Um Algoritmo Paralelo em Hadoop para Cálculo de Centralidade em Grafos Grandes

João Paulo B. Nascimento[1], **Cristina D. Murta**[1]

[1]Departamento de Computação – CEFET-MG
Belo Horizonte – MG

Resumo. *Neste artigo apresentamos um algoritmo paralelo para cálculo de métricas de centralidade em redes complexas de grande porte, modeladas por grafos. O algoritmo é implementado em MapReduce/Hadoop. Os resultados calculados para as métricas diâmetro e raio são exatos, enquanto os algoritmos da literatura provêem valores estimados para as mesmas métricas. Para avaliar o algoritmo, experimentos foram executados em vários grafos e comparados com resultados providos por outros algoritmos. Os resultados indicam que o algoritmo apresenta valores de speedup próximos à linearidade, boa eficiência e escalabilidade.*

Abstract. *In this paper we present a parallel algorithm for centrality metrics in large complex networks modeled by graphs. The algorithm is implemented in MapReduce/Hadoop. The algorithm finds exact values for network metrics diameter and radius, while the algorithms in the literature provide estimated values for the same metrics. To evaluate the algorithm, experiments were executed in several graphs and compared to results provided by other algorithms. The results indicate that the algorithm presents speedup close to linearity, high efficiency and scalability.*

1. Introdução

Muitos sistemas presentes em nosso cotidiano são parte de redes grandes e dinâmicas, atualmente denominadas redes complexas. Estas redes estão no núcleo da revolução que estamos experimentando e estudá-las tornou-se imperativo. Nesses estudos, as redes são modeladas como grafos e suas características são medidas. Dentre as métricas fundamentais estão as distâncias entre todos os pares de vértices e métricas derivadas, tais como as excentricidades dos vértices e o raio e diâmetro do grafo, que são medidas de centralidade. Processar esses grafos é uma tarefa que demanda elevada capacidade de processamento e requer espaço considerável, o que pode tornar a solução desses problemas, por meio da computação sequencial, inviável, dependendo do tamanho da rede.

A computação paralela ressurgiu nos últimos anos, com parâmetros de disponibilidade e custo sem precedentes. A possibilidade de reunir e coordenar recursos computacionais para processamento paralelo massivo apresenta-se como uma opção para tratar quantidades cada vez maiores de dados. O agrupamento dos recursos requer uma camada de software para gerenciá-los e para tratar problemas que não ocorrem ou que são menos importantes em programação sequencial, por exemplo, conflitos e tolerância a falhas. Além disso, é necessário um modelo de programação paralela. Para atender a estas necessidades foi proposto o modelo MapReduce [Dean and Ghemawat 2008], que ganhou uma implementação de código aberto, o Hadoop [White 2009]. Os programas projetados

nesse modelo são inerentemente paralelos, permitindo o tratamento e o processamento de grandes massas de dados de forma paralela e distribuída, por desenvolvedores que não têm intimidade com programação paralela [Ranger et al. 2007, Dean and Ghemawat 2008].

Neste trabalho, procuramos atender à necessidade de processamento de redes complexas cada vez maiores por meio da programação paralela, usando computadores conectados em rede, para prover paralelismo no processamento. Assim, propomos um algoritmo paralelo denominado HEDA (*Hadoop-based Exact Diameter Algorithm*), que é projetado de acordo com o modelo MapReduce e implementado em Hadoop. O HEDA é baseado no algoritmo de busca em largura e realiza paralelamente o cálculo do menor caminho entre todos os pares de vértices.

O projeto de algoritmos paralelos em grafos no modelo MapReduce ainda é pouco discutido na literatura [Lin and Schatz 2010]. Encontramos apenas um algoritmo projetado de acordo com o paradigma MapReduce/Hadoop para medir a centralidade de grafos. Esse algoritmo é o HADI (HAdoop DIameter and radii estimator) [Kang et al. 2011], que produz estimativas para o raio e o diâmetro de uma rede complexa de grande porte. O HADI não faz cálculos exatos destas medidas. De fato, ele privilegia o tempo de execução, o que resulta em um algoritmo escalável e de bom desempenho. Contudo, em contrapartida, ele provê somente estimativas para as métricas. Diante da inexistência de um algoritmo para cálculo exato de centralidades de grafos grandes, que execute no ambiente paralelo MapReduce/Hadoop, criamos o HEDA.

A principal contribuição desse trabalho é a proposta de um algoritmo paralelo, implementado no paradigma MapReduce/Hadoop, que faz o cálculo exato de métricas de centralidade em redes complexas. Além disso, apresentamos um estudo experimental desse algoritmo, comparado-o com o HADI e com um algoritmo sequencial. Experimentos foram feitos com vários grafos de redes complexas de diversos tamanhos. Foram testados ajustes finos de desempenho em parâmetros do Hadoop. Os resultados foram validados e medidas de tempo de execução, escalabilidade e *speedup* são discutidas.

2. Contexto e Trabalhos Relacionados

Nas últimas décadas, a tecnologia do silício permitiu duplicar a capacidade dos processadores a cada dezoito meses. No entanto, depois de alcançar poucas unidades de GHz, não foi possível aumentar ainda mais as frequências de operação, devido principalmente a problemas de superaquecimento, cujas soluções se mostraram muito difíceis e caras [Asanovic et al. 2009]. Assim, surgiu a arquitetura *multi-core*, que consiste em inserir múltiplos núcleos de processamento no mesmo *chip* de um processador. Esta foi a opção da indústria de hardware para dar continuidade à melhoria de desempenho nos computadores [Breshears 2009].

Esse contexto revelou um novo problema, que é o desenvolvimento de software paralelo e concorrente. A programação paralela é mais complexa do que a programação sequencial e requer atenção para uma série de novos problemas tais como divisão de tarefas, balanceamento de carga, sincronização e tolerância a falhas. Além disso, os programas devem apresentar bom desempenho e escalabilidade à medida que aumentamos as unidades de processamento [Breshears 2009, Asanovic et al. 2009, Dean and Ghemawat 2010].

Muitos modelos e ambientes para programação paralela são conhecidos, por

exemplo, o MPI. No entanto, o processamento em larga escala nesses modelos pode ser comprometido devido a problemas de distribuição dos dados, tratamento de falhas e balanceamento de carga. MapReduce é um novo modelo de programação paralela projetado para tratar estes problemas sem intervenção do programador [Dean and Ghemawat 2008]. O MapReduce é inspirado em linguagens funcionais e projetado para processamento paralelo e distribuído de massas de dados da ordem de petabytes.

Nesse modelo, o processamento é dividido em duas fases, Map e Reduce, que são funções definidas pelo programador. A função Map é aplicada aos dados de entrada e produz uma lista de pares intermediários $< chave, valor >$. A função Reduce é aplicada a conjuntos de pares intermediários que têm a mesma chave. Tipicamente, sua tarefa é agrupar dados rotulados com uma mesma chave e produzir os pares de saída. Finalmente, os pares de saída são ordenados com base em suas chaves. Um esquema desse modelo é apresentado na Figura 1. O HDFS (*Hadoop Distributed File System*) é responsável por coordenar o acesso aos dados pelos nós no *cluster*, replicando-os e distribuindo-os de maneira confiável.

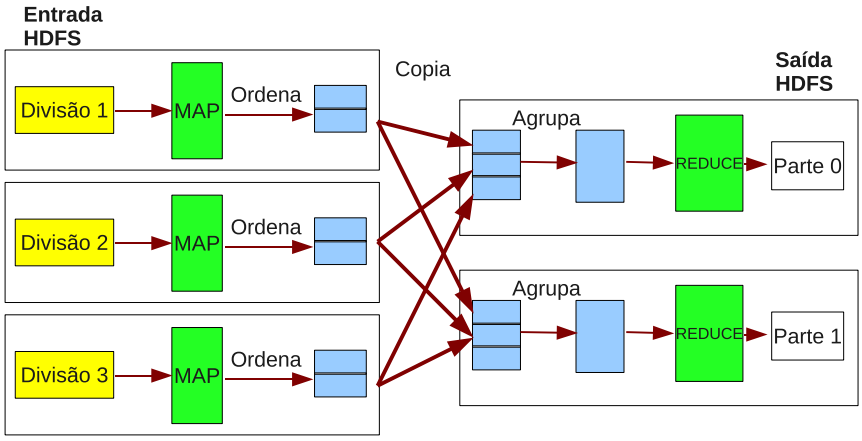

Figura 1. Funcionamento do modelo MapReduce, baseado em [White 2009].

Redes complexas são sistemas físicos, biológicos ou sociais caracterizados por um conjunto grande de entidades bem definidas que interagem dinamicamente entre si. Alguns exemplos destes sistemas são a Internet, redes sociais reais ou virtuais, redes de distribuição (energia, telefone, transporte), redes biológicas e muitas outras [Newman 2003]. Redes complexas são modeladas por grafos. Formalmente, um grafo G é um par (V, A), em que V é um conjunto finito de vértices e A é um conjunto finito de arestas composto por relações binárias em V. A topologia das redes é representada em seus respectivos grafos, que encerram em sua estrutura as informações de conectividade da rede. Considerando o crescimento acelerado de redes dos mais diversos tipos, grafos cada vez maiores precisam ser processados. Pode-se encontrar redes com milhões e até bilhões de vértices e arestas.

Medidas de centralidade de uma rede estão entre as suas principais características [Gross and Yellen 1999, Newman 2003]. Várias métricas são definidas a partir de medidas de distância entre dois vértices [Buckley and Harary 1990]. A distância é o

comprimento, em número de arestas, do menor caminho entre dois vértices da rede. Assim, inicialmente calcula-se a distância entre todos os pares de vértices da rede. A partir desse cálculo, para cada vértice, calcula-se a sua excentricidade.

A excentricidade de um vértice v em um grafo G, denotada por $e(v)$, é a maior distância entre v e qualquer outro vértice de G, pelo menor caminho [Buckley and Harary 1990, Gross and Yellen 1999]. Métricas de centralidade podem ser definidas a partir da excentricidade dos vértices de um grafo. O raio $R(G)$ de um grafo G corresponde à excentricidade mínima de seus vértices: $R(G) = min\{e(v)|v \in V\}$. O diâmetro $D(G)$ de um grafo G corresponde à excentricidade máxima de seus vértices: $D(G) = max\{e(v)|v \in V\}$. Além disso, um nodo é central se $e(v) = R(G)$. O centro de G é o conjunto de vértices de menor excentricidade. Um nodo é periférico se $e(v) = D(G)$. De forma análoga ao centro de G, a periferia de G é o conjunto de vértices do grafo com a maior excentricidade.

O projeto de algoritmos em grafos no modelo MapReduce está em fase inicial. Os artigos são poucos e recentes, e algumas propostas são encontradas em fontes informais da Web [Lin and Schatz 2010, Cohen 2009]. Uma discussão abrangente é apresentada em [Lin and Schatz 2010], que propõe três padrões de projeto para esses algoritmos: *In-Mapper Combining*, *Schimmy* e *Range Partitioner*. Esses padrões abordam aspectos de particionamento, serialização e distribuição do grafo, que podem se tornar fatores limitantes do desempenho do algoritmo. Experimentos mostraram ganhos consideráveis na aplicação de cada uma das técnicas e melhorias de até 70% no tempo de execução do algoritmo quando as técnicas são aplicadas em conjunto. Grafos com até 1,4 bilhão de vértices foram usados nos testes.

A aplicação do modelo MapReduce no projeto de algoritmos de grafos é também tratada em [Cohen 2009]. Ao longo do trabalho, o autor discute o projeto e a implementação de algoritmos para grafos em MapReduce, analisando sua complexidade. O autor argumenta que, mesmo que sejam feitas melhorias na complexidade dos algoritmos em grafos, em algum ponto a acomodação do grafo em memória se tornará impraticável e proibitivamente cara, e propõe que o grafo seja armazenado e processado em sistemas remotos (*cloud computing*).

Pregel é outro modelo de computação para o processamento distribuído e paralelo de grandes grafos [Malewicz et al. 2010]. Segundo os autores, esse modelo é escalável, tolerante a falhas e possui uma expressiva e flexível API. O modelo Pregel é baseado no modelo *Valiant's Bulk Synchronous Parallel* e, diferentemente do modelo MapReduce, o Pregel mantém os vértices e arestas do grafo na máquina que executa o processamento e usa a rede apenas para trafegar mensagens.

O algoritmo HADI [Kang et al. 2011] foi projetado para processar o diâmetro e o raio de grafos na escala de Tera e Petabytes. Segundo os autores, o HADI é escalável e o tempo de execução é linear em relação ao número de arestas processado. O HADI é um algoritmo de aproximação que trabalha com o conceito de diâmetro efetivo, definido como o número mínimo de saltos em que 90% de todos os pares de nós conectados podem alcançar uns aos outros. Com isso, o HADI encontra valores estimados para o raio e diâmetro de um grafo grande. Pegasus é um conjunto de algoritmos paralelos para grafos baseado no modelo MapReduce, que inclui o algoritmo HADI [Kang et al. 2009]

e implementa algoritmos para realizar mineração em grafos.

Este artigo se diferencia dos trabalhos anteriores citados pois propõe um novo algoritmo para calcular métricas de centralidade em grafos grandes projetado no modelo MapReduce. Esse novo algoritmo calcula os menores caminhos a partir de todos os vértices para todos os outros vértices do grafo, as excentricidades dos vértices, o raio, diâmetro, periferia e centro de um grafo grande. Todos esses cálculos são exatos. Não foi encontrado na literatura nenhum algoritmo em MapReduce/Hadoop que provê resultados exatos para essas métricas.

3. Descrição do algoritmo HEDA

O algoritmo proposto nesse trabalho, denominado HEDA (*Hadoop-based Exact Diameter Algorithm*), é apresentado nessa seção. O HEDA é um algoritmo paralelo que faz o cálculo exato dos menores caminhos entre todos os pares de vértices, das excentricidades de cada vértice, e do diâmetro, raio, periferia e centro do grafo. O HEDA foi inspirado em uma proposta de um algoritmo de busca em largura a partir de um único vértice fonte, apresentada em [Google Developers 2007], e considera as recomendações da literatura [Cohen 2009, Lin and Schatz 2010].

O algoritmo é dividido em duas fases. A primeira fase tem como objetivo calcular o menor caminho entre todos os pares de vértices. Para isso, são implementadas uma função Map e uma função Reduce. Na segunda fase do algoritmo são calculadas as medidas de excentricidade dos vértices, o diâmetro e o raio do grafo, e as demais medidas de centralidade. Essa fase é também composta por uma função Map e uma função Reduce.

Algoritmo 1: Algoritmo HEDA - Fluxo Principal

Entrada: Um arquivo contendo as arestas *arquivoArestas*, um arquivo contendo as distâncias *arquivoDistancias* e o caminho do arquivo de saída *caminhoSaída*

Saída: A centralidade do grafo e os menores caminhos entre todos os pares de vértices

```
1 iteracao ← 1; vetorArestas ← arquivoArestas;
2 Enquanto PossuiVerticesAProcessar() faça
3     caminhoSaída ← caminhoSaída + iteracao;
4     EncontrarMenorCaminho(vetorArestas, arquivoDistancias, caminhoSaída);
5     iteracao ← iteracao + 1;
6 fimEnquanto
7 EncontrarCentralidade();
```

O Algoritmo 1 apresenta o fluxo principal do algoritmo HEDA, que recebe como parâmetros de entrada o endereço do arquivo de arestas, o arquivo de distâncias e o endereço no HDFS para gravação dos dados de saída. Inicialmente, o arquivo de arestas é carregado em um vetor, cujo índice i indica o vértice de origem da aresta e o conteúdo da posição i do vetor armazena uma lista dos vértices de destino da aresta. A seguir, o algoritmo realiza um laço para processar todos os vértices do grafo. Ao finalizar esse laço, será chamada uma função para o cálculo das métricas de centralidade.

O Algoritmo 2 descreve o método Map da função que encontra os menores caminhos entre pares de vértices do grafo. A função Map recebe como parâmetros de entrada

um vetor com todas as arestas do grafo, o arquivo de distâncias e um diretório do HDFS no qual os dados de saída serão gravados. A função Map é executada para cada linha do arquivo de distâncias.

Antes de entrar no método Map, o método Inicializa é executado. Nessa função, na linha 3, o HEDA atribui a um vetor global da classe Mapper chamado ARESTAS, o vetor de arestas passado como parâmetro e que não é visível para toda a classe Mapper. Dessa forma, o vetor de arestas poderá ser acessado tanto no método Inicializa quanto no método Map. Em seguida, na linha 4, é criado um segundo vetor de tipo booleano que controla se o vértice, representado no índice i do vetor, já foi processado. Esse vetor implementa o padrão de projeto *In-Mapper Combining* para algoritmos em grafos utilizando MapReduce, apresentado no trabalho [Lin and Schatz 2010]. Na linha 6 é criada uma instância de um objeto do tipo Node, que representa um vértice. Durante a criação do objeto, o estado do vértice é atribuído de acordo com o seu estado na iteração anterior do algoritmo. Caso o vértice já tenha sido processado, após o registro, ele é enviado diretamente para a fase Reduce, por meio do comando SAÍDA (*output.collect* do Hadoop).

Algoritmo 2: Algoritmo HEDA - Função *EncontrarMenorCaminho* (MAP)

Entrada: Arquivo de arestas, Arquivo de distâncias, Caminho do arquivo de saida
Saída: Uma nova expansão da busca pelo menor caminho entre todos os pares de vértices

```
1  classe MAPPER
2      método INICIALIZA (Vetor arestas)
3          ARESTAS ← novo Vetor(arestas);
4          VISITADOS ← VETORASSOCIATIVO;
5      método MAP (chave, valor)
6          Node n1 ← novo Node( id chave, valor, ARESTAS[chave] );
7          Se n1.estado = 2 então
8              VISITADOS[n1.verticeDestino] ← verdadeiro;
9              SAÍDA( n1.verticeDestino, n1.dados )
10         Senão
11             Para todos ( id a ∈ n1.listaArestas ) faça
12                 Se a = n1.verticeDestino ou VISITADOS[ a ] = verdadeiro então
13                     continue;
14                 Node n2 ← novo Node(a);
15                 n2.atribuiDistancia( n1.distancia + 1);
16                 n2.atribuiEstado(1);
17                 SAÍDA( n2.verticeDestino, n2.dados );
18                 VISITADOS[ a ] ← verdadeiro;
19             fimPara
20             n1.atribuiEstado(2);
21         fimSe
22         VISITADOS[ n1.verticeDestino ] ← verdadeiro;
23         SAÍDA( n1.verticeDestino, n1.dados);
```

Para calcular a distância de um vértice fonte *i* para um vértice destino *j*, a propriedade *verticeFonte* do objeto Node armazena o identificador do vértice *i* e a propriedade *verticeDestino* armazena o identificador do vértice *j*. A propriedade *distancia*

armazena a distância acumulada do vértice fonte até o vértice destino e o construtor do objeto atribui valores iniciais para as propriedades *verticeFonte*, *verticeDestino*, *distancia* e *estado*.

Caso o vértice esteja em processamento, o HEDA recupera todas as arestas do vértice destino na linha 11 e inicia um laço percorrendo aresta por aresta, até que o processamento seja finalizado. Após essa etapa, o vértice é marcado no vetor de visitados e enviado para a fase Reduce, na linha 23. Ao final da execução do Algoritmo 2, os dados processados são enviados para a fase Reduce.

O Algoritmo 3 descreve o método Reduce da função *EncontrarMenorCaminho*. O Hadoop agrupa automaticamente os valores de mesma chave em uma lista armazenada no parâmetro *P*. Na linha 4 é criado um laço para percorrer todos os vértices de mesma chave que, ao final, encontra a maior distância e instancia um novo objeto para armazenar esses valores (linha 10). Em seguida, os dados são gravados no HDFS (linha 13).

Algoritmo 3: Algoritmo HEDA - Função *EncontrarMenorCaminho* (REDUCE)

```
   Entrada: Dados de saída da função Map
   Saída: Distâncias calculadas e gravadas no HDFS
1  classe REDUCER
2      método REDUCE(id chave, P[p1, p2, ..., pn] )
3          distancia ← INT-MAX; estado ← 0;
4          Enquanto (existirem valores em P) faça
5              valor ← P.próximo();
6              Node n1 ← novo Node(chave, valor);
7              Se n1.distancia < distancia então distancia ← n1.distancia;
8              Se n1.estado > estado então estado ← n1.estado;
9          fimEnquanto
10         Node n2 ← novo Node(chave);
11         n2.atribuiDistancia(distancia);
12         n2.atribuiEstado(estado);
13         SAÍDA(n2.id, n2.dados);
```

A função Map da tarefa de calcular a centralidade, apresentada no Algoritmo 4, tem um trabalho bem simples em comparação aos demais algoritmos do HEDA. Sua tarefa é buscar no HDFS os dados da última iteração da fase *EncontrarMenorCaminho* e repassá-los para a função Reduce da fase *EncontrarCentralidade*, transformando as linhas que são buscadas no arquivo em pares $< chave, valor >$. A fase Reduce é apresentada no Algoritmo 5.

Algoritmo 4: Algoritmo HEDA - Função *EncontrarCentralidade* (MAP)

```
1  classe MAPPER
2      método MAP (chave, valor)
3          chaveTransformada ← Copia posições anteriores ao TAB;
4          valorTransformado ← Copia posições posteriores ao TAB;
5          SAÍDA(chaveTransformada, valorTransformado);
```

A função Reduce calcula a excentricidade de cada vértice. Para isso, é criado um

laço que percorrerá todos os valores de uma determinada chave, verificando se o valor corrente é maior que a maior excentricidade já encontrada (linha 8). Nas linhas 10 e 11 são calculados os valores para o diâmetro e o raio da rede. Para cada conjunto de chaves processado é gravado no HDFS o identificador do vértice fonte, o vértice destino e o valor da excentricidade. Ao final, os valores de diâmetro e raio do grafo são gravados em arquivo. Mais detalhes do algoritmo podem ser encontrados em [Nascimento 2011].

Algoritmo 5: Algoritmo HEDA - Função *EncontrarCentralidade* (REDUCE)

```
   Entrada: Todas as excentricidades dos vértices calculadas
   Saída: As medidas de raio e diâmetro do grafo
1  classe REDUCER
2      método INICIALIZA()
3          diametro ← 0; raio ← 2048;
4      método REDUCE (id chave, P[p1, p2, ..., pn)
5          excentricidade ← 0;
6          Enquanto (existirem valores em P ) faça
7              valor ← P.próximo();
8              Se valor > excentricidade então excentricidade ← valor;
9          fimEnquanto
10         Se excentricidade > diametro então diametro ← excentricidade;
11         Se excentricidade < raio então raio ← excentricidade;
12         SAÍDA(chave, excentricidade);
13     método FINALIZA()
14         SAÍDA("D(G) = ", diametro);
15         SAÍDA("R(G) = ", raio);
```

4. Planejamento dos Experimentos

Esta seção apresenta e discute a metodologia e o planejamento dos experimentos para avaliação do algoritmo proposto nesse trabalho. Para a realização dos experimentos foi utilizado um *cluster* composto por seis máquinas, cada uma com dois processadores *QuadCore* Intel Xeon E5620 2.4GHz totalizando 8 núcleos de processamento por máquina e 48 núcleos no *cluster*. Cada máquina tem 32 GB de RAM, 320 GB em disco e está conectada em um *switch* Gigabit. O sistema operacional utilizado é o *Linux Ubuntu Server* 64-Bit versão 10.04. A *JVM (Java Virtual Machine)* é a *OpenJDK* 64-Bit *Server* 1.6.0.20 (*build* 19.0-b09, *mixed mode*) e a versão do Hadoop é a 0.20.2.

Para a avaliação do algoritmo HEDA foram utilizados grafos de redes complexas reais e grafos sintéticos. Os grafos reais são da topologia dos sistemas autônomos da Internet, coletados pelo *Internet Research Lab*[1] da Universidade da Califórnia (UCLA) e o grafo IMDB[2] (*Internet Movie Database*) que relaciona informações sobre filmes e atores. Para os experimentos foram utilizados dois grafos dos sistemas autônomos da Internet. O primeiro grafo, de 01/01/2010, possui 42.089 vértices e 570.570 arestas. O segundo, do dia 01/07/2011, possui 47.224 vértices e 718.768 arestas. O grafo IMDB possui 757.395 vértices e 4.539.634 arestas. Esse grafo foi escolhido por fazer parte do trabalho HADI e por ter um número elevado de vértices e arestas.

[1]http://irl.cs.ucla.edu/index.html
[2]http://www.imdb.com

Geradores de grafos sintéticos foram utilizados por permitirem a construção de grafos com valores específicos de vértices e arestas, o que possibilita análises de escalabilidade. Assim, podemos analisar o comportamento de tempo e espaço do algoritmo à medida que aumentamos a quantidade de vértices e arestas. Grafos sintéticos foram gerados aleatoriamente utilizando o mesmo gerador do trabalho [Kang et al. 2011], baseado no modelo de Erdös-Rényi. Foram produzidos dois conjuntos de grafos sintéticos, o primeiro com quantidade fixa de vértices (50.000) e número de arestas variando de 100.000 a 1.000.000. O segundo conjunto tem quantidade de vértices e arestas variáveis, sendo que os vértices variam de 25.000 a 75.000 vértices e as arestas variam de 200.000 a 600.000.

Inicialmente, os resultados do algoritmo HEDA foram comparados, para grafos pequenos, com os resultados de um algoritmo sequencial [Gonçalves et al. 2010], com o objetivo de validação dos resultados do algoritmo. Os tempos de execução também foram comparados. Em seguida, os algoritmos HEDA e HADI são comparados quanto aos resultados e ao tempo de execução. A avaliação do impacto da alteração dos parâmetros do Hadoop no tempo de execução foi feita executando-se o algoritmo HEDA com e sem alteração nesses parâmetros, tendo como entrada os grafos da Internet. Os resultados foram comparados. Todos os experimentos desse trabalho foram executados três vezes e os resultados apresentados correspondem à média dos tempos de execução, sendo muito pequenos os valores dos desvios padrão. Devido a restrições de espaço, não são apresentados os valores do desvio padrão, que podem ser encontrados em [Nascimento 2011].

5. Resultados

Nessa seção são apresentados os resultados da execução do algoritmo HEDA, tendo como entrada os grafos descritos. Os resultados analisados são quanto ao tempo de execução em função do número de máquinas no *cluster* e em função do tamanho do grafo. Além disso, são analisadas a escalabilidade do algoritmo, o *speedup* e a eficiência. Comparações com os algoritmos HADI e sequencial são também apresentadas e discutidas, assim como resultados do ajuste de desempenho do Hadoop.

5.1. Resultados do Algoritmo HEDA

Esta seção apresenta os resultados do algoritmo HEDA obtidos no processamento de cada um dos grafos propostos. Os gráficos da Figura 2 apresentam resultados de tempo de execução para os grafos dos sistemas autônomos da Internet em função do número de máquinas agregadas ao *cluster*. Observamos que o tempo de execução vai diminuindo à medida que o número de máquinas aumenta, conforme esperado.

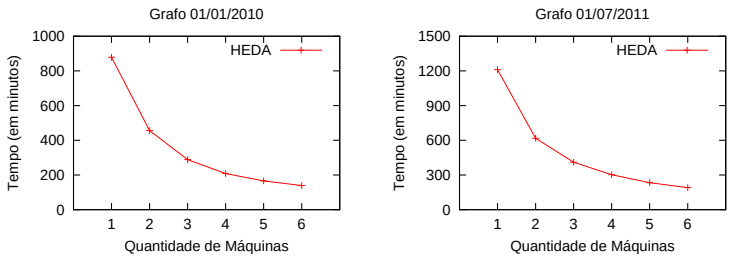

Figura 2. Resultados do HEDA para os grafos da Internet

Os resultados para os grafos sintéticos são apresentados na Figura 3. O gráfico à esquerda dessa figura apresenta o tempo de execução do algoritmo em função do aumento do número de arestas, para um grafo sintético com número de vértices fixo e igual a 50.000. Observamos que o algoritmo HEDA é escalável em relação à quantidade de arestas. Ao duplicarmos o número de arestas, de 100 mil para 200 mil, o HEDA apresentou tempo de execução apenas 0,13 vezes maior. Quando aumentamos o número de arestas de 100.000 para 1.000.000 arestas, ou seja, dez vezes mais, o tempo de execução foi 2,39 vezes maior.

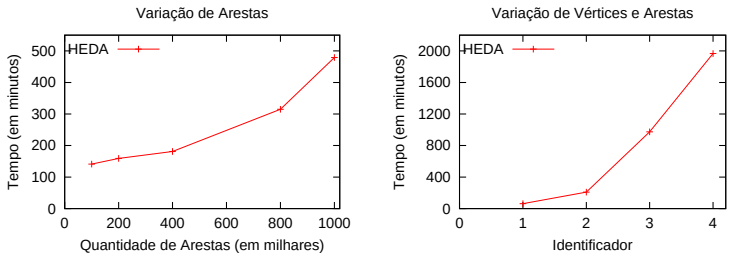

Figura 3. Resultados HEDA para grafos Sintéticos

O gráfico à direita da Figura 3 apresenta resultados para grafos sintéticos em que o número de vértices e o número de arestas variam. As instâncias testadas e o tempo de execução são apresentados na Tabela 1. Uma análise da tabela indica que, ao dobramos o tamanho do grafo, da instância 1 para a 2, por exemplo, o tempo de execução aumenta 2,37 vezes. Comparando as instâncias 3 e 4, com o mesmo aumento proporcional, observamos que tempo de execução aumentou 2,02 vezes. Entendemos que esse aumento menor no tempo de execução para instâncias maiores deve-se ao *overhead* do Hadoop, que é compensado no processamento de quantidades maiores de dados.

Grafo	V + A	Tempo (minutos)
1	100.000	62
2	200.000	209
3	400.000	974
4	800.000	1967

Tabela 1. Resultados HEDA: grafos sintéticos testados

Os resultados para o grafo IMDB são apresentados na Tabela 2. Para mostrar sua escalabilidade, executamos o algoritmo tendo como ponto de partida um número crescente de vértices. A tabela exibe a quantidade de vértices de partida e o tempo de execução em cada caso. Uma análise dos tempos apresenta evidência da escalabilidade do algoritmo quanto a sua principal tarefa: o cálculo das distâncias entre os vértices. Observamos que, para até 100 vértices iniciais, o tempo aumenta pouco, devido ao *overhead* do Hadoop. No entanto, a partir de 100 vértices iniciais, observamos que o tempo é aproximadamente proporcional ao aumento do número de vértices de partida.

A medida do *speedup* do algoritmo HEDA foi realizada para quatro grafos, apresentados na Tabela 3. Para esse cálculo, cada grafo foi processado três vezes em cada conjunto de máquinas (1 a 6) disponível no ambiente de teste.

Vértices de Partida	Tempo (em minutos)
1	25
10	26
100	32
1.000	122
10.000	1352

Tabela 2. Resultados HEDA: resultados para o grafo IMDB

Identificador	Tipo de Grafo	Vértices	Arestas
Int1	Internet	42.089	570.570
Int2	Internet	47.224	718.768
Sint1	Sintético	50.000	800.000
Sint2	Sintético	100.000	300.000

Tabela 3. Grafos utilizados no teste de *speedup*

A Figura 4 apresenta os resultados de *speedup* e eficiência do algoritmo HEDA para os conjuntos de dados testados. Observamos que o algoritmo HEDA apresenta *speedup* superlinear para os grafos da Internet (Int1 e Int2). Para os grafos sintéticos, o algoritmo apresentou valores de *speedup* próximos à linearidade. Os valores de eficiência ficaram entre $1,06$, para o grafo Int1, e $0,76$ para o grafo Sint1.

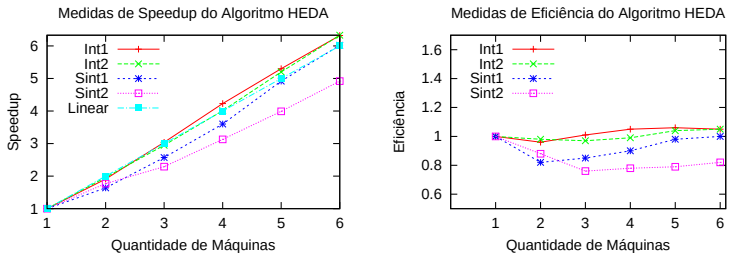

Figura 4. Resultados do HEDA: *speedup* e eficiência

O diâmetro do grafo tem influência direta no tempo de execução do algoritmo HEDA. Esse resultado pode ser avaliado nos dados apresentados na Tabela 4, para grafos sintéticos. O grafo que apresenta menor diâmetro apresenta também o menor tempo de execução, mesmo tendo 33,34% mais arestas. Isso ocorre devido ao fato de que o algoritmo HEDA executa em fases e o número de fases é igual ao valor do diâmetro do grafo.

5.2. Comparação dos algoritmos HEDA e HADI

Esta seção apresenta uma comparação dos algoritmos HEDA e HADI. O HADI foi escolhido por ser um algoritmo paralelo em Hadoop para processar grafos. No entanto, o HADI não faz o cálculo exato do diâmetro do grafo e sim um cálculo aproximado. Como resultado dessa abordagem de aproximação, seus tempos de resposta são bem inferiores aos tempos de resposta do HEDA, conforme pode ser visto nos gráficos da Figura 5, em que os dois algoritmos foram executados para as duas instâncias de grafos da Internet, em função da quantidade de máquinas. Para esses grafos, em relação ao diâmetro, o algoritmo HADI apresentou uma variação de 30% a menos no resultado de diâmetro e uma variação de 42,86% a menos nos resultados de raio.

Vértices	Arestas	Tempo (minutos)	Diâmetro
50.000	150.000	209	23
50.000	200.000	159	16

Tabela 4. Análise de diâmetros e tempos de execução

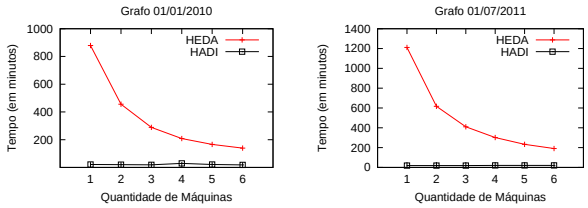

Figura 5. Comparação dos algoritmos HEDA e HADI para os grafos da Internet

5.3. Comparação entre o HEDA e o Algoritmo Sequencial

Para a realização da comparação dos resultados dos experimentos entre o algoritmo HEDA e o algoritmo sequencial foram utilizados os dois grafos da Internet e dois conjuntos de grafos sintéticos. Os resultados são apresentados nos gráficos da Figura 6, em função do número de máquinas. O tempo do algoritmo sequencial em cada caso é dado por um ponto em cada gráfico.

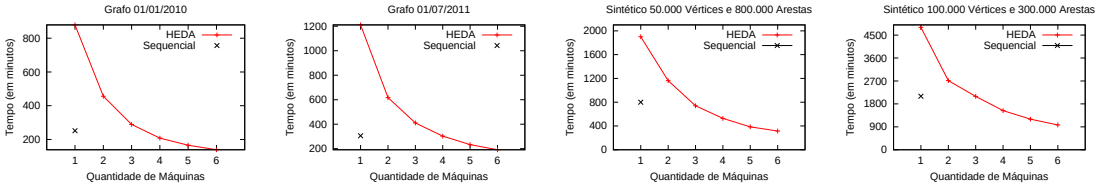

Figura 6. Comparação dos algoritmos HEDA e sequencial em quatro grafos

Podemos observar, em todos os casos, que o algoritmo sequencial é mais rápido que o HEDA em *clusters* pequenos. No entanto, à medida que o número de máquinas aumenta, o tempo de execução do HEDA diminui, conforme esperado. A ferramenta Hadoop foi projetada para processar grandes massas de dados. Há experimentos que utilizam *clusters* com milhares de máquinas [Kang et al. 2011]. Para gerenciar o conjunto de máquinas, é necessário uma camada de software que opera sobre o sistema operacional, criando um sistema de arquivos distribuído. Por isso, o Hadoop apresenta *overhead* ao executar em poucas máquinas. Esse *overhead* deve-se às várias tarefas necessárias ao gerenciamento do sistema, por exemplo, o particionamento dos dados entre as tarefas Map, a criação de tarefas Map e Reduce e o gerenciamento de falhas do Hadoop. A execução de grafos maiores com o algoritmo sequencial é inviável devido às limitações de memória, o que justifica a proposição e implementação do HEDA.

5.4. Ajuste de Desempenho no Hadoop

O Hadoop tem cerca de duzentos parâmetros configuráveis [White 2009]. Para avaliar o efeito do ajuste fino dos valores dos parâmetros do Hadoop no ambiente experimental, os parâmetros numéricos foram estudados e os mais importantes foram testados experimentalmente. Os valores dos parâmetros foram variados para mais e para menos, tendo como

base o valor *default*. A Tabela 5 apresenta uma lista de parâmetros testados, com seus valores *default*, bem como os valores que produziram os melhores resultados para tempo de execução.

Nome do Parâmetro	Valor *default*	Melhor Valor
mapred.child.java.opts	200	900
mapred.tasktracker.map/reduce.tasks.maximum	2	2
tasktracker.http.threads	50	60
io.sort.factor	10	120
mapred.reduce.parallel.copies	5	10
io.file.buffer.size	4096	98304
io.sort.mb	100	400
arquivos abertos	1024	16384

Tabela 5. Valores dos parâmetros do Hadoop em ajuste de desempenho

Os resultados de tempo de execução para o grafo Internet de 01/01/2010, com os dois conjuntos de parâmetros apresentados na Tabela 5, indicaram redução de até 33,6% no tempo de processamento. Estes resultados são importantes pois os ganhos são obtidos apenas ajustando-se os valores dos parâmetros do Hadoop, sem adição de nenhum recurso ao sistema.

6. Conclusão e Trabalhos Futuros

Nesse artigo apresentamos o algoritmo paralelo HEDA que calcula medidas de centralidade em grafos. O algoritmo segue o modelo MapReduce e é implementado em Hadoop. Diversos experimentos foram feitos, com grafos de redes complexas reais e grafos sintéticos. Os resultados do algoritmo foram comparados com os resultados dos algoritmos HADI e um algoritmo sequencial. Foram feitas medidas de tempo de execução, *speedup* e eficiência, além da validação da correção dos resultados providos pelo HEDA. A escalabilidade do algoritmo foi também analisada.

A principal contribuição desse trabalho é oferecer um algoritmo paralelo em Hadoop para o cálculo exato das medidas de centralidade. Seus principais concorrentes são o algoritmo HADI, que faz o cálculo aproximado destas medidas, e o algoritmo sequencial, que é mais rápido ao processar pequenos grafos, mas não é capaz de processar grafos grandes. Em conjunto, os três algoritmos oferecem ao pesquisador opções importantes que devem ser avaliadas em função do contexto do seu trabalho, isto é, o tamanho da rede complexa a ser analisada e o tempo disponível para essa análise, bem como a necessidade de se obter valores exatos para as medidas.

Os trabalhos futuros incluem executar experimentos em um *cluster* maior, com o objetivo de verificar o comportamento do algoritmo quanto ao tempo de execução, à medida que mais máquinas são adicionadas. Além disso, algumas melhorias podem ser feitas no próprio algoritmo. A principal delas é não repassar informações de vértices já processados para fases posteriores. Se as distâncias de um determinado vértice para todos os outros vértices do grafo já foram encontradas, estas deverão ser armazenadas em um arquivo diferente do arquivo de distâncias, para não sobrecarregar a comunicação entre os nós.

Agradecimentos

Os autores agradecem ao InWeb e à FAPEMIG pelo apoio.

Referências

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. (2009). A View of the Parallel Computing Landscape. *Commun. ACM*, 52:56–67.

Breshears, C. (2009). *The Art of Concurrency*. O'Reilly, San Francisco, CA, USA.

Buckley, F. and Harary, F. (1990). *Distance in Graphs*. Addison-Wesley.

Cohen, J. (2009). Graph Twiddling in a MapReduce World. *Computing in Science and Engineering*, 11:29–41.

Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51:107–113.

Dean, J. and Ghemawat, S. (2010). MapReduce: a Flexible Data Processing Tool. *Commun. ACM*, 53:72–77.

Gonçalves, M. R. S., Maciel, J. N., and Murta, C. D. (2010). Geração de Topologias da Internet por Redução do Grafo Original. In *XXVIII SBRC*, pages 959–972.

Google Developers (2007). Lecture 5: Parallel Graph Algorithms with MapReduce. http://www.slideshare.net/jhammerb/lec5-pagerank - Acesso em setembro de 2011.

Gross, J. and Yellen, J. (1999). *Graph Theory and its Applications*. CRC Press, Inc., Boca Raton, FL, USA.

Kang, U., Tsourakakis, C. E., Appel, A. P., Faloutsos, C., and Leskovec (2011). HADI: Mining Radii of Large Graphs. *ACM Trans. Knowl. Discov. Data*, 5:8:1–8:24.

Kang, U., Tsourakakis, C. E., and Faloutsos, C. (2009). PEGASUS: A Peta-Scale Graph Mining System. In *Proc. 2009 Ninth IEEE Int. Conf. on Data Mining*, ICDM'09, pages 229–238.

Lin, J. and Schatz, M. (2010). Design Patterns for Efficient Graph Algorithms in MapReduce. In *Proc. Eighth Workshop on Mining and Learning with Graphs*, pages 78–85. ACM.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a System for Large-scale Graph Processing. In *Proc. 2010 International Conference on Management of Data*, SIGMOD '10, pages 135–146.

Nascimento, J. P. B. (2011). Um Algoritmo Paralelo para Cálculo de Centralidade em Grafos Grandes. Master's thesis, PPGMMC, CEFET–MG.

Newman, M. E. J. (2003). The Structure and Function of Complex Networks. *SIAM Review*, 45(2):167–256.

Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and C., K. (2007). Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proc. 13th IEEE Int. Symp. on High Performance Computer Architecture*, pages 13–24.

White, T. (2009). *Hadoop: The Definitive Guide*. O'Reilly.