Um Algoritmo Paralelo Eficiente para Cálculo de Centralidade em Grafos

Leonardo Carlos da Cruz
Departamento de Computação
CEFET-MG
Belo Horizonte, Brasil
Email: leonardocruz@ufmg.br

Cristina Duarte Murta
Departamento de Computação
CEFET-MG
Belo Horizonte, Brasil
Email: cristina@decom.cefetmg.br

***Resumo*—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 dos grafos. Neste artigo apresentamos um novo algoritmo paralelo para o cálculo exato de centralidades em grafos grandes, usando o paradigma de programação MapReduce/Hadoop. O objetivo é computar as distâncias entre os vértices e extrair medidas de centralidades decorrentes dessas distâncias, fazendo uso eficiente dos recursos computacionais. Para avaliar o algoritmo proposto foram processados diversos grafos e os resultados comparados com algoritmos sequenciais e paralelos. Os experimentos mostraram que o algoritmo proposto faz uso eficiente da memória e do espaço em disco, comparado à outras implementações.**

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

I. Introdução

A crescente facilidade de aquisição de dados associada à ampla disponibilidade de dispositivos de armazenamento possibilitou o surgimento de grandes massas de dados em diversos contextos. Assim, em várias áreas do conhecimento, da coleta de amostras de moléculas de DNA, capazes 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 [1], há cada vez mais dados para processar e armazenar. Um modelo comumente usado para representar tais conjuntos de dados são os grafos. Portanto, explorar modelos baseados em grafos é uma importante tarefa para auxiliar a descoberta científica. Contudo, desenvolver técnicas eficientes capazes de extrair propriedades em grafos é uma tarefa não trivial, e o desafio torna-se ainda maior quando estamos trabalhando com grafos grandes, em que o número de arestas e vértices alcançam escalas de milhões ou maiores.

No contexto do processamento paralelo de grandes quantidades de dados, o modelo de programação paralela MapReduce foi adotado por empresas de tecnologia da informação e no meio acadêmico. Um dos fatores que permitiram seu uso foi o desenvolvimento da plataforma Hadoop, uma implementação *open source* do modelo, que computa com sucesso dados na escala de petabytes. Assim, tornou-se desejável aplicar esse modelo ao processamento de grafos grandes.

Nesse contexto, apresentamos o AHEAD (*Advanced Hadoop Exact Algorithm for Distances*), um algoritmo paralelo para processamento de grafos grandes, implementado em MapReduce/Hadoop. Nosso objetivo é investigar a aplicabilidade do modelo MapReduce ao processamento de grafos, particularmente ao processamento das distâncias entre os vértices, bem como a escalabilidade desse processamento em grafos com quantidades de vértices e arestas em várias escalas, e também o uso eficiente dos recursos computacionais no procedimento.

A computação de medidas de centralidade em grafos apresenta barreiras no que diz respeito à complexidade de sua computação [2]. Essas medidas, que incluem o diâmetro e o raio do grafo, tornam-se difíceis, senão proibitivas, de serem calculadas em grafos de escala muito grande [3]. Assim, 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, podemos ter apenas um cálculo aproximado destas medidas [3], [4].

Esse trabalho dá continuidade ao trabalho desenvolvido em [5], em que 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 HEDA apresenta limitações quanto ao uso do espaço de armazenamento de dados [6]. Essas limitações têm relação com o uso que o algoritmo faz da memória principal e o modelo de armazenamento de dados do Hadoop no sistema de arquivos. Para contornar essas limitações, o AHEAD aplica as técnicas propostas no trabalho [7], o que permite avanços na obtenção de medidas exatas de diâmetro e raio em grafos de escalas maiores. Os resultados apresentados nesse artigo mostram melhorias significativas quanto ao uso da memória principal e espaço em disco em relação a outras implementações.

II. Trabalhos Relacionados

A. Modelo MapReduce

O modelo de programação MapReduce tem por objetivo descrever uma maneira de processar grandes quantidades de dados de forma paralela [8]. 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

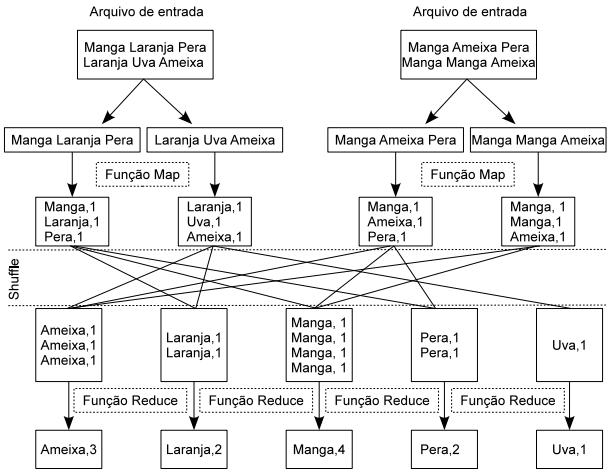


Figura 1. O modelo MapReduce para um contador de palavras.

de vários desses nós. Assim, o modelo inclui uma dinâmica de conexões entre os recursos computacionais, cujo objetivo é distribuir de forma coerente os dados e o processamento. Adicionalmente, o MapReduce foi projetado para ser usado em aglomerados de máquinas comuns [8]. Essa condição permite flexibilidade na redução de custos com as mesmas, o que é especialmente interessante quando a expectativa é usar 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 a uma menor robustez das máquinas, mas também devido ao número de máquinas presentes no sistema.

Do ponto de vista de programação paralela, o MapReduce especifica apenas duas funções que devem ser escritas pelo programador: a função *Map* e a função *Reduce*. A programação necessária para a sincronização dos processamentos, distribuição dos dados e o uso adequado 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/Hadoop. Esse paradigma de programação, portanto, procura reduzir a complexidade do desenvolvimento de programas paralelos para grandes volumes de dados, tipicamente na escala de petabytes, provendo confiabilidade no processamento através do gerenciamento de falhas nas máquinas constituintes do sistema distribuído [9].

Um esquema de funcionamento do modelo MapReduce é apresentado na Figura 1. A ideia básica do modelo é processar paralelamente vários arquivos de entrada usando uma função *Map* e uma função *Reduce*. No exemplo da Figura 1, o objetivo é contar as palavras contidas nos documentos armazenados no sistema de arquivo distribuído HDFS (*Hadoop Distributed File System*). Inicialmente, a função *Map* recebe como entrada um 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. Cada par chave/valor é armazenado em partições nos discos locais da máquina que executa a função *Map*, onde a partição de destino é determinada por uma função particionadora. Após todos os arquivos de entrada terem sido processados paralelamente, as saídas das funções *Map* são redistribuídas pelo sistema, em um procedimento chamado *shuffle* [10].

Nesse processo, as saídas das funções *Map* com palavras ou *chaves* idênticas são agrupadas na mesma partição, sendo estas armazenadas no sistema de arquivo local das máquinas que irão executar a função *Reduce*. Ao fim da redistribuição, a fase seguinte é realizada pela execução da função *Reduce*, aplicada paralelamente em cada partição. À medida que a função é aplicada, a saída é gravada no HDFS. No exemplo da Figura 1, a função *Reduce* soma todos os valores associados a mesma chave, resultando na contagem das palavras contidas nos arquivos de entrada.

B. Cálculo de centralidade

Para este trabalho consideramos grafos direcionados $G = (V, A)$, sendo V o conjunto de vértices e A o conjunto de arestas direcionadas, onde as arestas representam alguma relação existente entre dois vértices. Grafos não direcionados podem ser representados substituindo-se as arestas não direcionadas por duas arestas direcionadas em sentidos opostos. Os grafos aqui tratados são simples, ou seja, sem arestas múltiplas, sem laços e sem pesos nas arestas.

Em diversos modelos matemáticos que utilizam grafos, a noção de *centro* de uma rede é bastante explorada, pois muitas vezes deseja-se determinar a localização de elementos no grafo que atendam algum critério de satisfação. 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 seja fixo.

Nesse trabalho em particular, estamos interessados no cálculo do *diâmetro* e *raio* de um grafo, feito a partir do cálculo de excentricidades que, por sua vez, são calculadas a partir das distâncias entre os vértices do grafo. A *distância* $d(s, t)$ de um vértice s até um vértice t é o comprimento do menor caminho entre s e t. A excentricidade de um vértice é a maior distância entre esse vértice e os demais vértices alcançáveis da rede. Assim, denotamos a excentricidade de um vértice por $e(v) = \max_{t \in V}\{d(v, t)\}$.

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, denotado por $R(G) = \min_{v \in V}\{e(v)\}$, é a menor excentricidade entre os vértices do grafo. A análise da excentricidade revela que vértices com valores de excentricidade baixos são *centrais* ao grafo, enquanto valores de excentricidade elevados indicam os vértices *periféricos* ao grafo. Nota-se que essa medida de centralidade tem custo computacional elevado, pois a quantidade de menores caminhos é estimada supe-

riormente pelo cálculo $\binom{|V|}{2} = \frac{|V|^2-|V|}{2}$ para grafos de $|V|$ vértices. Além disso, deve-se levar em consideração o custo computacional necessário para encontrar um único menor caminho.

C. Processamento paralelo de grafos usando MapReduce

Antes de processar dados no modelo MapReduce é necessário fazer uma análise da adequabilidade da aplicação alvo a esse paradigma de programação. Alguns algoritmos mais complexos não são de fácil adaptação ao modelo, enquanto outros são inerentemente paralelizáveis, com o melhor desempenho do MapReduce sendo mais evidente neste último grupo [11]. Para o caso de processamento de grafos, é possível implementar operações em grafos de forma simples e eficiente, desde que essas operações possam ser caracterizadas por permitir processamento local [12].

No trabalho realizado em [7], 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 inspiradas na dinâmica de funcionamento do modelo. A técnica *In-Mapper Combiner* consiste em pré-processar a saída da fase *Map*, reduzindo a quantidade de dados transferidos pela fase *Shuffle* e diminuindo o trabalho realizado pela fase *Reduce*. Em outras palavras, a técnica objetiva reduzir o processamento 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 uma fase e outra, 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 particionadora que aloca vértices vizinhos na mesma partição de saída da fase *Map*, pois assim qualquer processamento que envolva ambos é feito localmente, ou seja, na memória do mesmo nó computacional. Combinando essas três práticas de projeto, os autores conseguiram ganhos de desempenho de até 70% em relação a implementações do algoritmo *PageRank* [13].

D. Algoritmo HADI

O algoritmo HADI *(HAdoop DIameter and radii estimator)* [4] 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 estimar, paralelamente, a contagem do número de pares conectados que se alcançam dentro de um determinado comprimento de caminho, usando para isso o *algoritmo de Flajolet-Martin* [14]. Ao estimar a contagem de pares conectados, é possível fazer o cálculo aproximado do diâmetro do grafo. 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. No entanto, o diâmetro estimado pode diferir significativamente de seu valor real [5].

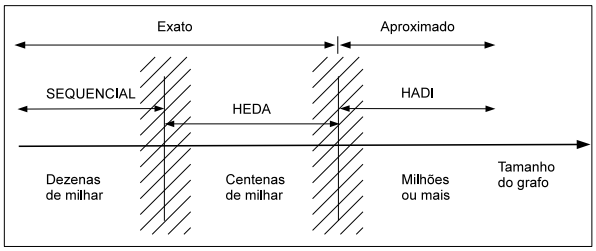


Figura 2. Faixas de atuação dos algoritmos HEDA, HADI e sequencial no cálculo de diâmetro em grafos de diversos tamanhos.

E. Algoritmo HEDA

Também desenvolvido para a plataforma MapReduce/ Hadoop, o algoritmo HEDA *(Hadoop-based Exact Diameter Algorithm)* [5] é um algoritmo baseado na expansão de fronteiras para descoberta de vértices vizinhos (busca em largura). O algoritmo faz a expansão paralelamente, calculando a distância entre um nó de origem e todos os outros de forma simultânea e exata, na medida que avança na descoberta de novos vértices.

A Figura 2 exibe as escalas de tamanhos de grafos em que os algoritmos HEDA, HADI e sequencial atuam, em seus cálculos exatos e aproximados do diâmetro do grafo. A região hachurada indica que os limites de atuação de cada algoritmo apresentam alguma flexibilidade, dependendo do grafo e dos recursos computacionais utilizados no processamento. Há um espaço para atuação do algoritmo HEDA que é explorado através do paralelismo, a fim de obter resultados exatos para grafos que são impossíveis de serem processados por soluções sequenciais.

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.

III. Descrição do algoritmo AHEAD

A. Visão Geral

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. Portanto, dado um certo número de *mappers*, que são máquinas virtuais Java instanciadas pela plataforma Hadoop, a estrutura do grafo deve ser distribuída entre os mesmos na forma de subconjuntos excludentes. Contudo, o *mapper* que executa a função *Map* em um subconjunto não compartilha os dados na sua memória com os demais *mappers* do sistema distribuído. Assim, para calcular e atualizar as distâncias entre vértices de partições ou máquinas diferentes, é necessário transmitir os resultados locais entre as partições do grafo. Para tanto os *mappers*

armazenam temporariamente os resultados intermediários no sistema de arquivo local para repassá-los às iterações seguintes do processamento.

A recepção e a síntese dos resultados parciais é feita pelos *reducers*, que assim como os *mappers*, são máquinas virtuais Java instanciadas e gerenciadas pela plataforma Hadoop. A síntese diz respeito a necessidade de, dentre as informações recebidas pelos *reducers*, selecionar aquelas que atendam algum critério de satisfação relacionado ao problema proposto. No caso do algoritmo AHEAD, o critério definido é a menor distância calculada até o momento entre dois vértices do grafo. Além disso, o *reducer* deverá acessar as partições as quais tais vértices pertencem, com o propósito de realizar a devida atualização da informação sobre a distância entre eles.

B. Funcionamento do Algoritmo

O algoritmo AHEAD utiliza as ideias expostas no trabalho realizado em [7], com a implementação das técnicas *In-Mapper Combiner* e *Schimmy*. A técnica *Range Partitioning* não é implementada por ser efetiva apenas para grafos que representam domínios da Web. Porém, a ação de particionar o grafo é de fato implementada no AHEAD, com o uso da função *Default Partitioner* do Hadoop. Essa função particiona os dados de entrada baseando-se em uma função *hash*, o que proporciona partições com tamanhos homogêneos. Partições de mesmo tamanho permitem que o processamento das mesmas termine em tempos aproximadamente iguais, evitando assim a contenção na execução paralela devido a um balanceamento ruim na distribuição dos dados. Referimemos a esse tipo de particionamento como *Hash Partitioning*.

O objetivo ao implementar as técnicas mencionadas foi otimizar o uso do espaço em disco e memória. Tal otimização possibilita o avanço nas escalas de tamanhos dos grafos que podem ser processados de forma exata, de acordo com o sentido exibido na Figura 2. O pseudocódigo do algoritmo AHEAD, mostrado na Figura 3, é composto pelas três fases listadas a seguir:

1. Linhas (1-6): Criação paralela no HDFS da matriz de distâncias do grafo. A matriz é particionada em R partições, onde R é o número de *reducers*;
2. Linhas (7-13): Cálculo paralelo das distâncias entre todos os vértices e atualização paralela no HDFS das partições matriciais representativas do grafo;
3. Linhas (14-24): 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 da matriz é ocupada pelo valor da distância entre o vértice de destino i e o vértice de origem j. A matriz é criada na fase 1 a partir da leitura de todos os registros da lista de adjacências do grafo. 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 adjacências para uma determinada partição G_i. Na linha

```
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} = ∅
```

Figura 3. Programa Principal com as três fases do algoritmo AHEAD.

(5), *reducers* recepcionam e fazem a expansão dos registros com as colunas correspondentes às distâncias para os outros vértices do grafo. A partição a qual o registro é enviado e posteriormente estendido é determinada pela função *Hash Partitioning*. Ao final dessa 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 [7].

Na linha (9) da Figura 3, *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. A operação de cálculo da distância é baseada no algoritmo de busca em largura, 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 à mesma origem. Portanto, para calcular a distância de um vértice vizinho, basta adicionar um à distância acumulada. A função *Map* que calcula os caminhos entre as diversas origens e destinos foi implementada de forma que somente os menores valores são gravados localmente antes de serem enviados aos *reducers*. Além disso, quando há caminhos diferentes entre origem e destino com valores de distâncias iguais, apenas um desses valores fica disponível, evitando redundância de informação no disco local. Essas duas técnicas de implementação da função *Map* contemplam o padrão de projeto *In-Mapper Combiner* [7].

Na linha (10), os *reducers* acessam a partição do grafo, isto é, o arquivo do grafo que receberá as atualizações das distâncias. Cada *reducer* é responsável por uma partição. A atualização das R partições ocorre simultaneamente, isto é, os *reducers* trabalham em paralelo assim como os *mappers* o fazem durante o cálculo das distâncias. Sempre que a distância de um vértice é atualizada, o cálculo da distância

de seus vértices vizinhos em relação a origem considerada é disparada na próxima iteração. A condição de parada na linha (12) é baseada nesse fato, pois o *reducer* que atualiza alguma partição também incrementa um contador global de atualizações. Uma iteração completa do laço na linha (7) corresponde a um ciclo *map-shuffle-reduce* ou um *Job*. Quando não houver atualizações durante um *Job*, a condição na linha (12) muda, encerrando o laço. Note que no laço somente as distâncias calculadas são transferidas entre *mappers* e *reducers*. Não há transferência da estrutura do grafo. Tal fato economiza o espaço usado para armazenamento local dos resultados parciais, como também economiza recursos de rede usados para a transferência dos mesmos entre as instâncias de máquinas virtuais Java.

C. Número de Mappers e Reducers

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

$$M = \frac{T_e}{B_{\text{hdfs}}}, \tag{1}$$

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 (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 *cores* disponíveis no sistema. Assim, considerando c o número de *cores* no *cluster* e $k = \frac{M}{R}$ a relação entre o número de *mappers* e *reducers* desejada, ao dividirmos os dois lados da equação (1) por R obtemos:

$$B_{\text{hdfs}} = \frac{T_e}{k \cdot c} \tag{2}$$

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. 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 *cores*/*mappers* e *cores*/*reducers* sejam 1/1.

IV. Resultados

A. Planejamento dos experimentos

O algoritmo AHEAD foi avaliado mediante o processamento de grafos reais e sintéticos. Os grafos reais foram obtidos na base de dados *Stanford Large Network Dataset Collection*[1] mantida pelo grupo de pesquisa *Stanford Network Analysis Project*. Os grafos reais utilizados são o da rede social *Epinions*, com 75.879 vértices e 508.837 arestas, o grafo da rede de comunicação por correio eletrônico de funcionários da empresa *Enron*, com 36.692 vértices e 367.662 arestas, e a rede de colaboração científica entre co-autores de artigos da categoria *Astro Physics* da revista eletrônica *arXiv*, com 18.772 vértices e 396.160 arestas.

Além desse conjunto, foram criados grafos sintéticos com número de vértices fixo em 50.000 (50K) e número de arestas variando entre 150K e 1.600K, com o objetivo de avaliar o escalonamento do AHEAD quanto à variação do número de arestas. Também foram gerados grafos sintéticos com número de vértices variando entre 25K e 200K e arestas variando entre 75K e 600K. Para a geração de grafos sintéticos foi utilizada a biblioteca *NetworkX* [15], que também foi a fonte de um dos dois algoritmos sequenciais utilizados. O segundo algoritmo sequencial testado no trabalho é citado em [16]. Os resultados de ambos são comparados com o AHEAD para verificação da correção dos resultados (raio e diâmetro dos grafos). Uma comparação entre HEDA e AHEAD é feita com o objetivo de verificar o uso de espaço em disco e memória. Os valores apresentados em gráficos e tabelas são a média dos resultados de três processamentos. Observamos que a variação dos valores em torno da média é pequena. Os dados podem ser encontrados em [17].

Os testes foram realizados em um *cluster* com 6 nós de processamento interligados via *switch* de 1 Gb/s, onde cada nós possui dois processadores Intel Xeon X5660 a 2,80 GHz, 98 GB de RAM e 300 GB de disco. Ao todo são 72 *cores* físicos ou 144 *cores* virtuais (*hyperthreading*) presentes no *cluster*. A versão Hadoop instalada é 1.1.2 e o sistema operacional é o Linux Gentoo Kernel 3.6.11. Somente durante a comparação entre os algoritmos HEDA e AHEAD, na Seção IV-F, a capacidade dos discos locais foram trocadas para o valor máximo disponível, que é de 2TB por máquina.

B. Correção do algoritmo

O AHEAD gerou resultados de raio e diâmetro iguais aos dos algoritmos sequenciais e do HEDA, para vários grafos testados. Por exemplo, para o grafo sintético com 50K vértices e 200K arestas, todos os algoritmos testados apresentaram os mesmos resultados, a saber, *Diâmetro* $= 15$ e $Raio = 11$. A Figura 4 mostra os tempos de execução dos algoritmos sequenciais (em uma máquina), e dos algoritmos HEDA e AHEAD, quando varia-se o número de máquinas. Os resultados exibidos na Figura 4 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.

C. Escalabilidade

1) Variação de arestas: O comportamento do algoritmo AHEAD frente à variação no número de arestas é exibido nas Figuras 5 e 6. Nesse experimento, o grafo tem 50K vértices e o número de arestas varia entre 200K e 1.600K. O aumento do número de arestas representa o aumento

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

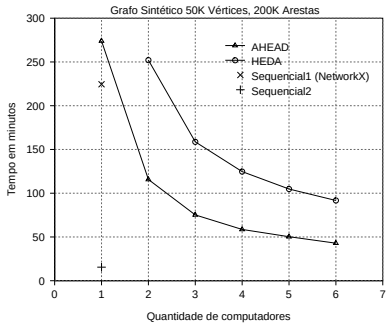


Figura 4. Comparação entre AHEAD, HEDA e algoritmos sequenciais.

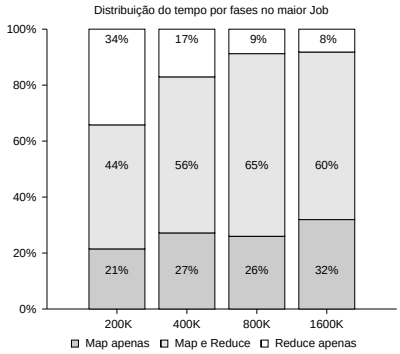


Figura 5. Tempo gasto em cada fase com o aumento do número de arestas. Os dados são coletados do *job* de maior duração.

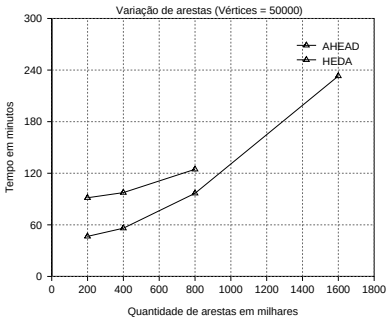


Figura 6. Variação no número de arestas em grafos de 50K vértices.

no número de relações entre uma quantidade fixa de entidades. A Figura 6 mostra que a taxa de crescimento do tempo de execução é lenta. No intervalo testado, o tempo de execução do AHEAD foi multiplicado por cerca de quatro enquanto o número de arestas foi multiplicado por oito. A Figura 5 mostra que, a partir de 400K arestas, a fase *map*, que faz a busca em largura no grafo, domina o tempo de processamento gasto no *Job* com maior tempo de duração. Para esse gráfico, o tempo de processamento da fase *reduce* inclui o tempo gasto no processo de transmissão de dados (*shuffle*). Nas Figuras 4 e 6, respectivamente, o algoritmo HEDA não processa o grafo com 200K arestas em uma máquina e 1.600K arestas em seis máquinas. Nos dois casos, o sistema acusa falta de espaço em disco no *cluster* durante o processamento.

2) Variação de vértices e arestas: A Tabela I mostra os valores de diâmetro e tempo de processamento de grafos sintéticos 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. Na prática, a inserção de vértices só tem efeito no cálculo das distâncias se os mesmos tiverem grau maior ou igual a um. Os grafos exibidos na Tabela I são conectados, o que garante a existência de no mínimo uma nova aresta a cada novo vértice. Nessa tabela, percebe-se que a variação do tempo de execução com a variação do tamanho do grafo é maior se comparada à variação mostrada na Figura 6. Ao aumentar o valor de vértices e arestas $(V + A)$, mantendo a relação arestas/vértices relativamente baixa $(A/V = 3)$, estamos aumentando a quantidade de vértices distantes entre si, o que dificulta a exploração do algoritmo sobre o grafo. Tal fato 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, resultando nos diâmetros cada vez maiores mostrados na Tabela I.

D. Speedup e Eficiência

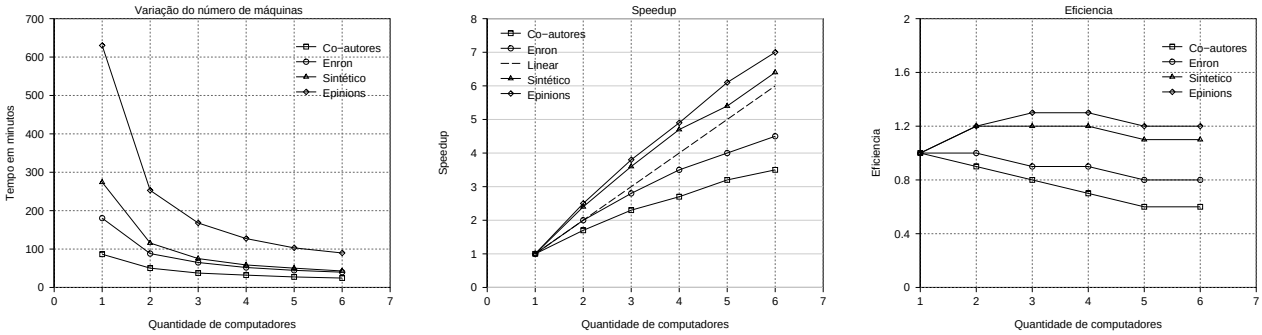
Para os testes de desempenho foram usados os grafos reais descritos na Seção IV-A, juntamente com um grafo sintético contendo 50K vértices e 200K arestas. A Figura 7 mostra os resultados de tempo de execução, *speedup* e eficiência frente à variação do número de máquinas do *cluster*. Nota-se no gráfico da Figura 7b 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 nas memórias *cache* de cada processador. Assim, as operações de memória serão executadas em cada nó computacional de maneira mais eficiente. Esse efeito também aparece no gráfico da Figura 7c, onde há valores de eficiência maiores do que 1. Para grafos menores, entendemos que tal efeito é sobreposto pelo *overhead* que o sistema apresenta, que é 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*.

E. Variação do tamanho dos blocos no HDFS

De acordo com a Equação 2 da Seção III-C, a relação entre o número de *mappers* e *reducers*, k, pode ser ajustada pela seleção adequada do valor de B_{hdfs}, que é o tamanho do bloco usado no HDFS. A Figura 8 mostra a distribuição de *mappers* e *reducers* no *cluster* para $k = 1$ e $k = 2$, em uma máquina com 23 *cores*. 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 à duração de uma tarefa, considerando as datas de início e de fim no formato *epoch* do Unix. Na Figura 8a, notamos a predominância

Tabela I
EFEITO DA VARIAÇÃO DO NÚMERO DE VÉRTICES E ARESTAS EM GRAFOS SINTÉTICOS.

Vértices / Arestas	Diâmetro	Tempo médio (minutos)	Variação tamanho	Variação tempo
25.000 / 75.000	21	19,5	-	-
50.000 / 150.000	23	47,1	2x	2,4x
100.000 / 300.000	24	163,3	4x	8,4x
200.000 / 600.000	26	1129,8	8x	57,9x



(a) Tempo total. (b) Speedup. (c) Eficiência.

Figura 7. Teste de desempenho.

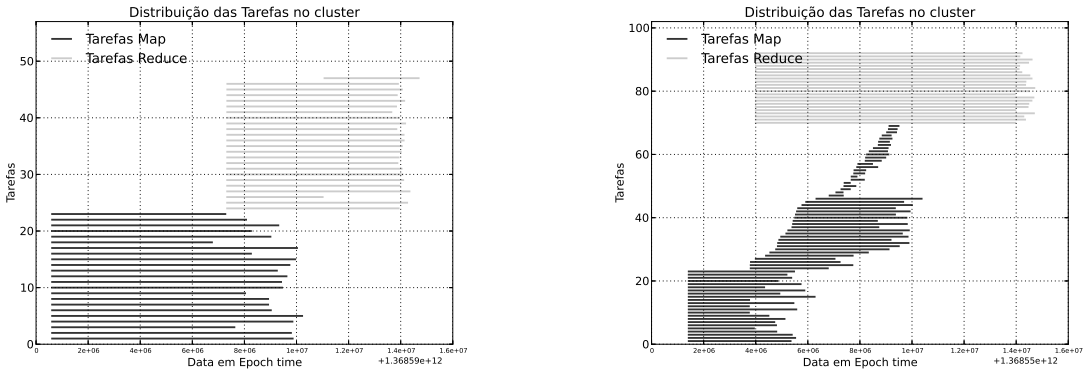
do paralelismo entre tarefas *map* e entre tarefas *reduce* de forma isolada, isto é, há pouca concorrência entre *mappers* e *reducers*. A Figura 8b mostra predominância na concorrência entre tarefas de natureza diferente. Notamos também que, para $k = 2$, o instanciamento de tarefas *map* ocorre em grupos de 23 *mappers*, que é o número de *cores*. No eixo Y, o último grupo de *mappers* é constitíudo pelas tarefas 47 a 69. Este grupo se deve ao fato de que o valor de T_e foi acima do valor estimado. Baseando-se nos experimentos realizados, notamos que os melhores tempos de execução do AHEAD, para diversos grafos, ocorrem para valores de k no intervalo $1 \leq k \leq 2$. 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.

F. Comparação HEDA e AHEAD

Ambos os algoritmos foram executados em diversos grafos usando o mesmo valor de B_{hdfs}. Um resumo das medições de uso de recursos é exibido na Tabela II. A Figura 9a, exibe os valores de uso da memória no processamento do grafo sintético de 50K vértices e 800K arestas. O valor fornecido pela plataforma Hadoop é a soma dos bytes usados por todas as tarefas instanciadas durante o *Job*. O algoritmo AHEAD reduz em 68,8% o uso de memória em relação ao algoritmo HEDA, quando somados os valores dos *Jobs*. A Figura 9b mostra o uso do espaço agregado de disco no *cluster*. Os valores medidos são referentes à quantidade de dados na saída da fase *map*, que fica armazenada nos discos locais antes da transferência (*shuffle*) para os *reducers*. O algoritmo AHEAD reduz em 34,5% o uso de espaço em disco, considerando-se a soma de dados armazenada em cada *Job*.

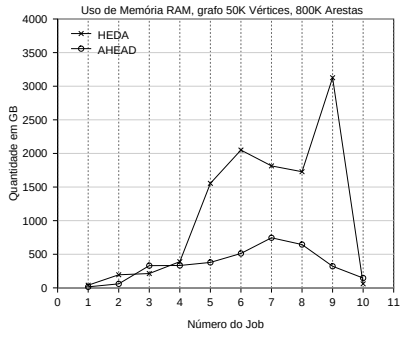
Tabela II
UTILIZAÇÃO DE RECURSOS PELOS ALGORITMOS HEDA E AHEAD

Recurso	Grafo	HEDA	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%

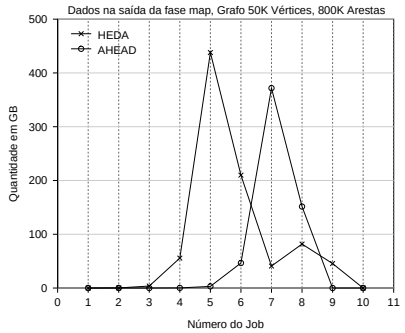


(a) (k = 1) (b) (k = 2)

Figura 8. Variação da relação entre o número de *mappers* e *reducers* no processamento do grafo *Epinions*. Para $k = 1$ o número de *mappers* é igual ao número de *reducers*. Para $k = 2$ o número de *mappers* esperado é o dobro do número de *reducers*, porém o valor de T_e foi um pouco acima do estimado, causando o instanciamento de tarefas *map* (tarefas 47 a 69) com curto tempo de processamento.



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



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

Figura 9. Comparação do uso do sistema de arquivo agregado e da memória RAM pelos algoritmos HEDA e AHEAD.

V. Conclusão e Trabalhos Futuros

Esse artigo apresentou o algoritmo paralelo AHEAD, desenvolvido para o processamento eficiente e exato de medidas de centralidades em grafos grandes. Experimentos foram feitos para demonstrar a escalabilidade, o *speedup* e a eficiência do algoritmo. Com a implementação das técnicas *In-Mapper Combiner* e *Schimmy* para algoritmos paralelos implementados no modelo MapReduce/Hadoop, foi possível reduzir em até 72,4% o uso da memória RAM e em até 64,3% o uso do espaço em disco em relação ao algoritmo HEDA. A principal contribuição desse trabalho é o projeto e a implementação de um algoritmo que possibilita o processamento de grafos maiores, a partir do uso mais eficiente dos recursos computacionais disponíveis.

Trabalhos futuros devem investigar 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 procurando minimizar o número de vértices vizinhos em partições diferentes e, ao mesmo tempo, manter o tamanho das mesmas relativamente iguais.

Agradecimentos

Os autores agradecem ao INCT InWeb, ao CNPQ e à FAPEMIG pelo apoio.

Referências

[1] R. Bryant, "Data-intensive scalable computing for scientific applications," *Computing in Science & Engineering*, vol. 13, no. 6, pp. 25–33, Nov.-Dec. 2011.

[2] L. C. Freeman, "Centrality in social networks conceptual clarification," *Social Networks*, vol. 1, no. 3, pp. 215 – 239, 1979.

[3] U. Kang, S. Papadimitriou, J. Sun, and H. Tong, "Centralities in large networks: Algorithms and observations," in *SIAM International Conference on Data Mining*. SIAM / Omnipress, Apr 2011b, pp. 119–130.

[4] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec, "Hadi: Mining radii of large graphs," *ACM Trans. Knowl. Discov. Data*, vol. 5, no. 2, pp. 8:1–8:24, Feb 2011a.

[5] J. P. B. Nascimento and C. D. Murta, "Um Algoritmo Paralelo para Cálculo de Centralidade em Grafos Grandes," in *Anais do XXX SBRC*, 2012, pp. 393–406.

[6] J. P. B. Nascimento, "Um Algoritmo Paralelo para Cálculo de Centralidade em Grafos Grandes," Master's thesis, PPGMMC, CEFET-MG, 2011.

[7] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in MapReduce," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, ser. MLG '10. ACM, 2010, pp. 78–85.

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *USENIX Association Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDE'04)*. USENIX ASSOC, 2004, pp. 137–149.

[9] ——, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan 2008.

[10] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for MapReduce," in *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA'10. Society for Industrial and Applied Mathematics, 2010, pp. 938–948.

[11] S. N. Srirama, P. Jakovits, and E. Vainikko, "Adapting scientific computing problems to clouds using MapReduce," *Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 184–192, Jan 2012.

[12] J. Cohen, "Graph Twiddling in a MapReduce World," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, Jul-Aug 2009.

[13] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999.

[14] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 182–209, Sep. 1985.

[15] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Aug. 2008, pp. 11–15.

[16] M. R. S. Gonçalves, J. N. Maciel, and C. D. Murta, "Geração de Topologias da Internet por Redução do Grafo Original." in *Anais do XXVIII SBRC*, 2010, pp. 959–972.

[17] L. C. da Cruz, "Um Algoritmo Paralelo Eficiente para Cálculo de Centralidade em Grafos," Master's thesis, PPGMMC, CEFET-MG, 2013.