MapReduce vs BSP para o cálculo de medidas de centralidade em grafos grandes

Francisco Sanches Banhos Filho[1] and Eduardo Javier Huerta Yero[1]

[1]Faculdade de Campo Limpo Paulista (FACCAMP). Rua Guatemala, 167
Jardim América – 13.231-230 - Campo Limpo Paulista / SP – Brasil

sanchesbanhos@gmail.com, huerta@cc.faccamp.br

Abstract. *Big Data techniques such as MapReduce have been used to calculate centrality measures of large graphs. MapReduce, however, has proven to be ill-suited for this kind of problem, where the communication rate among nodes is high in every iteration of the algorithm. Other models, such as Bulk Synchronous Parallel (BSP) have been considered in recent literature to be more appropriate for this task. This paper compares a BSP-based algorithm developed by the authors to calculate centrality measures in large graphs with HEDA (Hadoop-based Exact Diameter Algorithm) and HADI (HAdoop-based DIameter estimator), two algorithms based on the MapReduce model.*

Resumo. *Técnicas de Big Data, como MapReduce, tem sido utilizadas para calcular medidas de centralidade em grafos grandes. O modelo MapReduce, no entanto, não tem produzido resultados satisfatórios em problemas deste tipo. O modelo Bulk Synchronous Parallel (BSP) tem sido considerado na literatura recente como mais apropriado para resolver problemas em grafos. Este trabalho apresenta um algoritmo baseado no modelo BSP para calcular medidas de centralidade em grafos grandes e compara seus resultados com o HEDA (Hadoop-based Exact Diamenter Algorithm) e com o HADI (Hadoop-based DIameter estimator), ambos baseados no modelo MapReduce.*

1. Introdução

Redes sociais, rotas de transporte, telecomunicações e finanças são apenas algumas das áreas que costumeiramente utilizam grafos para modelar seus problemas. Em alguns casos estes grafos podem atingir milhões e até bilhões de nós e, portanto, exigir técnicas de processamento paralelo especializadas [Bondy and Murty, 2008].

A literatura recente mostra algumas iniciativas para processar grafos grandes usando o modelo MapReduce [Dean and Ghemawat, 2008]. O algoritmo HEDA (*Hadoop-based Exact Diameter Algorithm*) [Nascimento and Murta, 2012], baseado no modelo MapReduce, calcula medidas de centralidade como diâmetro e raio de um grafo. O algoritmo HADI (*Hadoop-based DIameter estimator*) [Kang et al. 2011], é também baseado no modelo MapReduce e calcula valores aproximados para estas mesmas medidas de centralidade de forma a produzir resultados em um tempo menor.

Alguns artigos na literatura recente apontam o modelo BSP [Valiant et al. 1989] como sendo mais apropriado para tratar com grafos [Pace, 2012] [Kajdanowicz et al. 2014). Neste trabalho será explorada esta hipótese utilizando o modelo BSP para

calcular as medidas de centralidade de um grafo e comparar seus resultados com o HEDA e o HADI. Para este fim os autores desenvolveram o algoritmo intitulado FastGDC que é baseado no modelo BSP.

O restante deste trabalho está organizado como a seguir. A seção 2 descreve brevemente o algoritmo FastGDC. A seção 3 apresenta os resultados dos testes realizados neste trabalho. A seção 4 apresenta a análise dos resultados e a seção 5 descreve as conclusões e trabalhos futuros.

2. FastGDC

Os autores criaram o algoritmo chamado FastGDC (*Fast Graph Diameter Calculator*), baseado no modelo BSP. O método *Compute()* é mostrado no Algoritmo 1, que é executado para cada nó do grafo. Ele começa no *superstep* 0, que é dedicado apenas a fazer as inicializações necessárias para o processamento (linhas 4 a 12).

Algoritmo 1. Método Compute () - FastGDC

Entrada: Grafo, Tamanho do vetor de distâncias
Saída: Arquivo contendo as excentricidades de todos os vértices

```
1 Método compute
2   if (superstep == 0)
3     distance_vector = new Vector(NODE_COUNT);
4     for each index in distance_vector
5           If (index == current_node)
6                 distance_vector[index] = 0;
7           else if (isNeighbour(index, current_node))
8                 distance_vector[index] = 1;
9           else
10                distance_vector[index] = MAX_INTEGER;
11          endif
12    endfor
13  else
14    shouldHalt = true;
15    for each message in messages_received_from_neighbouring_nodes
16          received_distance_vector = getContents(message);
17          for each index in received_distance_vector
18            current_entry = distance_vector[index];
19            remote_entry = received_distance_vector[index];
20                if (remote_entry < current_entry)
21                      shouldHalt = false;
22          distance_vector[index] = received_distance_vector[index];
23          if (getEccentricity(current_node) < remote_entry)
24            setEccentricity(current_node, remote_entry);
25          endif
26      endif
27    endfor
28  endfor
29    if (shoulHalt)
30      voteToHalt(current_node);
31    else
32      distance_vector_to_send = distance_vector
33      for each index in distance_vector_to_send
34        distance_vector_to_send[index] = distance_vector[index]+1;
35      endfor
36      for each node in neighbour_nodes
37            send(node, distance_vector_to_send);
38        endfor
39    endif
40  endif
41 end Método Compute
```

A partir da linha 13 o algoritmo se divide em duas partes: uma dedicada a processar as mensagens recebidas dos nós vizinhos e outra dedicada a enviar mensagens a estes mesmos nós. Cada mensagem recebida de um nó vizinho contém o vetor de distâncias desse nó (linhas 15 e 16) , que é usado para atualizar o vetor de distância do nó atual. Caso não tenha havido nenhuma atualização (linhas 17 a 27) após processar todas as mensagens o nó atual emite seu voto para que o processamento termine (linha 30). Caso contrário, o ele envia a todos os nós vizinhos seu vetor de distâncias acrescentando 1 em todas as entradas (linhas 32 a 34).

3. Testes

Os testes realizados neste trabalho foram executados em um *cluster* formado por computadores com processador Intel Core i7 com 8 (oito) núcleos de processamento, 8 (oito) gigabytes de memória RAM, 500 (quinhentos) gigabytes de espaço em disco e ligados em rede por meio de um switch gigabit ethernet. O sistema operacional utilizado no *cluster* desta pesquisa foi o Centos versão 5.9. Além disso utilizamos a versão 1.7 do Java JDK, a versão 1.0.4 do Hadoop e a versão 0.6.3 do Hama.

Para avaliação de desempenho dos algoritmos FastGDC, HEDA e HADI foram utilizados grafos sintéticos gerados usando o modelo Erdos-Renyi [Erdos and Renyi, 1959] e grafos que descrevem a topologia da Internet [Internet Research Lab, 2013]. Para calcular o *speedup* os autores desenvolveram uma solução sequencial baseada no algoritmo de busca em largura tradicional, sem utilizar MapReduce nem BSP.

3.1. Grafos sintéticos

A Figura 1 apresenta os tempos de execução do algoritmo sequencial, HEDA, HADI e FastGDC para 6 grafos com 200 mil arestas e 5, 10, 20, 30, 40 e 50 mil vértices respectivamente. Os tempos de execução são apresentados em minutos. O *speedup* de cada solução é mostrado na Figura 2.

A Figura 3 mostra os tempos de execução para um grafo com 10 mil vértices e quantidade de arestas variando entre 100 mil e 600 mil. Os tempos de execução são mostrados em minutos. A Figura 4 mostra o *speedup* os mesmos grafos da Figura 3.

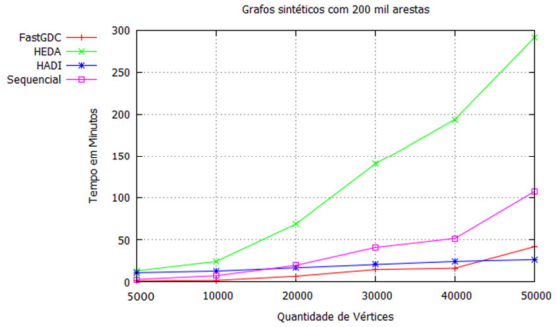

Figura 1. Grafos sintéticos com número variado de vértices

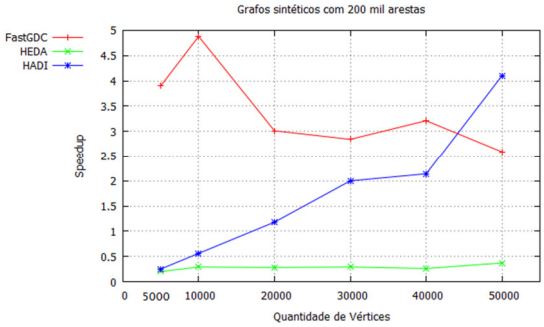

Figura 2. Speedup para grafos com 200 mil arestas

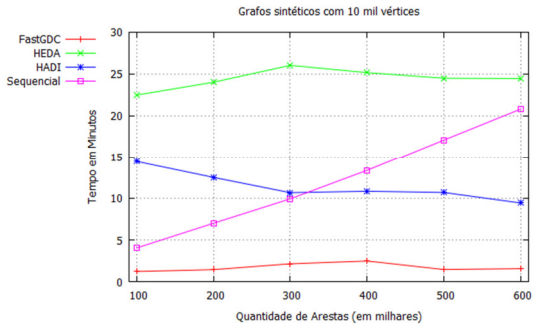

Figura 3. Grafos sintéticos com 10 mil vértices e quantidade variada de arestas

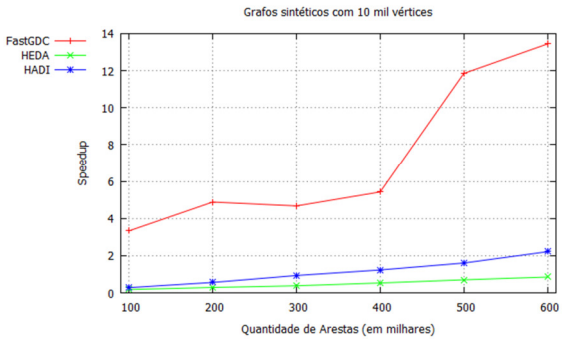

Figura 4. Speedup para um grafo com 50 mil vértices

3.2. Grafos reais

Os grafos testados nesta subseção representam a topologia da Internet [Internet Research Lab, 2013] entre os meses de março e julho de 2013. Eles têm em média 44.600 vértices e 340.000 arestas.

Os resultados dos tempos de execução para cada combinação de algoritmo e grafo para os grafos de internet podem ser visualizados na Figura 5. O *speedup* desse conjunto de grafos é mostrado na Figura 6.

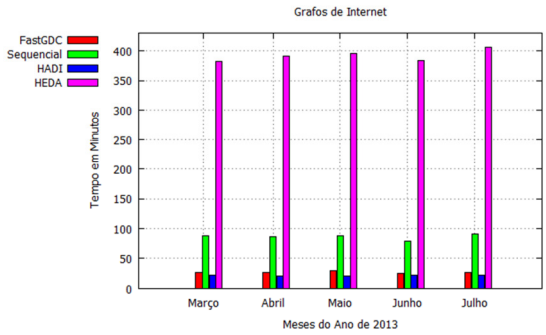

Figura 5. Tempos de execução para os grafos que descrevem a topologia da Internet.

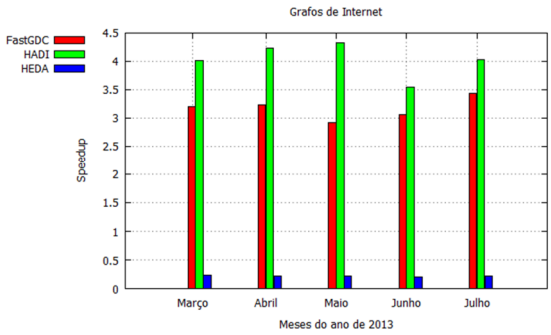

Figura 6. *Speedup* para os grafos que representam a topologia da Internet.

4. Análise dos resultados

O FastGDC obteve melhores resultados que o HEDA em todos os grafos testados. De fato, o FastGDC se comportou melhor do que o HADI na maioria dos testes feitos, apesar de que o HADI produz resultados aproximados. O modelo BSP, portanto, parece ser mais apropriado para tratar com grafos destas dimensões.

A razão para este comportamento está nas características intrínsecas de ambos modelos. No modelo MapReduce cada trabalhador executa uma sequencia de tarefas *map* e *reduce*, com a memória local do trabalhador sendo apagada ao término de cada tarefa. Desta forma, não é possível associar dados a um trabalhador específico e acessá-los em rodadas diferentes. Dados que precisem ser compartilhados entre rodadas devem ser armazenados na memória global, implementada pelo HDFS no caso do Hadoop, o que implica em custosas operações de leitura e escrita em um sistema de arquivos distribuído.

O modelo BSP funciona de forma diferente. Cada trabalhador executa uma sequencia de supersteps, e a memória local do trabalhador é preservada entre um superstep e outro. Além disso, cada trabalhador pode se comunicar diretamente com qualquer outro trabalhador do sistema, o que implica em um controle absoluto sobre como os dados são armazenados e transmitidos durante a execução. O algoritmo FastGDC faz uso destas facilidades ao criar um trabalhador para cada vértice do grafo e implementar a troca direta de informações entre os vértices vizinhos. Acreditamos que esta característica seja a principal responsável pelos melhores resultados obtidos pelo FastGDC.

5. Conclusões

Este trabalho comparou a capacidade dos modelos MapReduce e BSP para processar grafos grandes. O problema escolhido para fazer esta comparação foi o cálculo de medidas de centralidade, por exigir o cálculo das distâncias entre todos os pares de vértices do grafo e ser, portanto, exigente desde o ponto de vista computacional.

Para a comparação utilizamos o algoritmo HEDA, baseado no modelo MapReduce, e o algoritmo FastGDC, baseado no modelo BSP e desenvolvido pelos autores. Além destes dois algoritmos, apresentamos os resultados obtidos pelo HADI, também baseado no modelo MapReduce mas que calcula apenas resultados aproximados.

O FastGDC apresentou tempos de execução melhores do que o HEDA para todos os grafos testados, e chegou a superar o HADI na maior parte deles mesmo com o HADI oferecendo apenas resultados aproximados. Estes resultados indicam que o modelo BSP é de fato mais apropriado do que o MapReduce para tratar com grafos grandes. Acreditamos que a superioridade do BSP esteja diretamente ligada à sua capacidade de manter informação de estado nos nós trabalhadores entre um superstep e outro, diferente do modelo MapReduce que apaga a memória dos trabalhadores ao fim das etapas *map* e *reduce* e obriga a efetuar qualquer compartilhamento de informação através do HDFS.

Referências

Internet Research Lab. (2013). Fonte: Internet Research Lab: http://irl.cs.ucla.edu/index.html

Bondy, J., & Murty, U. (2008). *Graph Theory.* Springer.

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplied Data Processing on Large Clusters. *ACM Digital Library*, 107-113.

Erdos, P., & Renyi, A. (1959). On random graphs I. *Publ. Math. Debrecen. v. 6.*, 290-297.

Kajdanowicz, T., Kazienko, P., & Indyk, W. (2014). Parallel Processing of Large Graphs. *Future Generation Computer Systems 32*, 324-337.

Kang, U., Tsourakakis, C. E., Appel, A. P., Faloutsos, C., & Leskovec, J. (2011). HADI: Mining Radii of Large Graphs. *Journal ACM Transactions on Knowledge Discovery from Data (TKDD)*, Article 8.

Nascimento, J. P., & Murta, C. (2012). Um algoritmo paralelo em Hadoop para Cálculo de Centralidade em Grafos Grandes. *XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos.* Ouro Preto: SBC.

Pace, M. F. (2012). BSP vs MapReduce. *Procedia Computer Science. v. 9*, 246-255.

Valiant, L. G., Reeve, M., Zenith, S., & Wiley, E. (1989). Bulk-synchronous parallel computers. *In Parallel Processing and Artificial Intelligence*, 15-22.