grafo e a sua lista de vizinhos. No momento da criação da matriz, os valores da diagonal principal são preenchidos com zero. Os cálculos das excentricidades de todos os vértices do grafo são feitos na terceira fase. À medida que as excentricidades são calculadas, as mesmas são avaliadas para que seja selecionada a maior e a menor excentricidade. Como definido no Capítulo 2, Seção 2.3, a maior e a menor excentricidade são o diâmetro e raio do grafo, respectivamente.

O Algoritmo 1 mostra o pseudo-código principal do algoritmo AHEAD. As linhas de número (1) a (6) constituem a primeira fase da execução, na qual a lista de adjacências é convertida em uma matriz de distâncias particionada em R partes. A lista de adjacências contém em cada registro um vértice do grafo e a sua lista de vizinhos. Na linha (4), *mappers* enviam em paralelo os registros da lista de adjacência para uma determinada partição G_i. Na linha (5), *reducers* recepcionam e fazem a expansão dos registros com as colunas correspondentes as distâncias para os

Algoritmo 1: Programa Principal do AHEAD.

```
1: G ← lista_de_adjacências;
2: R ← número_de_reducers;
3: for i=1 to R do in parallel
4:     G_i ← CriaParticaoDeMatriz(G);
5:     HDFS ← G_i;
6: end for
7: while (condição de parada = false) do
8:     for all G_i ∈ G = {G_1 ∪ G_2 ∪ ... ∪ G_R} do in parallel
9:         CalculaMenoresCaminhos(G_i);
10:        AtualizaDistancias(G_i);
11:    end for
12:    Atualiza(condição de parada);
13: end while
14: for all G_i ∈ G = {G_1 ∪ G_2 ∪ ... ∪ G_R} do in parallel
15:    for all vértice v ∈ G_i do
16:        CalculaExcentricidade(v)
17:        D_i ← MaiorExcentricidade();
18:        R_i ← MenorExcentricidade();
19:    end for
20: end for
21: repeat in parallel
22:    Diâmetro ← RemoveMenor({D_1, D_2, ..., D_R});
23:    Raio ← RemoveMaior({R_1, R_2, ..., R_R});
24: until {D_1, D_2, ..., D_R} = ∅ and {R_1, R_2, ..., R_R} = ∅
```

outros vértices do grafo. Se G é o grafo representado pela lista de adjacências, então as partições da matriz serão criadas em paralelo no HDFS respeitando a relação $G = G_1 \cup G_2 \cup ... \cup G_R$. A partição à qual um registro é enviado e posteriormente estendido é determinada pela função de partição *Hash Partitioning*. Essa função distribui os registros baseando-se em uma função *hash*, o que proporciona partições com tamanhos homogêneos. Ao final dessa primeira fase, existirão R arquivos no HDFS, cada um contendo uma parte da matriz de distâncias do grafo. Essa é a única fase em que a estrutura do grafo é transmitida via processo *shuffle*, o que coloca o AHEAD de acordo com o padrão de projeto *Schimmy* proposto em [Lin e Schatz 2010]. A segunda fase de execução é realizada nas linhas (7) a (13). Na linha (9), *mappers* executam em paralelo as operações do cálculo de distâncias entre os vértices do grafo, de acordo com as informações disponíveis em uma partição G_i. Na linha (10), os *reducers* acessam a partição do grafo, ou seja, o arquivo do grafo o qual receberá

as atualizações das distâncias calculadas. O laço *while* na linha (7) é interrompido apenas quando todas as distâncias tiverem sido calculadas e atualizadas nas partições. Tanto o cálculo dos menores caminhos quanto as atualizações (linhas 9 e 10) são feitos nas partições de forma paralela, isto é, as partições $G_1, G_2, \ldots, G_R$ são processadas simultâneamente. Os cálculos feitos em uma partição G_i podem ser usados para atualizar qualquer uma das partições $G_i \in G = \{G_1, G_2, ..., G_R\}$, ou seja, não é necessariamente para atualizar a própria partição.

A terceira fase é executada nas linhas de número (14) a (24). Após o cálculo dos menores caminhos, cada partição é novamente processada em paralelo com as demais, agora para o cálculo de excentricidade de seus vértices. Para que seja calculada a excentricidade de um vértice (linha 16), seleciona-se a maior distância calculada entre ele e algum outro vértice no grafo. Porém, para o cálculo do diâmetro e raio do grafo, esses valores de excentricidade precisam ser comparados com os valores de excentricidade obtidos nas demais partições, de maneira que seja possível selecionar a maior e a menor excentricidade global. Para tanto, nas linhas (15) a (19) *mappers* identificam as maiores e as menores excentricidades locais, para que nas linhas (21) a (24) *reducers* selecionem a maior e a menor excentricidade global. Especificamente, dois *reducers* trabalham em paralelo (linhas 22 e 23) - um para identificação do diâmetro e outro para identificação do raio. Nas próximas subseções serão detalhadas as fases acima descritas.

3.3.1 1ª Fase: Partição do grafo

A forma como o grafo é particionado influencia no desempenho do algoritmo AHEAD, uma vez que o mesmo é desenvolvido na plataforma Hadoop que apresenta melhor desempenho quando há predominância de processamento local de dados. Especificamente para a execução do AHEAD, a melhoria de desempenho ocorrerá quando vértices com grande quantidade de vizinhos em comum forem colocados na mesma partição. Tal melhoria ocorre pela possibilidade de descobrir mais facilmente caminhos equivalentes dentro da mesma partição, isto é, dado um único par destino/origem que possui caminhos diferentes, mas com o mesmo valor de distância, é possível selecionar localmente apenas um deles, descartando os outros e evitando

o uso de espaço em disco para armazenar informações redundantes.

O tamanho de cada partição também é importante para o desempenho do algoritmo, visto que partições de tamanhos iguais devem ter aproximadamente o mesmo tempo gasto no processamento. Caso a distribuição do tamanho das partições não seja bem balanceada, uma partição maior que as outras poderia tornar-se um ponto de contenção para o processamento paralelo. Por exemplo, consideremos a primeira fase do algoritmo AHEAD, a qual a estrutura do grafo é particionada. Se ao executar o AHEAD for especificado o uso de dois *reducers* no Hadoop, serão criadas duas partições locais em cada nó computacional que instanciar *mappers* (reveja Figura 2.2). Suponha que a função *Map* é especificada para que os *mappers*, ao processar a lista de adjacência, emitam pares chave/valor cujas chaves são números naturais identificadores dos vértices e os valores sejam as listas de vizinhos dos respectivos vértices. A função *Reduce* é especificada para concatenar, ao receber o par chave/valor, a lista de vizinhos com as respectivas linhas da matriz de distâncias. De acordo com o funcionamento do Hadoop, cada par produzido deverá ser alocado em uma das duas partições criadas, e a função particionadora é quem define o destino do par. Suponha também que o grafo tenha dez vértices numerados de 1 a 10 e que vamos testar o uso de duas funções particionadoras, F_1 e F_2. A função F_1 coloca vértices identificados com números ímpares na mesma partição enquanto a função F_2 coloca vértices identificados com números múltiplos de sete na mesma partição. Ao usar a função F_1, teremos ambas partições com o mesmo número de pares chave/valor, enquanto o uso de F_2 coloca nove pares a mais em uma das partições. No caso de uso da F_2, o *reducer* reponsável pela partição maior poderá ter o tempo de processamento degradado, assim como o *mapper* que atuará sobre essa partição nas iterações seguintes do AHEAD (cálculo de distâncias) poderá experimentar o mesmo efeito. Portanto, dependendo dos critérios usados para a função particionadora, poderiam ser criadas R partições com tamanhos muito diferenciados, desbalanceando o tempo de processamento dos *mappers* e dos *reducers*.

Para o algoritmo AHEAD, a função particionadora ideal é aquela que cria partições, contendo partes da matriz de distâncias, com tamanhos homogêneos e que possuam a maior quantidade possível de vértices vizinhos em comum. Contudo,

dividir um grafo com partições de tamanhos iguais (i.e. mesmo número de linhas da matriz de distâncias) minimizando a quantidades de vértices vizinhos presentes em partições diferentes é um problema NP-difícil [Muntés-Mulero et al. 2010]. A solução desse problema, apesar de estar relacionada com a melhoria de desempenho do AHEAD, está fora do escopo deste trabalho e assim optamos por implementar uma função particionadora cujo objetivo é apenas criar partições de tamanhos aproximadamente iguais. Essa maneira de criar partições privilegia o processamento paralelo em termos de manter homogêneo o tempo de duração entre as tarefas *mappers* e entre as tarefas *reducers*.

Como mencionado anteriormente, a lista de adjacências contém em cada linha (registro) um vértice do grafo e a sua lista de vizinhos. Em uma sequência *Map-Shuffle-Reduce* auxiliar ou *Job* auxiliar, os registros da lista de adjacências são numerados de acordo com a sua posição dentro do arquivo. Além disso, o número da posição do último registro na lista de adjacências é armazenado, pois o mesmo determina o número de vértices do grafo e consequentemente determina o número de colunas que serão expandidas em cada registro por um *reducer*. Para implementar esse *Job* auxiliar foi utilizado o algoritmo desenvolvido por Gordon Linoff[2], que faz a contagem em paralelo das posições das linhas de um arquivo de entrada no Hadoop. Após essa etapa, a lista de adjacências numerada poderá ser processada para criação das partições da matriz de distâncias do grafo.

A função *Map* processa um registro por vez do arquivo da lista de adjacências e está implementada por meio de um método da classe *Mapper* mostrada no Algoritmo 2. A entrada da função é um par chave/valor, cujo conteúdo da chave é a posição do registro na lista de adjacências e o valor é o próprio registro. A classe *Mapper* implementa uma função imagem, isto é, ele apenas copia a entrada para a saída. O objetivo principal é atingido quando a função de partição é aplicada automaticamente pela plataforma Hadoop em cada saída emitida na linha 3 da classe *Mapper*.

[2]Disponível em: *Data Miners Blog - Hadoop and MapReduce: A Parallel Program to Assign Row Numbers. November 25, 2009* - `http://blog.data-miners.com/2009/11/hadoop-and-mapreduce-parallel-program.html` - Último acesso em maio de 2013.

Algoritmo 2: Classes *Mapper*, *Partitioner* e *Reducer* para a partição do grafo.

```
1: classe MAPPER
2:   método MAP(chave, registro)
3:     EMITE(chave, registro)

1: classe HASHPARTITIONERMODIFICADO
2:   método SELECIONAPARTICAO(chave, registro, numero_de_reducers)
3:     R ← numero_de_reducers
4:     chave ← SEPARA(string_do_vertice, registro)
5:     número_da_particao ← hashcode(chave)MODULO(R)
6:     return numero_da_particao

1: classe REDUCER
2:   método REDUCE(chave, registro)
3:     registro ← CONCATENA(registro, linha_da_matriz_de_distancias)
4:     EMITE(chave, registro)
```

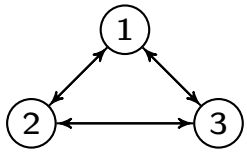
Como foi descrito no Capítulo 2, Seção 2.1.2, cada *reducer* instanciado pela plataforma Hadoop é responsável por processar uma partição de dados. No caso do algoritmo AHEAD, uma partição de dados significa uma partição da matriz de distâncias que representa o grafo. Para estabeler essa relação por meio de uma função particionadora, a função de partição *default HashPartitioner* do Hadoop foi modificada para distribuir as linhas da matriz entre as partições baseando-se na identificação do vértice. Os parâmetros de entradas da função de partição são o número de reducers usados e um par chave/valor emitido por um *mapper*.

A classe *HashPartitionerModificado* no Algoritmo 2 mostra o pseudo-código da função particionadora. Na linha (3) dessa classe, a variável R recebe o número de *reducers* que serão instanciados pela plataforma Hadoop. Na linha (4), a *String* identificadora do vértice, presente no registro, é atribuída a variável chave. Esta chave é convertida em um valor arbitrário via função *hashcode* do Java na linha (5). O resto da divisão entre o valor do *hashcode* e o número de *reducers* especificados é o número da partição a qual o registro será alocado. Assim, todos os vértices e suas respectivas listas de vizinhos serão alocados em uma partição cuja identificação está entre os números 0 e $(R-1)$. Segundo [White 2012], dependendo da qualidade da função *hash* usada, os registros serão alocados de maneira homogênea entre os *reducers*.

A função *Reduce* do *Job* de partição do grafo cria as linhas da matriz com as

posições dos caracteres que representarão as distâncias para uma determinada origem. A classe *Reducer* do Algoritmo 2 mostra a função *Reduce*, que é executada por cada um dos R *reducers* do cluster em suas respectivas partições. A função *Reduce* recebe os pares chave/valor gerados pela função *Map*, descrita na classe *Mapper*, e entregue pelo particionador, descrito na classe *HashPartitionerModificado*. Como descrito anteriormente, o número de vértices do grafo foi armazenado, permitindo que na linha (3) da classe *Reducer* seja criada a linha da matriz com o tamanho necessário para armazenar as distâncias em relação aos outros vértices do grafo.

Como sabemos, cada posição i, j da matriz de distâncias representa a distância da origem j para o destino i. Portanto, cada coluna da linha da matriz de distâncias (linha 3, classe *Reducer*) contém um valor de distância para cada vértice do grafo. Nesse esquema, a n-ésima coluna da linha armazena a distância para o vértice que está n-ésimo registro da lista de adjacências. Por essa razão, as chaves dos pares emitidos na classe *Mapper* devem conter as posições dos registros dentro da lista de adjacências. Usando essa chave, é possível atribuir o valor zero na coluna da linha referente a diagonal da matriz, ou em outras palavras, a diagonal da matriz de distâncias é preenchida com zeros. Finalmente, o par chave/registro emitido na linha (4) da classe *Reducer* contém a linha da matriz de uma determinada partição. Portanto, a emissão dos vários pares recebidos pelo *reducer* formarão uma partição da matriz de distâncias.



(a) Grafo de entrada.

#1	2;3
#2	1;3
#3	1;2

(b) Lista de adjacência.

#1	2;3	0 d d
#2	1;3	d 0 d
#3	1;2	d d 0

(c) Partição no HDFS.

Figura 3.1: Exemplo de criação de uma partição da matriz de distâncias.

A Figura 3.1 mostra um exemplo de criação da partição, a partir da lista de adjacências do grafo. A Figura 3.1(a) é o grafo a ser processado e tem a sua lista de adjacências exibida na Figura 3.1(b). Após processamento do *Job* de partição do grafo, usando apenas um *reducer*, a partição da matriz no HDFS é como está

na Figura 3.1(c). Nessa figura, a primeira linha exibe o nome do vértice destino "#1", a sua respectiva lista de vizinhos "2;3" e a *string* de distâncias "0 d d". Essa *string* descreve que a distância a partir do próprio vértice é zero e a partir dos outros vértices de origem, vértices 2 e 3, a distância é desconhecida. Distâncias desconhecidas são marcadas com o caracter "d". Nesse exemplo foi criada apenas uma partição - caso fossem usados três *reducers*, haveriam três partições e uma linha da matriz mostrada na Figura 3.1(c) por partição. Para cada partição é criado um arquivo, sendo que o *reducer* emite por vez uma linha do arquivo de saída, conforme mostrado na linha 4 da classe *Reducer* do Algoritmo 2. O nome do arquivo de saída no Hadoop é *part-r-0000n* por *default*, onde n é o mesmo número da partição o qual aquele *reducer* é responsável.

Antes do início do cálculo das distâncias entre os vértices, as únicas distâncias conhecidas são aquelas que estão na diagonal da matriz, ou seja, quando o vértice de destino e origem são os mesmos. Essa condição inicial é necessária para que o cálculo das distâncias entre todos os vértices seja feita nos *Jobs* seguintes da execução do AHEAD.

3.3.2 2ª Fase: Cálculo das Distâncias

As operações de cálculo dos menores caminhos são baseadas no algoritmo sequencial de busca em largura [Cormen et al. 2009, Ziviani 2004], onde a distância acumulada de um vértice em relação a uma origem serve de referência para o cálculo da distância de seus vizinhos em relação a mesma origem. Na medida que as distâncias para os novos vértices são calculadas, estas servirão de referência para o cálculo dos menores caminhos na próxima iteração do algoritmo. Dessa maneira, ocorre a expansão da fronteira entre os vértices cujas distâncias para alguma origem são conhecidas e vértices cujas distâncias para a mesma origem ainda serão descobertas. Mais precisamente, o algoritmo de busca em largura (*Breadth-Search First*) encontra todos os vértices com distância k em relação a uma origem antes de descobrir todos aqueles com distância $k + 1$, num procedimento em que a árvore de busca em largura [Ziviani 2004] é criada. Um caminho simples entre o vértice fonte s e um vértice v nessa árvore é o menor caminho entre s e v.

No algoritmo AHEAD, a expansão paralela da fronteira ocorre conforme explicado na Seção 2.4. Mais especificamente, durante o cálculo das distâncias realizado pelo AHEAD, várias árvores de busca em largura são criadas simultâneamente, com cada vértice do grafo sendo a raiz de uma árvore. Assim, em relação ao Algoritmo 1, na iteração k do laço presente nas linhas (7) a (13) serão descobertos todos os pares de vértices destino/origem (ou folha/raiz) cujas distâncias entre si é k. A condição inicial é dada na matriz construída na primeira fase do algoritmo, pois a diagonal é preenchida com zeros, ou seja, as únicas distâncias conhecidas antes do início do laço são aquelas em que o destino é igual a origem. Portanto, na primeira iteração do laço após a criação da matriz, são descobertos os pares cuja distância entre si é $d = k = 1$.

Sempre que os vértices com distâncias entre si iguais a k são descobertos, a atualização de suas distâncias ocorre em uma ou mais partições da matriz do grafo. Essas novas distâncias atualizadas são marcadas de forma que o cálculo da distância dos vértices vizinhos, isto é, dos vértices com distâncias iguais a $k+1$, sejam disparadas na iteração $k+1$. A condição de parada na linha (12) do Algoritmo 1 é baseada nesse fato, pois quando não houver atualização em nenhuma das partições da matriz do grafo, significa que a fronteira de conhecimento sobre as distâncias do grafo chegou ao valor máximo. Uma iteração completa do laço na linha (7) corresponde a um ciclo *map-shuffle-reduce* ou *Job* e, somente quando não houver atualizações durante um *Job* a condição na linha (12) mudará, encerrando o laço. Durante essa fase do AHEAD, os *mappers* instanciados no Hadoop cumprem o papel de calcular a expansão da fronteira, enquanto os *reducers* são responsáveis por atualizar tais distâncias nas respectivas partições da matriz que contêm os respectivos vértices origem e destino. O valor de k correspondente à iteração em que não há atualização feita por algum *reducer* é exatamente o valor do diâmetro do grafo.

No Algoritmo AHEAD, um par chave/valor está no formato *destino.origem/distância*, representando a distância entre vértice destino (folha) e vértice origem (raiz). Dentro de uma partição, durante a busca em largura paralela, poderá ocorrer de caminhos diferentes com o mesmo valor de distância sejam calcu-

Algoritmo 3: Classe *Mapper* para cálculo de distâncias.

```
1: classe Mapper
2:   método Setup()
3:     Visitados ← novo HashSet(chave)
4:     InMapperCombiner ← novo HashMap(chave, valor)
5:   método Map(chave, registro)
6:     vértice_id ← Separa(registro, vértice_id)
7:     vetor_de_vizinhos ← Separa(registro, lista_de_vizinhos)
8:     vetor_de_distâncias ← Separa(registro, lista_de_distancias)
9:     flag ← 0
10:    for all vértice_destino ∈ vetor_de_vizinhos do
11:      for i=0 to Tamanho(vetor_de_distâncias)
12:        d_i ← vetor_de_distâncias[i]
13:        if d_i != infinito and flag == 0 then
14:          chave ← vértice_id@vertice_origem_i
15:          if memória_livre == true then
16:            Visitados.Armazena(chave)
17:          end if
18:        end if
19:        if d_i recém_calculada == true then
20:          if memória_livre == true then
21:            chave ← vértice_destino@vertice_origem_i
22:            InMapperCombiner.Armazena(chave, d_i + 1)
23:          else
24:            chave ← vértice_destino@vertice_origem_i
25:            Emite(chave, d_i + 1)
26:            for all chave/valor ∈ InMapperCombiner do
27:              if chave/valor ∉ Visitados then
28:                Emite(chave, valor)
29:              end if
30:            end for
31:            InMapperCombiner.Apaga()
32:          end if-else
33:        end if
34:      end for
35:      flag ← 1
36:    end for
37:  método Cleanup()
38:    for all chave/valor ∈ InMapperCombiner do
39:      if chave/valor ∉ Visitados then
40:        Emite(chave, valor)
41:      end if
42:    end for
43:    InMapperCombiner.Apaga()
44:    Visitados.Apaga()
```

lados para o mesmo par destino/origem e, portanto, apenas um deles deve ser armazenado no disco local na forma chave/valor. Além disso, se um ou mais vértices vizinhos de um vértice presente na partição em processamento for ou forem vértices de registros presentes nessa mesma partição, deverá ser verificado na lista de distâncias desses últimos se a distância emitida no par *destino.origem/distância* já foi calculada em *Jobs* anteriores. Em outras palavras, deverá ser verificado em uma mesma partição quais vértices já foram descobertos (visitados) e, por isso, já possuem as distâncias calculadas. Finalmente, o algoritmo deverá emitir pares *destino.origem/distância* somente em relação aos vértices que estão na fronteira. Isso se deve ao fato do grafo não ser ponderado, portanto a primeira vez que uma distância for atualizada em uma determinada iteração ou *Job*, está já será a distância mínima.

A classe *Mapper* que implementa a busca em largura está descrita no Algoritmo 3. Na linha (2) está implementado o método *Setup()*, que é executado apenas uma vez durante a existência de uma instância *mapper*. A execução ocorre sempre antes das repetidas execuções do método *Map()*, que é chamado uma vez por registro da partição da matriz. O método *Setup()* é utilizado para a alocação das estruturas de dados que serão usadas durante a existência do *mapper* que processa uma partição. Na linha (3) há a alocação da estrutura *Hashset* do *Java*. Esta estrutura armazena itens não repetidos, ou seja, é um conjunto que deverá armazenar os vértices que já foram visitados dentro de uma partição. Na linha (4) é alocada a estrutura *Hashmap* do Java, que implementará a técnica *In-Mapper Combiner* e é onde os pares *destino.origem/distância* serão armazenados. Os pares contidos nessa estrutura não são emitidos imediatamente no disco local, de forma que toda vez que houver repetições apenas uma será armazenada.

Na linha (5) do Algoritmo 3 está implementada a função *Map*, por meio do método *Map()* da classe *Mapper*. Nos laços aninhados da linha (10) e (11) são executadas operações referentes a cada um dos vizinhos do vértice presente no registro carregado. A primeira operação é realizada no comando condicional da linha (13). A condição [d_i ! = infinito] garante que apenas os vértices já visitados dentro da partição sejam registrados no *Hashset*. A condição [*flag == 0*] muda na linha (35), garantindo que a operação condicional da linha (13) seja executada apenas na

primeira iteração do laço da linha (10). Isto é assim implementado porque estamos interessados nas distâncias que já foram calculadas para o vértice do registro carregado, não para seus vizinhos.

Nas linhas (19) a (23) os cálculos das distâncias dos vizinhos dos vértices na fronteira são feitos e armazenados no *Hashmap*. A condicional da linha (19) garante que apenas sejam usadas referências de distâncias atualizadas no *Job* imediatamente anterior para o cálculo feito na linha (22). Nas linhas (23) a (29) desenvolvemos uma técnica a qual chamaremos de *In-Mapper Buffer Combiner*. Nessa técnica, toda vez que a memória virtual da instância *mapper* chega ao seu limite de capacidade, os pares armazenados no *Hashmap* são emitidos no disco por meio das linhas (25) e (28). A linha (27) garante que as emissões ocorram desde que não sejam de pares com valores já calculados em *Jobs* anteriores. Por fim, o método *Cleanup()* é chamado na linha (37). Esse método também é executado apenas uma vez durante a existência do *mapper*, sempre depois que todas as chamadas do método *Map()* ocorrerem, isto é, depois que todos os registros da partição tiverem sido processados. Nas linhas (38) a (42) são emitidos os últimos pares presentes no *Hashmap InMapperCombiner*. Isso pode ser necessário caso a condição da linha (20) não seja satisfeita, ou seja, quando a técnica *In-Mapper Buffer Combiner* não precisou ser utilizada.

A função particionadora que atua nos pares emitidos pelos *mappers* dessa fase tem importância fundamental no funcionamento do algoritmo AHEAD. Esta função deve distribuir os pares entre as partições exatamente da mesma forma que a função particionadora da primeira fase o fez. Dessa maneira, será garantido que os pares com os valores de distância a serem atualizados serão encaminhados aos *reducers* responsáveis pela partição que contém os vértices envolvidos na atualização.

A classe *HashPartitionerMenoresCaminhos* mostrada no Algoritmo 4 implementa a função particionadora da fase de cálculo das distâncias. Na linha (2), o método *SelecionaParticao()* recebe como parâmetro o par *destino.origem/distância* emitido por um *mapper* e o número de *reducers* especificado. Na linha(4), a *String* que representa o vértice de destino é armazenda na variável chave. Essa chave é convertida em um valor pela mesma função *hash* usada

Algoritmo 4: Classe *Partitioner* do cálculo de distâncias.

```
1: classe HashPartitionerMenoresCaminhos
2:   método SelecionaParticao(destino@origem, distancia, numero_de_reducers)
3:     R ← numero_de_reducers
4:     chave ← SeparaDestino(destino@origem, distancia)
5:     número_da_particao ← hashcode(chave)MODULO(R)
6:     return numero_da_particao
```

durante a fase de partição do grafo. Assim, o número da partição retornado na linha (5) deverá ser coerente com a função particionadora da primeira fase do algoritmo AHEAD (Algoritmo 2).

A função *Reduce* da fase de cálculo das distâncias deve receber todos os pares chave/valor distribuídos pela função particionadora, isto é, todos os pares *destino.origem/distância* pertencentes a uma determinada partição. Do ponto de vista do *reducer*, o par *destino.origem/distância* é equivalente ao par *registro.coluna(j)/valor*, ou seja, o *vértice destino* está no registro contido na partição da matriz que ele acessa, o *vértice origem* é a coluna j da lista de distâncias desse registro e a *distância* é o valor que ocupa a coluna j. Como vimos anteriormente, as distâncias calculadas pelos *mappers* em um determinado *Job* possuem sempre o mesmo valor de distância, ou seja, dentro de mesmo ciclo *map-shuffle-reduce* são encontrados todos os pares cuja distância entre si é k. Assim, dentre os valores de distância recebidos no *reducer* para uma mesma chave do par *origem.destino/distância*, basta selecionar uma distância por chave, pois todos os valores são iguais, mesmo para chaves diferentes. Além disso, a distância calculada e emitida por um *mapper* dentro de um *Job* é a distância mínima, e portanto a função *reduce* não precisa comparar o valor recebido com o valor presente na partição da matriz, afim de verificar se a distância recebida é menor. De fato, se houver algum valor já presente na posição *registro.coluna(j)* da partição, diferente do caracter "d", então este é o mínimo e não precisa ser substituído. Por outro lado, se o valor presente na posição for o caracter "d", este deverá ser substituído pelo novo valor de distância encontrado, significando que a fronteira foi expandida.

A funcão *Reduce* também deverá acumular na memória RAM a maior quantidade possível de pares *destino.origem/distância* antes de abrir o arquivo referente a sua

Algoritmo 5: Classe *Reducer* do cálculo de distâncias.

```
1: classe REDUCER
2:   método SETUP()
3:     Leitor ← novo PonteiroDeArquivo();
4:     Distancias ← novo HASHMAP(chave, valor)
5:   método FLUSH()
6:     Leitor.AbrePartiçãodoGrafo()
7:     for all registro ∈ particao do
8:       existe ← Distancias.Consulta(registro)
9:       if existe == true then
10:        ATUALIZA(registro.coluna (j), valor)
11:        MARCA(registro.coluna (j), recem_atualizada)
12:        INCREMENTACONTADORGLOBAL()
13:      end if
14:    end for
15:    Leitor.FechaPartiçãodoGrafo()
16:  método REDUCE(chave, [d_1, d_2, ...])
17:    if memória_livre == true then
18:      Distâncias.Armazena(chave, d_1)
19:    else
20:      Reducer.Flush()
21:      Distâncias.Armazena(chave, d_1)
22:    end if-else
23:  método CLEANUP()
24:    Reducer.Flush()
25:    Leitor.AbrePartiçãodoGrafo()
26:    for all registro ∈ particao do
27:      EMITE(chave, registro)
28:    end for
29:    Leitor.FechaPartiçãodoGrafo()
30:    Distancias.Apaga()
```

partição. Isto deve ser feito para reduzir o número de vezes que o *reducer* n atualiza o arquivo *part-r-0000n*, onde n é o número da partição a qual o *reducer* é responsável. O Algoritmo 5 mostra a classe *Reducer* responsável por atualizar uma partição da matriz. Na linha (3) do método *Setup()* é alocado um ponteiro de arquivo para abertura e fechamento do arquivo *part-r-0000n*. Na linha (4) é alocado uma estrutura *Hashmap* do Java para o armazenamento dos pares chave/valor emitidos pelos *mappers* e recebidos no *reducer* por meio da função particionadora (Algoritmo 4).

O método *Reduce()* na linha (16) é chamado para cada par chave/valores, com *valores* sendo uma lista de todas distâncias recebidas para um mesma chave *des-*

tino.origem. Nas linhas (17) a (19), a função *Reduce* seleciona apenas um valor por chave, sendo o mesmo armazenado no *Hashmap* se existir espaço suficiente na memória do *reducer*. Caso a memória não seja suficiente, as linhas (19) a (22) atualizam a partição com as distâncias recebidas até o momento, por meio do método *Flush()*. Este método, descrito nas linhas (6) a (14), consulta se cada registro da partição possui alguma atualização presente no *Hashmap*. O arquivo de saída *part-r-0000n* no Hadoop é sequencial, pemitindo que sejam adicionados registros apenas ao final do arquivo. Por esse motivo que cada registro da partição deve ser lido na linha (7), mesmo que não haja necessidade de sua atualização. A medida que os registros são lidos, uma cópia atualizada do arquivo *part-r-0000n* é criada, de forma que a mesma subtituirá o arquivo original após o fim da leitura dos registros. Na linha (8) é consultado se o registro lido na partição necessita ser atualizado e caso seja necessário, nas linhas (9) a (13) são realizadas operações de atualização do registro. A linha (11) tem a função de marcar a distância atualizada como uma nova referência de distância na fronteira, sendo que esta informação será usada por *mappers* do próximo *Job* (linha 19 do Algoritmo 3). Na linha (12) um contador global é incrementado, de forma que todo *reducer* instanciado registre a ocorrência de uma atualização. Esse registro é responsável por atualizar a condição de parada na linha (12) do Algoritmo 1, isto é, a ocorrência de uma atualização é sinal de que uma nova fronteira foi encontrada, e que há possibilidade de expansão no cálculo do diâmetro do grafo. Finalmente, o método *Cleanup()* é chamado na linha (23), sempre depois que todos os pares *destino.origem/distância* tiverem sido processados pelo método *Reduce()*. Na linha (24) são atualizados os últimos registros que remanesceram na memória do *reducer*. Na linhas (25) a (28), a partição que está completamente atualizada é gravada no HDFS.

3.3.3 3ª Fase: Cálculo do Diâmetro e Raio

A terceira fase do algoritmo AHEAD consiste em calcular em cada registro de uma partição a excentricidade do vértice naquele registro. Para tanto, a lista de distâncias do registro deve ser percorrida para determinar qual é a maior distância

Algoritmo 6: Classes *Mapper*, *Partitioner* e *Reducer* para centralidades.

```
1: classe Mapper
2:   método Setup()
3:     excentricidade_min ← MAX_INT
4:     excentricidade_max ← MIN_INT
5:   método Map(chave, registro)
6:     excentricidade ← MIN_INT
7:     vetor_de_distancias ← Seleciona(lista_de_distancias, registro)
8:     for i=0 to Tamanho(vetor_de_distâncias)
9:       if vetor_de_distâncias[i] != d and vetor_de_distâncias != 0 then
10:        if vetor_de_distâncias[i] > excentricidade then
11:          excentricidade ← vetor_de_distâncias[i]
12:        end if
13:      end if
14:    end for
15:    if excentricidade > excentricidade_max then
16:      excentricidade_max ← excentricidade
17:    end if
18:    if excentricidade < excentricidade_min then
19:      excentricidade_min ← excentricidade
20:    end if
21:  método Cleanup()
22:    Emite(D, excentricidade_max)
23:    Emite(R, excentricidade_min)
```

```
1: classe HashPartitionerCalculaRaioDiametro
2:   método SelecionaParticao(chave, valor)
3:     if chave == R then
4:       numero_da_particao ← 0
5:     else
6:       numero_da_particao ← 1
7:     end if-else
8:     return numero_da_particao
```

```
1: classe Reducer
2:   método Reduce(chave, [v1, v2, ...])
3:     numero_do_reducer ← ConfiguracaoDoSistema()
4:     menor ← MAX_INT
5:     maior ← MIN_INT
6:     if numero_do_reducer == 0 then
7:       for all vi ∈ [v1, v2, ...] do
8:         if vi < menor then
9:           menor ← vi
10:      end for
11:      Emite(Raio:, menor)
12:    else
13:      for all vi ∈ [v1, v2, ...] do
14:        if vi > maior then
15:          maior ← vi
16:      end for
17:      Emite(Diâmetro:, maior)
18:    end if-else
```

encontrada para algum outro vértice do grafo. As maiores e menores excentricidades das partições devem ser comparadas, de forma a encontrar a maior e a menor excentricidade global. Portanto, um *Job* que realiza essa operação precisa de no máximo dois *reducers*. O Algoritmo 6 mostra as classes *Mapper*, *HashPartitionerCalculaRaioDiametro* e *Reducer* para o cálculo de diâmetro e raio do grafo. Na classe *Mapper*, nas linhas (3) e (4) do método *Setup()*, são criadas as variáveis que armazenarão a menor e a maior excentricidade local. Essas variáveis são iniciadas com as constantes MAX_INT e MIN_INT, que são respectivamente os valores máximo e mínimo de números inteiros na linguagem Java. O método *Map()* na linha (5) recebe como parâmetro o par chave/valor correspondente a um registro da partição que ele processa. A chave é criada automaticamente[3] pelo Hadoop, e o valor é o próprio registro da partição. Esse registro contém um vértice e as respectivas lista de vizinhos e de distâncias, como descrito na Seção 3.3.1. Na linha (7) a lista de distâncias para os outros vértices do grafo é repassada para um vetor de distância, que será percorrido com o propósito de encontrar a maior distância do vértice desse registro para algum outro vértice do grafo.

Portanto, o valor encontrado na linha (11) da classe *Mapper* é a excentricidade do vértice no registro processado, e não poderá ser o valor 0, que significa um vértice isolado (a única distância encontrada é a distância para ele mesmo) nem o valor "d", que significa distância não encontrada. Nas linhas (15) a (20) do método *Map()* são identificadas a maior e menor excentricidade locais, e portanto são candidatos a raio e diâmetro do grafo. Nas linhas (22) e (23), o *mapper* emite esses possíveis valores de raio e diâmetro com as chaves "R" para identificar um valor como candidato a raio e "D" para identificar um valor como candidato a diâmetro.

A classe *HashPartitionerCalculaRaioDiametro* deve apenas enviar os pares chave/valor emitidos pelo *mapper* para duas possíveis partições: a que seleciona a maior excentricidade global (chave = D) e a que seleciona a menor excentricidade global (chave = R). Essa distribuição ocorre nas linhas (3) a (8) da classe. Finalmente, os *reducers* instanciados da classe *Reducer* recebe todos os pares emitidos pelos *mappers*. Na linha (3) da classe Reducer, é definido se o *reducer* instanciado

[3]Corresponde ao *offset* em bytes da posição do registro em relação ao início do arquivo.

receberá os pares referentes ao valores candidatos a raio ou candidatos a diâmetro do grafo. Caso seja um *reducer* de raios locais, as linhas (7) a (11) são executadas, retornando o raio do grafo. Caso contrário, as linhas (13) a (17) são executadas, retornando o diâmetro do grafo.

3.3.4 Funções *Combiner*

A plataforma Hadoop oferece ao usuário a opção de especificar o uso de uma função *combiner*, que é aplicada nos pares chave/valor que foram escritos nos discos locais, ou seja, que foram emitidos por *mappers* e distribuídos em partições pela função particionadora. A saída de uma função *combiner*, portanto, é a entrada da função *reduce*. A principal vantagem de usar essa função é a possibilidade de realizar o pré-processamento dos dados que serão enviados aos *reducers* [White 2012]. Contudo, a aplicação da função *combiner* depende unicamente da plataforma Hadoop, não sendo possível prever com exatidão quando ela será executada [White 2012, Lin e Schatz 2010]. Além disso, a função só pode ser aplicada depois que os pares chave/valor já foram escritos no disco, sendo ainda necessário operações adicionais de leitura dos pares. Portanto, há claras vantagens em executá-la quando os pares chave/valor estão na memória RAM. Esses são os motivos pelos quais se justifica a criação da técnica *In-Mapper Combiner* pelos autores [Lin e Schatz 2010], em que a função *combiner* é executada na memória dos *mappers*, antes da serialização dos pares chave/valor no disco.

Na Seção 3.3.2, introduzimos a técnica *In-Mapper Buffer Combiner*, que descarrega os pares chave/valor quando não é mais possível utilizar a técnica *In-Mapper Combiner*, devido as limitações de memória de um *mapper* e a quantidade de pares criados por ele. Como vimos naquela Seção, o objetivo do *In-Mapper Buffer Combiner* no cálculo de distâncias é evitar que pares chave/valor repetidos sejam armazenados em disco e enviados via rede na fase *shuffle*. Portanto, quando os *mappers* descarregam no disco os pares chave/valor por falta de espaço livre na memória RAM, poderá ocorrer repetição de pares entre os conjuntos descarregados.

Para abordar esse problema, utilizamos uma função *combiner*, cujo objetivo é eliminar as repetições de pares chave/valor escritos no disco. Quanto maior é a quan-

Algoritmo 7: Classe *Combiner* do cálculo de distâncias.

```
1: classe COMBINER estende REDUCER
2:   método REDUCE(chave, [d_1, d_2, ...])
3:     valor ← (d_1)
4:     EMITE(chave, valor)
```

tidade de memória RAM disponível, menor será a quantidade de pares chave/valor repetidos no disco, mas aqueles que ainda forem duplicados devido as limitações da memória, serão eliminados pela função *combiner*, quando a mesma for utilizada pela plataforma Hadoop. A classe *Combiner* herda o método Reduce da classe *Reducer*, como mostrado no algoritmo 7. A linha (3) seleciona um apenas valor de distância, pois todos os valores de distâncias são iguais dentro do mesmo *Job*. Portanto qualquer outro valor para a mesma chave certamente será uma repetição. Na linha (4) o par não repetido é emitido e fica disponível para a transmissão aos *reducers*. É interessante notar que o método *Reduce()* na linha (2) é idêntico ao utilizado pelos *reducers* da segunda fase no algoritmo AHEAD, durante o cálculo das distâncias (linhas 16 a 19, Algoritmo 5).

3.3.5 Número de *Mappers* e *Reducers*

A quantidade de *mappers* instanciados pela plataforma Hadoop depende do tamanho da entrada de dados, onde o número de *mappers* M é o resultado da divisão:

$$M = \frac{T_e}{B_{\text{hdfs}}}, \tag{3.1}$$

sendo T_e a quantidade de bytes na entrada da fase *Map* e B_{hdfs} um parâmetro configurável do Hadoop que especifica o tamanho em bytes do bloco de dados usado pelo HDFS. Um arquivo armazenado no HDFS é constituído por um ou mais blocos dependendo do valor de B_{hdfs} e, de acordo com a equação (3.1), a plataforma Hadoop aloca por *default* um *mapper* para cada bloco de dados. O número R de *reducers* instanciados pelo Hadoop é especificado via parâmetro no AHEAD, sendo que o valor usado nesse trabalho é sempre igual a quantidade de núcleos disponíveis no sistema. Assim, considerando c o número de núcleos no *cluster* e $k = \frac{M}{R}$ a relação entre o número de *mappers* e *reducers* desejada, ao dividirmos os dois lados da equação (3.1)

por R obtemos:

$$B_{\text{hdfs}} = \frac{T_e}{k \cdot c} \tag{3.2}$$

Se $k = 1$ e T_e é conhecido, o valor calculado de B_{hdfs} é tal que o número de *mappers* ou *reducers* alocados pela plataforma Hadoop iguala ao número de *cores* disponíveis no sistema. É importante ressaltar que o número de blocos de dados no HDFS, criados automaticamente pela plataforma, não é necessariamente igual ao número de partições processadas pelos *reducers*. O número de partições é sempre igual ao números de *reducers* especificados pelo usuário, com a saída de uma partição processada por um *reducer* sendo escrita em um arquivo no HDFS. Assim uma partição pode ser constituída por um ou mais blocos, dependendo de seu tamanho e do valor de B_{hdfs}. A implementação modular do AHEAD permite que a sua primeira fase seja executada isoladamente, o que viabiliza a verificação prévia do valor de T_e. Assim é possível estimar o ajuste de B_{hdfs} de forma que as relações núcleos/*mappers* e núcleos/*reducers* sejam 1/1.

3.4 Considerações finais

Neste capítulo fizemos a descrição completa do algoritmo AHEAD. Na Seção 3.1 foi feito um histórico com os trabalhos que influenciaram no desenvolvimento do algoritmo AHEAD. Na Seção 3.2, há uma visão geral do algoritmo, relacionando conceitos do modelo MapReduce com a dinâmica necessária para o processamento paralelo de grafos. Na Seção 3.3 são descritas em detalhes todas as fases executadas pelo algoritmo AHEAD durante o processamento paralelo de um grafo grande. Finalmente, nas Seções 3.3.4 e 3.3.5 são feitas análises sobre algumas particularidades da plataforma Hadoop que afetam no funcionamento do algoritmo AHEAD.

Capítulo 4

Projeto de Experimentos

Neste Capítulo apresentamos os métodos e os meios utilizados para a análise do desempenho do algoritmo AHEAD mediante o processamento de grafos.

4.1 Ambiente computacional

Os experimentos foram realizados em um *cluster* com seis nós de processamento interligados por meio de um *switch* com velocidade de conexão de 1 Gb/s, onde cada nó possui dois processadores Intel Xeon X5660 com seis núcleos físicos a 2,80 GHz, 98 GB de RAM, 12 MB de Cache e 300 GB de disco. Os discos possuem 15K RPM de taxa de rotação e 6 GB/s de taxa de transferência de dados. Ao todo são 72 núcleos físicos ou 144 núcleos virtuais (*hyperthreading*) presentes no *cluster*. Para o nó de gerenciamento *Namenode* foi utlizada uma máquina com um processador Intel Xeon E5440 de 4 núcleos físicos a 2,83GHz, 16 GB de RAM, 12 MB de Cache e 530 GB de disco. O nó de gerenciamento *Secondary Namenode* é uma máquina com processador Intel Xeon X5310 de 4 núcleos físicos ou 8 núcleos virtuais a 1,60GHz, 16 GB de RAM, 12 MB de Cache e 60 GB de espaço em disco. A distribuição Hadoop instalada é fornecida pela fundação Apache[1], versão 1.1.2, e o sistema operacional é o Linux Gentoo Kernel 3.6.11. A versão da máquina virtual Java instalada nas máquinas é *Java(TM) SE Runtime Environment (build 1.6.0_ 39-b04)*. Os computadores utilizados fazem parte da infraestrutura do Laboratório de Computação Científica da Universidade Federal de Minas Gerais (LCC-UFMG [2]).

[1] `http://hadoop.apache.org`

[2] http://www.lcc.ufmg.br

4.2 Conjunto de Dados

O algoritmo AHEAD foi avaliado mediante o processamento de grafos reais e sintéticos. Considerando a contribuição que pretende-se obter com o algoritmo AHEAD, que é avançar nas escalas de tamanho de grafos em que é possível computar métricas exatas de centralidade, a análise do desempenho deste algoritmo em redes complexas que modelam diversos sistemas reais se faz importante. Grafos sintéticos são usados para experimentos que variam o tamanho da entrada de dados, com o objetivo de verificar o comportamento do algoritmo nessas condições. Nas seções a seguir os dois tipos de conjuntos são descritos com mais detalhes.

4.2.1 Grafos Reais

Os experimentos realizados com os grafos reais são para coleta das métricas discutidas na Seção 2.5, a saber, escalabilidade, *speedup* e eficiência. São três os grafos reais em questão, que modelam o sítio da rede social *Epinions* com 75.879 vértices e 508.837 arestas, a rede de comunicação por correio eletrônico entre funcionários da empresa *Enron* com 36.692 vértices e 367.662 arestas, e a rede de colaboração científica entre co-autores que publicaram artigos na categoria *Astro Physics* da revista eletrônica *arXiv* com 18.772 vértices e 396.160 arestas. O grafos reais mencionados estão disponíveis na base de dados *Stanford Large Network Dataset Collection*[3] mantida pelo grupo de pesquisa *Stanford Network Analysis Project*.

4.2.2 Grafos Sintéticos

Esses grafos possuem quantidades variáveis de vértices e arestas, e para tanto foram gerados usando a biblioteca *NetworkX* [Hagberg, Schult e Swart 2008], desenvolvida para a linguagem orientada à objetos *Python*. Essa biblioteca disponibiliza a função *gnm_ random_ graph(n,m)*, que gera aleatoriamente um *objeto* do tipo grafo, não direcionado, sem arestas repetidas, pertencente ao conjunto de todos os possíveis grafos $G(V, A)$ com n vértices e m arestas. Um programa escrito na linguagem *Python* que usa essa função foi elaborado para criar grafos direcionados e não direcionados, cujos vértices são nomeados com números naturais entre 0 e $(n-1)$. Para

[3] `http://snap.stanford.edu/data/index.html`

gerar um grafo não direcionado é assegurado que, dado um par de vértices $u, v \in V$, se u pertence à lista de vizinhos de v, então v pertence a lista de vizinhos de u. Portanto entre dois vértices vizinhos do grafo não direcionado há duas arestas direcionais em sentidos opostos. Para garantir que o grafo seja conectado, o mesmo é modificado de tal forma que todo vértice i presente no conjunto $V = \{0, 1, \ldots, n-1\}$ possui como vizinho o vértice $i + 1$. Para o caso do vértice $n - 1$, é adicionado à sua vizinhança um vértice escolhido aleatoriamente do conjunto $V - \{n - 1\}$. Além disso, é verificado na lista de vizinhos de cada vértice i se há a presença do próprio vértice i. Se assim for, o vértice i é retirado da lista para a eliminação da ocorrência de *self-loops*.

Foram criados grafos conectados não direcionados com o número de vértices fixo em 50.000 e número de arestas variando entre 200.000 e 1.600.000. Para verificar o comportamento do algoritmo com a variação de vértices e arestas, foram criados dois conjuntos de grafos, um com grafos direcionados e outro com grafos não direcionados. Ambos os conjuntos possuem grafos com o número de vértices variando entre 25.000 e 200.000, e o número de arestas variando entre 75.000 e 600.000. O estudo de grafos direcionados é importante uma vez que são usados para modelar situações específicas como, por exemplo, ligações entre endereços da *Web* (*hyperlinks*).

4.3 Comparação com algoritmos sequenciais

Foram usados dois programas sequenciais para o cálculo de diâmetro e raio, com o objetivo de fazer o teste de correção no algoritmo AHEAD. Além disso, são comparados o tempo de processamento de um grafo com 50.000 vértices e 200.000 arestas. O algoritmo sequencial desenvolvido no trabalho em [Gonçalves, Maciel e Murta 2010] é usado para a correção do cálculo de diâmetro. Foi desenvolvido um programa na linguagem *Python*, usando a biblioteca *NetworkX*, para o teste de corretude de diâmetro e de raio. A biblioteca *NetworkX* implementa duas funções para o cálculo do raio e diâmetro do grafo, sendo a implementação de ambas feitas para o trabalho realizado em [Hagberg, Schult e Swart 2008].

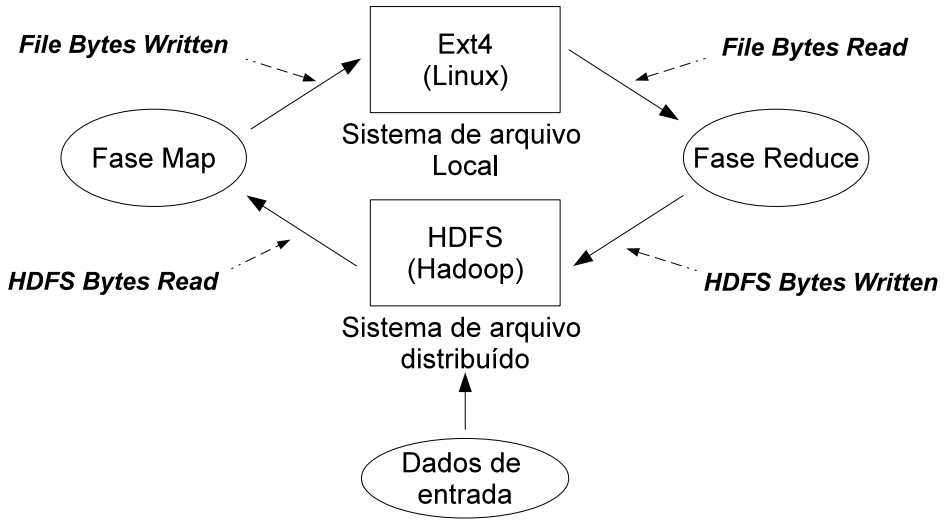


Figura 4.1: Representação da execução de *Jobs* no Hadoop.

4.4 Comparação com o algoritmo HEDA

A comparação entre os algoritmos AHEAD e HEDA tem o objetivo de verificar o uso dos recursos computacionais do *cluster* durante a execução dos algoritmos. Em particular, o uso da memória RAM e do espaço em disco tem especial importância, pois esses constituem fatores decisivos nas possibilidades de processamento de grafos em escalas maiores. A Figura 4.1 representa o ciclo do uso de espaço em disco durante a execução de um *job* qualquer na plataforma Hadoop. Um ciclo *map-shuffle-reduce* equivale a um ciclo no sentido horário da Figura 4.1, começando pelos dados de entrada. As principais métricas fornecidas pela plataforma Hadoop envolvidas no processo estão em itálico na figura:

- *File Bytes Written*: Escrita de dados no sistema de arquivo local. Ocorre na saída da fase *map*;
- *File Bytes Read*: Leitura de dados do sistema de arquivos local. Ocorre na entrada da fase *reduce*. Também conhecido como fase *shuffle*;
- *HDFS Bytes Written*: Escrita de dados no sistema de arquivos distribuído. Ocorre na saída da fase *reduce*;

- *HDFS Bytes Read*: Leitura de dados do sistema de arquivos distribuído. Ocorre na entrada da fase *map*.

Tanto o AHEAD quanto o HEDA executam vários ciclos *map-shuffle-reduce* durante o processamento de um grafo. O uso eficiente da memória RAM e do espaço em disco durante os *Jobs* são observados na comparação. Em especial, estaremos interessados no tamanho da saída na fase *map*, pois a mesma está diretamente relacionada à conectividade e o número de vértices do grafo processado. Quanto maior for essas duas características, maior é a quantidade de pares chaves/valor emitidos na memória e posteriormente armazenados em disco.

4.5 Considerações finais

Nesse capítulo apresentamos os métodos e meios que são usados para avaliar o desempenho do algoritmo AHEAD. Na Seção 4.1 foram descritas as características do *cluster* onde os experimentos são realizados. O conjunto de grafo usado para a análise do comportamento do algoritmo AHEAD quanto a escalabilidade, eficiência e *speedup* está descrito na Seção 4.2. Planejamento de experimentos relacionados às comparações com algoritmos sequenciais e paralelos foram descritos nas Seções 4.3 e 4.4 respectivamente.

Capítulo 5

Resultados dos Experimentos

Nesse capítulo apresentamos os resultados obtidos por meio dos experimentos propostos no Capítulo 4. Todos os dados coletados foram anexados no Apêncice A. Na Seção 5.2 mostramos os resultados de correção do algoritmo. Na Seção 5.4, grafos reais são usados para testes de *speedup* e eficiência, enquanto na seção 5.5 grafos sintéticos são usados para a análise do comportamento do algoritmo com grafos de tamanhos variados. O capítulo é encerrado com a comparação dos resultados entre os algoritmos HEDA e AHEAD, na Seção 5.7.

5.1 Medidas de tempo

As medidas apresentadas para cada experimento com os grafos são a média dos resultados de três processamentos. Mais especificamente, o tempo médio de execução é o resultado de três medidas de tempo retornadas pelo programa *time* do Linux. O valor denominado *real*, retornado por esse programa, é o tempo passado desde a chamada até a conclusão da aplicação submetida via *shell* do Linux. O sistema operacional Linux Gentoo permite instalação customizada, sendo portanto possível instalar as configurações mínimas para o seu funcionamento. Dessa maneira, não foi instalado com o sistema operacional as interfaces gráficas ou qualquer programa que necessite delas. A intenção foi minimizar a interferência de processos que são executados em *background* no sistema operacional, que podem interferir na medida de tempo. A submissão do programa AHEAD é sempre feita no nó computacional que executa o processo *namenode*, responsável pelo gerenciamento do *Job* submetido ao *cluster*.

5.2 Correção do AHEAD e comparação com algoritmos sequenciais

Ao processar um grafo sintético com 50.000 vértices e 200.000 arestas, os algoritmos AHEAD e sequenciais apresentaram os mesmos resultados, com $Diâmetro = 15$ e $Raio = 11$. Ao usar o algoritmo HEDA no teste de correção, os valores de raio e diâmetro para o grafo sintético, que é conectado, também foram iguais. Contudo, para o caso de grafos com componentes desconectados, os valores de raio retornados pelo algoritmo HEDA não foram iguais aos retornados pelo algoritmo AHEAD. Isto ocorre quando há um vértice isolado no grafo processado, situação em que o algoritmo HEDA calcula a excentricidade deste vértice igual a zero. Como este é o menor valor possível de distância, o raio retornado pelo HEDA também é zero. No algoritmo AHEAD será retornado o valor de raio do menor componente conectado, o que desconsidera vértices isolados no cálculo das excentricidades. Portanto o valor de raio retornado pelo AHEAD será sempre maior ou igual a um.

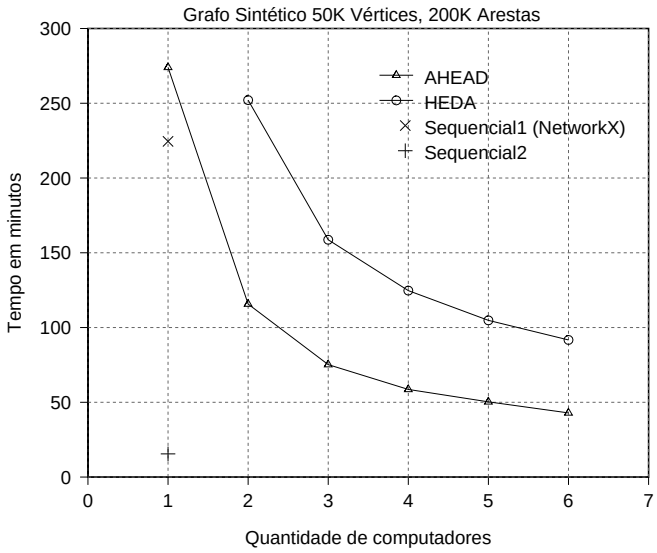


Figura 5.1: Correção do algoritmo AHEAD e comparação com os algortimos sequências e HEDA.

A Figura 5.1 mostra os tempos de execução dos algoritmos sequenciais com uma máquina e dos algoritmos HEDA e AHEAD, quando varia-se o número de máquinas.

O algoritmo *sequencial1* usa a biblioteca *NetworkX* [Hagberg, Schult e Swart 2008] para cálculo de raio e diâmetro enquanto o *sequencial2* calcula o diâmetro de acordo com o trabalho realizado em [Gonçalves, Maciel e Murta 2010]. Os resultados exibidos na Figura 5.1 indicam que o modelo MapReduce/Hadoop apresenta um *overhead* significativo, e seu uso só se justifica para quantidades de dados que não podem ser executadas em uma única máquina. Na Figura 5.1 também observamos que só foi possível processar o grafo de 50K vértices e 200K arestas por meio do algoritmo HEDA usando o mínimo de duas máquinas, pois com uma máquina o sistema acusa falta de espaço em disco. Vale rever a Figura 2.5, que representa essa ideia de que algoritmos diferentes devem ser escolhidos de acordo com o tamanho do grafo e os recursos computacionais disponíveis.

5.3 Resultados do AHEAD em grafos reais

Esta seção discute os resultados obtidos com o algoritmo AHEAD ao processar o conjunto de grafos reais, constituído pelos grafos descritos na Seção 4.2.1. O ambiente computacional nesse teste usa a capacidade máxima do cluster, com seis nós. O resumo dos resultados obtidos é apresentado na Tabela 5.1. Para avaliar o programa gerador de grafos descrito na Seção 4.2.2, foram acrescentados na tabela os resultados do processamento de um grafo sintético. Assim é possível comparar os grafos sintéticos gerados em relação aos grafos reais, por meio do comportamento do algoritmo AHEAD ao processar os dois tipos de grafos.

Tabela 5.1: Grafos reais processados pelo AHEAD

Nome	Tipo	Número de vértices	Número de arestas	Raio	Diâmetro	Tempo médio em minutos	Desvio Padrão
Co-autores	Não direcionado	18.772	396.160	1	14	24,5	0,4
Enron	Não direcionado	36.692	367.662	1	13	39,8	0,9
Sintético	Não direcionado	50.000	200.000	11	15	42,9	0,8
Epinions	Direcionado	75.879	508.837	1	16	103,1	1,6

Nos resultados, é notável o valor de raio igual a um, que pode significar duas coisas: há presença de componentes desconectados no grafo ou há um vértice que tem todos os outros vértices como vizinhos. No segundo caso, se o grafo fosse conectado, o diâmetro seria no máximo dois. Portanto trata-se de um grafo com componente

desconectado. Nesse caso, o valor do diâmetro é do maior componente conectado. No algoritmo AHEAD, um *Job* completo durante a fase de cálculo de distâncias aumenta em 1 o valor conhecido sobre a maior distância entre dois vértices no grafo. Portanto existe uma relação direta entre o diâmetro do grafo e o número de *Jobs* executados pelo algoritmo. Contudo, o diâmetro do grafo não é o único fator que afeta o custo do processamento, conforme visto na Tabela 5.1. O número de vértices foi o fator que apresentou relação direta com o tempo de execução, não ocorrendo o mesmo com o número de arestas ou com o diâmetro do grafo.

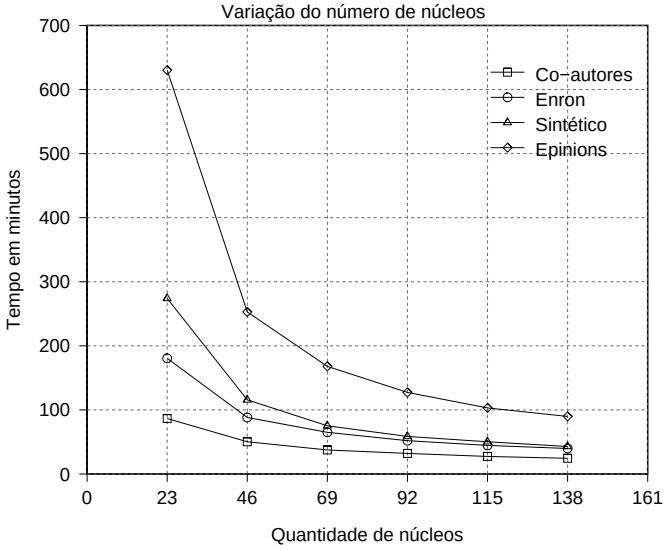
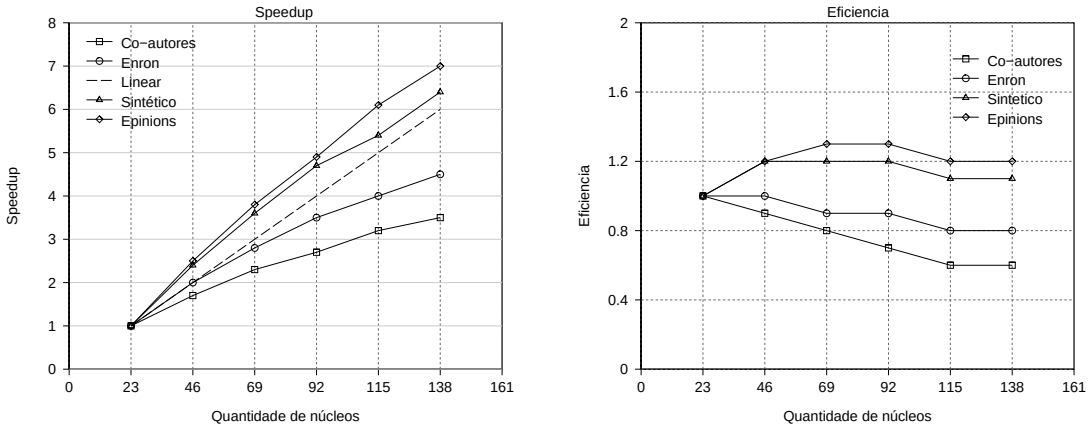


Figura 5.2: Variação do número de núcleos.

O gráfico da Figura 5.2 apresenta a variação do tempo médio de processamento dos grafos quando varia-se o número de núcleos presentes no cluster. No gráfico da Figura 5.2, a melhoria do tempo de execução com a adição de núcleos é mais evidente no processamento de grafos com maior quantidade de vértices e arestas. Para o grafo *Epinions*, a carga de processamento que apenas uma máquina experimenta (23 núcleos) é melhor distribuída em seis máquinas (138 núcleos) do que quando o mesmo ocorre no menor grafo, Co-autores. Esse resultado evidencia que o algoritmo AHEAD apresenta bons resultados para o seu principal propósito, que é atuar no processamento de grafos em escalas maiores.

5.4 Speedup e Eficiência

A Figura 5.3 mostra os resultados do *speedup* e da eficiência para o mesmo conjunto de dados da Tabela 5.1. Nota-se no gráfico da Figura 5.3(a) que, para grafos maiores, o AHEAD apresenta *Speedup* superlinear. Isso pode ser explicado pelo fato de que ao acrescentar máquinas no *cluster*, a quantidade de dados que um nó computacional deve processar é menor, permitindo que os dados em processamento possam ser melhor ajustados na memória *cache* do processador. Assim, as operações de memória serão executadas em cada nó computacional de maneira mais eficiente. Esse efeito também é percebido no gráfico da Figura 5.3(b), onde há registro de valor de eficiência maior que 1. Para grafos menores, entendemos que tal efeito é sobreposto pelo *overhead* que o sistema apresenta devido a vários fatores relacionados ao processamento distribuído dos *mappers* e *reducers*, como o instanciamento das máquinas virtuais, o gerenciamento de falhas e a comunicação de dados via rede *ethernet* na fase *shuffle*. Novamente, os resultados apresentados indicam que o AHEAD tem desempenho melhor em grafos de escalas maiores.



(a) Speedup.

(b) Eficiência.

Figura 5.3: Teste de desempenho.

5.5 Avaliação empírica do comportamento do algoritmo

Nessa seção apresentamos os resultados obtidos com os grafos sintéticos, que permitem a realização de experimentos em que há variação do número de vértices e

arestas do grafo. O ambiente de execução é constituído por seis nós computacionais.

5.5.1 Variação de arestas

A Tabela 5.2 apresenta o resultado do processamento de grafos sintéticos não direcionados quando altera-se o número de arestas e mantém-se o número de vértices em 50000. O aumento no número de arestas nesse caso representa o aumento no número de relações entre uma quantidade fixa de entidades. A consequência imediata é a existência de novos caminhos entre os vértices, o que tem o efeito de diminuição nos valores de diâmetro e raio do grafo.

Tabela 5.2: Variação do número de arestas em grafos sintéticos não direcionados com 50K vértices.

Arestas	Raio	Diâmetro	Tempo médio (minutos)	Desvio Padrão	Variação de Arestas	Variação tempo
200.000	11	15	46,6	0,4	-	-
400.000	7	9	56,1	0,0	2x	1,2x
800.000	5	6	96,7	1,9	4x	2,1x
1.600.000	4	5	233,1	1,0	8x	5x

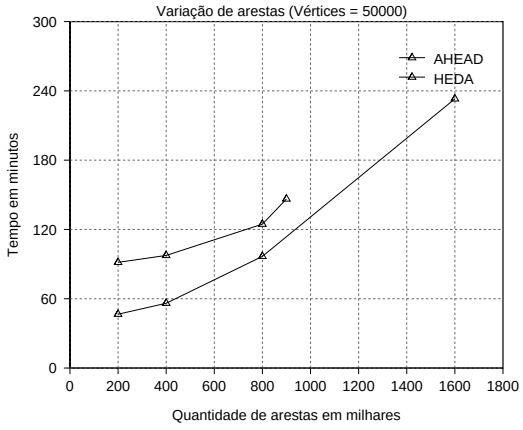
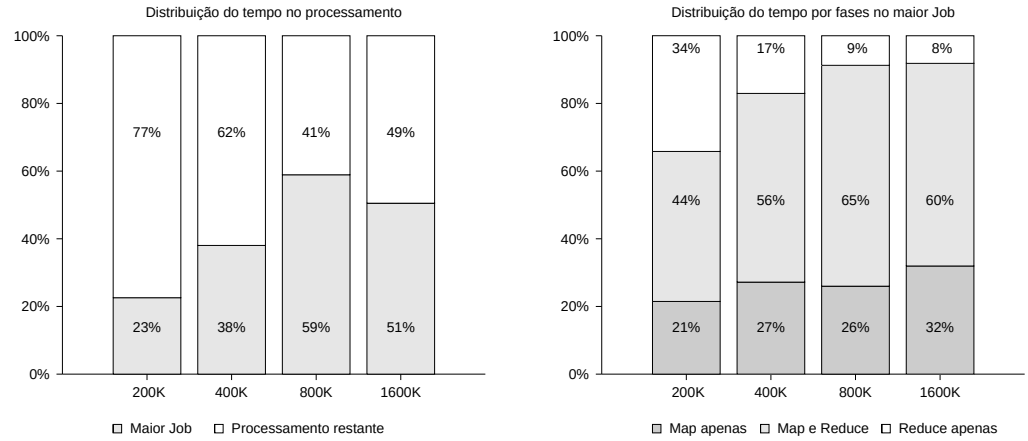


Figura 5.4: Tempo de execução do algoritmo AHEAD, para grafos sintéticos com 50000 vértices.

No caso do comportamento do algoritmo, a Figura 5.4 mostra que a taxa de crescimento do tempo de execução do algoritmo AHEAD é lenta. No intervalo testado, o tempo de execução foi multiplicado por cerca de cinco enquanto o número



(a) Proporção entre tempo do *Job* de maior duração e tempo total de processamento.

(b) Proporção entre os tempos das fases *map* e *reduce* e tempo do *Job* de maior duração.

Figura 5.5: Distribuição dos tempos de execução.

de arestas foi multiplicado por oito. No caso do algoritmo HEDA, não foi possível realizar o experimento na mesma faixa de variação arestas, pois o sistema indica falta de espaço em disco quando o número de arestas é maior que 900K.

Os gráficos da Figura 5.5 mostram a distribuição de tempo despendido no processamento dos grafos da Tabela 5.2 sob duas perspectivas. Esses gráficos foram construídos a partir de métricas fornecidas pelo processo *jobtracker* e acessíveis via protocolo HTTP na porta 50030 da máquina *namenode*. Foram coletadas as métricas que indicam a data, hora, minutos e segundos de início e fim das fases *map* e *reduce* de cada *Job* executado durante o processamento de distâncias do grafo. No gráfico da Figura 5.5(a), é exibida a proporção de tempo do *Job* de maior duração em relação ao tempo total de processamento do grafo, este último medido pelo comando *time* do Linux. A Figura 5.5(b) mostra a proporção de tempo gasto apenas com a fase *map*, apenas com a fase *reduce* e com a concorrência entre ambas para o *Job* de maior duração (mais longo).

O número de *Jobs* que são executados durante processamento de um grafo pode ser grande o suficiente para que a soma dos custos de cada *Job* seja significativamente maior que os custos envolvidos apenas no *Job* mais longo. Isso pode ser observado na Figura 5.5(a), para o grafo de 200K arestas. Este grafo possui o maior diâmetro

dentre os grafos processados no teste, portanto foi o que executou o maior número de *Jobs* durante o processamento. O processamento do mesmo grafo também é o que apresentou a menor proporção de tempo do *Job* mais longo, confirmando a afirmação feita acima.

Como podemos notar no gráfico da Figura 5.5(b), a principal mudança que ocorre na distribuição do tempo, ao aumentar o número de arestas presentes no grafo, está no aumento da proporção de tempo em que ocorre concorrência de tarefas *map* e tarefas *reduce* no *cluster*. Tal concorrência é indicada pela faixa de cor cinza intermediária. Também é possível verificar que, a partir de quatrocentos mil arestas, a fase *map* domina o tempo de processamento gasto no *Job* com maior tempo de duração. Durante o cálculo de distâncias, os *mappers* realizam operações de busca em largura no grafo, o que de fato deve ser afetado com aumento do número de arestas a serem percorridas. As técnicas de projeto *In-Mapper Combiner* e *Schimmy* atuam justamente nessa fase, a primeira otimizando o uso do espaço em disco por meio do uso eficiente da memória e a segunda evitando que os *mappers* despendam processamento para transmitir a estrutura do grafo durante a busca em largura.

5.5.2 Variação de vértices e arestas

A Tabela 5.3 mostra os dados referentes ao processamento de grafos sintéticos direcionados quando altera-se o número de arestas e vértices. O aumento do número de vértices e arestas nesse caso representa o aumento do número de entidades e do número de relações entre elas.

Tabela 5.3: Variação do número de arestas e vértices em grafos sintéticos direcionados.

Grafo	Número de vértices	Número de arestas	V + A	Raio	Diâmetro	Tempo médio (minutos)	Desvio padrão	Variação Grafo	Variação tempo
1	25.000	75.000	100.000	13	21	19,3	0,1	-	-
2	50.000	150.000	200.000	13	22	44,1	0,5	2x	2,3x
3	100.000	300.000	400.000	15	22	170,5	3,6	4x	8,8x
4	200.000	600.000	800.000	15	24	1212,8	9,4	8x	62,8x

Na prática, a inserção de vértices só tem efeito no cálculo das distâncias se os mesmos tiverem grau maior que um. Os grafos na Tabela 5.3 são conectados, o que

garante a existência de no mínimo uma nova aresta a cada novo vértice. A constatação da pouca variação dos raios e diâmetros nos grafos descritos na Tabela 5.3, mesmo havendo aumento de vértices e arestas, é devido à proporção constante entre o número de arestas e de vértices, a saber, 3 para 1. Verificando a Tabela 5.2, a proporção de arestas para vértices chegou a 32 para 1, quando há ocorrência dos menores valores de raio e diâmetro.

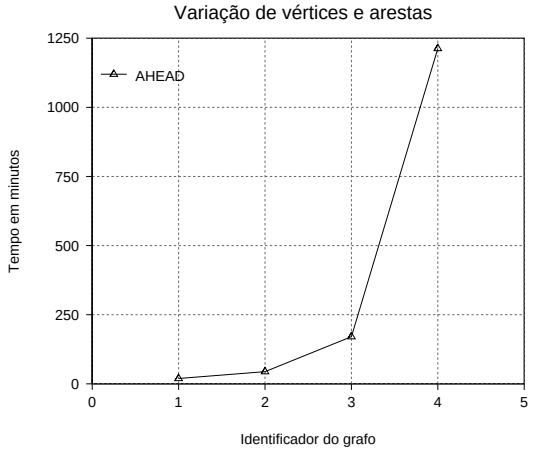


Figura 5.6: Tempo de execução do algoritmo AHEAD para os grafos identificados na Tabela 5.3.

Essa situação afeta o comportamento do algoritmo, pois quando tem-se uma relação arestas/vértice baixa, haverá também menos opções de caminhos entre os vértices. O número reduzido de caminhos faz o algoritmo ter que encontrar vértices mais isolados ou distantes entre si, calculando diâmetros que variam pouco na Tabela 5.3 e que são significativamente maiores em relação aos da Tabela 5.2. Tal fato fica também evidenciado no gráfico da Figura 5.6. Nesse gráfico, a variação do tempo de execução com a variação do tamanho do grafo é maior se comparada com a variação mostrada na Figura 5.4. Ao aumentar o valor (V+A), mantendo a relação arestas/vértices relativamente baixa, estamos aumentando a quantidade de vértices distantes entre si, o que dificulta a exploração do algoritmo sobre o grafo.

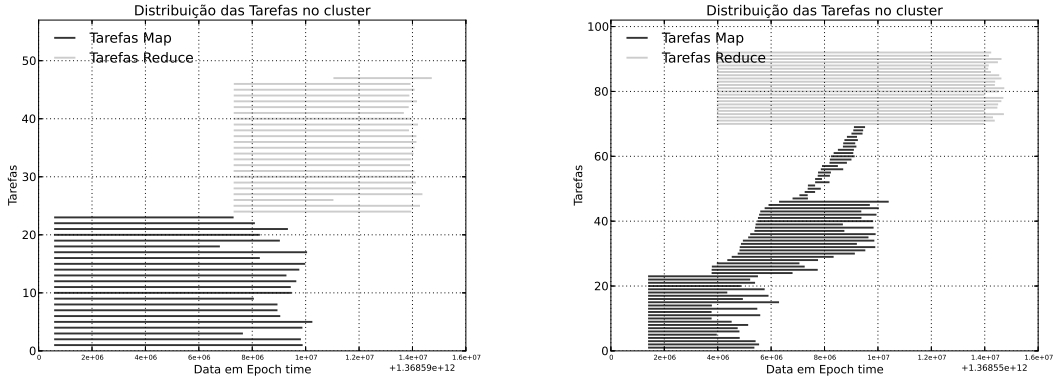
No experimento de variação de vértices e arestas, também foram processados grafos sintéticos não direcionados. O resultados obtidos são fornecidos na Tabela 5.4.

Tabela 5.4: Variação do número de arestas e vértices em grafos sintéticos não direcionados.

Grafo	Número de vértices	Número de arestas	V + A	Raio	Diâmetro	Tempo médio (minutos)	Desvio padrão	Variação Grafo	Variação tempo
1	25.000	75.000	100.000	15	21	19,5	0,2	-	-
2	50.000	150.000	200.000	15	23	47,1	0,8	2x	2,4x
3	100.000	300.000	400.000	16	24	163,3	0,9	4x	8,4x
4	200.000	600.000	800.000	18	26	1129,8	1,7	8x	57,9x

A análise feita para os grafos direcionados é válida também nesse caso. A pequena diferença que aparece nos valores de variação do tempo de execução, se comparada com a obtida para grafos direcionados, deve-se ao fato que arestas bidirecionais propiciam a existência de caminhos alternativos, facilitando a exploração do grafo pelo algoritmo.

5.6 Variação do tamanho dos blocos no HDFS



(a) (k = 1) (b) (k = 2)

Figura 5.7: Variação da relação entre o número de *mappers* e *reducers* no processamento do grafo *Epinions*.

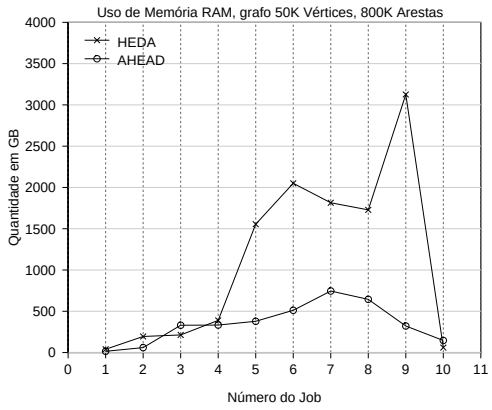
De acordo com a Equação 3.2 da Seção 3.3.5, a relação entre o número de *mappers* e *reducers*, k, pode ser ajustada através da seleção adequada de B_{hdfs}, que é o tamanho do bloco usado no HDFS. A Figura 5.7 mostra a distribuição de *mappers* e *reducers* para $k = 1$ e $k = 2$. O *cluster* foi configurado para usar apenas uma máquina com 23 *cores* disponíveis. Os gráficos são gerados a partir dos dados da sequência

map-shuffle-reduce (*Job*) de maior tempo de duração, durante o processamento do grafo *Epinions*. Uma barra horizontal no gráfico tem o comprimento relativo a duração de uma tarefa, considerando as datas de início e de fim no formato *epoch* do Unix. Na Figura 5.7(a), notamos a predominância do paralelismo entre tarefas *map* e entre tarefas *reduce* de forma isolada, isto é, há pouca concorrência entre *mappers* e *reducers*. O mesmo não ocorre na Figura 5.7(b), onde há a predominância na concorrência entre tarefas de natureza diferente. Notamos também que, para $k = 2$, o instanciamento de tarefas *map* ocorre em número de núcleos disponíveis, nesse caso, em grupos de 23 *mappers*.

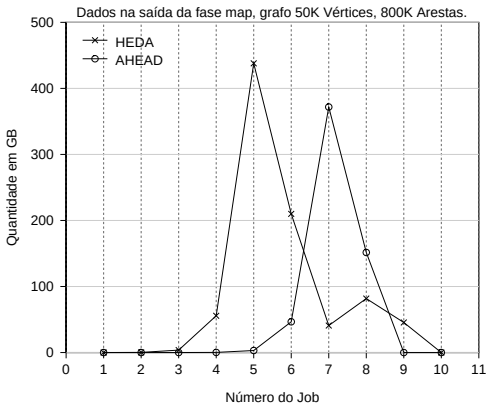
O número total esperado de *mappers* para $k = 2$ é o dobro do número de *reducers*, e este último é sempre configurado como igual ao número de *cores*. Portanto eram esperados 46 *mappers* instanciados durante o *Job* mostrado na Figura 5.7(b). Contudo, no eixo Y, nota-se o instanciamento de um terceiro grupo de *mappers* constitíudo pelas tarefas 47 a 69. A explicação para tal fato é que o valor de T_e foi um pouco acima do estimado, causando o instanciamento de tarefas *map* com curto tempo de processamento. Baseando-se nos experimentos realizados, notamos que os melhores tempos de execução do AHEAD, para diversos grafos, ocorrem quando k encontra-se na faixa de valores definida em $1 \leq k \leq 2$. Mais especificamente, para esse trabalho os ajustes de B_{hdfs} foram feitos de forma a manter o valor de k próximo de 1. Contudo, o estudo dos efeitos da variação de k devem ser feitos de maneira mais aprofundada e em diferentes contextos do uso da plataforma Hadoop.

5.7 Comparação entre HEDA e AHEAD no uso de recursos

Para o teste de comparação foram usados os grafos Co-autores, Enron e um grafo sintético contendo 50.000 vértices e 800.000 arestas. Nesse teste, a capacidade de espaço em disco de cada um dos seis nós computacionais foi ampliada para 2TB. Contudo, cada disco possui 7200 RPM de taxa de rotação e 3 GB/s de taxa de transferência de dados, tornando-os mais lentos que os discos de 300GB usados nos testes anteriores. Também foram feitos testes de correção da versão do HEDA fornecida pelo autor [Nascimento 2011]. Na versão utilizada, o algoritmo HEDA executa até que o número de iterações fornecido como parâmetro de entrada do algoritmo seja



(a) Uso acumulativo da memória RAM no *cluster*.



(b) Uso de espaço do sistema de arquivo agregado no *cluster*.

Figura 5.8: Comparação do uso do sistema de arquivo agregado e da memória RAM pelos algoritmos HEDA e AHEAD durante o processamento do grafo sintético.

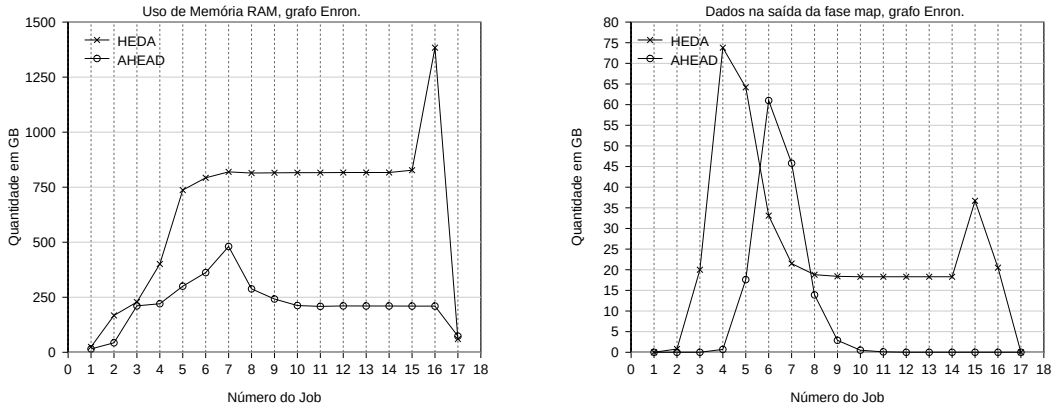
alcançado. Espera-se que o número de iterações fornecido na entrada seja maior do que o diâmetro do grafo. O algoritmo HEDA retorna valor de raio e diâmetro iguais aos retornados pelos algoritmos sequenciais e AHEAD, quando o número de iterações passado por parâmetro é suficiente para o cálculo. O número mínimo de iterações necessárias é justamente o diâmetro do grafo, pois o algoritmo HEDA também faz uso do procedimento de busca em largura para cálculo de distâncias.

O mesmo valor de B_{hdfs} é usado na execução dos dois algoritmos. Na Figura 5.8(a), os valores exibidos de uso da memória são acumulados por *mapper* e *reducer* alocados dentro do *Job*. Essa medida é fornecida pela plataforma Hadoop e significa a soma dos bytes usados por todas as tarefas que foram instanciadas durante o *Job*. Por exemplo, no *Job* de número seis, o AHEAD usou 500 GB de memória, sendo este valor a soma dos bytes usados por cada tarefa que foi instanciada durante o ciclo *map-shuffle-reduce*. Os bytes usados não referem-se apenas àqueles usados para a criação de uma máquina virtual Java (*mapper* ou *reducer*), mas também à memória que a própria maquina virtual usa para realizar o processamento.

Observamos durante os experimentos que o uso de uma quantidade menor de memória pelo algoritmo AHEAD em relação ao algoritmo HEDA foi devido a um

número menor de máquinas virtuais Java instanciadas para realizar o mesmo trabalho. Os algoritmos AHEAD e HEDA utilizam vários ciclos para o processamento de um grafo, e portanto as análises a seguir levam em consideração a soma do uso de recursos de todos os ciclos. No caso da Figura 5.8(a), o algoritmo AHEAD usa cerca de um terço da quantidade de memória que o algoritmo HEDA usa, quando somamos os valores usados em todos os *Jobs*.

A Figura 5.8(b), mostra o uso do espaço agregado de disco no *cluster* durante o processamento do grafo. Os valores medidos são referentes a quantidade de dados na saída da fase *map*, que fica armazenada nos discos locais antes da transferência (*shuffle*) para os *reducers*. A quantidade de espaço em disco usado pelo algoritmo HEDA é 52,8% maior que o usado pelo algoritmo AHEAD, somando-se a quantidade de dados armazenada em cada *Job*. Esse resultado mostra que é possível processar grafos com maior conectividade e número de vértices ao se utilizar as técnicas implementadas no algoritmo AHEAD, considerando o uso do mesmo recurso computacional.

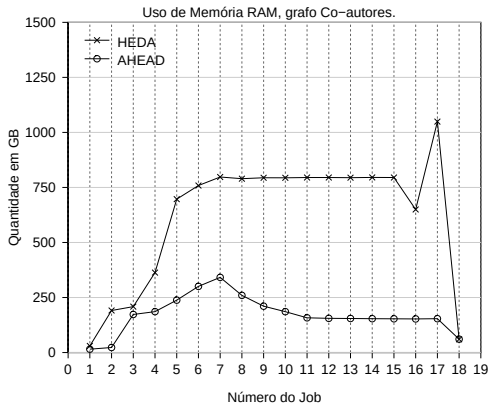


(a) Uso acumulativo da memória RAM no *cluster*.

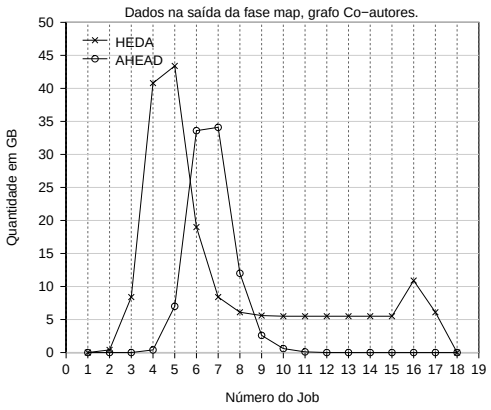
(b) Uso de espaço do sistema de arquivo agregado no *cluster*.

Figura 5.9: Comparação do uso do sistema de arquivo agregado e da memória RAM pelos algoritmos HEDA e AHEAD durante o processamento do grafo Enron.

Grafos com maior número de vértices e arestas geram quantidades maiores de dados na saída da fase *map*. Portanto as técnicas implementadas permitem que



(a) Uso acumulativo da memória RAM no *cluster*.



(b) Uso de espaço do sistema de arquivo agregado no *cluster*.

Figura 5.10: Comparação do uso do sistema de arquivo agregado e da memória RAM pelos algoritmos HEDA e AHEAD durante o processamento do grafo Co-autores.

grafos maiores sejam processados utilizando a mesma quantidade de espaço em disco. As Figuras 5.9 e 5.10 mostram resultados semelhantes, obtidos no processamento dos grafos Enron e Co-autores respectivamente. O resumo das otimizações observadas durante o teste de comparação são exibidas na Tabela 5.5. Além da possibilidade de processar grafos maiores, o tempo de execução é reduzido, devido ao menor número de operações de entrada/saída nos discos dos nós computacionais.

Tabela 5.5: Utilização de recursos pelos algoritmos HEDA e AHEAD

Recurso	Grafo	Algoritmo HEDA	Algoritmo AHEAD	Redução
Memória (GB)	Sintético	11161,6	3487,2	68,8%
	Enron	11152,7	3714,1	66,7%
	Co-autores	11157,6	3075,5	72,4%
Discos (GB)	Sintético	875,7	573,2	34,5%
	Enron	399,4	142,4	64,3%
	Co-autores	182,2	90,5	50,3%
Tempo (Min.)	Sintético	147,9	102,8	30,5%
	Enron	67,1	40,2	40,0%
	Co-autores	46,7	24,5	47,5%

5.8 Considerações finais

Neste capítulo apresentamos os resultados referentes ao experimentos do algoritmo AHEAD mediante o processamento de grafos reais e sintéticos. Também são apresentados resultados referentes à comparação com algoritmos sequenciais e paralelos. Na Seção 5.2 constatamos que o algoritmo AHEAD calcula de forma exata o diâmetro e raio de um grafo. Os resultados apresentados na seção mencionada também indicam que o modelo MapReduce/Hadoop apresenta *overhead* significativo dependendo do tamanho do grafo e dos recursos computacionais usados. Os resultados apresentados na Seção 5.1 e 5.4 indicam que algoritmo AHEAD apresenta melhor desempenho quando usado no processamento de grafos maiores, apresentando *speedup* superlinear e eficiência entre 1 e 1,2 nesses casos. Na Seção 5.5, os resultados indicaram que o algoritmo AHEAD apresenta taxa lenta de variação do tempo de execução com o aumento do número de arestas em grafos com alta conectividade, ou seja, grafos em que a relação arestas/vértices é alta. Finalmente, na Seção 5.7 os resultados mostram redução do uso de recursos do *cluster* ao se utilizar as técnicas implementadas no algoritmo AHEAD. Em especial, há redução do uso de espaço em disco de até 64% quando comparado com o algoritmo HEDA, o que indica a possibilidade de processamento de grafos maiores quando há restrição do espaço disponível em disco.

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 Conclusões

Esse trabalho apresenta o algoritmo paralelo AHEAD, desenvolvido para o processamento eficiente e exato de medidas de centralidades em grafos grandes. Experimentos foram feitos para análise da escalabilidade, do *speedup* e da eficiência do algoritmo. Também foram feitos testes de correção, além da comparação com algoritmos sequenciais e paralelos. Nos resultados obtidos por meio dos experimentos propostos, observamos que o algoritmo AHEAD apresentou melhor desempenho no processamento dos maiores grafos do conjunto de dados utilizado, com *speedup* superlinear e eficiência maior que a unidade nesses casos. Em relação ao comportamento do algoritmo AHEAD, quando varia-se o tamanho do grafo processado, os resultados indicaram que o mesmo apresenta variação lenta no tempo de execução com o aumento do número de arestas em grafos com alto número de conexões, ou seja, grafos em que o grau médio dos vértices é maior. Por meio da implementação das técnicas *In-Mapper Combiner* e *Schimmy* para algoritmos paralelos implementados no modelo MapReduce/Hadoop, foi possível reduzir em até 72% o uso da memória RAM e em até 64% o uso do espaço em disco, comparado ao algoritmo HEDA. A principal contribuição desse resultado é a possibilidade de expandir a escala de tamanhos de grafos que podem ser processados, considerando o cálculo exato e os recursos computacionais disponíveis.

O desafio de se adaptar o modelo MapReduce/Hadoop no processamento paralelo de grafos está na necessidade de superar os custos envolvidos na produção de resul-

tados parciais nos discos locais das máquinas do sistema distribuído. Os *mappers* e *reducers* gerenciados pela plataforma Hadoop não compartilham endereçamento global de memória RAM, e assim a computação distribuída nessa plataforma é incapaz de compartilhar informações sobre o processamento das partes sem que antes elas sejam armazenadas temporariamente no disco. Nesse sentido, as memórias individuais de cada nó computacional são estendidas por meio do uso intensivo dos discos, isto é, por operações de entrada/saída. Assim, os discos e a rede são os meios de comunicação no compartilhamento de dados que devem ocorrer na computação distribuída. Este é um custo que pode ser alto, principalmente se o processamento envolver vários ciclos *map-shuffle-reduce*. Em um sistema em que a troca de informação sobre a computação distribuída ocorre com acesso direto à memória entre as partes, o desempenho será certamente melhor.

A questão é, portanto, entender em que situações se justifica usar o modelo MapReduce/Hadoop. A resposta está no fato de que esse tipo de sistema distribuído apresenta facilidade e baixo custo operacional no aumento de sua estrutura. No nível de programação, há facilidades na implementação das operações que envolvem o processamento paralelo, com o programador focando apenas nas funções *map* e *reduce* mesmo para atividades que envolvem o processamento de dados na escala de petabytes. Além disso, a plataforma oferece gerenciamento automático de falhas, com a recuperação do processamento local em outras partes do sistema distribuído.

No caso particular de processamento de grafos, a vantagem está justamente na possibilidade de trabalhar com grafos que modelam redes complexas. Essas redes contém um alto número de elementos participantes, com o número de conexões igualmente elevado, de forma que a quantidade de memória necessária para processar tais grafos pode justificar o uso de modelos como o proposto no MapReduce/Hadoop. Mesmo que o processamento dessas redes dure horas ou até mesmo dias, não podemos dizer que os dados obtidos sejam obsoletos, pois redes desse porte mudam muito lentamente.

6.2 Trabalhos Futuros

Trabalhos futuros podem ser feitos no sentido de determinar qual é a melhor relação entre o número de *mappers* e *reducers* em um *cluster* Hadoop executando o AHEAD. Além disso, a implementação do AHEAD pode ser aprimorada por meio do particionamento do grafo de forma a minimizar o número de vértices vizinhos em partições diferentes, ao mesmo tempo que mantêm o tamanho das mesmas relativamente iguais. Uma possível intervenção que também pode trazer benefícios é o aprimoramento da maneira como a atualização das partições ocorrem. Os arquivos criados no *Hadoop Distributed File System* são sequenciais e, portanto, a busca por um registro específico passa necessariamente pela leitura de outros registros não relacionados. Por fim, a busca de alternativas para descrever a estrutura do grafo nos arquivos de saída também pode contribuir para que o algoritmo utilize menos espaço em disco.

Apêndice A

Dados de coleta

Neste apêndice são mostrados os dados referentes às figuras apresentadas no Capítulo 5, a exceção da Figura 5.7. Esta última é criada automaticamente por uma ferramenta de coleta desenvolvida para este trabalho. As seções estão organizadas de acordo com a ordem em que as figuras aparecem naquele capítulo.

A.1 Dados da Figura 5.1

Tabela A.1: Algoritmos sequenciais - Tempo de execução em um núcleo, em minutos. Grafo sintético com 50K vértices e 200K arestas.

Algoritmo	1º Teste	2º Teste	3º Teste	Média	Desvio Padrão
Sequencial1[1]	224,3	224,2	225,0	224,5	0,5
Sequencial2[2]	15,8	15,6	16,3	15,9	0,4

Tabela A.2: Algoritmo HEDA - Tempo de execução em minutos - Grafo sintético com 50K vértices e 200K arestas.

Número de Máquinas	1º Teste	2º Teste	3º Teste	Média	Desvio Padrão
1	-	-	-	-	-
2	252,2	252,8	251,2	252,1	0,6
3	158,6	158,9	158,5	158,7	0,2
4	123,6	125,1	125,3	124,7	0,8
5	104,9	105,0	104,4	104,8	0,3
6	91,5	92,0	91,4	91,7	0,2

[1]Implementado em [Hagberg, Schult e Swart 2008].

[2]Implementado em [Gonçalves, Maciel e Murta 2010].

Tabela A.3: Algoritmo AHEAD - Tempo de execução em minutos - Grafo sintético com 50K vértices e 200K arestas.

Número de Máquinas	1º Teste	2º Teste	3º Teste	Média	Desvio Padrão
1	274,9	272,1	274,9	274,0	1,7
2	112,7	114,8	119,5	115,7	3,5
3	75,2	75,0	75,2	75,2	0,1
4	58,7	58,3	58,9	58,6	0,3
5	50,7	51,2	49,1	50,3	1,1
6	43,7	42,9	42,2	42,9	0,8

A.2 Dados da Figura 5.2

Tabela A.4: Algoritmo AHEAD - Efeito no tempo de execução em minutos com a variação do número de núcleos.

Configuração do *cluster*	Grafo	1º Teste	2º Teste	3º Teste	Tempo médio	Desvio Padrão
6 máquinas (138 núcleos)	Co-Autores	24,2	24,3	24,9	24,5	0,4
	Enron	39,0	40,8	39,7	39,8	0,9
	Sintético	43,7	42,9	42,2	42,9	0,8
	Epinion	88,6	91,7	89,3	89,8	1,6
5 máquinas (115 núcleos)	Co-Autores	27,3	27,7	27,2	27,4	0,3
	Enron	46,1	43,8	43,9	44,6	1,3
	Sintético	50,7	51,2	49,1	50,3	1,1
	Epinion	102,5	104,7	102,1	103,1	1,4
4 máquinas (92 núcleos)	Co-Autores	30,9	32,8	32,5	32,1	1,0
	Enron	52,6	52,0	51,4	52,0	0,6
	Sintético	58,7	58,3	58,9	58,6	0,3
	Epinion	126,1	130,6	125,2	127,3	2,9
3 máquinas (69 núcleos)	Co-Autores	37,8	37,1	37,5	37,5	0,3
	Enron	65,3	64,1	65,5	65,0	0,8
	Sintético	75,2	75,0	75,2	75,2	0,1
	Epinion	167,3	168,7	167,6	167,9	0,7
2 máquinas (46 núcleos)	Co-Autores	49,7	50,3	50,8	50,3	0,6
	Enron	88,1	88,3	88,2	88,2	0,1
	Sintético	112,7	114,8	119,5	115,7	3,5
	Epinion	254,9	254,3	249,5	252,9	3,0
1 máquina (23 núcleos)	Co-Autores	86,9	85,6	86,9	86,5	0,7
	Enron	182,7	179,6	179,3	180,5	1,9
	Sintético	274,9	272,1	274,9	274,0	1,7
	Epinion	636,7	629,3	624,5	630,2	6,1

A.3 Dados da Figura 5.3(a)

Tabela A.5: Algoritmo AHEAD - *Speedup*

Configuração do *cluster*	Grafo	1º Teste	2º Teste	3º Teste	Speedup Médio	Desvio Padrão
6 máquinas (138 núcleos)	Co-Autores	3,59	3,53	3,44	3,52	0,08
	Enron	4,68	4,41	4,52	4,54	0,14
	Sintético	6,29	6,34	6,52	6,38	0,12
	Epinion	7,19	6,87	6,99	7,02	0,16
5 máquinas (115 núcleos)	Co-Autores	3,19	3,10	3,20	3,16	0,06
	Enron	3,96	4,10	4,08	4,05	0,07
	Sintético	5,42	5,31	5,60	5,45	0,15
	Epinion	6,21	6,01	6,12	6,11	0,10
4 máquinas (92 núcleos)	Co-Autores	2,81	2,61	2,67	2,70	0,10
	Enron	3,47	3,45	3,49	3,47	0,02
	Sintético	4,69	4,67	4,67	4,67	0,01
	Epinion	5,05	4,82	4,99	4,95	0,12
3 máquinas (69 núcleos)	Co-Autores	2,30	2,31	2,32	2,31	0,01
	Enron	2,80	2,80	2,74	2,78	0,04
	Sintético	3,65	3,63	3,65	3,64	0,02
	Epinion	3,80	3,73	3,73	3,75	0,04
2 máquinas (46 núcleos)	Co-Autores	1,75	1,70	1,71	1,72	0,03
	Enron	2,07	2,03	2,03	2,05	0,02
	Sintético	2,44	2,37	2,30	2,37	0,07
	Epinion	2,50	2,47	2,50	2,49	0,02

A.4 Dados da Figura 5.3(b)

Tabela A.6: Algoritmo AHEAD - Eficiência

Configuração do *cluster*	Grafo	1º Teste	2º Teste	3º Teste	Speedup Médio	Desvio Padrão
6 máquinas (138 núcleos)	Co-Autores	0,60	0,59	0,57	0,59	0,01
	Enron	0,78	0,73	0,75	0,76	0,02
	Sintético	1,05	1,06	1,09	1,06	0,02
	Epinion	1,20	1,14	1,17	1,17	0,03
5 máquinas (115 núcleos)	Co-Autores	0,64	0,62	0,64	0,63	0,01
	Enron	0,79	0,82	0,82	0,81	0,01
	Sintético	1,08	1,06	1,12	1,09	0,03
	Epinion	1,24	1,20	1,22	1,22	0,02
4 máquinas (92 núcleos)	Co-Autores	0,70	0,65	0,67	0,67	0,03
	Enron	0,87	0,86	0,87	0,87	0,00
	Sintético	1,17	1,17	1,17	1,17	0,00
	Epinion	1,26	1,20	1,25	1,24	0,03
3 máquinas (69 núcleos)	Co-Autores	0,77	0,77	0,77	0,77	0,00
	Enron	0,93	0,93	0,91	0,93	0,01
	Sintético	1,22	1,21	1,22	1,21	0,01
	Epinion	1,27	1,24	1,24	1,25	0,01
2 máquinas (46 núcleos)	Co-Autores	0,87	0,85	0,85	0,86	0,01
	Enron	1,04	1,02	1,02	1,02	0,01
	Sintético	1,22	1,18	1,15	1,18	0,03
	Epinion	1,25	1,24	1,25	1,25	0,01

A.5 Dados da Figura 5.4

Tabela A.7: Algoritmo AHEAD - Efeito no tempo de execução em minutos com a variação do número de arestas em grafos não direcionados.

Vértices/ Arestas	1º Teste	1º Teste	1º Teste	Tempo médio	Desvio padrão
50K/200K	47,1	46,3	46,4	46,6	0,4
50K/400K	56,1	56,0	56,1	56,1	0,0
50K/800K	94,8	98,6	96,8	96,7	1,9
50K/1600K	233,7	233,6	231,9	233,1	1,0

Tabela A.8: Algoritmo HEDA - Efeito no tempo de execução em minutos com a variação do número de arestas em grafos não direcionados.

Vértices/ Arestas	1º Teste	1º Teste	1º Teste	Tempo médio	Desvio padrão
50K/200K	91,6	91,5	91,5	97,5	0,0
50K/400K	98,2	98,3	96,1	97,5	1,2
50K/800K	124,0	125,4	124,3	124,6	0,7
50K/900K	146,3	148,3	144,5	146,4	1,9

A.6 Dados da Figura 5.5(a)

Tabela A.9: Algoritmo AHEAD - Proporção do *Job* mais longo em relação ao tempo total de execução.

Vértices/ Arestas	1º Teste	2º Teste	3º Teste	Média	Desvio Padrão
50K/100K	7,1%	6,8%	6,4%	6,8%	0,3%
50K/200K	22,3%	22,5%	22,9%	22,6%	0,3%
50K/400K	38,0%	38,0%	38,1%	38,0%	0,1%
50K/800K	59,8%	58,1%	58,9%	58,9%	0,9%
50K/1600K	50,1%	51,0%	50,5%	50,5%	0,5%

A.7 Dados da Figura 5.5(b)

Tabela A.10: Algoritmo AHEAD - Proporção das fases MapReduce - Grafo sintético com 50K vértices e 200K arestas.

Fase	1º Teste	2º Teste	3º Teste	Média	Desvio Padrão
Map	20,9%	21,9%	21,6%	21,5%	0,5%
Map/Reduce	43,6%	43,7%	45,7%	44,3%	1,2%
Reduce	34,4%	32,7%	34,2%	33,8%	0,9%

Tabela A.11: Algoritmo AHEAD - Proporção das fases MapReduce - Grafo sintético com 50K vértices e 400K arestas.

Fase	1º Teste	2º Teste	3º Teste	Média	Desvio Padrão
Map	27,4%	27,3%	26,8%	27,2%	0,3%
Map/Reduce	55,5%	55,8%	56,0%	55,8%	0,2%
Reduce	16,9%	17,2%	17,0%	17,0%	0,1%

Tabela A.12: Algoritmo AHEAD - Proporção das fases MapReduce - Grafo sintético com 50K vértices e 800K arestas.

Fase	1º Teste	2º Teste	3º Teste	Média	Desvio Padrão
Map	26,3%	25,7%	26,0%	26,0%	0,3%
Map/Reduce	64,9%	65,6%	65,2%	65,3%	0,4%
Reduce	8,7%	8,8%	8,7%	8,7%	0,1%

Tabela A.13: Algoritmo AHEAD - Proporção das fases MapReduce - Grafo sintético com 50K vértices e 1600K arestas.

Fase	1º Teste	2º Teste	3º Teste	Média	Desvio Padrão
Map	32,1%	31,7%	32,1%	32,0%	0,2%
Map/Reduce	59,3%	60,9%	59,4%	59,9%	0,9%
Reduce	7,4%	8,5%	8,2%	8,0%	0,5%

A.8 Dados da Figura 5.6

Tabela A.14: Algoritmo AHEAD - Efeito no tempo de execução em minutos com a variação do número de vértice e arestas em grafos direcionados.

Vértices/Arestas	1º Teste	2º Teste	3º Teste	Tempo médio	Desvio padrão
25K/75K	19,4	19,3	19,3	19,3	0,1
50K/150K	44,2	43,6	44,6	44,1	0,5
100K/300K	168,7	168,1	174,7	170,5	3,6
200K/600K	1221,5	1202,8	1214,1	1212,8	9,4

A.9 Dados da Figura 5.8(a)

Tabela A.15: Algoritmo AHEAD - Uso da memória principal em gigabytes - Grafo sintético com 50K vértices e 800K arestas.

Ciclo	1º Teste	2º Teste	3º Teste	Média de Uso	Desvio padrão
job1	15,2	15,4	15,3	15,3	0,1
job2	60,0	59,9	59,9	59,9	0,0
job3	331,1	330,7	330,4	330,7	0,4
job4	333,7	332,9	332,6	333,1	0,5
job5	378,9	378,3	378,5	378,6	0,3
job6	511,3	511,1	513,0	511,8	1,1
job7	746,0	744,2	745,8	745,3	1,0
job8	645,7	644,1	643,7	644,5	1,0
job9	322,5	320,6	321,5	321,5	0,9
job10	146,5	146,4	146,5	146,4	0,1

Tabela A.16: Algoritmo HEDA - Uso da memória principal em gigabytes - Grafo sintético com 50K vértices e 800K arestas.

Ciclo	1º Teste	2º Teste	3º Teste	Média de Uso	Desvio padrão
job1	40,0	39,5	40,2	39,9	0,3
job2	194,5	194,6	195,1	194,7	0,3
job3	213,6	213,8	212,6	213,3	0,6
job4	389,2	390,1	389,7	389,7	0,4
job5	1547,7	1551,4	1535,4	1544,8	8,4
job6	2051,2	2049,7	2049,6	2050,2	0,9
job7	1814,8	1814,0	1814,5	1814,4	0,4
job8	1728,3	1728,0	1728,2	1728,2	0,1
job9	3126,7	3125,6	3126,4	3126,2	0,6
job10	60,0	60,1	60,4	60,2	0,2

A.10 Dados da Figura 5.8(b)

Tabela A.17: Algoritmo AHEAD - Uso do espaço em disco em gigabytes - Grafo sintético com 50K vértices e 800K arestas.

Ciclo	1º Teste	2º Teste	3º Teste	Média de Uso	Desvio padrão
job1	0,0	0,0	0,0	0,0	0,0
job2	0,0	0,0	0,0	0,0	0,0
job3	0,0	0,0	0,0	0,0	0,0
job4	0,2	0,2	0,2	0,2	0,0
job5	3,1	3,1	3,1	3,1	0,0
job6	46,6	46,6	46,6	46,6	0,0
job7	371,8	371,8	371,8	371,8	0,0
job8	151,5	151,5	151,5	151,5	0,0
job9	0,0	0,0	0,0	0,0	0,0
job10	0,0	0,0	0,0	0,0	0,0

Tabela A.18: Algoritmo HEDA - Uso do espaço em disco em gigabytes - Grafo sintético com 50K vértices e 800K arestas.

Ciclo	1º Teste	2º Teste	3º Teste	Média de Uso	Desvio padrão
job1	0,0	0,0	0,0	0,0	0,0
job2	0,2	0,2	0,2	0,2	0,0
job3	3,7	3,7	3,7	3,7	0,0
job4	55,7	55,7	55,7	55,7	0,0
job5	437,9	437,9	437,9	437,9	0,0
job6	210,0	210,0	210,0	210,0	0,0
job7	40,9	40,9	40,9	40,9	0,0
job8	81,8	81,8	81,8	81,8	0,0
job9	45,5	45,5	45,5	45,5	0,0
job10	0,0	0,0	0,0	0,0	0,0

A.11 Dados da Figura 5.9(a)

Tabela A.19: Algoritmo AHEAD - Uso da memória principal em gigabytes - Grafo Enron.

Ciclo	1º Teste	2º Teste	3º Teste	Média de Uso	Desvio padrão
job1	15,3	15,3	15,2	15,3	0,1
job2	49,1	49,1	49,1	49,1	0,0
job3	210,8	211,0	211,2	211,0	0,2
job4	220,3	220,5	220,7	220,5	0,2
job5	300,5	300,1	299,9	300,2	0,3
job6	362,2	361,7	362,4	362,1	0,4
job7	480,7	480,8	480,5	480,6	0,1
job8	287,1	288,5	288,5	288,0	0,8
job9	241,8	242,2	241,9	242,0	0,2
job10	211,5	214,3	211,3	212,4	1,7
job11	211,6	207,0	207,3	208,6	2,6
job12	210,5	210,6	210,7	210,6	0,1
job13	210,3	210,3	210,4	210,3	0,1
job14	210,1	210,5	210,2	210,3	0,2
job15	210,3	209,8	209,3	209,8	0,5
job16	209,1	209,9	210,2	209,7	0,6
job17	73,5	73,3	73,4	73,4	0,1

Tabela A.20: Algoritmo HEDA - Uso da memória principal em gigabytes - Grafo Enron.

Ciclo	1º Teste	2º Teste	3º Teste	Média de Uso	Desvio padrão
job1	24,7	24,6	24,6	24,6	0,0
job2	167,1	167,6	167,4	167,3	0,3
job3	228,0	228,1	228,1	228,0	0,0
job4	401,8	400,7	400,7	401,1	0,6
job5	737,0	737,0	736,9	737,0	0,1
job6	792,6	792,6	792,5	792,6	0,1
job7	819,5	819,5	819,8	819,6	0,2
job8	814,0	815,0	813,8	814,3	0,6
job9	815,9	815,1	815,0	815,3	0,5
job10	816,5	816,5	815,1	816,0	0,8
job11	815,6	816,2	816,2	816,0	0,4
job12	817,3	816,1	816,4	816,6	0,6
job13	816,7	817,0	816,5	816,7	0,3
job14	816,2	817,7	816,2	816,7	0,9
job15	828,2	826,7	826,2	827,1	1,0
job16	1383,7	1383,5	1384,1	1383,7	0,3
job17	60,1	59,8	60,3	60,1	0,3

A.12 Dados da Figura 5.9(b)

Tabela A.21: Algoritmo AHEAD - Uso do espaço em disco em gigabytes - Grafo Enron.

Ciclo	1º Teste	2º Teste	3º Teste	Média de Uso	Desvio padrão
job1	0,0	0,0	0,0	0,0	0,0
job2	0,0	0,0	0,0	0,0	0,0
job3	0,0	0,0	0,0	0,0	0,0
job4	0,7	0,7	0,7	0,7	0,0
job5	17,6	17,6	17,6	17,6	0,0
job6	61,0	61,0	61,0	61,0	0,0
job7	45,8	45,8	45,8	45,8	0,0
job8	13,9	13,9	13,9	13,9	0,0
job9	2,9	2,9	2,9	2,9	0,0
job10	0,5	0,5	0,5	0,5	0,0
job11	0,1	0,1	0,1	0,1	0,0
job12	0,0	0,0	0,0	0,0	0,0
job13	0,0	0,0	0,0	0,0	0,0
job14	0,0	0,0	0,0	0,0	0,0
job15	0,0	0,0	0,0	0,0	0,0
job16	0,0	0,0	0,0	0,0	0,0
job17	0,0	0,0	0,0	0,0	0,0

Tabela A.22: Algoritmo HEDA - Uso do espaço em disco em gigabytes - Grafo Enron.

Ciclo	1º Teste	2º Teste	3º Teste	Média de Uso	Desvio padrão
job1	0,0	0,0	0,0	0,0	0
job2	0,8	0,8	0,8	0,8	0
job3	20,0	20,0	20,0	20,0	0
job4	73,8	73,8	73,8	73,8	0
job5	64,2	64,2	64,2	64,2	0
job6	33,1	33,1	33,1	33,1	0
job7	21,5	21,5	21,5	21,5	0
job8	18,8	18,8	18,8	18,8	0
job9	18,4	18,4	18,4	18,4	0
job10	18,3	18,3	18,3	18,3	0
job11	18,3	18,3	18,3	18,3	0
job12	18,3	18,3	18,3	18,3	0
job13	18,3	18,3	18,3	18,3	0
job14	18,3	18,3	18,3	18,3	0
job15	36,7	36,7	36,7	36,7	0
job16	20,5	20,5	20,5	20,5	0
job17	0,0	0,0	0,0	0,0	0

A.13 Dados da Figura 5.10(a)

Tabela A.23: Algoritmo AHEAD - Uso da memória principal em gigabytes - Grafo Co-autores.

Ciclo	1º Teste	2º Teste	3º Teste	Média de Uso	Desvio padrão
job1	15,2	15,3	15,4	15,3	0,1
job2	22,9	22,9	23,0	22,9	0,1
job3	172,4	173,4	173,1	173,0	0,5
job4	183,9	187,1	186,3	185,8	1,7
job5	238,1	238,3	237,8	238,1	0,3
job6	300,4	300,6	300,8	300,6	0,2
job7	341,4	341,4	341,5	341,5	0,1
job8	259,5	260,3	259,6	259,8	0,4
job9	212,1	209,5	210,2	210,6	1,3
job10	185,7	186,0	185,4	185,7	0,3
job11	158,8	158,0	156,7	157,8	1,1
job12	155,3	155,2	155,6	155,4	0,2
job13	155,1	154,2	154,6	154,7	0,5
job14	154,4	153,5	153,7	153,8	0,5
job15	153,3	154,2	152,4	153,3	0,9
job16	152,1	153,1	152,8	152,7	0,5
job17	153,4	155,3	152,6	153,8	1,4
job18	60,8	61,0	60,8	60,8	0,1

Tabela A.24: Algoritmo HEDA - Uso da memória principal em gigabytes - Grafo Co-autores.

Ciclo	1º Teste	2º Teste	3º Teste	Média de Uso	Desvio padrão
job1	30,1	30,0	29,8	30,0	0,1
job2	190,9	191,1	191,2	191,0	0,1
job3	208,7	208,6	208,6	208,6	0,0
job4	363,4	363,1	363,2	363,2	0,1
job5	697,0	696,5	696,8	696,8	0,2
job6	758,5	759,1	758,5	758,7	0,3
job7	797,8	796,9	797,5	797,4	0,5
job8	788,9	790,2	790,4	789,8	0,8
job9	793,7	793,3	795,0	794,0	0,9
job10	794,1	793,9	793,4	793,8	0,4
job11	795,0	794,2	796,0	795,1	0,9
job12	795,2	794,9	795,2	795,1	0,2
job13	793,8	794,4	795,0	794,4	0,6
job14	794,8	796,3	795,4	795,5	0,7
job15	795,5	794,4	795,2	795,1	0,6
job16	649,8	650,1	649,9	649,9	0,2
job17	1049,5	1049,2	1048,6	1049,1	0,5
job18	60,0	60,3	59,9	60,1	0,2

A.14 Dados da Figura 5.10(b)

Tabela A.25: Algoritmo AHEAD - Uso do espaço em disco em gigabytes - Grafo Co-autores.

Ciclo	1º Teste	2º Teste	3º Teste	Média de Uso	Desvio padrão
job1	0,0	0,0	0,0	0,0	0
job2	0,0	0,0	0,0	0,0	0
job3	0,0	0,0	0,0	0,0	0
job4	0,4	0,4	0,4	0,4	0
job5	7,0	7,0	7,0	7,0	0
job6	33,6	33,6	33,6	33,6	0
job7	34,1	34,1	34,1	34,1	0
job8	12,0	12,0	12,0	12,0	0
job9	2,6	2,6	2,6	2,6	0
job10	0,6	0,6	0,6	0,6	0
job11	0,1	0,1	0,1	0,1	0
job12	0,0	0,0	0,0	0,0	0
job13	0,0	0,0	0,0	0,0	0
job14	0,0	0,0	0,0	0,0	0
job15	0,0	0,0	0,0	0,0	0
job16	0,0	0,0	0,0	0,0	0
job17	0,0	0,0	0,0	0,0	0
job18	0,0	0,0	0,0	0,0	0

Tabela A.26: Algoritmo HEDA - Uso do espaço em disco em gigabytes - Grafo Co-autores.

Ciclo	1º Teste	2º Teste	3º Teste	Média de Uso	Desvio padrão
job1	0,0	0,0	0,0	0,0	0,0
job2	0,4	0,4	0,4	0,4	0,0
job3	8,4	8,4	8,4	8,4	0,0
job4	40,8	40,8	40,8	40,8	0,0
job5	43,4	43,4	43,4	43,4	0,0
job6	19,0	19,0	19,0	19,0	0,0
job7	8,4	8,4	8,4	8,4	0,0
job8	6,1	6,1	6,1	6,1	0,0
job9	5,6	5,6	5,6	5,6	0,0
job10	5,5	5,5	5,5	5,5	0,0
job11	5,5	5,5	5,5	5,5	0,0
job12	5,5	5,5	5,5	5,5	0,0
job13	5,5	5,5	5,5	5,5	0,0
job14	5,5	5,5	5,5	5,5	0,0
job15	5,5	5,5	5,5	5,5	0,0
job16	10,9	10,9	10,9	10,9	0,0
job17	6,1	6,1	6,1	6,1	0,0
job18	0,0	0,0	0,0	0,0	0,0

A.15 Dados de tempo de execução da Tabela 5.5

Tabela A.27: Algoritmo AHEAD - Tempo de execução em minutos. Grafo sintético com 50K vértices e 800K arestas, Enron e Co-autores.

Grafo	1º Teste	2º Teste	3º Teste	Média	Desvio Padrão
Sintético	102,7	102,5	103,1	102,8	0,3
Enron	39,8	40,1	40,8	40,2	0,5
Co-autores	24,7	24,4	24,3	24,5	0,2

Tabela A.28: Algoritmo HEDA - Tempo de execução em minutos. Grafo sintético com 50K vértices e 800K arestas, Enron e Co-autores.

Grafo	1º Teste	2º Teste	3º Teste	Média	Desvio Padrão
Sintético	147,1	146,6	150,0	147,9	1,8
Enron	67,0	67,1	67,0	67,1	0,1
Co-autores	46,5	46,8	46,7	46,7	0,2

Referências Bibliográficas

[Bader, Cong e Feo 2005]BADER, D.; CONG, G.; FEO, J. On the architectural requirements for efficient execution of graph algorithms. In: *Parallel Processing, 2005. ICPP 2005. International Conference on*. [S.l.: s.n.], 2005. p. 547 – 556.

[Bader e Madduri 2006]BADER, D.; MADDURI, K. Parallel algorithms for evaluating centrality indices in real-world networks. In: *Parallel Processing, 2006. ICPP 2006. International Conference on*. [S.l.: s.n.], 2006. p. 539 –550.

[Bavelas 1948]BAVELAS, A. A mathematical model of Group Structure. *Human Organizations*, v. 7, p. 16–30, 1948.

[Brandes 2001]BRANDES, U. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, Gordon Breach Publishing, Taylor & Francis Group, 25, n. 2, p. 163–177, 2001.

[Bryant 2011]BRYANT, R. Data-intensive scalable computing for scientific applications. *Computing in Science & Engineering*, v. 13, n. 6, p. 25–33, Nov.-Dec. 2011.

[Buckley e Harary 1990]BUCKLEY, F.; HARARY, F. *Distance in graphs*. [S.l.]: Addison-Wesley Pub. Co., 1990. (The Advanced Book Program).

[Buluç e Gilbert 2011]BULUÇ, A.; GILBERT, J. R. The Combinatorial BLAS: design, implementation, and applications. *International Journal of High Performance Computing Applications*, Sage Publications LTD, 25, n. 4, p. 496–509, Nov 2011.

[Buluç e Madduri 2011]BULUÇ, A.; MADDURI, K. Parallel breadth-first search on distributed memory systems. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. [S.l.]: ACM, 2011. (SC'11), p. 65:1–65:12.

[Cohen 2009]COHEN, J. Graph Twiddling in a MapReduce World. *Computing in Science & Engineering*, IEEE Computer Soc, 11, n. 4, p. 29–41, Jul-Aug 2009.

[Cormen et al. 2009]CORMEN, T. H. et al. *Introduction to Algorithms, Third Edition*. 3rd. ed. [S.l.]: The MIT Press, 2009.

[Crauser et al. 1998]CRAUSER, A. et al. A parallelization of Dijkstra's shortest path algorithm. In: Brim, L and Gruska, J and Zlatuska, J (Ed.). *Mathematical Foundations of Computer Science 1998*. [S.l.]: Springer-Verlag Berlim, 1998. (Lecture Notes in Computer Science, 1450), p. 722–731.

[Dean e Ghemawat 2004]DEAN, J.; GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In: *USENIX Association Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDE'04)*. [S.l.]: USENIX ASSOC, 2004. p. 137–149.

[Dean e Ghemawat 2008]DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, ACM, v. 51, n. 1, p. 107–113, Jan 2008.

[Dijkstra 1959]DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik*, Springer Berlin / Heidelberg, v. 1, p. 269–271, 1959.

[Flajolet e Martin 1985]FLAJOLET, P.; MARTIN, G. N. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, Academic Press, Inc., v. 31, n. 2, p. 182–209, set. 1985.

[Foster 1995]FOSTER, I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1995.

[Freeman 1977]FREEMAN, L. C. A set of measures of centrality based on betweenness. *Sociometry*, American Sociological Association, v. 40, n. 1, p. pp. 35–41, 1977.

[Freeman 1979]FREEMAN, L. C. Centrality in social networks conceptual clarification. *Social Networks*, v. 1, n. 3, p. 215 – 239, 1979.

[Gago, Hurajova e Madaras 2012]GAGO, S.; HURAJOVA, J.; MADARAS, T. Notes on the betweenness centrality of a graph. *Mathematica Slovaca*, Versita, 62, n. 1, p. 1–12, FEB 2012.

[Gonçalves, Maciel e Murta 2010]GONÇALVES, M. R. S.; MACIEL, J. N.; MURTA, C. D. Geração de Topologias da Internet por Redução do Grafo Original. In: *Anais do XXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. [S.l.: s.n.], 2010. p. 959–972.

[Gross e Yellen 2006]GROSS, J.; YELLEN, J. *Graph Theory and Its Applications*. Boca Raton, Florida: Chapman & Hall/CRC, 2006. (Discrete Mathematics And Its Applications).

[Hagberg, Schult e Swart 2008]HAGBERG, A. A.; SCHULT, D. A.; SWART, P. J. Exploring network structure, dynamics, and function using NetworkX. In: *Proceedings of the 7th Python in Science Conference (SciPy2008)*. [S.l.: s.n.], 2008. p. 11–15.

[Hage e Harary 1995]HAGE, P.; HARARY, F. Eccentricity and Centrality In Networks. *Social Networks*, 17, n. 1, p. 57–63, Jan 1995.

[Harary 1969]HARARY, F. *Graph theory*. [S.l.]: Addison-Wesley Pub. Co., 1969. (Addison-Wesley series in Mathematics).

[Hong, Oguntebi e Olukotun 2011]HONG, S.; OGUNTEBI, T.; OLUKOTUN, K. Efficient parallel graph exploration on multi-core CPU and GPU. In: *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. [S.l.: s.n.], 2011. p. 78 –88.

[Kang et al. 2011]KANG, U. et al. Centralities in large networks: Algorithms and observations. In: *SIAM International Conference on Data Mining*. [S.l.]: SIAM / Omnipress, 2011. p. 119–130.

[Kang et al. 2011]KANG, U. et al. Hadi: Mining radii of large graphs. *ACM Trans. Knowl. Discov. Data*, ACM, v. 5, n. 2, p. 8:1–8:24, Feb 2011.

[Karloff, Suri e Vassilvitskii 2010]KARLOFF, H.; SURI, S.; VASSILVITSKII, S. A model of computation for MapReduce. In: *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*. [S.l.]: Society for Industrial and Applied Mathematics, 2010. (SODA'10), p. 938–948.

[Kimball, Michels-Slettvet e Bisciglia 2008]KIMBALL, A.; MICHELS-SLETTVET, S.; BISCIGLIA, C. Cluster computing for web-scale data processing. *SIGCSE Bull.*, ACM, v. 40, n. 1, p. 116–120, mar. 2008.

[Lee et al. 2011]LEE, K.-H. et al. Parallel data processing with MapReduce: a survey. *SIGMOD Rec.*, ACM, v. 40, n. 4, p. 11–20, Jan 2011.

[Lin e Dyer 2010]LIN, J.; DYER, C. *Data-Intensive Text Processing with MapReduce*. [S.l.]: Morgan and Claypool Publishers., 2010.

[Lin e Schatz 2010]LIN, J.; SCHATZ, M. Design patterns for efficient graph algorithms in MapReduce. In: *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*. [S.l.]: ACM, 2010. (MLG '10), p. 78–85.

[Malewicz et al. 2010]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*. [S.l.]: ACM, 2010. (SIGMOD'10), p. 135–146.

[Möhring et al. 2007]MöHRING, R. H. et al. Partitioning graphs to speedup Dijkstra's algorithm. *J. Exp. Algorithmics*, ACM, v. 11, fev. 2007.

[Muntés-Mulero et al. 2010]MUNTÉS-MULERO, V. et al. Graph partitioning strategies for efficient bfs in shared-nothing parallel systems. In: *Proceedings of the 2010 international conference on Web-age information management*. [S.l.]: Springer-Verlag, 2010. (WAIM'10), p. 13–24.

[Nascimento 2011]NASCIMENTO, J. P. B. *Um Algoritmo Paralelo para Cálculo de Centralidade em Grafos Grandes*. Dissertação (Mestrado) — PPGMMC, CEFET-MG., 2011.

[Nascimento e Murta 2012]NASCIMENTO, J. P. B.; MURTA, C. D. Um Algoritmo Paralelo para Cálculo de Centralidade em Grafos Grandes. In: *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, Ouro Preto. Anais do XXX SBRC*. [S.l.: s.n.], 2012.

[Page et al. 1999]PAGE, L. et al. *The PageRank Citation Ranking: Bringing Order to the Web*. [S.l.], November 1999.

[Patterson e Hennessy 2008]PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. 4th. ed. [S.l.]: Morgan Kaufmann Publishers Inc., 2008.

[Sabidussi 1966]SABIDUSSI, G. Centrality Index of a Graph. *Psychometrika*, Psychometric Soc, 31, n. 4, p. 581, 1966.

[Srirama, Jakovits e Vainikko 2012]SRIRAMA, S. N.; JAKOVITS, P.; VAINIKKO, E. Adapting scientific computing problems to clouds using MapReduce. *Future Generation Computer Systems - The International Journal of Grid Computing and eScience*, ELSEVIER SCIENCE BV, 28, n. 1, p. 184–192, Jan 2012.

[Tan, Tu e Sun 2009]TAN, G.; TU, D.; SUN, N. A parallel algorithm for computing betweenness centrality. In: *Parallel Processing, 2009. ICPP '09. International Conference on*. [S.l.: s.n.], 2009. p. 340 –347.

[Townsend 1987]TOWNSEND, M. *Discrete mathematics: applied combinatorics and graph theory*. [S.l.]: Benjamin/Cummings Pub. Co., 1987.

[White 2012]WHITE, T. *Hadoop: The Definitive Guide*. 3rd. ed. [S.l.]: O'Reilly Media, Inc., 2012.

[Ziviani 2004]ZIVIANI, N. *Projeto de algoritmos: com implementações em Pascal e C*. [S.l.]: Thomson Pioneira, 2004.