MapReduce vs BSP no cálculo de medidas de centralidade em grafos grandes

Francisco Sanches Banhos Filho

Março / 2014

Dissertação de Mestrado em
Ciência da Computação

MapReduce vs BSP no cálculo de medidas de centralidade em grafos grandes

Esse documento corresponde a Dissertação de mestrado apresentada à Banca Examinadora no curso de Mestrado em Ciência da Computação da Faculdade Campo Limpo Paulista.

Campo Limpo Paulista, **31 de Março de 2014**.

Francisco Sanches Banhos Filho

Prof. Dr. Eduardo Javier Huerta Yero
Orientador

Faculdade Campo Limpo Paulista

Programa de Mestrado em Ciência da Computação

"MapReduce vs BSP no cálculo de medidas de centralidade em grafos grandes"

FRANCISCO SANCHES BANHOS FILHO

Dissertação de Mestrado apresentada ao Programa de Mestrado em Ciência da Computação da Faculdade Campo Limpo Paulista, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:

Prof. Dr. Eduardo Javier Huerta Yero
(FACCAMP)

Prof. Dr. Luis Mariano Del Val Cura
(FACCAMP)

Prof. Dr. Marcelo Paiva Guimarães
(FACCAMP)

Prof. Dr. Marco Aurélio Amaral Henriqu
(UNICAMP)

Campo Limpo Paulista – SP

28 de Março de 2014

FICHA CATALOGRÁFICA

Dados Internacionais de Catalogação na Publicação (CIP)

Câmara Brasileira do Livro, São Paulo, Brasil.

Banhos Filho, Francisco Sanches
MapReduce vs BSP no cálculo de medidas de centralidade em grafos grandes / Francisco Sanches Banhos Filho. Campo Limpo Paulista, SP: FACCAMP, 2014.

Orientador: Profº. Dr. Eduardo Javier Huerta Yero
Dissertação (mestrado) – Faculdade Campo Limpo Paulista – FACCAMP.

1. MapReduce. 2. Bulk Synchronous Parallel – BSP. 3. Grafos. I. Huerta Yero, Eduardo J. II. Campo Limpo Paulista. III. Título.

CDD-005.1

À minha esposa e meu Filho que sempre estiveram comigo em todos os momentos.

Agradecimentos

Agradeço à Deus por todas as experiências vivenciadas neste curso, pois com toda a certeza servirão para que eu seja uma pessoa melhor e sobretudo, um professor melhor.

Agradeço ao meu orientador professor Dr. Eduardo Javier Huerta Yero, pois sempre demonstrou serenidade para conduzir meus estudos mostrando sempre qual seria o melhor caminho a seguir. Agradeço também toda a paciência e dedicação com que conduziu minha orientação.

Agradeço a todos da FACCAMP – Faculdade Campo Limpo Paulista, em especial o Professor Dr. Osvaldo Luiz de Oliveira pelo incentivo e pelo pronto atendimento solucionando dúvidas sobre o funcionamento do mestrado.

Agradeço à professora Dr. Cristina Duarte Murta e ao Msc. João Paulo Barbosa Nascimento, pois gentilmente cederam o algoritmo HEDA.

Agradeço à equipe do Hama, especialmente ao Suraj Menon pois, quando eu detectava alguma anomalia no framework, o Suraj sempre estava disposto a ajudar.

Agradeço a todos os amigos que fiz no decorrer do curso: Eduardo Machado Real, Thiago Fernandes, Valter Ferlet, Luiz Francisco Contri, Denilson Peramos, Waldomiro Moreira, Fabiano Fernandes, Marta Angélica Ferreira; todos vocês contribuíram de alguma forma para que este trabalho fosse concluído, seja apenas com uma conversa sobre banco de dados, ou até mesmo como companhia para os poucos Happy Hour que fizemos.

Agradeço os amigos da Unemat, especialmente ao Ivan Pires que enquanto estava na direção geral do Campus não poupou esforços para me ajudar cedendo o espaço do Laboratório de Hardware e os 20 computadores utilizados nesta pesquisa.

"Há homens que lutam um dia e são bons.
Há outros que lutam um ano e são melhores.
Há os que lutam muitos anos e são muito bons.
Porém, há os que lutam toda a vida. Esses são os imprescindíveis"

Bertolt Brecht

Resumo. *O termo Big Data é atribuído a um conjunto de dados cujo tamanho e complexidade é tal que não podem ser gerenciados por ferramentas convencionais (e.g. sistemas de gerenciamento de bancos de dados relacionais) em um período razoável de tempo. Nos últimos anos houve uma explosão de técnicas inovadoras para o gerenciamento de conjuntos de dados deste porte, normalmente utilizando soluções capazes de escalar até milhares de nós de processamento. Dentre os problemas normalmente resolvidos usando técnicas de Big Data destacam-se aqueles associados a grafos. 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, o que implica que seu processamento apresenta um grande desafio desde o ponto de vista computacional . Este trabalho terá como foco o cálculo de medidas de centralidade em grafos grandes, que envolvem o cálculo da distância entre todos os vértices do grafo. As medidas de centralidade foram escolhidas porque seu cálculo implica na análise exaustiva dos grafos e, portanto, são ideais para analisar o desempenho das técnicas de processamento paralelo comumente associadas ao fenômeno Big Data. Em particular, o objetivo do trabalho é comparar a capacidade dos modelos MapReduce e Bulk Synchronous Parallel (BSP) para resolver este tipo de problema. Para isso, o trabalho utiliza os resultados obtidos pelos algoritmos HEDA (Hadoop Exact Diameter Algorithm) e HADI (Hadoop based DIameter estimator), já relatados na literatura e que usam MapReduce, comparando-os com o FastGDC (Fast Graph Diameter Calculator), um algoritmo que utiliza o modelo BSP.*

Abstract*: The term Big Data is attributed to a collection of data sets whose size and complexity is such that they cannot be processed by conventional tools (e.g. relational database management systems) in a reasonable period of time. In recent years a number of innovative techniques have been created to deal with data sets this size, usually capable of scaling to thousands of processing nodes. Among the problems usually solved using Big Data techniques are those associated with graphs. Social networks, transportation, telecommunications and finance are just some of the areas that routinely use graphs to model their problems. In some cases, these graphs can reach millions and even billions of nodes, thus creating an enormous challenge from the computational standpoint. This work will focus on the calculation of measures of centrality in large graphs, involving the calculation of the distance between all pairs of vertices of the graph. The centrality measures were selected because their calculation implies in the exhaustive processing of the graph and are therefore well-suited to assess the performance of the various parallel processing techniques usually associated with Big Data. In particular, the goal of this work is to compare the ability of MapReduce and Bulk Synchronous Parallel (BSP) models to solve these kinds of problems. To that effect, we present the results obtained by HEDA (Hadoop Exact Diameter Algorithm) and HADI (Hadoop based DIameter estimator), already reported in recent literature and that use MapReduce, and compare them with the peformance of FastGDC (Fast Graph Diameter Calculator), an algorithm that uses the BSP model.*

Sumário

Capítulo 1. Introdução 1

Capítulo 2. Grafos 4

2.1. Conceitos básicos de grafos 4

2.2. Tipos de grafos 5

2.3. Representação de um grafo 7

2.4. Passeios, caminhos, ciclos e distância entre vértices 8

2.5. Problema de menor caminho em grafos 8

2.6. Centralidades em grafos 10

2.7. Considerações finais do capítulo 15

Capítulo 3. Processamento paralelo 16

3.1. Sistemas de computação paralela 16

3.2. Algoritmos paralelos 19

3.3. Algoritmos paralelos para grafos 21

3.4. Medidas de desempenho 24

3.5. Modelos específicos para Big Data 26

3.6. Modelo MapReduce X Modelo BSP 37

3.7. Considerações finais do capítulo 38

Capítulo 4. Algoritmos para cálculo de centralidade em grafos grandes 39

4.1. *Hadoop based DIameter estimator* (HADI) 39

4.2. *Hadoop-based Exact Diameter Algorithm* (HEDA) 39

4.3. Algoritmo FastGDC 42

4.4. Considerações finais do capítulo 47

Capítulo 5. Projeto dos experimentos 48

5.1. Metodologia 48

5.2. Infraestrutura e ambiente computacional 48

5.3. Conjunto de dados que compõem os experimentos 49

5.4. Considerações finais do capítulo 52

Capítulo 6. Resultados experimentais 53

6.1. Tempos de execução 53

6.2. Escalabilidade 68

6.3. Armazenamento de vértices do grafo em disco 75

6.4. Análise dos resultados 78

Capítulo 7. Conclusões e trabalhos futuros 80

Referências 82

Apêndice A 87

Anexos 88

1. Tabela de resultados algoritmo Sequencial 88

2. Tabela de resultados algoritmo HEDA 89

3. Tabela de resultados algoritmo HADI 90

4. Tabela de resultados algoritmo FastGDC 92

5. Resultados *speedup* variando o número de computadores 93

6. Resultados eficiência variando o número de computadores 94

7. Algoritmo Sequencial 95

8. Comprovante de submissão de artigo 99

9. Artigo Submetido 99

Lista de Tabelas

1. Distância entre todos os vértices do grafo H 14

2. Subprojetos que compõem o Apache Hadoop (White, 2009) 32

3. Exemplo Hadoop: iterações do algoritmo HEDA (Nascimento & Murta, 2011) 41

4. Tabela comparativo entre os algoritmos 46

5. Configuração do Apache Hama 49

6. Grafos sintéticos 50

7. Grafos *DataSet* de Stanford (Stanford Large Nertwork Dataset Collection, 2012) 51

8. Grafos que descrevem a topologia da Internet. 52

9. Tempos de execução para grafos com 5 mil vértices 53

10. Tempos de execução para grafos com 10 mil vértices 56

11. Diâmetro e raio de grafos com 10 mil vértices 57

12. Tempos de execução para grafos com 20 mil vértices 58

13. Tempos de execução para grafos com 30 mil vértices 59

14. Tempos de execução para grafos com 40 mil vértices 61

15. Tempos de execução para um grafo com 50 mil vértices. 63

16. Tempos de execução dataset da Universidade de Standford. 64

17. Tempos de execução para os grafos reais (internet) 66

Lista de Figuras

1. Exemplo de grafo: grafo G 4
2. Tipos de arestas 5
3. Grafo dirigido e não dirigido 5
4. Grafo completo 6
5. Grafo bipartido 6
6. Multigrafos 7
7. Tipos de representações para grafos 7
8. Exemplo de grafos: grafos G e H 11
9. Exemplo de grafos: grafos K e L (centralidade de intermediação) 13
10. Exemplo de grafo: grafo H (centralidade de eficiência) 14
11. Arquitetura de acesso a memória compartilhada (Kumar, 2002) 18
12. *Speedup* sublinear, linear e superlinear 24
13. Arquitetura de um CEP Server 27
14. Stream Processing (Stonebraker, Çetintemel, & Zdonik, 2005) 28
15. Fases Map e Reduce (contador de palavras) 30
16. Visão geral execução MapReduce (Dean & Ghemawat, 2008) 31
17. Passos de um algoritmo BSP (Bisseling, 2004) 34
18. Exemplo máximo valor (Malewicz, et al., 2010) 35
19. Máquina de estados do vértice (Malewicz, et al., 2010) 36
20. Arquitetura Hama (Apache Hama, 2013) 36
21. Exemplo de grafo: grafo I (HEDA) (Nascimento & Murta, 2011) 40
22. Diagrama de blocos algoritmo FastGDC 45
23. Paralelização do algoritmo FastGDC 46

24. Grafo sintético com 5 mil vértices. 54

25. Speedup para grafos com 5 mil vértices. 55

26. Tempos de execução para grafos com 10 mil vértices. 56

27. Speedup para grafos com 10 mil vértices. 57

28. Tempos de execução para grafos com 20 mil vértices. 58

29. Speedup para grafos com 20 mil vértices. 59

30. Tempos de execução para grafos com 30 mil vértices. 60

31. Speedup para grafos com 30 mil vértices. 61

32. Tempos de execução para grafos com 40 mil vértices. 62

33. Speedup para grafos de 40 mil vértices. 62

34. Tempo de execução para um grafo de 50 mil vértices. 63

35. Speedup para um grafo com 50 mil vértices. 64

36. Tempos de execução para os grafos do dataset de Standford. 65

37. Speedup para os grafos do dataset da Universidade de Standford. 66

38. Tempos de execução para os grafos que descrevem a topologia da Internet. 67

39. Speedup para os grafos que representam a topologia da Internet. 67

40. Speedup para um grafo com 10 mil vértices e 100 mil arestas. 69

41. Eficiência para um grafo com 10 mil vértices e 100 mil arestas. 70

42. Speedup para um grafo com 10 mil vértices e 200 mil arestas. 71

43. Eficiência para um grafo com 10 mil vértices e 200 mil arestas. 71

44. Speedup para um grafo com 10 mil vértices e 400 mil arestas. 72

45. Eficiência para um grafo com 10 mil vértices e 400 mil arestas. 73

46. Speedup para um grafo com 10 mil vértices e 600 mil arestas. 74

47. Eficiência para um grafo com 10 mil vértices e 600 mil arestas. 75

48. Grafo com 10 mil vértices e 100 mil arestas com e sem despejo em disco. 76

49. Grafo com 10 mil vértices e 200 mil arestas com e sem despejo em disco. 77

50. Grafo com 10 mil vértices e 600 mil arestas com e sem despejo em disco. 77

Capítulo 1. Introdução

Durante a última década a quantidade de dados disponível digitalmente tem aumentado de forma significativa. A explosão no uso das redes sociais e de sistemas para compartilhamento de músicas e vídeos, além da consolidação do comércio eletrônico, são os maiores responsáveis por este aumento. A maior disponibilidade de dados coletados através de logs de servidores, sensores e satélites, dentre outras fontes de dados digitais, completam este cenário.

Em 2012, aproximadamente 2,5 exabytes foram criados diariamente, e a quantidade total de dados armazenados atingiu 2,7 zetabytes (Mewawalla, 2012). Este crescimento está se dando de forma tão acelerada que 90% dos dados armazenados hoje foram criados nos últimos dois anos (Mewawalla, 2012). Para nomear este fenômeno – a geração, armazenamento, processamento e análise de grandes massas de dados – tem-se utilizado o termo Big Data (Pettey & Goasduff, 2011).

A empresa Gartner define os desafios e as oportunidades associados a Big Data em três dimensões (Pettey & Goasduff, 2011): **V**olume, onde a complexidade esta associada à quantidade, tanto em dados tradicionais quanto em novos tipos de dados; **V**elocidade, onde se busca analisar os dados com a mesma velocidade em que são produzidos; e **V**ariedade, relacionado ao processamento de diferentes tipos de informações (e.g. tabular, hierárquica, e-mails, fotos, áudio, etc.).

Para processar estas grandes massas de dados tem-se utilizado técnicas como *Complex Event Processing* (Wasserkrug, Gal , Etzion, & Turchin, 2008), *Stream Processing* (Carney, et al., 2002), MapReduce (Dean & Ghemawat, 2008) e *Bulk-synchronous parallel* (BSP) (Valiant, Reeve, Zenith, & Wiley, 1989). O ponto em comum entre todas elas é a preocupação por resolver de forma inovadora um ou vários dos "**V**" definidos pela Gartner. *Complex Event Processing* e *Stream Processing* focam em soluções em tempo real, enquanto MapReduce e BSP tem como alvo problemas com grandes volumes de dados.

Dentre os problemas normalmente resolvidos usando técnicas de Big Data destacam-se aqueles associados a grafos (Bondy & Murty, 2008). 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 normalmente utilizadas para Big Data. No entanto, os grafos apresentam desafios únicos que fazem com que a aplicabilidade destas técnicas não seja tão natural: por exemplo, há comumente uma interdependência forte entre os nós do grafo, o que dificulta a paralelização da maioria dos algoritmos que tratam com eles. Estes problemas magnificam-se quando se trata de grafos com milhões de nós e arestas.

Este trabalho tem como foco a comparação dos modelos MapReduce e BSP no cálculo de medidas de centralidade em grafos grandes. Cálculos de centralidade envolvem o cálculo da distância mínima entre todos os vértices do grafo. Para cada vértice e o conjunto dos demais alcançáveis a partir dele, a maior destas distâncias é sua excentricidade. A maior excentricidade dentre todos os vértices é o diâmetro do grafo, e a menor, o raio (Hage & Harary, 1995). Estas medidas de centralidade são essenciais para muitos dos problemas que tratam com grafos, como por exemplo, aqueles que precisam determinar pontos de distribuição para uma rede determinada (e.g. localização de centrais telefônicas). Além disso, seu cálculo implica na análise exaustiva de todos os nós e arestas do grafo e, portanto, são ideais para avaliar e comparar o desempenho dos modelos MapReduce e BSP.

Existem soluções para o problema de centralidade em grafos grandes que já foram implementadas utilizando o modelo MapReduce. Em (Nascimento & Murta, 2012) é apresentado o algoritmo HEDA, baseado no modelo MapReduce, cujo objetivo é calcular a excentricidade de todos os nós do grafo oferecendo resultados exatos. O algoritmo HADI (Kang, Tsourakakis, Appel, Faloutsos, & Leskovec, 2011) calcula valores aproximados para a excentricidade de cada nó do grafo usando o modelo MapReduce.

Apesar do sucesso obtido pelo modelo MapReduce, alguns artigos na literatura mais recente apontam o modelo BSP como sendo mais apropriado para tratar com grafos

(Pace, 2012). Neste trabalho pretendemos explorar esta hipótese comparando os algoritmos HEDA e HADI, baseados no modelo MapReduce, com o FastGDC, um algoritmo baseado em busca em largura e que usa o modelo BSP.

O restante deste trabalho está organizado como a seguir. O capítulo 2 apresenta alguns conhecimentos elementares sobre teoria de grafos que são necessários para o entendimento dos diferentes tipos de centralidades. O capítulo 3 introduz os conceitos básicos sobre processamento paralelo e apresenta alguns dos modelos mais relevantes para Big Data. O capítulo 4 apresenta os algoritmos HEDA e HADI, que utilizam o modelo MapReduce para resolver o problema de centralidade em grafos grandes, e o algoritmo baseado em busca em largura que foi proposto neste trabalho, chamado de FastGDC (*Fast Graph Diameter Calculator*). No capítulo 5 são apresentados o projeto dos experimentos, a infraestrutura utilizada e o conjunto de dados utilizados nos testes. O capítulo 6 discute os resultados dos experimentos relacionados no capítulo 5. O capítulo 7 apresenta as considerações finais e os trabalhos futuros.

Capítulo 2. Grafos

Este capítulo define o conceito de grafo e apresenta alguns dos tipos de grafo mais conhecidos. Define-se também o conceito de centralidade e alguns dos tipos de medidas de centralidade existentes. Como parte desta introdução, o texto aborda também o problema do menor caminho entre um par de vértices, que é parte fundamental no cálculo das medidas de centralidade.

2.1. Conceitos básicos de grafos

Um grafo G = (V, E) é definido por um conjunto V de vértices e um conjunto de arestas E. A função de incidência Ψ^G associa cada aresta em E com o par de vértices (não necessariamente distintos) de V que formam a aresta. Quando este par de vértices não é ordenado, o grafo é chamado de simples (Bondy & Murty, 2008). A Figura 1 mostra um exemplo.

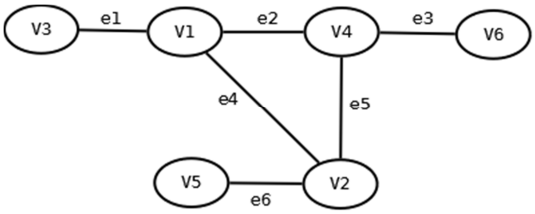


Figura 1. Exemplo de grafo: grafo G

Na Figura 1 é possível identificar os vértices e as arestas do grafo G: G = (V, E), onde V = {v1,v2,v3,v4,v5,v6} e E = {e1,e2,e3,e4,e5,e6}, e a função de incidência definida por Ψ^G é:

Ψ^G (e1) = (v1,v3) Ψ^G (e2) = (v1,v4) Ψ^G (e3) = (v4,v6)

Ψ^G (e4) = (v1,v2) Ψ^G (e5) = (v2,v4) Ψ^G (e6) = (v2,v5)

O grau de um vértice é igual à quantidade de arestas que o vértice possui. O grau mínimo de um grafo corresponde ao grau do vértice do grafo com menor grau. Na Figura 1 os vértices v3, v6 e v5 possuem grau 1: como nenhum outro vértice tem grau

menor, então 1 é o grau mínimo para o grafo. Da mesma forma os vértices v1, v2 e v4 têm três arestas cada um e este é o grau máximo para o grafo, pois não há nenhum outro vértice com grau superior a este.

Arestas paralelas (ou múltiplas) são arestas que conectam os mesmos vértices. O item a) da Figura 2 mostra que os vértices v1 e v2 são extremos em comum das arestas e1 e e2 e, portanto, e1 e e2 são paralelas. Arestas são adjacentes quando possuem um extremo em comum. No item b) da Figura 2, as arestas e1 e e2 possuem o vértice v1 como extremo em comum. Quando uma aresta possui vértices extremos iguais ela é chamada de laço, como é o caso do item c) da Figura 2.

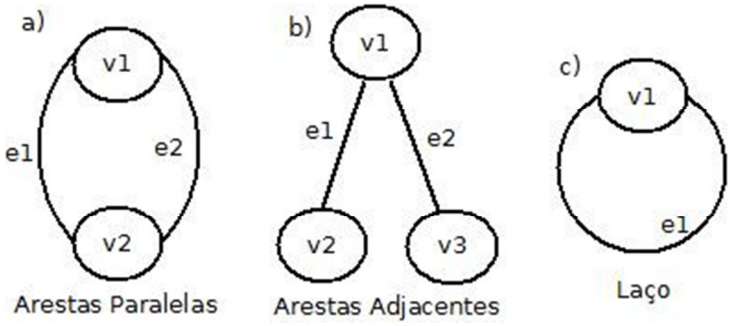


Figura 2. Tipos de arestas

2.2. Tipos de grafos

Em um grafo dirigido as arestas são chamadas de arcos, arestas direcionadas ou setas, e a ordem entre os vértices que formam o arco é relevante. Um arco $\Psi^G(e) = u,v$ em um grafo dirigido indica que é possível chegar no vértice v a partir do vértice u, mas não o inverso (Cormen, Leiserson, Rivest, & Stein, 2001). A Figura 3 mostra um exemplo de grafo dirigido (Figura 3-a) e não dirigido (Figura 3-b).

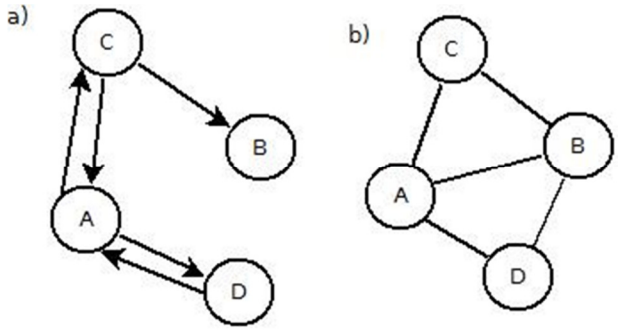


Figura 3. Grafo dirigido e não dirigido

Um grafo é conexo se, para cada divisão de seu conjunto de vértices em dois conjuntos não vazios X e Y, existe uma aresta com uma extremidade em X e uma extremidade em Y, caso contrário o grafo é chamado de desconexo (ou desconectado) (Bondy & Murty, 2008).

Um grafo é *infinito* se seu conjunto de vértices e/ou arestas é infinito, caso contrário é chamado de *finito*. Já um grafo é chamado de *ponderado* se a cada aresta é associado um valor real que representa seu peso (Bondy & Murty, 2008).

Um grafo H = (V, E) é completo, se todos os seus vértices são ligados entre si. Grafos completos com *n* vértices são chamados de *Kn*. O número de arestas de um grafo *Kn*, é dado por: $E(H) = \frac{n(n-1)}{2}$. A Figura 4 ilustra um grafo completo com quatro vértices e seis arestas, chamado de *K4*.

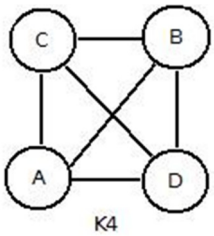


Figura 4. Grafo completo

Um grafo é chamado de bipartido se o conjunto de vértices pertencentes ao grafo pode ser particionado em dois subconjuntos U e V e cada aresta possui uma extremidade em U e outra em V. A Figura 5 ilustra um grafo bipartido e os subconjuntos U e V.

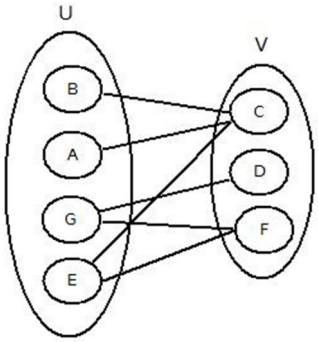


Figura 5. Grafo bipartido

Grafos dinâmicos (ou com características temporais) são grafos que em períodos de tempo diferentes podem sofrer alterações nos pesos de suas arestas.

Multigrafos são grafos nos quais dois vértices podem ser conectados através de mais de uma aresta. Para grafos não dirigidos é suficiente ter duas arestas entre o mesmo par de vértices, já para grafos dirigidos o par de vértices deve ser conectado por duas arestas na mesma direção ou pelo menos por 3 arestas. A Figura 6, ilustra dois multigrafos.

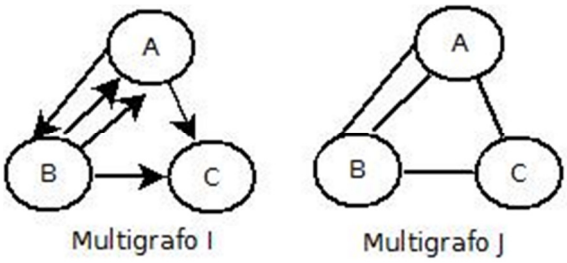
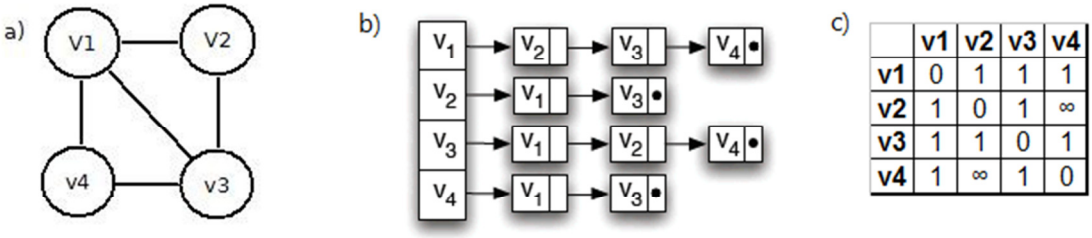


Figura 6. Multigrafos

2.3. Representação de um grafo

As estruturas de dados mais conhecidas para representar grafos são a matriz de adjacência e a lista de adjacência. A matriz de adjacência é uma matriz quadrada onde cada linha e coluna corresponde a um vértice do grafo. Se o grafo tem uma aresta entre os vértices v_i e v_j então na matriz de adjacência tem-se que $M[i,j]=1$.

A lista de adjacência é na verdade uma coleção de listas não ordenadas, com uma lista para cada vértice do grafo. A lista associada ao vértice v_i contém todos os vértices vizinhos de v_i . A Figura 7 mostra um grafo e sua representação através de uma lista de adjacência e uma matriz de adjacência.



	v1	v2	v3	v4
v1	0	1	1	1
v2	1	0	1	∞
v3	1	1	0	1
v4	1	∞	1	0

Figura 7. Tipos de representações para grafos

2.4. Passeios, caminhos, ciclos e distância entre vértices

Dado um grafo G= (V, E), um passeio é a sequencia de vértices e arestas que começam e terminam com vértices pertencentes ao conjunto V. A sequencia v_0, e_1, v_1, .., ,e_k, v_k, onde v_0, v_1, .., v_k pertencem ao conjunto V e Ψ^G (e_i) = v_{i-1}, v_i para i=1...k é um passeio de v_1 a v_k. O comprimento do passeio é o número de arestas do mesmo (Kleinberg & Tardos, 2006). É chamado de ciclo um passeio em que todos os vértices e arestas são distintas, excetuando-se os vértices origem e destino.

Um caminho de um vértice *u* até um vértice *u*' em um grafo G = (V,E) é uma sequência (v_0, v_1, v_2, ..., v_k) de vértices tais que *u* = v_0, *u'*= v_k e existe e_j tal que Ψ^G (e_j) = (v_{i-1}, v_i) para i = 1...k. Além disso, todos os vértices do caminho precisam ser distintos, ou seja, $v_i \neq v_j$ para i≠j e i,j=0...k. O comprimento do caminho é o número de arestas no caminho, neste caso *k*. Se existe um caminho p de *u* até *u'*, afirma-se que *u'* é acessível a partir de *u* via *p*, ou $u \overset{p}{\rightsquigarrow} u'$ (Cormen, Leiserson, Rivest, & Stein, 2001). A distância entre dois vértices *u* e *v* pode ser calculada apenas se $u \overset{p}{\rightsquigarrow} v$, e seu valor é o menor comprimento de todos os caminhos entre *u* e *v*.

2.5. Problema de menor caminho em grafos

Seja um grafo orientado e ponderado G = (V, E), com função peso w: E → R mapeando arestas para pesos de valores reais. O peso do caminho p = {v_0, v_1,...,v_k} é o somatório dos pesos de suas arestas, sendo que w(v_{i-1},v_i) representa o peso da aresta entre o vértice v_i e seu antecessor v_{i-1} (Cormen, Leiserson, Rivest, & Stein, 2001):

$$w(p)=\sum_{k}^{i=1} w(v_{i-1}, v_i)$$

O peso do caminho mais curto desde u até v pode ser definido como (Russell & Norvig, 2004):

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{se existe um} \\ & \text{caminho de } u \text{ até } v \\ \infty & \text{caso contrário} \end{cases}$$

Há algumas variantes para problemas de caminhos em grafos (Cormen, Leiserson, Rivest, & Stein, 2001):

- **Caminhos mais curtos de única origem (*single source shortest path - SSSP*)**: encontrar o caminho mais curto desde um vértice de origem s ϵ V até todo vértice v ϵ V.
- **Caminhos mais curtos de destino único**: encontrar um caminho mais curto até um determinado vértice de destino t a partir de cada vértice V, com t e v ϵ V.
- **Caminho mais curto de par único**: encontrar o caminho mais curto desde u até v para determinados vértices u e v.
- **Caminho mais curto de todos os pares *(all pairs shortest path - APSP)*** : encontrar o caminho mais curto desde u até v para todo par de vértices u e v.

Existem vários algoritmos que tratam o problema de menor caminho e suas variantes. A maior parte destas soluções são baseadas em estratégias de busca. Os algoritmos sequenciais mais conhecidos que tratam desse problema incluem os algoritmos Dijkstra, Bellman-Ford, e Floyd-Warshall (Cormen, Leiserson, Rivest, & Stein, 2001).

2.5.1. Estratégias de busca

As estratégias de busca mais utilizadas em grafos são busca em largura e busca em profundidade. Na busca em profundidade, as arestas são exploradas a partir do vértice mais recentemente descoberto que ainda tem arestas inexploradas saindo dele. Quando todas as arestas deste vértice são exploradas, a busca retorna para explorar as arestas que deixam o vértice a partir do qual ele foi descoberto. Esse processo continua até se descobrir todos os vértices acessíveis a partir do vértice inicial. Se restarem quaisquer vértices não descobertos, então um deles será selecionado como uma nova origem e a busca se repetirá a partir dele. Esse processo inteiro será repetido até que todos os vértices sejam descobertos (Cormen, Leiserson, Rivest, & Stein, 2001).

A estratégia de busca em profundidade não é ótima, ou seja, nem sempre encontra a melhor solução quando existem soluções diferentes. Ela também não é

completa, pois nem sempre encontra uma solução quando existe alguma. Sua maior vantagem é que para problemas com várias soluções esta estratégia pode ser mais rápida em achar uma das soluções (Cormen, Leiserson, Rivest, & Stein, 2001).

Na busca em largura o nó mais recentemente descoberto é expandido primeiro, em seguida todos os sucessores, depois os sucessores deles e assim até não restarem mais vértices por analisar. Todos os nós a partir do nó recentemente descoberto são expandidos antes que todos os nós no nível seguinte sejam expandidos. Caso exista uma solução e se o grafo é finito, a solução é encontrada. Esta estratégia é ótima e completa, pois sempre encontra a melhor solução quando existem soluções diferentes e sempre encontra solução quando existe alguma. A execução dessa técnica envolve um alto consumo de memória, pois o número de vértices pode chegar a ser exponencial em relação à profundidade. Ela exige um esforço computacional relativamente grande se o comprimento dos caminhos não for pequeno, e esse esforço é gasto com todos os vértices (Cormen, Leiserson, Rivest, & Stein, 2001), (Russell & Norvig, 2004).

A busca em largura é um dos algoritmos mais utilizados para se pesquisar um grafo. Algoritmos utilizados para resolver o problema de menor caminho entre vértices como Dijkstra e para encontrar a árvore geradora mínima como o Prim utilizam esse tipo de busca (Cormen, Leiserson, Rivest, & Stein, 2001).

2.6. Centralidades em grafos

Hage e Harary classificam três tipos de centralidade em (Hage & Harary, 1995): centralidade de grau (*Degree Centrality*), centralidade de proximidade (*Closeness Centrality*) e centralidade de intermediação (*Betweenness Centrality*). Estes modelos são aplicados principalmente na área de antropologia em estudos que envolvem redes sociais. Os autores propuseram uma nova medida de centralidade baseada no cálculo da excentricidade de cada vértice que eles chamaram de *Efficiency Centrality*. Na teoria dos grafos esta medida é conhecida apenas como centralidade, e será ela a utilizada neste trabalho.

2.6.1. Centralidade de Grau

Centralidade de grau é uma medida baseada na contagem do número de adjacências que cada vértice pertencente ao grafo possui. Essa medida foi utilizada por vários autores na década de 70, mas foi Nieminem quem chamou de centralidade de grau (Nieminem, 1974). Dado um grafo G= (V, E) com *n* vértices, seja v_k um vértice qualquer de G. A centralidade de grau de v_k é dada pelo número de arestas incidentes a v_k. Os vértices centrais serão os vértices com o maior grau.

Dados os grafos G e H presentes na Figura 8, é possível observar que o vértice central dos dois grafos é o vértice E. Ambos possuem o mesmo valor de centralidade de grau, que é 3. No entanto, se esses grafos representassem redes de comunicação o vértice E do grafo G dominaria apenas uma pequena parcela da rede, enquanto o vértice E do grafo H dominaria quase que completamente essa rede. Devido a esta característica, criou-se a definição de centralidade relativa de grau. Esta centralidade é baseada não apenas no grau de cada vértice, mas também no número total de vértices do grafo.

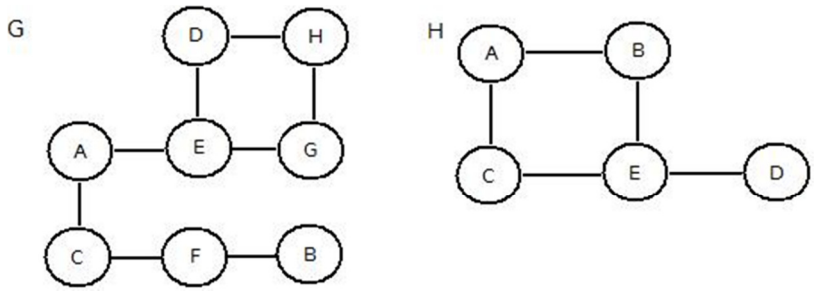


Figura 8. Exemplo de grafos: grafos G e H

A fórmula para calcular a centralidade relativa de grau é dada por:

$\mathrm{CR} = \frac{grau_do_vértice}{número_de_vertices - 1} =$ Então para o grafo H a centralidade relativa de grau dos vértices A, B e C é 0,5, do vértice D é 0,25 e do vértice E, 0,75 enquanto esta mesma centralidade para o vértice E do grafo G seria 0,42.

2.6.2. Centralidade de Proximidade

Sabidussi propôs em (Sabidussi, 1966) uma medida de centralidade chamada de centralidade de proximidade. Essa medida é baseada na soma das distâncias de um vértice em relação a todos os demais do grafo. Dado um grafo G= (V, E) com *n* vértices, seja v_k um vértice qualquer de G. A centralidade de proximidade de v_k pode ser obtida pelo inverso da soma de todas as distâncias do vértice v_k para todos os outros vértices pertencentes ao grafo. Para tanto, é necessário utilizar a fórmula: $\mathrm{CP} = \frac{1}{soma_das_distâncias_do_vértice}$. Os vértices centrais são aqueles cuja centralidade de proximidade for maior.

Considerando o grafo H presente na Figura 8 é possível calcular a centralidade de proximidade para todos os vértices pertencentes ao grafo. O vértice A possui a centralidade de proximidade que é igual a: $A = \frac{1}{1+1+2+3} = 0{,}14$, para o restante dos vértices pertencentes ao grafo, a centralidade de proximidade é: B=0,16, C=0,16, E=0,2 e D=0,12. Com esses resultados é possível concluir que o vértice central do grafo H é o vértice E.

2.6.3. Centralidade de Intermediação

Centralidade de intermediação é uma medida que observa quais vértices em um grafo são indispensáveis para manter o grafo conexo. Por exemplo, em uma rede social quais pessoas figuram entre os grupos fechados de alunos e professores? Essa pessoa (vértice) realiza a intermediação entre os dois grupos. Este tipo de centralidade é baseada na relação de todos os caminhos mínimos entre pares de vértices em um grafo. Os vértices que estiverem em um maior número de caminhos serão os vértices que possuem a maior intermediação. Esta medida for proposta por Freeman em (Freeman, 1977).

Observando o grafo K na Figura 9, é possível determinar o conjunto de caminhos entre todos os pares de vértices: {B**A**C,C**A**B}, {B**A**E,E**A**B}, {B**A**D,D**A**B}, {B**A**F,F**A**B}, {E**A**C,C**A**E}, {E**A**D,D**A**E}, {E**A**F,F**A**E}, {C**A**D,D**A**C}, {C**A**F,F**A**C}, {D**A**F,F**A**D}. Neste conjunto é possível observar que o vértice A se repete em cada

caminho entre pares de vértices. Quando um vértice se repete em um par de caminhos aumenta-se em 1 o grau de intermediação do vértice. No caso do grafo K, o vértice A aparece repetido 10 vezes nos caminhos entre os pares de vértices, enquanto os outros vértices não aparecem repetidos em nenhum caminho. Conclui-se, portanto, que o grau de intermediação do grafo é 10.

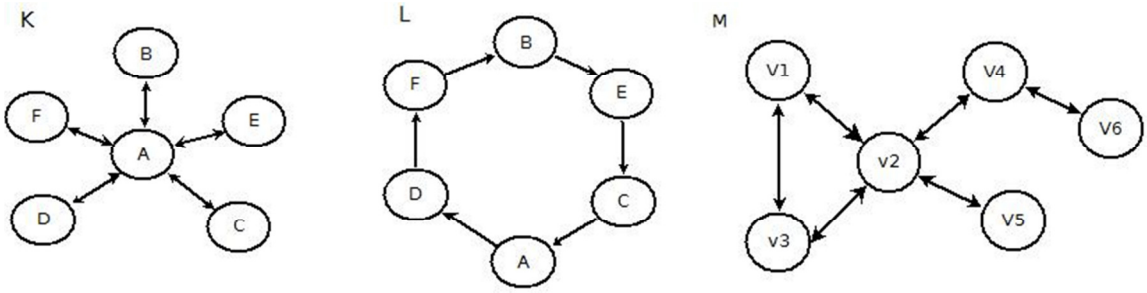


Figura 9. Exemplo de grafos: grafos K e L (centralidade de intermediação)

Na Figura 9, o grafo L apresenta grau de intermediação igual a 0 (zero). Isso porque, no caminho entre quaisquer dois vértices, nenhum outro vértice se repete. Por exemplo, no caminho entre os vértices A e B, tem-se: {A,D,F,B}, e no caminho entre B e A, tem-se: {B,E,C,A}. Nenhum vértice, desconsiderando a origem e o destino (vértices A e B), são repetidos. No grafo M o vértice V2 esta presente em 8 menores caminhos. Desta forma, o conjunto de intermediação dos vértices do grafo M é: V1=0, V2=8, V3=0, V4=4, V5=0, V6=0. E o grau de intermediação do grafo M é 8.

2.6.4. Centralidade de Eficiência (*Efficiency Centrality*)

A medida de centralidade de eficiência foi proposta por Hage e Haray em (Hage & Harary, 1995). Esta medida é baseada no cálculo da excentricidade de cada vértice pertencente ao grafo. O cálculo de excentricidade é baseado solução do problema de caminho entre todos os pares de vértices. Com o cálculo dessa centralidade é possível obter, diâmetro, raio, centro e periferia do grafo.

Como exemplo, a Figura 10 mostra um grafo H conexo e não dirigido com vértices $V=\{v_1, v_2, v_3, v_4, v_5, v_6\}$ e arestas $E=\{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$. A excentricidade de um vértice v_i é a maior entre as distâncias mínimas desde este vértice

até qualquer outro vértice v_j do grafo. Para calcular a excentricidade de cada vértice, é então necessário encontrar o menor caminho partindo de um vértice para todos os demais. Desta maneira, considerando o grafo mostrado da Figura 10, é possível construir a Tabela 1, contendo as menores distâncias entre todos os vértices e suas respectivas excentricidades Ɛ.

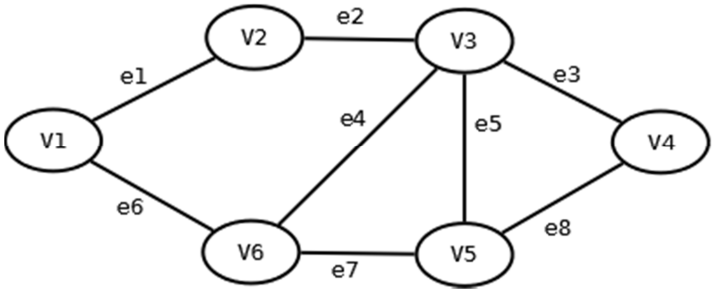


Figura 10. Exemplo de grafo: grafo H (centralidade de eficiência)

Tabela 1. Distância entre todos os vértices do grafo H

v1 v1 a v2 = 1 v1 a v3 = 2 v1 a v4 = 3 v1 a v5 = 2 v1 a v6 = 1 Ɛ (v1) = 3	v2 v2 a v1 = 1 v2 a v3 = 1 v2 a v4 = 2 v2 a v5 = 2 v2 a v6 = 2 Ɛ (v2) = 2	v3 v3 a v2 = 1 v3 a v1 = 2 v3 a v4 = 1 v3 a v5 = 1 v3 a v6 = 1 Ɛ (v3) = 2
v4 v4 a v2 = 2 v4 a v3 = 1 v4 a v1 = 3 v4 a v5 = 1 v4 a v6 = 2 Ɛ (v4) = 3	v5 v5 a v1 = 2 v5 a v3 = 1 v5 a v4 = 1 v5 a v2 = 2 v5 a v6 = 1 Ɛ (v5) = 2	v6 v6 a v2 = 2 v6 a v1 = 1 v6 a v4 = 2 v6 a v5 = 1 v6 a v3 = 1 Ɛ (v6) = 2

Tem-se então que, de acordo com a Tabela 1, o conjunto das excentricidades dos vértices do grafo H, Ɛ = (3, 2, 2, 3, 2, 2). De posse desse conjunto é possível definir medidas de centralidade como diâmetro, raio, centro e periferia do grafo. O **diâmetro** D é o maior valor do conjunto Ɛ, ou seja, D(H) = 3. Por sua vez, o **raio** é o menor valor do conjunto, ou seja, R(H) = 2. O **centro do grafo** é formado por todos os vértices que possuem excentricidade igual ao raio, neste caso C(H) = {v2, v3, v5, v6}. A **periferia** P é formada por todos os vértices que possuem excentricidade igual ao diâmetro, neste

caso P(H) = {v1, v4}. Essas serão as medidas calculadas nesta pesquisa.

2.7. Considerações finais do capítulo

Este capítulo apresentou a definição formal de um grafo e alguns dos tipos de grafos mais conhecidos. Definiram-se também algumas das medidas de centralidade em grafos propostas da literatura, tais como centralidade de grau, de proximidade, de intermediação e de eficiência, que acabou sendo a escolhida neste trabalho. O capítulo definiu também o conceito de caminho em um grafo e apresentou as diversas variantes do problema de menor caminho, essencial para o cálculo da centralidade de eficiência.

O próximo capítulo apresenta conceitos básicos de processamento paralelo e alguns dos algoritmos paralelos para grafos mais conhecidos.

Capítulo 3. Processamento paralelo

Este capítulo introduz conceitos e definições básicas sobre processamento paralelo. No que diz respeito à infra-estrutura de hardware, o capítulo apresenta uma taxonomia das arquiteturas paralelas mais conhecidas. Em termos de software, o texto define o que é um algoritmo paralelo e apresenta uma classificação de acordo com a organização das tarefas executadas em paralelo e de como a troca de informações ocorre entre elas. Completando o embasamento teórico, o capítulo define as métricas de *speedup* e eficiência, utilizadas para avaliar o desempenho de um determinado algoritmo paralelo.

Além destes conceitos gerais, o capítulo aborda também os algoritmos paralelos para grafos mais conhecidos, fazendo ênfase no *caminho mais curto para todos os pares*, que é o problema estudado neste trabalho para calcular a excentricidade de cada nó. Por fim, texto apresenta os modelos específicos que tratam de problemas de Big Data e estabelece um paralelo entre os modelos MapReduce e BSP.

3.1. Sistemas de computação paralela

Um sistema de computação paralela consiste em uma coleção de processadores interconectados que colaboram na solução de um único problema. Tipicamente estes sistemas são utilizados para solucionar problemas onde o processamento sequencial é inviável ou quando há restrições de tempo para produzir os resultados (Coulouris, Dollimore, & Kindberg, 2007).

A arquitetura de um sistema de computação paralela pode ser classificada utilizando a taxonomia proposta por Flynn (Flynn, 1972). Esta classificação é realizada de acordo com o número de fluxos de dados e instruções.

- *Single Instruction Single Data* (SISD): representa todos os computadores que possuem apenas um único fluxo de instrução e dados. Computadores classificados como SISD não exploram qualquer tipo de paralelismo.

- *Single Instruction Multiple Data* (SIMD): neste modelo são incluídos computadores com vários processadores que são capazes de executar a mesma instrução sobre um conjunto de dados independente em cada processador. Processadores vetoriais e, mais recentemente, GPUs, aderem a este modelo.
- *Multiple Instruction Single Data* (MISD): todos os computadores capazes de executar diversas instruções sobre uma mesma sequência de dados são classificados como MISD. Esta arquitetura é pouco utilizada pelo fato de não existirem muitas aplicações práticas. O exemplo mais famoso é o computador que controla o *Space Shuttle*, que permite executar o mesmo cálculo de diversas maneiras e depois verificar se os resultados concordam entre si, de forma a se proteger de falhas bizantinas.
- *Multiple Instruction Multiple Data* (MIMD): cada processador é capaz de executar diferentes instruções sobre diferentes fluxos de dados. Esta arquitetura é a mais comumente utilizada em computadores paralelos de grande escala. Sistemas distribuídos convencionais também podem ser classificados como MIMD, assim como, mais recentemente, computadores com processadores multi-core. Em 2006, os 10 supercomputadores mais rápidos do mundo aderiam a esta arquitetura. A literatura especializada considera ainda duas subdivisões para esta categoria:
    - SPMD (*Single Program, Multiple Data*): os processadores executam o mesmo programa sobre dados diferentes, mas sem a obrigação de sincronizar a cada passo do programa como acontece nas arquiteturas SIMD e
    - MPMD (*Multiple Program, Multiple Data*): os processadores executam pelo menos dois programas diferentes sobre dados diferentes.

3.1.1. Arquiteturas de memória compartilhada

Computadores da arquitetura MIMD (*Multiple Instruction Multiple Data*) também podem ser classificados de acordo com o tipo de compartilhamento de memória. Um sistema fortemente acoplado (*tightly coupled system*) compartilha um único espaço de endereçamento de memória entre todos os processadores, enquanto em um sistema fracamente acoplado (*loosely coupled system*) cada processador possui seu próprio espaço de endereçamento de memória.

Para sistemas fortemente acoplados os processadores utilizam redes de interconexão para acessar informações na memória. Considerando a localização física da memória, o acesso a uma posição pode ser uniforme ou não. As arquiteturas de acesso

uniforme a memória são chamadas de UMA (*Uniform Memory Access*) enquanto as arquiteturas com acesso não uniforme a memória são chamadas de NUMA (*Non-Uniform Memory Access*).

Na arquitetura UMA o espaço de endereçamento de memória é único para todos os processadores. Desta forma, o tempo gasto por um processador para acessar qualquer palavra de memória do sistema é idêntico (Coulouris, Dollimore, & Kindberg, 2007), (Kumar, 2002). A Figura 11, nas letras a e b, ilustram como é realizado o acesso a memória de computadores que utilizam esta arquitetura.

Na arquitetura NUMA a memória global é fracionada em módulos. Cada processador utiliza um desses módulos como memória local. Desta forma, o tempo de acesso a uma palavra de memória do módulo local é menor que o tempo de acesso a uma palavra de memória de um módulo remoto. A Figura 11, letra c, ilustra o acesso a memória utilizando esta arquitetura.

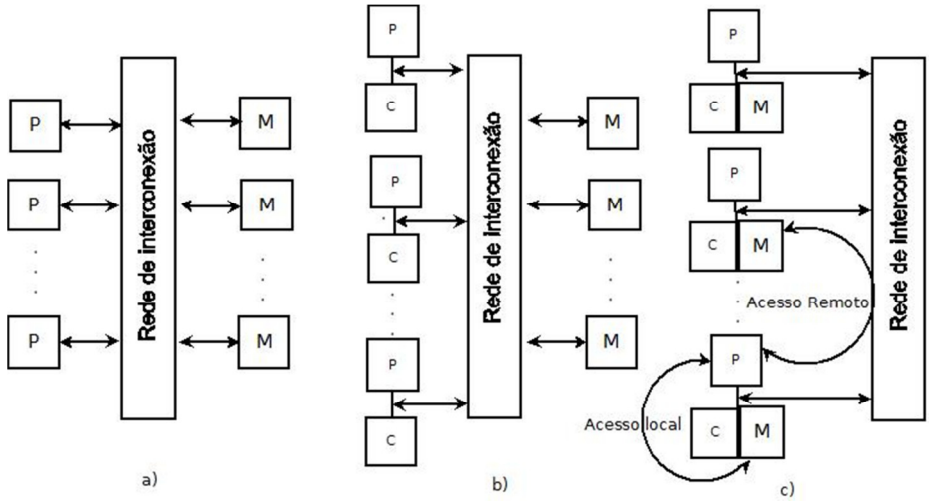


Figura 11. Arquitetura de acesso a memória compartilhada (Kumar, 2002)
(a) UMA sem cache (b) UMA com cache (c) NUMA com cache.

A arquitetura UMA é mais simples e intuitiva, o que implica que é mais fácil de implementar softwares para ela. No entanto, na medida em que a quantidade de processadores aumenta a rede de interconexão tende a se sobrecarregar, o que limita sua escalabilidade. Redes de interconexão mais complexas, como a crossbar e a omega, contornam este problema mas seu custo é elevado. Na arquitetura NUMA a existência de

módulos de memória local diminui a quantidade de dados que trafegam pela rede de interconexão ao custo de transferir parte da complexidade para os programadores, que passam a se preocupar com a localização física dos seus dados.

3.2. Algoritmos paralelos

Um algoritmo paralelo é composto por várias sequências de instruções, ou *tarefas*, que são executadas simultaneamente em unidades de processamento independentes. Eventualmente, estas sequências podem sincronizar seus fluxos de execução e trocar informações entre si, usando para isso os mecanismos de sincronização e comunicação disponíveis na infraestrutura computacional onde o algoritmo é executado.

3.2.1. Modelos de Algoritmos Paralelos

Os algoritmos paralelos podem ser classificados de acordo com a natureza de cada uma das tarefas e a forma em que os dados são divididos entre elas. A classificação mais ampla e conhecida estabelece apenas duas categorias, apresentadas a seguir.

- **Paralelismo de dados**: cada tarefa executa a mesma sequência de instruções sobre conjuntos de dados diferentes. Algoritmos com estas características podem ser eficientemente executados em arquiteturas SIMD ou MIMD/SPMD. Uma aplicação paralela em que cada tarefa executa o mesmo processamento em registros diferentes de uma tabela em um banco de dados é um exemplo típico deste tipo de paralelismo (Kumar, 2002).
- **Paralelismo de tarefas**: neste tipo de paralelismo o programa é dividido em tarefas que executam sequências de instruções diferentes sobre dados que podem ser iguais ou diferentes entre si. Algumas implementações paralelas de algoritmos genéticos, que tentam chegar na solução usando em paralelo diferentes abordagens sobre os mesmos dados, exemplificam este tipo de paralelismo (Kumar, 2002).

Os algoritmos paralelos também podem ser classificados de acordo com a forma em que seus componentes são organizados e colaboram entre si.

- **Pool de tarefas (Mestre-Escravo)**: um processo mestre divide o processamento em tarefas pequenas que são distribuídas entre um conjunto de processos escravos. Os resultados parciais gerados pelos escravos são coletados e

combinados pelo mestre para produzir o resultado final do processamento. Usualmente o processamento de cada tarefa individual é independente (Kumar, 2002). Um exemplo típico deste tipo de aplicações é o projeto SETI@Home (Anderson, Werthimer, Cobb, Korpela, & Lebofsky, 2000), que examina sinais captados por radares à procura de provas da existência de inteligência extraterrestre;

- **SPMD**: este modelo aproveita características intrínsecas aos dados (por exemplo, se eles estão organizados em forma de grafo, matriz, cubo, etc.) para dividi-los entre várias tarefas que os processam simultaneamente. Estas tarefas podem comunicar-se entre si para chegar ao resultado final. Exemplos incluem plataformas como Hama (Apache Hama, 2013) que criam uma tarefa para cada nó de um grafo e executa o mesmo algoritmo em cada nó, permitindo a comunicação entre nós vizinhos a intervalos regulares.
- ***Pipeline***: as tarefas são organizadas em uma fila, de forma tal que a saída de uma tarefa é usada como entrada da próxima, simulando uma linha de produção em uma fábrica. Cada tarefa representa um estágio do *pipeline*. A medida que o fluxo de dados trafega pelos estágios do *pipeline*, cada tarefa executa seu processamento sobre ele (Kumar, 2002). Exemplos incluem aplicações que processam conjuntos de imagens aplicando um filtro diferente em cada estágio do pipeline.
- **Dividir e conquistar**: o problema é dividido em subproblemas menores, que são resolvidos em paralelo e cujos resultados são combinados para formar a solução final. Diferente do modelo mestre-escravo, as tarefas criadas desta forma podem gerar uma árvore de tarefas de vários níveis. Este modelo está normalmente associado a soluções recursivas como aquelas aplicadas em algoritmos paralelos de ordenação de dados.
- **Paralelismo especulativo**: neste tipo de paralelismo um mesmo problema é resolvido usando algoritmos diferentes ou o mesmo algoritmo com pontos iniciais diferentes. O resultado final é aquele oferecido pela primeira tarefa a terminar. Em algoritmos genéticos, este processo pode ser repetido até achar uma solução satisfatória.
- **Híbrido**: aplicações paralelas de grande escala podem ter a necessidade de utilizar mais do que um dos paradigmas básicos apresentados acima. O modelo resultante deste caso é considerado híbrido.

A próxima seção apresenta alguns algoritmos conhecidos na literatura que oferecem soluções paralelas para os problemas de grafos apresentados.

3.3. Algoritmos paralelos para grafos

Cáceres et. al. em (Cáceres, et al., 1997) apresentam uma lista de problemas para grafos com suas soluções sequenciais e paralelas. Dentre os problemas relacionados estão o *list ranking*, caminho de Euler, ancestral comum mais baixo, componentes conexos e os problemas de caminhos (*single source shortest path* e *all pairs shortest path*).

3.3.1. List ranking

O problema *list ranking* consiste em determinar a posição, ou *rank*, de cada elemento de uma lista ligada. O primeiro elemento da lista deve receber o valor 1, o segundo, 2, e assim sucessivamente até chegar ao último elemento. Este problema é equivalente a calcular o *prefix sum* de uma lista ligada onde todos os elementos têm valor 1.

O algoritmo sequencial para este problema é linear ($O(n)$) e consiste em percorrer a lista do início ao fim atribuindo a cada elemento valores correspondentes com a sua posição na lista. Os algoritmos paralelos, no entanto, são mais complexos e tem merecido estudos detalhados na literatura. O mais eficiente deles deve-se a Anderson e Miller (Anderson & Miller, 1990) e tem uma complexidade de tempo de $O(\log(n))$ em $O(n/\log(n))$ processadores (i.e. a quantidade total de passos do algoritmo é *n*, consistente com o algoritmo sequencial.)

3.3.2. Caminho de Euler

O problema do caminho de Euler consiste em realizar um percurso sobre um grafo conectado não dirigido G=(V,E) onde cada aresta deverá ser percorrida exatamente uma vez. Se o vértice origem e destino são distintos então o resultado do algoritmo é chamado de caminho de Euler. Se o caminho começa e termina no mesmo vértice o resultado é conhecido como circuito de Euler ou *Euler tour*.

A melhor solução sequencial conhecida para este problema tem ordem $O(m + n)$ para um grafo com *m* arestas e *n* vértices (Cáceres & Nasu, 2003). Soluções paralelas

como a apresentada em (Cáceres & Nasu, 2003) consistem em encontrar todos os ciclos do grafo e posteriormente "costurar" aqueles ciclos que possuam algum vértice em comum, gerando uma complexidade de tempo de $O({}^{m+n}/_{p})$ e $O(\log(p))$ rodadas de comunicação, onde *p* é o número de processadores utilizados na execução paralela.

3.3.3. Ancestral Comum mais Baixo

Dada uma árvore T, um ancestral comum de dois vértices *u* e *v* é um vértice *w* tal que *w* tem simultaneamente *u* e *v* como descendentes. Um nó *w* ancestral comum de *u* e *v* será o mais baixo se for o ancestral comum mais distante da raiz de T.

Para achar o ancestral comum mais baixo entre dois vértices há vários algoritmos sequenciais documentados na literatura (Aho, Hopcroft, & Ullman, 1973). Os algoritmos mais eficientes pré-processam a árvore em tempo linear e conseguem responder as consultas para determinar o ancestral comum mais baixo entre dois nós em tempo constante.

Dentre os algoritmos paralelos sugeridos para resolver este problema destaca-se o proposto em (Cáceres, et al., 1997), que utiliza a solução para o Euler Tour para determinar o ancestral comum mais baixo numa árvore de *n* nós e utilizando *p* processadores. Este algoritmo é completado em $O(\log(p))$ rodadas de comunicação, cada uma delas com complexidade computacional em cada processador de $O({}^{n}/_{p})$.

3.3.4. Componentes conexos

Seja G um grafo não dirigido G = (V,E) e dois vértices $u, v \in V$, diz-se que *u* e *v* estão conectados se G contém um caminho entre *u* e *v*. Um componente conexo é definido como um subgrafo de G tal que todos seus vértices estão conectados entre si e não há outros vértices no supergrafo conectados a eles.

Os algoritmos sequenciais para calcular componentes conexos são bastante conhecidos e usualmente são baseados em busca em largura ou profundidade para atingir seu objetivo. A complexidade computacional destas abordagens é de $O(n + m)$ para um grafo com *n* vértices e *m* arestas, visto que todos os vértices e arestas do grafo precisam ser visitados.

Cáceres et. al propõem em (Cáceres, et al., 1997) uma abordagem paralela que, assim como a proposta para o cálculo do ancestral comum mais baixo, transforma o problema original em um problema de cálculo do Euler Tour. O resultado é um algoritmo capaz de gerar o resultado em $O(\log(p))$ rodadas de comunicação, cada uma delas com complexidade computacional em cada processador de $O({(n+m)}/{p})$.

3.3.5. Caminho mais curto de todos os pares (*all pairs shortest path*)

O problema de caminho mais curto de todos os pares (*All Pairs Shortest Path*, ou APSP) consiste em resolver o SSSP (*Single Source Shortest Path*) para todos os vértices do grafo. Uma das maneiras de calcular centralidade em grafos é resolver o APSP e usar os valores calculados para determinar a excentricidade de cada vértice e, consequentemente, o raio e o diâmetro do grafo.

Tiskin em (Tiskin, 2001) apresenta uma solução paralela para o problema de menor caminho entre todos os pares utilizando um modelo criado por ele e chamado de BSPRAM. Este modelo é uma combinação das características do modelo BSP (*Bulk Synchronous Parallel*) e o modelo PRAM (*Parallel Random Access Machine*). No modelo proposto, cada processador tem uma memória local rápida, e há uma memória global compartilhada que pode ou não ser acessada concorrentemente pelos processadores.

O algoritmo proposto por Tiskin está fortemente baseado em matrizes, que tem complexidade espacial de $O(n^2)$ para um grafo com n vértices. Além disso, há a necessidade de acessar a memória global compartilhada, o que pode criar um gargalo quando o número de nós de processamento aumenta. Estas características fazem com que a solução proposta não seja adequada para tratar grafos grandes.

A próxima seção apresenta as medidas de desempenho utilizadas neste trabalho.

3.4. Medidas de desempenho

Quando o objeto de estudo é medir desempenho de programas paralelos, a literatura recente tem utilizado principalmente as métricas de *speedup* (aceleração) e eficiência. A seguir serão definidas essas métricas.

3.4.1. Speedup

O *speedup*, ou fator de aceleração, representa o quanto um trabalho é mais rápido se executado em um computador utilizando alguma melhoria em relação ao computador original (Gustafson, 1992), (Patterson & Hennessy, 2005). Para o caso de um algoritmo paralelo ele é definido como a razão do tempo de execução do algoritmo em um único processador pelo tempo de execução do algoritmo paralelo utilizando *N* processadores (Eager, Zahorjan, & Lazowska, 1989).

$$Speedup = \frac{Tempo_de_execução_em_um_processador}{Tempo_de_execução_em_N_processadores}$$

O resultado obtido pelo cálculo de *speedup* pode ser classificado em 3 categorias. Essa classificação consiste em dizer se o *speedup* é sublinear, linear ou superlinear (Alba, 2002). O *speedup* é dito sublinear quando o resultado obtido é menor do que o número de processadores utilizados. Já o *speedup* linear indica que a medida que novos processadores vão sendo adicionados a aceleração do processamento cresce na mesma proporção. O *speedup* superlinear indica que a aceleração obtida com a adição de novos processadores é maior que o número de processadores adicionados. A Figura 12 ilustra a diferença entre as categorias (Tomassini, 2005).

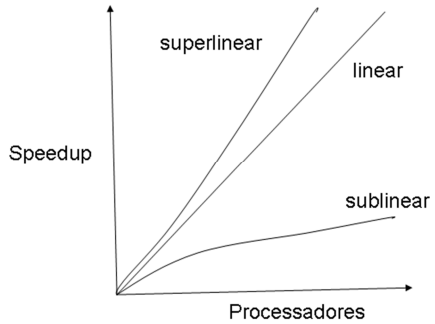


Figura 12. *Speedup* sublinear, linear e superlinear

Quando o tempo do algoritmo com processamento paralelo é comparado com o algoritmo sequencial mais eficiente conhecido na literatura, esse *speedup* é classificado como absoluto (*Strong Speedup*). Quando o tempo do algoritmo com processamento paralelo é comparado com o próprio algoritmo paralelo executado em um único computador, o *speedup* é chamado de relativo (*Weak Speedup*) (Gustafson, 1992). Para todos os testes realizados neste trabalho utilizou-se o *speedup* absoluto (*Strong Speedup*).

3.4.2. Eficiência

A eficiência é usada para determinar o grau de utilização dos recursos computacionais durante a execução do algoritmo. Para calcular a eficiência basta dividir o valor do *speedup* pelo número de processadores utilizados (Eager, Zahorjan, & Lazowska, 1989). A fórmula utilizada para se obter resultados utilizando esta métrica é a que segue:

$$Eficiência = \frac{Speedup}{número_de_processadores}$$

Se o resultado do cálculo da eficiência é igual a 1 (um), isso indica que o *speedup* é linear. Se o resultado é maior que 1 (um), o *speedup* é superlinear e menor que 1 (um) é sublinear. Em sistemas paralelos a linearidade raramente é conseguida. Isto se deve a que, com a adição de novos recursos computacionais, cresce também o custo de gerenciamento e comunicação. A demora introduzida pelas tarefas de gerenciamento é comumente chamada de *overhead*.

Outro ponto que impacta na eficiência é que existem partes do algoritmo que são inerentemente sequenciais e não podem ser paralelizadas. A lei de Amdahl (Patterson & Hennessy, 2005) prova que, para um algoritmo com uma parcela s ($0 \leq s \leq 1$) estritamente sequencial o *speedup* estará limitado por $1/s$, independentemente da quantidade de processadores utilizados.

A próxima seção apresenta os modelos utilizados para Big Data.

3.5. Modelos específicos para Big Data

Esta seção apresenta modelos paralelos que foram desenvolvidos para trabalhar com grandes massas de dados: Processamento de Eventos Complexos (*Complex Event Processing*), Processamento de Fluxo (*Stream Processing*), GraphLab, MapReduce e BSP.

3.5.1. Processamento de eventos complexos (*Complex Event Processing* – CEP)

O registro de informações de log em um sistema, a leitura de um sensor de temperatura, ou mesmo uma transação de negócio, podem ser considerados como eventos em um sistema de informação (Luckham, 2002). Esses eventos podem ser classificados em dois tipos: eventos simples, quando sua área de escopo não gera outros eventos (e.g. passar dentro do limite de velocidade em uma lombada eletrônica) e eventos complexos, que podem gerar uma reação em cadeia criando vários outros eventos (e.g. passar acima da velocidade em uma lombada eletrônica).

Operações sobre eventos incluem leitura, criação, transformação, e exclusão de eventos (Etzion & Niblett, 2011). O processamento de eventos complexos considera não apenas a cronologia dos eventos, mas também as suas relações causais (Cugola & Margara, 2012). Neste tipo de processamento o sistema considera os eventos em uma base de dados e os eventos que estão sendo criados em tempo de execução, capturados a partir de dispositivos móveis, sensores, etc (Wasserkrug, Gal , Etzion, & Turchin, 2008).

A Figura 13 ilustra a arquitetura de um servidor que implementa o modelo *Complex Event Processing* (CEP). Nela é possível observar que os eventos têm origem a partir de diversas fontes. Todos os eventos passam por adaptadores de entrada e são submetidos às consultas. As consultas abstraem os eventos do fluxo submetido, passando o resultado pelos adaptadores de saída. Os eventos identificados são então enviados para diferentes dispositivos para serem analisados.

Os eventos gerados pelas fontes de eventos são eventos simples. As consultas extraem os eventos complexos e os encaminham a um dispositivo de destino de eventos.

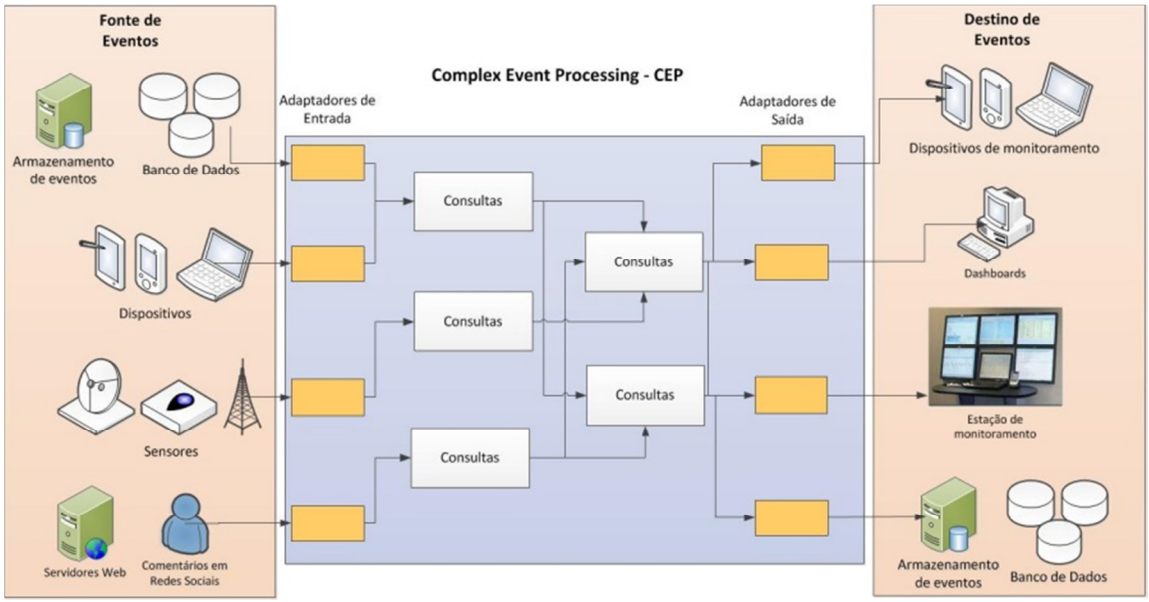


Figura 13. Arquitetura de um CEP Server

3.5.2. Stream Processing (processamento de fluxo)

Em determinados cenários os bancos de dados tradicionais são insuficientes para fazer análise de dados em tempo real (Luckham, 2002). Um exemplo típico é a execução de consultas contínuas em fluxos de dados de entrada onde a resposta é válida apenas na janela atual de tempo. Com a janela de tempo deslizando para frente a resposta da consulta também deveria ser atualizada para incluir os novos eventos e excluir os antigos (Luckham, 2006). Este cenário é chamado de *Data Management Streams* (DSM) e originou o modelo *Stream Processing* (processamento de fluxo).

O modelo *Stream Processing* (processamento de fluxo) surgiu com a ideia de executar consultas contínuas em fluxos de dados de entrada. Este modelo é focado na consulta de alta velocidade e aplicação de algoritmos matemáticos em fluxo de dados. Um exemplo da aplicação desse modelo aparece nos *feeds* do mercado de ações em sistemas financeiros onde o objetivo é o processamento em tempo real de informações sobre negócios.

Para atingir baixa latência, o sistema deve ser capaz de processar grandes quantidades de mensagens sem a necessidade de realizar operações de armazenamento. Operações de armazenamento geralmente possuem um alto custo computacional, pois são necessários acessos em banco de dados, registro de logs, e principalmente, acesso ao

disco físico (Stonebraker, Çetintemel, & Zdonik, 2005). Para muitas aplicações de processamento de fluxo não é aceitável, nem necessário, exigir uma operação tão demorada antes que o processamento de mensagens possa ocorrer . Em vez disso, as mensagens devem ser processadas "em fluxo", em tempo real, quando os eventos realmente acontecem.

A Figura 14, ilustra como funciona o modelo *Stream Processing*. Nela é possível observar que *feeds* (ou notícias, eventos) em tempo real são tidos como entrada. Todo processamento realizado pelo modelo é feito em memória, a utilização do disco (ou mesmo de banco de dados) é opcional. Como resultado do processamento do modelo, são geradas ações de alerta.

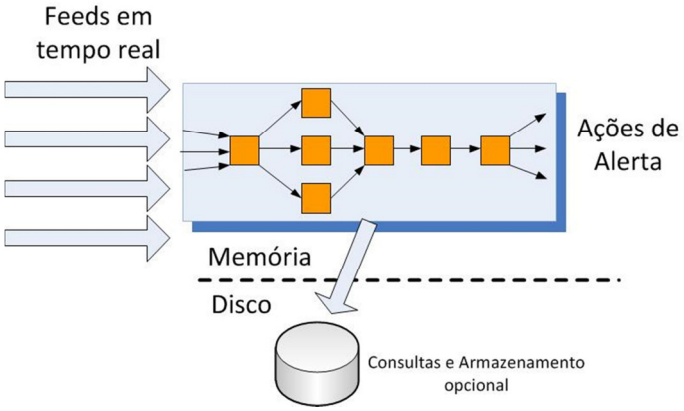


Figura 14. Stream Processing (Stonebraker, Çetintemel, & Zdonik, 2005)

Sistemas que utilizam o modelo *Stream Processing* possuem algumas características que são inerentemente diferentes de outros modelos (Stonebraker, Çetintemel, & Zdonik, 2005): os dados deverão estar em constante movimento; as consultas devem ser realizadas no fluxo de dados; o sistema deve ser capaz de lidar com imperfeições, por exemplo, dados em atraso, falta de dados, dados fora de ordem; os resultados devem ser previsíveis, ou seja, os resultados do processamento devem ser determinísticos e repetíveis; o sistema deve integrar armazenamento e transmissão de dados; garantir a segurança e disponibilidade dos dados; ser capaz de realizar o particionamento e escalonamento de aplicativos de maneira autônoma; e processar e responder instantaneamente;

3.5.3. Modelo MapReduce

MapReduce é um modelo de programação criado por Google para processar conjuntos grandes de dados (Dean & Ghemawat, 2008). O modelo consiste em duas funções, *map* e *reduce* que, combinadas, permitem dividir os dados a serem processados entre um conjunto potencialmente grande de nós de processamento e posteriormente gerar o resultado final a partir dos resultados parciais de cada nó. A função *map* recebe um conjunto de pares chave/valor de entrada, e produz um conjunto de pares chave/valor intermediários. A implementação do modelo MapReduce é então encarregada de agrupar todos os valores intermediários que têm a mesma chave e entregá-los à função *reduce*, que produz o conjunto final de pares chave/valor.

A principal vantagem do modelo MapReduce é que permite ao programador se concentrar no problema específico a ser resolvido, enquanto os detalhes de distribuição de carga, tolerância a falhas e processamento de resultados intermediários são responsabilidade da implementação do modelo. Um problema que comumente é apresentado como primeiro exemplo de aplicação MapReduce é o de contar a quantidade de vezes que cada palavra aparece em um conjunto de documentos potencialmente grande. O pseudocódigo das funções *map* e *reduce* para esta aplicação é mostrado a seguir (Dean & Ghemawat, 2008).

Algoritmo 1. Função Map

```
1 map(String key, String value):
2 // key: document name
3 // value: document contents
4   for each word w in value:
5           EmitIntermediate(w, "1");
```

Algoritmo 2. Função Reduce

```
1 reduce(String key, Iterator values):
2 // key: a word
3 // values: a list of counts
4         int result = 0;
5         for each v in values:
6                 result += ParseInt(v);
7         Emit(AsString(result));
```

A Figura 15 ilustra como é o funcionamento do modelo MapReduce para os algoritmos 1 e 2. Nela é possível observar que primeiro o texto é fragmentado em partes e é passado para a fase Map, nesta fase cada palavra é emitida com um contador associado. Na fase reduce ele agrupa todas as palavras que possuem o mesmo contador contando-as. Cada palavra com a quantidade de ocorrências no texto é oferecida como resultado.

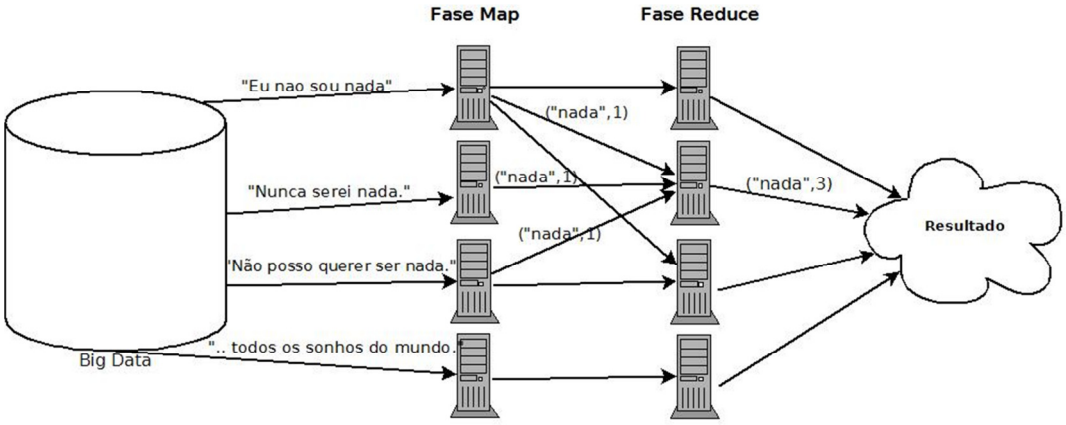


Figura 15. Fases Map e Reduce (contador de palavras)

A função *map* emite cada palavra com um contador associado, que contém o valor "1" nesta implementação mais simples. A função *reduce* soma os contadores de todas as entradas geradas para cada palavra.

A Figura 16 mostra a execução de uma aplicação MapReduce na plataforma originalmente criada pelo Google para executar suas aplicações (Dean & Ghemawat, 2008). A sequência de ações mostrada na Figura 16 é explicada a seguir.

1. A biblioteca MapReduce divide a entrada em M segmentos.

2. Dentre as cópias do programa há um mestre e vários trabalhadores. O mestre atribui as tarefas *map* e *reduce* aos trabalhadores que estiverem ociosos.

3. Um trabalhador encarregado de executar uma tarefa *map* lê o conteúdo do segmento de entrada correspondente, faz *parsing* dos pares chave/valor e passa cada um à função *map* definida pelo usuário. Os pares intermediários chave/valor produzidos pela função *map* são armazenados na memória.

4. Periodicamente os dados armazenados no *buffer* são particionados em R regiões pela

função de particionamento e gravados no disco local. A localização destes dados é informada ao mestre, que é encarregado de encaminhar esta informação aos trabalhadores que executam as tarefas *reduce*.

5. Quando um trabalhador encarregado de executar uma tarefa *reduce* é notificado pelo mestre sobre a localização dos dados intermediários, ele usa chamadas de procedimento remoto para ler os dados dos discos locais onde foram executadas as tarefas *map*. Uma vez lidos, os dados são ordenados pelas chaves intermediárias, usando ordenação em disco se forem muito grandes.

6. O trabalhador *reduce* percorre os dados intermediários já ordenados e para cada chave única encontrada, ele passa a chave e o conjunto correspondente de valores intermediários para a função *reduce* definida pelo usuário. A saída de cada função *reduce* é anexada a um arquivo final de saída.

7. Quando todas as tarefas *map* e *reduce* forem concluídas, a chamada *MapReduce* é concluída e a execução retorna ao código do usuário.

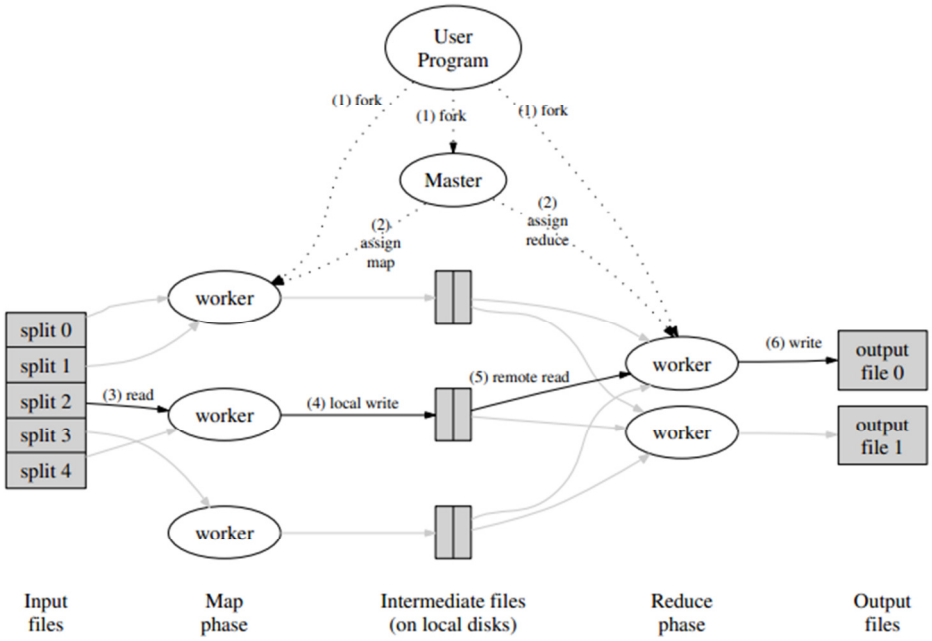


Figura 16. Visão geral execução MapReduce (Dean & Ghemawat, 2008)

Além da implementação original mostrada por Dean e Ghemawat em (Dean & Ghemawat, 2008), há outras plataformas que também implementam o modelo MapReduce, tais como BashReduce (Frey, 2006), Disco Project (Schmidt, 2009) e Apache Hadoop (White, 2009). Dentre elas, o Apache Hadoop tem sido a mais bem sucedida, gerando até distribuições comerciais como Cloudera (Cloudera Ask Bigger Questions, 2013), Hortonworks (Kopytoff, 2013) e MapR (MapR Technologies, 2013).

A seguir é apresentado o *framework* Apache Hadoop.

Apache Hadoop

O *framework* Apache Hadoop desenvolvido pela *Apache Software Foudation* é o mais conhecido dentre os *frameworks* de livre utilização que implementam o modelo MapReduce. Escrito em Java, o Hadoop é utilizado para a execução de aplicativos em *clusters* de computadores. Grandes empresas, como Yahoo, utilizam o Hadoop na implementação de partes do motor de busca (Yahoo Launches World's Largest Hadoop Production Application, 2008). De modo geral, Hadoop é utilizado como solução para problemas que envolvam data *warehouse*, inteligência de negócio, aplicações analíticas e mídias sociais.

O Hadoop é hoje uma coleção de subprojetos correlatos que juntos fornecem uma infraestrutura completa para computação distribuída em grande escala. Embora o Hadoop seja mais conhecido por MapReduce e seu sistema de arquivos distribuídos (HDFS - *Hadoop Distributed File System*), os outros subprojetos fornecem serviços complementares. A Tabela 2, apresenta os subprojetos que compõem o Apache Hadoop, e uma breve descrição de cada subprojeto de acordo com White e Lam (White, 2009) e (Lam, 2010).

Tabela 2. Subprojetos que compõem o Apache Hadoop (White, 2009)

<table>
<tr><td>Pig</td><td>Chukwa</td><td>Hive</td><td>Hbase</td></tr>
<tr><td>MapReduce</td><td colspan="2">HDFS</td><td>Zookeeper</td></tr>
<tr><td colspan="2">Core</td><td colspan="2">Avro</td></tr>
</table>

- **Core**: conjunto de componentes e interfaces para sistemas de arquivos distribuídos e I/O (*Input/Output*) universal.
- **Avro**: sistema de serialização de dados eficiente com armazenamento de dados persistente.
- **MapReduce**: responsável pelo processamento distribuído de dados e ambiente de execução que roda em grandes *clusters*.
- **HDFS**: sistema de arquivos distribuído que funciona em grandes *clusters*.

- **Pig**: linguagem de fluxo de dados e ambiente de execução para explorar conjuntos de dados muito grandes.
- **HBase**: banco de dados distribuído orientado a coluna.
- **ZooKeeper**: serviço distribuído de administração altamente disponível, fornecendo primitivas como bloqueios distribuídos.
- **Hive**: sistema para *data warehouse* distribuído que gerencia os dados armazenados no HDFS e fornece uma linguagem de consulta baseada em SQL.
- **Chukwa**: sistema para análise de dados distribuídos.

3.5.4. Modelo *Bulk Synchronous Parallel* (BSP)

O modelo *Bulk Synchronous Parallel* (BSP), proposto em (Valiant, Reeve, Zenith, & Wiley, 1989), define um computador abstrato que age como uma ponte entre *hardware* e *software*. Este tipo de modelo, conhecido como *bridging models*, se caracteriza por conter elementos de arquitetura de *hardware* e de *software*. O exemplo mais conhecido deste tipo de modelo é a máquina de Von Neumann, que teve papel protagônico no desenvolvimento da computação ao permitir que desenvolvedores de *hardware* e *software* pudessem trabalhar de forma relativamente isolada, focando apenas nas características do modelo descrito.

Um computador BSP consiste em uma coleção de processadores, cada um com memória privada, conectados por uma rede de comunicação. Um algoritmo BSP consiste de uma sequência de *supersteps*. Um *superstep* compreende três componentes, explicados a seguir.

- **Computação concorrente**: cada processador participante executa processos independentes e assíncronos que usam apenas dados armazenados nas suas respectivas memórias locais.
- **Comunicação**: os processos trocam informações entre eles usando primitivas unilaterais *put* e *get*, que acessam e modificam posições de memória de outros processadores.
- **Sincronização global**: quando um processo atinge este ponto, ele aguarda até que todos os outros processos concluam suas ações de comunicação.

A Figura 17 apresenta o fluxo de um algoritmo de BSP. Nela é possível observar

um algoritmo BSP com cinco *supersteps* executados em cinco processadores. As linhas verticais indicam a computação local de cada processador; as setas indicam a comunicação com outro processador. As linhas marcadas com a palavra *comp* indicam *supersteps* de computação. As linhas marcadas com a palavra *comm* indicam *supersteps* de comunicação. No primeiro *superstep* de comunicação o processador P(0) envia dados para todos os outros processadores. Cada *superstep* é terminado por uma sincronização global.

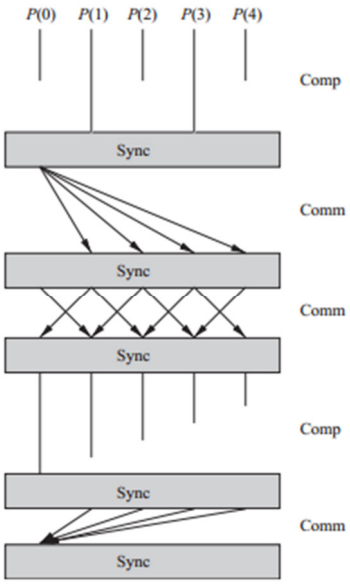


Figura 17. Passos de um algoritmo BSP (Bisseling, 2004)

A seguir uma breve descrição de duas implementações do modelo BSP.

Pregel

Pregel é um *framework* desenvolvido pela Google que é utilizado especificamente para resolver problemas que envolvam grafos (Malewicz, et al., 2010). Este *framework* é baseado no modelo BSP proposto em (Valiant, Reeve, Zenith, & Wiley, 1989). A entrada para uma computação Pregel é um grafo direcionado em que cada vértice é unicamente identificado por uma *string,* chamada de *vertex identifier*. Cada vértice está associado a um valor modificável definido pelo usuário. As arestas direcionadas estão associadas com seus respectivos vértices fontes e são constituídas por um valor modificável definido pelo usuário e um identificador de vértice alvo (Malewicz, et al., 2010).

Uma execução típica em Pregel começa com a inicialização do grafo, seguida de uma sequência de *supersteps* separados por pontos de sincronização global até o algoritmo terminar, e finalmente a geração da saída. A saída de um programa Pregel é o conjunto de valores explicitamente emitidos pelos vértices. Frequentemente, a saída é um grafo isomorfo ao grafo de entrada, mas esta não é uma propriedade necessária do sistema, porque é possível adicionar e remover vértices e arestas durante o cálculo. Um algoritmo de agrupamento, por exemplo, pode gerar um pequeno conjunto de vértices desconectados selecionados a partir de um grande grafo. (Malewicz, et al., 2010).

A Figura 18 ilustra estes conceitos através de um exemplo simples: dado um grafo fortemente conectado, onde cada vértice contém um valor, procura-se propagar o maior valor para cada vértice. Em cada *superstep*, qualquer vértice que aprendeu um valor maior de suas mensagens envia para todos os seus vizinhos. Quando não há mais vértices para mudar em um *superstep*, o algoritmo termina.

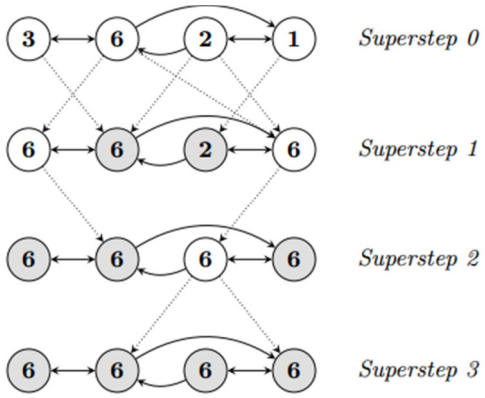


Figura 18. Exemplo máximo valor (Malewicz, et al., 2010)

Cada vértice em Pregel pode votar para parar durante um *superstep*. Após votar para parar, o vértice permanece em estado inativo a menos que seja ativado por uma nova mensagem. O algoritmo termina quando todos os vértices estão simultaneamente inativos e não há mensagens em trânsito. A Figura 19, mostra uma máquina de estados para cada vértice.

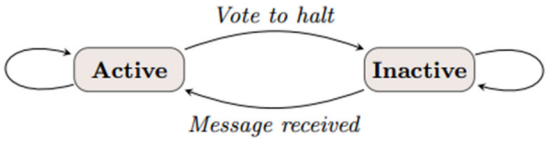


Figura 19. Máquina de estados do vértice (Malewicz, et al., 2010)

Apache Hama

Hama (Apache Hama, 2013) é um *framework* para computação distribuída baseado no modelo BSP. Assim como o Hadoop, o Hama é mantido pela *Apache Software Foundation* e, portanto, usufrui de parte do ecossistema de projetos mostrado na Tabela 2. Em particular, o *Hadoop Distributed File System* (HDFS) é extensivamente utilizado para armazenar os dados a serem processados e os resultados gerados pelos aplicativos Hama.

A Figura 20 mostra o fluxo de execução de um aplicativo Hama.

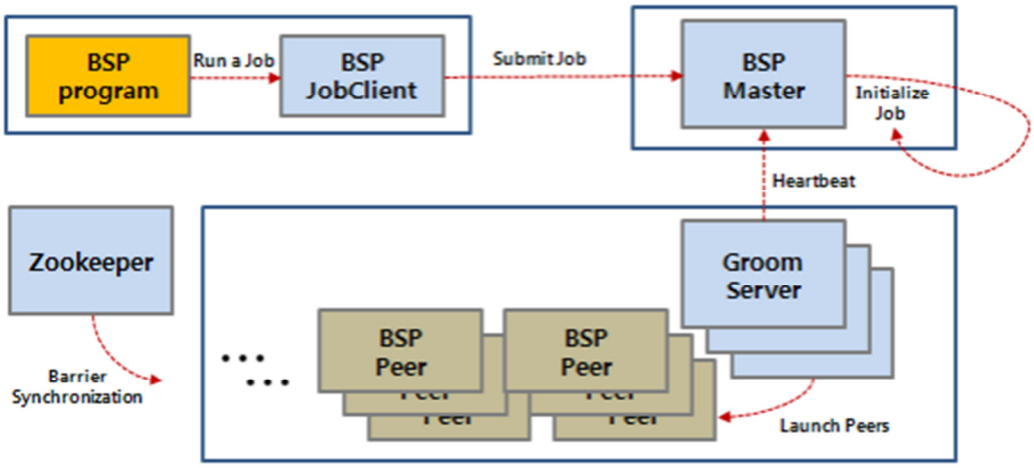


Figura 20. Arquitetura Hama (Apache Hama, 2013)

A solicitação para iniciar a execução de um aplicativo em Hama é feita pelo programa do usuário através de uma cópia local do *BSP JobClient*, cuja função é a de contatar o *BSP Master*, submeter o pedido de execução e aguardar pelos resultados. Ao receber um pedido o *BSP Master* contata os *Groom Servers* e aloca as tarefas a serem executadas por cada um. Cada *Groom Server* é responsável por pelo menos uma instância de *BSP Peer*, que é encarregada de executar de fato o código próprio do aplicativo. Os *Groom Servers* contatam periodicamente o *BSP Master* através de um

heartbeat para informar seu estado, que inclui a quantidade de tarefas sendo executadas, quantidade de memória livre, entre outras informações necessárias para que o *BSP Master* possa executar suas funções. Por fim, o *Zookeeper* é encarregado de implementar as primitivas de sincronização global que fazem parte do modelo BSP.

3.6. Modelo MapReduce X Modelo BSP

O modelo MapReduce tem revolucionado a maneira com que a análise de grandes massas de dados é feita por apresentar um modelo escalável e tolerante a falhas. Em seu funcionamento cada trabalhador executa uma sequencia de tarefas Map e Reduce, após concluir uma tarefa a memória local do trabalhador é apagada. Isto implica que não é possível associar dados a um trabalhador e acessá-los em rodadas diferentes. Dados que devem ser compartilhados entre rodadas tem que ser armazenados na memória global. Não é possível garantir que um determinado trabalhador receba dados específicos para processar durante a fase Map.

O modelo de processamento *Bulk Synchronous Parallel* é um modelo bastante conhecido. Ele tem como característica principal a troca de mensagens entre os pares para informar estado, gerenciar e sincronizar o processamento. Este modelo se mostrou eficiente na implementação de diversos algoritmos paralelos (Cáceres, et al., 1997). Em seu funcionamento cada processador executa uma sequencia de *supersteps*. A memória local do processador é preservada entre *supersteps*. Cada processador pode se comunicar diretamente com qualquer outro processador oferecendo controle total sobre como os dados são distribuídos e mantidos no sistema.

Embora os dois modelos difiram na maneira com que tratam as informações, eles também possuem algumas semelhanças. Pace propõe dois teoremas que provam que os dois modelos são equivalentes para algumas classes de problemas (Pace, 2012). Dentre seus resultados destaca-se a conclusão de que existem alguns problemas para os quais o modelo MapReduce não apresenta resultados satisfatórios.

Pace analisou os problemas de *Sorting*, *Dense Matrix Multiplication* e *Breadth First Search* (busca em largura) analisando a eficiência de uma implementação utilizando o modelo MapReduce. Os teoremas propostos por ele provam que o problema

Breadth First Search não poderá ser implementado de maneira eficiente utilizando o modelo MapReduce.

Outro indicativo de que o modelo MapReduce não escala bem para todos os problemas é mostrado por Jain e Mallozi em (Jain & Mallozzi, 2010). Neste artigo, os autores apresentam uma implementação mal sucedida do problema do Caixeiro Viajante utilizando MapReduce.

Algoritmos para grafos tendem a ser paralelizados usando o vértice como unidade de processamento e trocando informações entre vértices. O Hama adota esta estratégia, criando um *BSP Peer* para cada vértice do grafo capaz de manter um estado local e trocar informações com vértices adjacentes. Estas características indicam que o modelo BSP é de fato mais apropriado para implementar algoritmos paralelos em grafos do que o MapReduce.

3.7. Considerações finais do capítulo

Este capítulo apresentou conceitos básicos de computação paralela e descreveu as taxonomias mais comumente aceitas na literatura para arquiteturas e algoritmos paralelos. O capítulo enumerou também os algoritmos paralelos existentes para problemas em grafos, dando especial ênfase ao problema do caminho mais curto para todos os pares, que é essencial para calcular as medidas de centralidade que são o foco deste trabalho. Por fim, o capítulo apresentou as técnicas e modelos de computação paralela utilizados especificamente para resolver problemas classificados como *Big Data*, focando a discussão nas vantagens e desvantagens dos modelos MapReduce e BSP para resolver problemas em grafos.

O próximo capítulo apresenta os algoritmos que tratam com o problema de centralidade em grafos grandes utilizando os modelos MapReduce ou BSP.

Capítulo 4. Algoritmos para cálculo de centralidade em grafos grandes

Este capítulo apresenta os algoritmos HADI (*Hadoop based DIameter estimator*) e HEDA (*Hadoop-based Exact Diameter Algorithm*), ambos baseados no modelo MapReduce, e o FastGDC (*Fast Graph Diameter Calculator*), baseado no modelo BSP. Todos os algoritmos apresentados neste capítulo resolvem o problema de centralidade em grafos.

4.1. *Hadoop based DIameter estimator* (HADI)

O HADI (*Hadoop based DIameter estimator*) é um algoritmo que calcula o raio e o diâmetro de grafos utilizando técnicas de aproximação. O algoritmo é implementado seguindo o modelo MapReduce e executado usando Hadoop. Ele possui uma versão mais simples chamada HADI-*naive* e uma versão mais performática que gasta menos recursos chamada HADI-*plain*, que possui melhoras na fase Reduce garantindo um melhor desempenho. Nos testes realizados neste trabalho com o algoritmo HADI foi utilizada a versão *plain*.

O HADI usa o algoritmo de Flajolet-Martin para estimar o diâmetro efetivo do grafo, que é a quantidade mínima de saltos necessária para conectar pelo menos 90% dos nós do grafo. Este algoritmo consegue diminuir a complexidade de tempo e espaço necessárias para o cálculo exato da centralidade em grafos ao custo de fornecer apenas resultados aproximados. A complexidade de tempo do HADI para um grafo de diâmetro d com n nós e m arestas processado usando M máquinas é de $O(\frac{d(m+n)}{M}\log\frac{m+n}{M})$, e a complexidade de espaço de $O((m+n)\log n)$.

4.2. *Hadoop-based Exact Diameter Algorithm* (HEDA)

Como alternativa ao HADI, Nascimento e Murta propuseram o HEDA (*Hadoop-based Exact Diameter Algorithm*) (Nascimento & Murta, 2012). Diferente do HADI, o

HEDA determina o diâmetro exato a partir do cálculo do caminho mais curto entre todos os pares de nós do grafo. Este problema, chamado de *All Pairs Shortest Path*, é implementado em (Nascimento & Murta, 2012) através da busca em largura partindo de cada um dos nós do grafo. Nesta implementação, os autores utilizaram o modelo MapReduce e o *framework* Hadoop.

A implementação do HEDA utiliza dois arquivos. O primeiro deles contém o conjunto de vértices e suas respectivas arestas, enquanto o segundo contém as distâncias e a situação dos vértices sobre o processamento. O formato destes dois arquivos é mostrado na Tabela 3, que descreve o grafo da Figura 21.

O arquivo de arestas lista cada um dos vértices, seguido de um TAB e uma lista de vértices vizinhos separados por uma vírgula. Assim, a entrada 1TAB2,5 descreve o vértice de número 1, que tem arestas que o ligam aos vértices 2 e 5. O arquivo de distâncias, por sua vez, descreve a distância entre cada par de vértices, assim como o estado de processamento de cada vértice (1 para "em processamento" e 2 para "processado"). A título de exemplo, a entrada 1.5TAB1|1 indica que a distância atual entre os vértices 1 e 5 é de 1, mas que esse par de vértices está ainda em processamento. Já a entrada 3.4TAB1|2 indica que a distância entre os vértices 3 e 4 é de 1, e que o processamento para esse par de vértices já terminou.

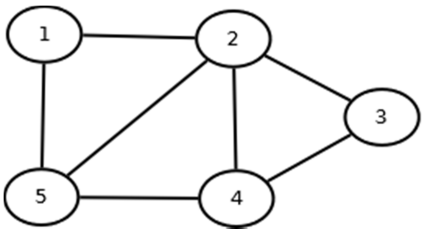


Figura 21. Exemplo de grafo: grafo I (HEDA) (Nascimento & Murta, 2011)

Para descrever o funcionamento do algoritmo HEDA basta observar a sequência de iterações apresentadas na Tabela 3. Inicialmente o arquivo de distâncias é preenchido com uma entrada para cada vértice do grafo com o valor de 1 para a distância e indicando que essas entradas ainda estão em processamento. Para explicar o funcionamento do algoritmo nos passos seguintes, basta observar a entrada 1.1TAB0|1,

associada ao vértice 1 no estado inicial.

Tabela 3. Exemplo Hadoop: iterações do algoritmo HEDA (Nascimento & Murta, 2011)

Formato do arquivo de Arestas	Formato do arquivo de distâncias / Estado inicial	Primeira iteração	Segunda iteração	Terceira iteração
1TAB2,5	1.1TAB0|1	1.1TAB0|2	1.1TAB0|2	1.1TAB1|2
2TAB1,3,4,5	2.2TAB0|1	1.2TAB1|1	1.2TAB1|2	1.2TAB1|2
3TAB2,4	3.3TAB0|1	1.5TAB1|1	1.3TAB2|1	1.3TAB2|2
4TAB2,3,5	4.4TAB0|1	2.2TAB0|2	1.4TAB2|1	1.4TAB2|2
5TAB1,2,4	5.5TAB0|1	2.1TAB1|1	1.5TAB1|2	1.5TAB1|2
		2.3TAB1|1	2.2TAB0|2	2.1TAB1|2
		2.4TAB1|1	2.1TAB1|2	2.2TAB1|2
		2.5TAB1|1	2.3TAB1|2	2.3TAB1|2
		3.3TAB0|2	2.4TAB1|2	2.4TAB1|2
		3.2TAB1|1	2.5TAB1|2	2.5TAB1|2
		3.4TAB1|1	3.3TAB0|2	3.1TAB2|2
		4.4TAB0|2	3.1TAB2|1	3.2TAB1|2
		4.2TAB1|1	3.2TAB1|2	3.3TAB1|2
		4.3TAB1|1	3.4TAB1|2	3.4TAB1|2
		4.5TAB1|1	3.5TAB2|1	3.5TAB2|2
		5.5TAB0|2	4.4TAB0|2	4.1TAB2|2
		5.1TAB1|1	4.1TAB2|1	4.2TAB1|2
		5.2TAB1|1	4.2TAB1|2	4.3TAB1|2
		5.4TAB1|1	4.3TAB1|2	4.4TAB1|2
			4.5TAB1|2	4.5TAB1|2
			5.5TAB0|2	5.1TAB1|2
			5.1TAB1|2	5.2TAB1|2
			5.2TAB1|2	5.3TAB2|2
			5.3TAB2|1	5.4TAB1|2
			5.4TAB1|2	5.5TAB1|2

A cada passo do algoritmo o vértice á direita do ponto é examinado, procurando no arquivo de arestas se há outros vértices do grafo que possam ser acessíveis a partir dele e que ainda não tenham sido examinados. Desta forma, ao processar a entrada 1.1TAB0|1 na primeira iteração aparecem as entradas 1.2TAB1|1 e 1.5TAB1|1, que correspondem às arestas (1,2) e (1,5) extraídas do arquivo de arestas. A entrada original passa a ser 1.1TAB1|2, para refletir o fato de que já foi processada.

No passo seguinte, ao examinar a entrada 1.2TAB1|1, geram-se as entradas 1.3TAB2|1 e 1.4TAB2|1, decorrentes das arestas (2,3) e (2,4) do arquivo de arestas.

Cabe observar que a aresta (2,5) é desconsiderada, visto que já existe um caminho entre 1 e 5 representado pela entrada 1.5TAB1|1. Pela mesma razão, as arestas (5,2) e (5,4) geradas pela entrada 1.5TAB1|1 são desconsideradas, já que os nós 3 e 4 foram alcançados a partir do nó 3. Por fim, as entradas 1.2TAB1|1 e 1.5TAB1|1 também tem seu estado de processamento atualizado para 2 (processado).

Na última iteração as entradas 1.3TAB2|1 1.4TAB2|1 são processadas mas não geram novas entradas, sendo apenas atualizado seu estado para 2 ("processado"). Este algoritmo é repetido para cada uma das entradas no arquivo de distâncias até o ponto em que não há entradas com estado de processamento 1 (em processamento).

A complexidade de tempo é $(\frac{n(n+m)}{M})$ para um grafo processado usando M computadores. Esta expressão é decorrente da complexidade de executar uma busca em largura partindo de um nó, que é de $(n + m)$, visto que todos os nós e arestas precisam ser visitados. Como o algoritmo executa uma busca em largura partindo de cada um dos nós do grafo, a complexidade passa a ser $n(n + m)$. Por fim, pelo fato do trabalho ser dividido entre M computadores, a expressão final é de $\frac{n(n+m)}{M}$.

4.3. Algoritmo FastGDC

O algoritmo apresentado neste trabalho é chamado de FastGDC (*Fast Graph Diameter Calculator*). Ele tem como objetivo efetuar o cálculo exato de medidas de centralidade em grafos, principalmente diâmetro e raio. O algoritmo FastGDC utiliza o modelo *Bulk Synchronous Parallel* (BSP) na manipulação de grafos grandes. Dentre os *frameworks* que implementam este modelo, destacam-se o GPS (Salihoglu & Widom, 2012), Giraph (Apache Incubator Giraph, 2012), e o Apache Hama. Este último acabou sendo escolhido para os testes por aparentar estar em um estágio mais avançado de desenvolvimento.

O algoritmo FastGDC é baseado em busca em largura. O Algoritmo 3 apresenta o algoritmo FastGDC desenvolvido para calcular as medidas de centralidade de um grafo usando o modelo BSP. O método Compute apresentado pelo algoritmo é executado em cada vértice pertencente ao grafo a cada *superstep*.

Algoritmo 3. Método Compute ()

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

```
1 Método compute
2 // First superstep. Create and initialize distance_vector.
3 // It will contain "0" for the current node, "1" for neighbours
4 // of the current node and "MAX_INTEGER" for all other nodes
5   if (superstep == 0)
6           distance_vector = new Vector(NODE_COUNT);
7           for each index in distance_vector
8                   If (index == current_node)
9                           distance_vector[index] = 0;
10                  else if (isNeighbour(index, current_node))
11                          distance_vector[index] = 1;
12                  else
13                          distance_vector[index] = MAX_INTEGER;
14                  endif
15          endfor
16  else
18           // shouldHalt will indicate whether this node made any changes in this
19          // superstep. Otherwise, this node will vote to halt the processing.
20          // When all nodes vote to halt, the processing stops.
21          shouldHalt = true;
22          // Now he process every message received looking for path to nodes
23          // that are shorter than the ones he have.
24          for each message in messages_received_from_neighbouring_nodes
25                  received_distance_vector = getContents(message);
26                  // he got the distance vector from a neighbouring node. Now he analyze
27                  // every entry looking for better paths
28                  for each index in received_distance_vector
29                          current_entry = distance_vector[index];
30                          remote_entry = received_distance_vector[index];
31                          if (remote_entry < current_entry)
32                                  // a better path has been found. Update the distance vector
33                                  // and the eccentricity of this node. Set  shouldHalt to false,
34                                  // since the current node was updated.
35                                  shouldHalt = false;
36                                  distance_vector[index] = received_distance_vector[index];
37                                  if (getEccentricity(current_node) < remote_entry)
38                                          setEccentricity(current_node, remote_entry);
39                                  endif
40                          endif
41                  endfor
42          endfor
43
44          // If nothing was updated, vote to halt processing.
45          if (shoulHalt)
46                  voteToHalt(current_node);
47          else
48                  // Start the communication phase. Prepare a distance vector to be sent
49                  // to neighbouring nodes which contains the current distance values for this node plus 1.
50                  // In other words, it contains the distance from the neighbouring nodes to any other
51                  // node when using the current node as the first step of the path.
52                  distance_vector_to_send = distance_vector
53                  for each index in distance_vector_to_send
54                          distance_vector_to_send[index] = distance_vector[index]+1;
```

```
55                  endfor
56
57                  // Send the calculated vector to neighbouring nodes.
58                  for each node in neighbour_nodes
59                              send(node, distance_vector_to_send);
60                  endfor
61          endif
62    endif
63 end Método Compute
```

O Algoritmo 3 é executado para cada vértice do grafo. Ele começa no *superstep* 0, que é dedicado apenas a fazer as inicializações necessárias para o processamento. Em particular, um vetor de distâncias é criado e inicializado contendo 0 na entrada correspondente ao vértice atual (chamado de *current_node* no algoritmo), 1 para todos os vizinhos do vértice atual e o valor máximo possível para números inteiros em todas as outras entradas.

A partir deste ponto começa uma sequência de *supersteps* que contém duas partes: uma dedicada a processar as mensagens recebidas dos vértices vizinhos e outra dedicada a enviar mensagens a estes mesmos vértices. Cada mensagem recebida de um vértice vizinho contém um vetor de distâncias que reflete o tamanho do caminho desde o vértice vizinho até qualquer outro vértice. Como cada vizinho envia uma mensagem similar, o vértice atual é informado do comprimento de todos os caminhos possíveis até qualquer outro vértice. Usando esta informação, o vértice atual é capaz de calcular o menor caminho até todos os outros vértices do grafo. Inicialmente muitos destes valores ainda não foram calculados e, portanto, o processo precisa ser repetido até que não haja novas atualizações.

Caso não tenha havido nenhuma atualização no vetor de distâncias do vértice atual após processar todas as mensagens, ele emite seu voto para que o processamento termine. Caso contrário, ele envia a todos seus nós vizinhos seu próprio vetor de distâncias incrementado em 1 em cada entrada. Quando o algoritmo termina, cada vértice contém sua excentricidade no grafo. O menor valor de excentricidade é o raio do grafo e o maior valor seu diâmetro.

A Figura 22 ilustra o funcionamento do algoritmo FastGDC através de um diagrama de blocos. Na Figura 23 é possível observar como é realizada a paralelização do algoritmo FastGDC. No passo 0 o grafo é copiado para o HDFS. No passo 1 o

programa de usuário é submetido ao computador mestre (*BSPMaster*). O *BSPMaster* consulta se o grafo utilizado no programa está no HDFS e, caso positivo, copia o grafo em memória no passo 2. No passo 3 o mestre contata os outros computadores do *cluster* e aloca para cada um pelo menos um vértice para ser processado utilizando o método compute. O passo 4 corresponde à sincronização dos *supersteps* utilizando o servidor *Zookeeper*. No passo 5 o resultado da execução do método compute realizado pelos *GroomServers* é informado ao *BSPMaster*. O passo 6 grava o resultado no HDFS.

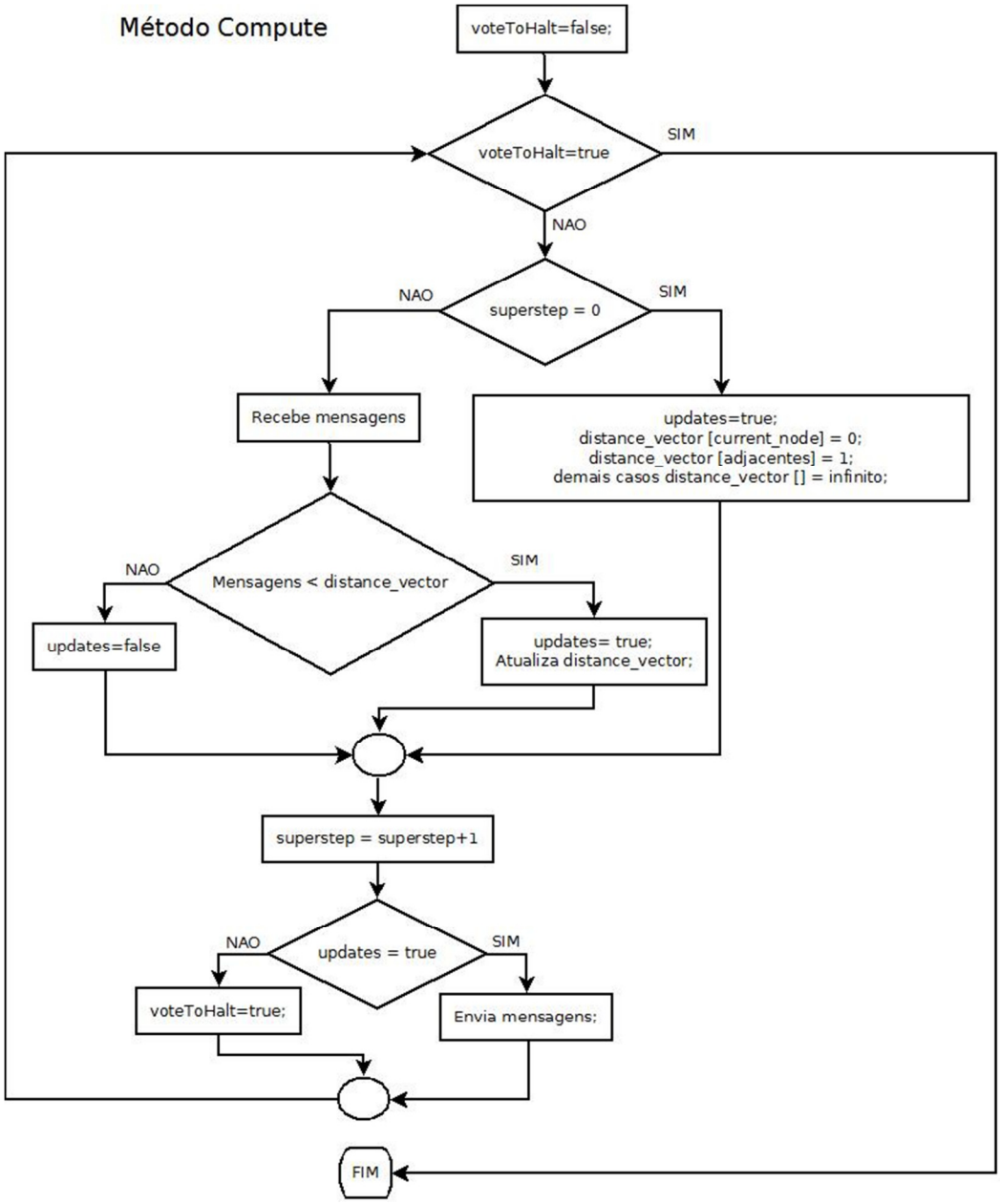


Figura 22. Diagrama de blocos algoritmo FastGDC

Em termos de complexidade computacional para um grafo com n nós e m arestas processados usando M computadores, o FastGDC é equivalente ao HEDA ($\frac{n(n+m)}{M}$), já que precisa executar uma busca em largura partindo de cada nó do grafo. A complexidade de espaço, no entanto, é menor, visto que cada nó armazena apenas dois vetores de distâncias que contém uma entrada para cada vértice do grafo, o que gera uma complexidade de $O(n^2)$.

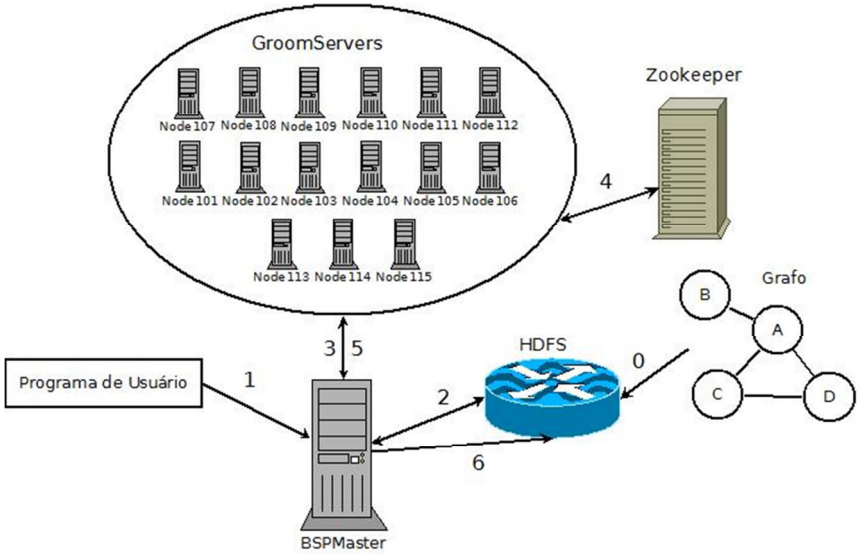


Figura 23. Paralelização do algoritmo FastGDC

A Tabela 4 mostra um comparativo entre os algoritmos HADI, HEDA, FastGDC, Sequencial, considerando algumas das suas características mais relevantes.

Tabela 4. Tabela comparativo entre os algoritmos

Algoritmos	Modelo	Persistência	Comunicação	Baseado em
HADI	MapReduce	HDFS	Através do HDFS	Algoritmo Flajolet-Martin
HEDA	MapReduce	HDFS	Através do HDFS	Busca em Largura
FastGDC	BSP	HDFS	Troca de mensagens	Busca em Largura
Sequencial	Sequencial	Sistemas de arquivos do sistema operacional	N/A	Busca em Largura

4.4. Considerações finais do capítulo

Este capítulo apresentou os algoritmos HEDA, HADI e FastGDC para o cálculo de medidas de centralidade em grafos. O algoritmo HEDA é baseado no modelo MapReduce e utiliza o framework Hadoop para ser executado, oferecendo uma solução exata para o problema. O algoritmo HADI também é baseado no modelo MapReduce e executado utilizando o framework Hadoop, mas oferece uma solução aproximada. O algoritmo FastGDC é uma implementação de busca em largura baseado no modelo BSP e executado utilizando o framework Hama e, assim como o HEDA, calcula as medidas de centralidade de forma exata.

O próximo capítulo mostra como foram planejados e executados os testes comparativos entre os algoritmos HEDA, HADI, FastGDC e Sequencial.

Capítulo 5. Projeto dos experimentos

Este capítulo descreve a metodologia usada nos experimentos, os dados utilizados e a infra-estrutura computacional onde eles foram conduzidos. O objetivo dos experimentos foi comparar os algoritmos HEDA, HADI, FastGDC e Sequencial no cálculo do raio e o diâmetro de um grafo. O código do HEDA foi gentilmente cedido pelos autores, enquanto o algoritmo HADI foi encontrado em uma página web. Já o algoritmo Sequencial e o FastGDC foram desenvolvidos pelo autor deste trabalho.

5.1. Metodologia

Para comparar adequadamente os algoritmos HEDA, HADI, FastGDC e Sequencial dentro das restrições impostas pela infra-estrutura computacional disponível foi projetado um conjunto extensivo de experimentos capazes de avaliar diferentes aspectos do problema sendo resolvido. Assim, foram processados grafos de vários tamanhos e características utilizando diferentes números de computadores a cada execução e comparados os tempos de execução, *speedup* e eficiência de cada solução. O *speedup* calculado foi o **absoluto**, ou seja, utilizando como referência o tempo de execução do algoritmo Sequencial, e não o próprio algoritmo paralelo executado em 1 computador.

5.2. Infraestrutura e ambiente computacional

A infraestrutura utilizada na pesquisa é formada por computadores localizados no Laboratório de *Hardware* da Universidade do Estado de Mato Grosso, Campus Universitário Vale do Teles Pires, Colider – MT. Cada um destes computadores contém processador Intel Core i7 com 8 (oito) núcleos de processamento, 8 (oito) gigabytes de memória RAM, 500 (quinhentos) gigabytes de espaço em disco e estão ligadas em rede por meio de um switch gigabit.

O sistema operacional utilizado no *cluster* desta pesquisa foi o Centos versão 5.9. A versão da Java Virtual *Machine* (JVM) é a Oracle JDK 1.7. O *framework* Hadoop foi instalado na versão 1.0.4. Para o *framework* Hama, foi instalado a versão 0.6.2.

5.2.1. Configurações dos *frameworks*

Para a configuração do Hadoop foram utilizadas as informações disponibilizadas por Noll (Noll, 2012), e os parâmetros recomendados para execução do Hadoop disponibilizados por Nascimento e Murta em (Nascimento & Murta, 2012).

O Hama foi configurado seguindo um documento disponibilizado pelo desenvolvedor do *framework* em (Apache Hama Installation Guide, 2012) o resumo desta configuração pode ser visualizado na Tabela 5.

Tabela 5. Configuração do Apache Hama

conf/hama-site.xml	Neste arquivo é especificando quem será o bsp-master, localizações de diretórios usados pelo Hama e principalmente especificações sobre o HDFS;
conf/groomservers	Neste arquivo foi criada uma lista com o nome de todos os microcomputadores que pertence ao *cluster*;
conf/hama-env.sh	Neste arquivo são configuradas variáveis como JAVA_HOME, HAMA_LOG_DIR, HAMA_HEAPSIZE, HAMA_SSH_OPTS, HAMA_MANAGES_ZK.

Para testar a configuração do *cluster* utilizando o Apache Hama foram executados exemplos de algoritmos disponibilizados pelo desenvolvedor do *framework*, tais como *Shortest Path* (menor caminho de única origem), cálculo de pi e emparelhamentos de grafos bipartidos (*Bipartite Matching*).

5.3. Conjunto de dados que compõem os experimentos

Para avaliação de desempenho dos algoritmos FastGDC, HEDA e HADI foram utilizados grafos sintéticos (Erdos & Renyi, 1959), grafos do *DataSet* da Universidade

de Stanford (Stanford Large Nertwork Dataset Collection, 2012), e grafos que descrevem a topologia da Internet (Internet Research Lab, 2013).

5.3.1. Grafos sintéticos

Os grafos sintéticos foram gerados utilizando o modelo Erdos-Renyi (Erdos & Renyi, 1959). Este modelo também foi utilizado nos artigos originais dos algoritmos HEDA e HADI. Uma peculiaridade dos grafos gerados por esse modelo é que todos os grafos são direcionados. No entanto, os algoritmos comparados na pesquisa trabalham com grafos não direcionados, o que implicou na necessidade de escrever um programa que convertesse grafos direcionados em grafos não direcionados.

Os grafos sintéticos foram utilizados pela facilidade de manipulação do número de vértices e arestas. A Tabela 6 mostra a relação de grafos sintéticos gerados e utilizados nos experimentos. Tentativas de processar grafos maiores se depararam com problemas inerentes ao Hama, discutidos com maior detalhe na seção 6.3.

Tabela 6. Grafos sintéticos

	Quantidade de Vértices					
	5.000	**10.000**	**20.000**	**30.000**	**40.000**	**50.000**
Quantidade de arestas	100.000	100.000	200.000	200.000	200.000	200.000
	200.000	200.000	300.000	300.000	300.000	
	400.000	300.000	400.000			
	600.000	400.000	500.000			
		500.000				
		600.000				

5.3.2. Grafos do *DataSet* de Stanford

A Universidade de Stanford mantem um conjunto de dados (*dataset*), com grafos de vários tipos. Neste *dataset* é possível encontrar grafos direcionados, não direcionados, bipartidos, multigrafos (grafos com várias arestas entre um par de vértices), grafos com características temporais (grafo que representa em qual momento determinada aresta

apareceu no grafo), grafos rotulados (grafos que contem pesos e atributos entre seus vértices e arestas) entre outros. Nos experimentos realizados foram utilizados apenas os grafos não direcionados. A Tabela 7 mostra a relação de grafos utilizados nos experimentos, juntamente com a descrição de cada um.

Tabela 7. Grafos *DataSet* de Stanford (Stanford Large Nertwork Dataset Collection, 2012)

Nome do Grafo	Quant. Vértices	Quant. Arestas	Descrição
ca-HepTh (High-Energy Physics - Theory)	9.877	51.971	Este grafo apresenta colaborações científicas entre os artigos submetidos à Física de Altas Energias na categoria Teoria.
ca-HepPh (High-energy physics citation network)	12.008	237.010	Este grafo apresenta colaborações científicas entre os artigos submetidos à Física de Altas Energias na categoria Fenomenologia.
ca-CondMat (Condense Matter Physics)	23.133	186.936	Este grafo apresenta colaborações científicas entre os artigos submetidos à Física da Matéria Condensada na categoria Matéria Condensada.
email-Enron (Enron email Network)	36.692	367.662	Este grafo apresenta toda a comunicação de e-mail dentro de um conjunto de dados de cerca de meio milhão de e-mails. Estes dados foram originalmente publicados e lançados na web, pela Comissão Federal Regulatória de Energia durante a sua investigação.
ca-AstroPh (Astro Physics collaboration network)	18772	396.160	Este grafo apresenta colaborações científicas entre os artigos submetidos à Astronomia na categoria Física.

5.3.3. Grafos Reais (Internet)

O *Internet Research Lab* (IRL) da Universidade da Califórnia (UCLA) coleta e trata informações reais sobre a topologia da internet. O IRL realiza pesquisas sobre tolerância a falhas em sistemas distribuídos de larga escala, infraestrutura de roteamento da internet, protocolos de roteamento entre domínios (BGP) e os princípios de design de protocolo para sistemas de auto-organização em larga escala (Internet Research Lab, 2013).

O IRL mantem um *dataset* que serve como arquivo histórico da Internet sobre dados de topologia, sendo bastante utilizado para pesquisas acadêmicas. Essa fonte de

dados foi utilizada principalmente por representar informações reais e por estar disponível para *download* de maneira gratuita. O IRL disponibiliza informações classificadas em dias ou meses. Neste trabalho foram utilizados grafos correspondentes aos meses de Março a Julho de 2013, como mostrado na Tabela 8.

Tabela 8. Grafos que descrevem a topologia da Internet.

Mês	Vértices	Arestas
Março	44447	336992
Abril	44729	341470
Maio	44961	350626
Junho	44819	305278
Julho	44457	353768

5.4. Considerações finais do capítulo

Este capítulo descreveu a metodologia, o conjunto de dados e a infra-estrutura computacional onde foram realizados os experimentos reportados neste trabalho. O próximo capítulo apresenta os resultados obtidos com a execução dos algoritmos HEDA, HADI, FastGDC e Sequencial utilizando os grafos descritos na seção 5.3.

Capítulo 6. Resultados experimentais

Este capítulo apresenta os tempos de execução e *speedup* obtidos ao processar cada grafo utilizando os algoritmos Sequencial, FastGDC, HEDA e HADI.

6.1. Tempos de execução

Esta seção apresenta uma análise comparativa dos tempos de execução obtidos para cada algoritmo.

6.1.1. Grafos sintéticos

Esta subseção mostra os tempos de execução obtidos ao processar os grafos sintéticos.

a) Grafos com 5 mil vértices

A Tabela 9 apresenta os resultados numéricos dos tempos de execução de quatro grafos com cinco mil vértices e quantidade variada de arestas. Os tempos de execução são apresentados em minutos.

Tabela 9. Tempos de execução para grafos com 5 mil vértices

Qtd. Arestas	Sequencial	HEDA	HADI	FastGDC
100.000	1,35	12,29	11,89	0,73
200.000	2,60	12,71	10,61	0,67
400.000	5,12	13,36	8,69	1,19
600.000	7,70	13,53	8,72	2,06

A Figura 24 permite apreciar mais claramente os dados da Tabela 9.

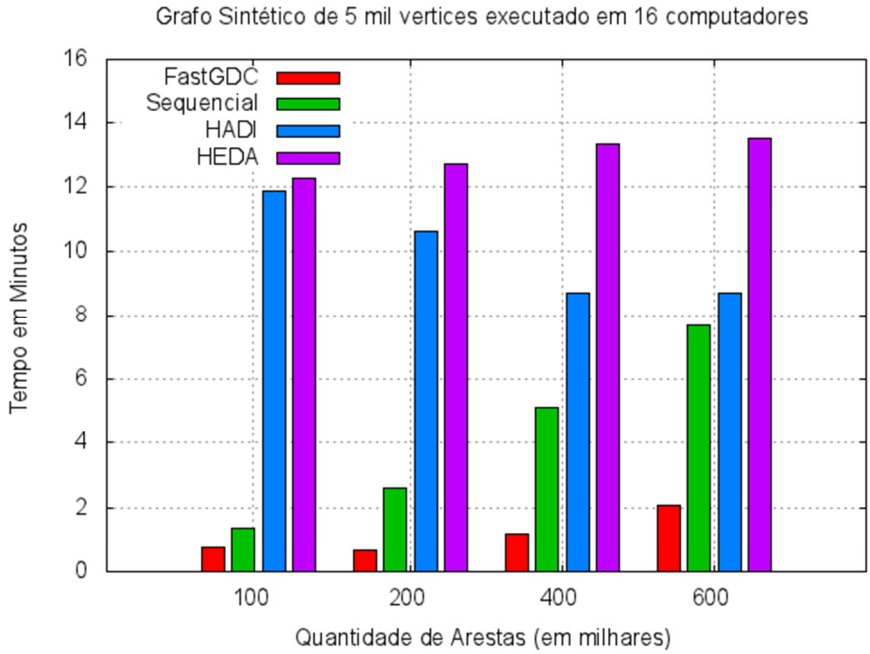


Figura 24. Grafo sintético com 5 mil vértices.

Nota-se na Figura 24 que existe uma leve redução no tempo de execução do algoritmo FastGDC no grafo com 200 (duzentas) mil arestas, em relação ao grafo com 100 (cem) mil arestas. A razão para isso é que o grafo com 200 mil arestas possui diâmetro menor, e consequentemente precisa de um menor número de *supersteps* para ser analisado. O algoritmo Sequencial manteve o crescimento do tempo de execução de acordo com o aumento no número de arestas. O HEDA manteve essa mesma característica, embora tenha tido um tempo de execução superior a todos os outros algoritmos. Já o algoritmo HADI para o grafo com 100 mil arestas teve um tempo similar ao do algoritmo HEDA, mesmo não oferecendo a solução exata para o problema.

A Figura 24 mostra que o FastGDC produz os melhores resultados, apesar de que a solução sequencial é uma alternativa a ser considerada para grafos com esse tamanho. Nenhuma das soluções que utiliza o modelo MapReduce – HEDA e HADI – obtiveram desempenho satisfatório, mesmo o HADI não oferecendo solução exata. No caso destes algoritmos, o tamanho relativamente pequeno do grafo mostrou que mesmo adicionando mais recursos computacionais a sobrecarga de gerenciamento imposta pelo modelo

MapReduce é maior que os ganhos na capacidade de processamento e memória, principalmente no que diz respeito a leitura e escrita no sistema de arquivos.

A Figura 25 mostra o speedup absoluto de cada algoritmo, calculado a partir dos resultados apresentados na Figura 24.

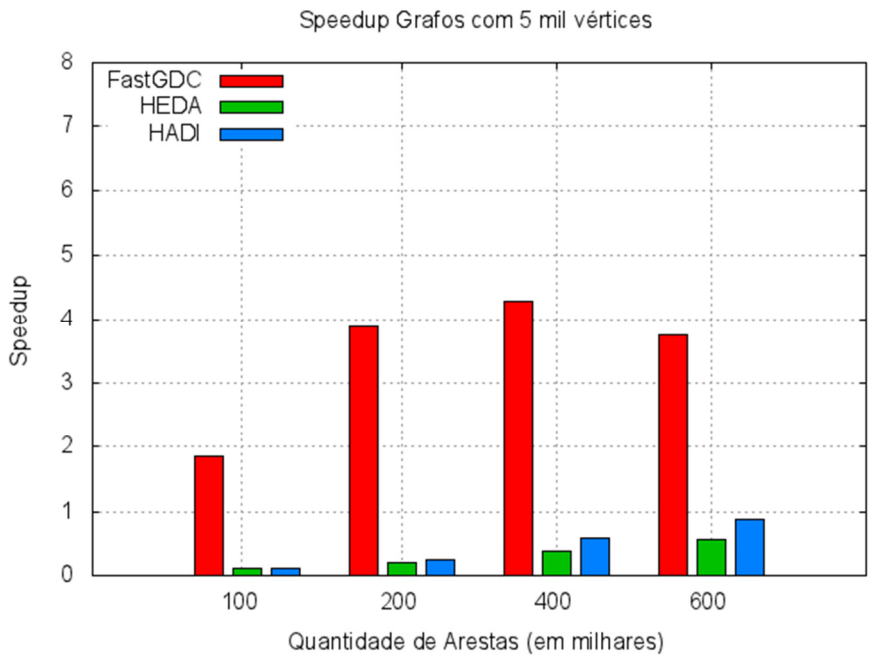


Figura 25. Speedup para grafos com 5 mil vértices.

Os resultados apresentados Figura 25 mostram que o FastGDC aproveitou melhor os recursos computacionais disponíveis. Os outros algoritmos – HEDA e HADI – tiveram *speedup* inferior ao algoritmo FastGDC, mas parecidos entre eles. No grafo com 600 mil arestas o desempenho do algoritmo FastGDC cai em relação ao grafo com 400 mil arestas pois diferente dos grafos anteriores, neste grafo o framework Hama começa a despejar as informações que passam do limite da memória física no disco. Apenas o FastGDC apresenta speedup maior do que 1, indicando que o HEDA e o HADI não são soluções apropriadas para grafos destas dimensões.

b) Grafos com 10 mil vértices

A Tabela 10 apresenta as médias dos tempos de execução em minutos para todos os grafos sintéticos com 10 mil vértices.

Tabela 10.Tempos de execução para grafos com 10 mil vértices

Qtd. Arestas	Sequencial	HEDA	HADI	FastGDC
100.000	4,03	22,49	14,44	1,21
200.000	7,00	24,03	12,51	1,43
300.000	9,93	26,04	10,67	2,13
400.000	13,37	25,17	10,84	2,47
500.000	17,12	24,5	10,69	1,44
600.000	20,83	24,47	9,43	1,55

O gráfico na Figura 26 mostra os dados da Tabela 10. Nela é possível visualizar as médias dos tempos de execução de cada algoritmo para grafos que tem 10 mil vértices e arestas que variam entre 100 mil e 600 mil. O algoritmo Sequencial, nos grafos com 100 mil e 200 mil arestas, desempenha melhor que os algoritmos que utilizam o modelo MapReduce (HEDA e HADI). Para os outros grafos o algoritmo Sequencial ainda consegue superar o HEDA. O tempo do HADI é melhor que do algoritmo Sequencial para os grafos com 400, 500 e 600 mil arestas. O algoritmo FastGDC novamente apresentou o menor tempo de execução para todos os grafos com 10 mil vértices.

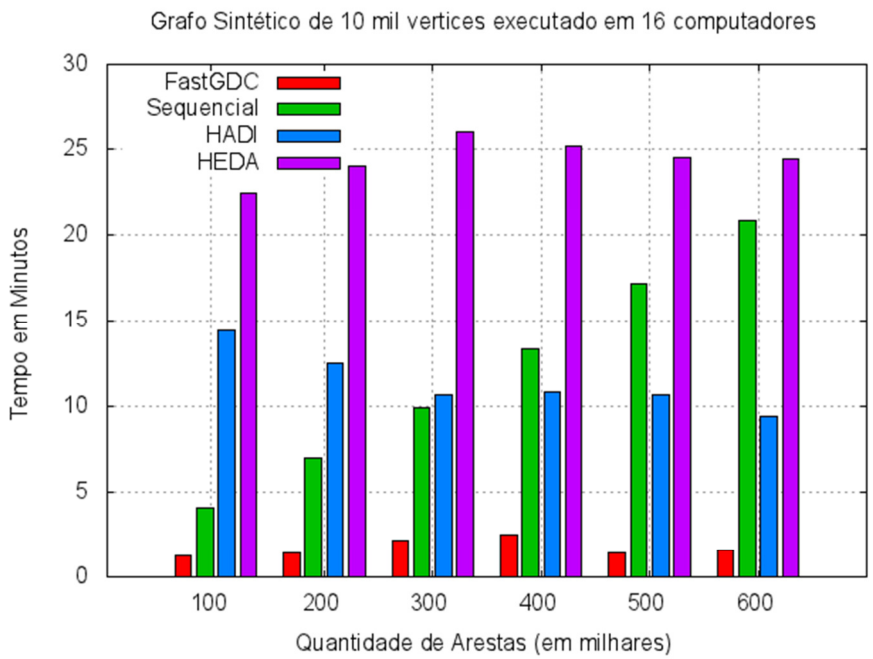


Figura 26. Tempos de execução para grafos com 10 mil vértices.

O FastGDC teve novamente nos grafos com 500 mil e 600 mil arestas um tempo de execução menor que no grafo com 400 mil arestas, devido a que estes grafos possuem menor diâmetro, tal como mostrado na Tabela 11.

Tabela 11. Diâmetro e raio de grafos com 10 mil vértices

Qtd. Arestas	Diâmetro Exato	Raio Exato
100.000	7	5
200.000	5	4
300.000	4	4
400.000	5	3
500.000	4	3
600.000	4	3

A Figura 27 apresenta o *speedup* absoluto calculado a partir dos resultados mostrados na Tabela 10. O algoritmo FastGDC obteve o maior *speedup* em todos os grafos, seguido pelo HADI, com o HEDA apresentando os piores resultados. É de se destacar o speedup próximo de 14 do FastGDC para o grafo com 600 mil arestas, resultado próximo do ideal em um ambiente com 16 computadores.

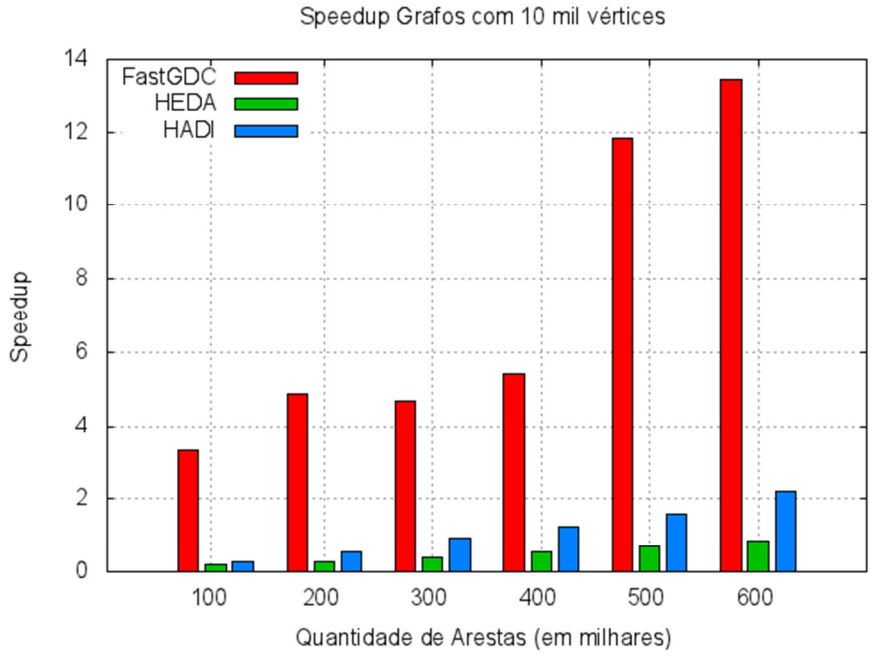


Figura 27. Speedup para grafos com 10 mil vértices.

c) Grafos com 20 mil vértices

Esta subseção apresenta os resultados dos testes com os algoritmos Sequencial, FastGDC, HEDA e HADI utilizando grafos sintéticos com 20 mil vértices. Os tempos obtidos, em minutos, são apresentados na Tabela 12.

Tabela 12. Tempos de execução para grafos com 20 mil vértices

Qtd.Arestas	Sequencial	HEDA	HADI	FastGDC
200.000	19,33	68,37	16,38	6,43
300.000	27,18	73,68	13,83	8,76
400.000	34,88	78,51	12,62	8,74
500.000	43,28	81,31	12,87	9,23

A Figura 28 permite visualizar os dados apresentados na Tabela 12. Nela fica claro que o FastGDC continua sendo a melhor opção, mas o desempenho do HADI já supera o do algoritmo Sequencial para grafos com 20 mil vértices independente da quantidade de arestas. Isso indica que para grafos destas dimensões a sobrecarga imposta pelo uso do modelo MapReduce já é compensada pelo fato do HADI calcular diâmetros aproximados. O HEDA continua apresentando os piores resultados.

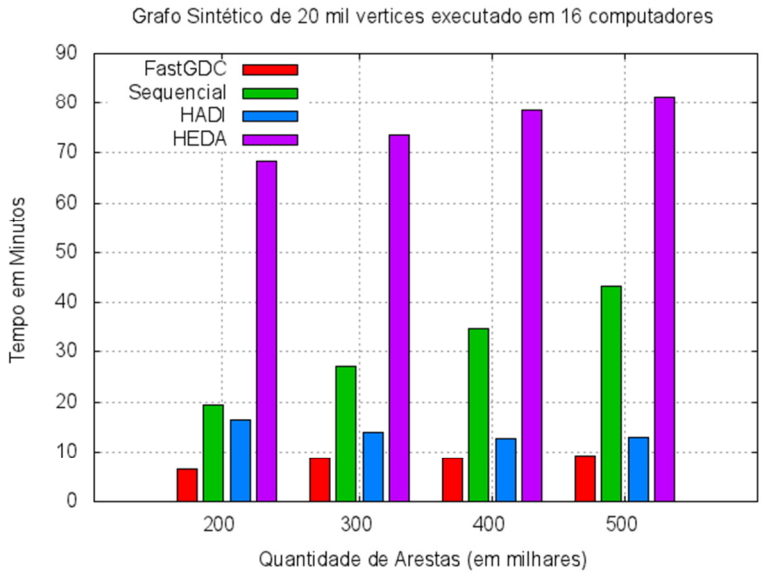


Figura 28. Tempos de execução para grafos com 20 mil vértices.

Com as informações apresentadas na Tabela 12 e na Figura 28 foi possível calcular o quanto cada algoritmo é mais rápido que a solução Sequencial. A Figura 29

apresenta o *speedup* absoluto para todos os grafos com 20 mil vértices. Os algoritmos FastGDC e HADI apresentaram *speedup* maior que 1, enquanto o HEDA não conseguiu melhorar o desempenho do algoritmo Sequencial. A queda no *speedup* do algoritmo FastGDC em relação aos grafos com 10 mil vértices se dá principalmente pela necessidade de utilizar o disco para gravar informações intermediárias entre um *superstep* e outro.

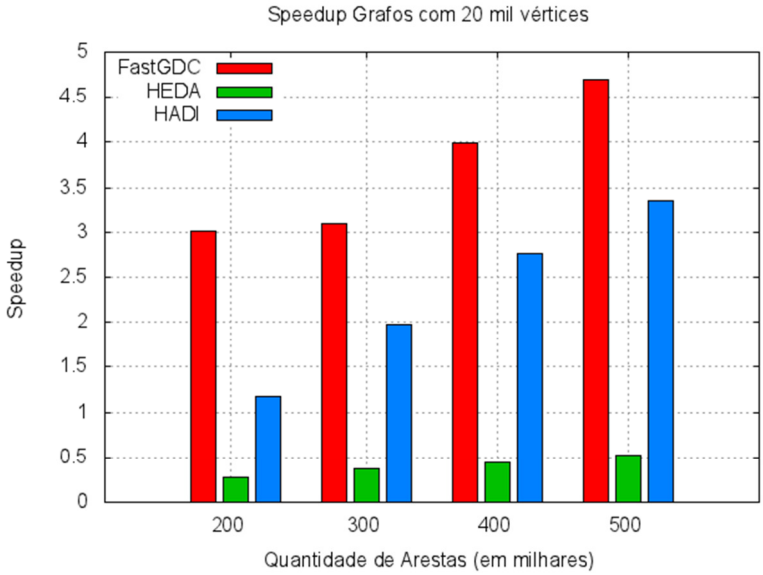


Figura 29. Speedup para grafos com 20 mil vértices.

d) Grafos com 30 mil vértices

Esta subseção apresenta os resultados obtidos da execução dos algoritmos Sequencial, FastGDC, HEDA e HADI para grafos com 30 mil vértices. A Tabela 13 e a Figura 30 apresentam os tempos de execução em minutos de cada algoritmo.

Tabela 13. Tempos de execução para grafos com 30 mil vértices

Qtd.Arestas	Sequencial	HEDA	HADI	FastGDC
200	40,6	140,31	20,29	14,31
300	48,62	169,44	17,16	24,27

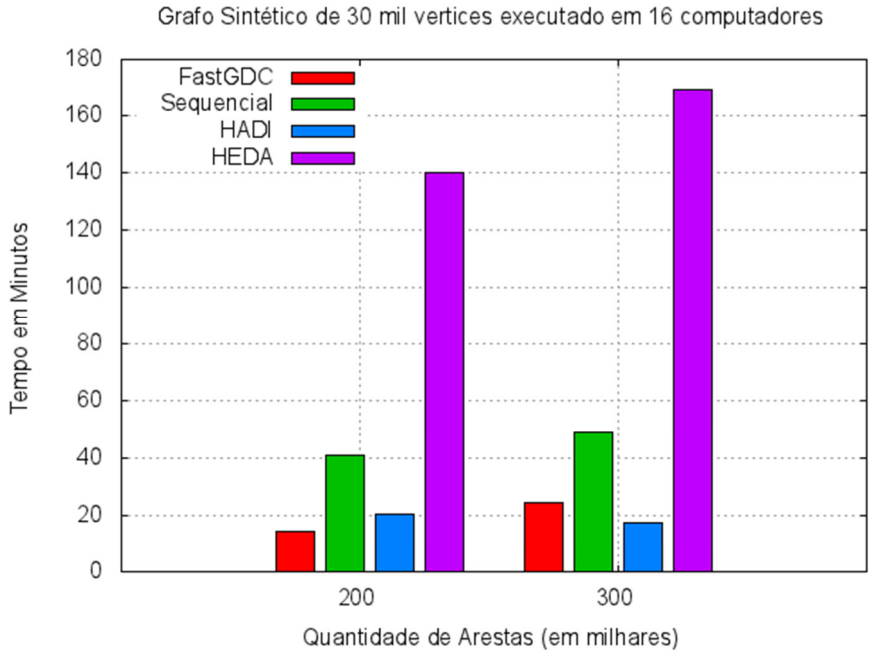


Figura 30. Tempos de execução para grafos com 30 mil vértices.

Os tempos de execução mostram que, para o grafo com 30 mil vértices e 300 mil arestas o HADI consegue calcular o diâmetro aproximado no menor tempo entre todos os algoritmos, enquanto o FastGDC produz resultados exatos em um tempo ligeiramente superior. Em outras palavras, este grafo é suficientemente grande para que as aproximações feitas pelo HADI compensem a sobrecarga imposta pelo modelo MapReduce. Já o HEDA, que usa o MapReduce para produzir resultados exatos, continua apresentando resultados muito piores que os algoritmos concorrentes, inclusive o Sequencial.

A Figura 31 mostra o speedup absoluto para grafos de 30 mil vértices calculados a partir dos dados mostrados na Tabela 13.

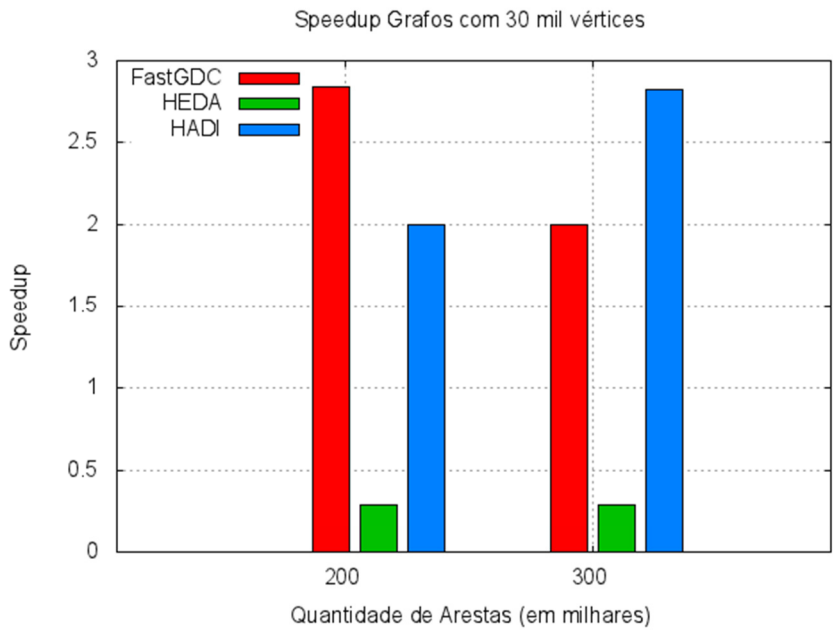


Figura 31. Speedup para grafos com 30 mil vértices.

e) Grafos com 40 mil vértices

Esta subseção apresenta o resultado da execução dos algoritmos Sequencial, FastGDC, HEDA e HADI para grafos com 40 mil vértices e 200 mil e 300 mil arestas, respectivamente. Os tempos de execução, em minutos, são apresentados na Tabela 14 e podem ser visualizados na Figura 32.

Tabela 14. Tempos de execução para grafos com 40 mil vértices

Qtd.Arestas	Sequencial	HEDA	HADI	FastGDC
200	51,42	194,18	24,02	16,01
300	75,37	278,82	20,32	48,78

Os resultados para os grafos com 40 mil vértices confirmam a tendência dos resultados anteriores. O FastGDC se mostra bastante sensível ao número de arestas, praticamente triplicando o tempo gasto para produzir resultados quando o número de arestas aumenta em 50%. Por outro lado, o HADI apresenta um comportamento oposto: seu tempo diminui quando a quantidade de arestas aumenta, em parte porque o diâmetro do grafo diminui também. No entanto, ambos os algoritmos conseguem produzir resultados melhores do que o Sequencial, algo que não acontece com o HEDA, que continua produzindo os piores resultados do grupo.

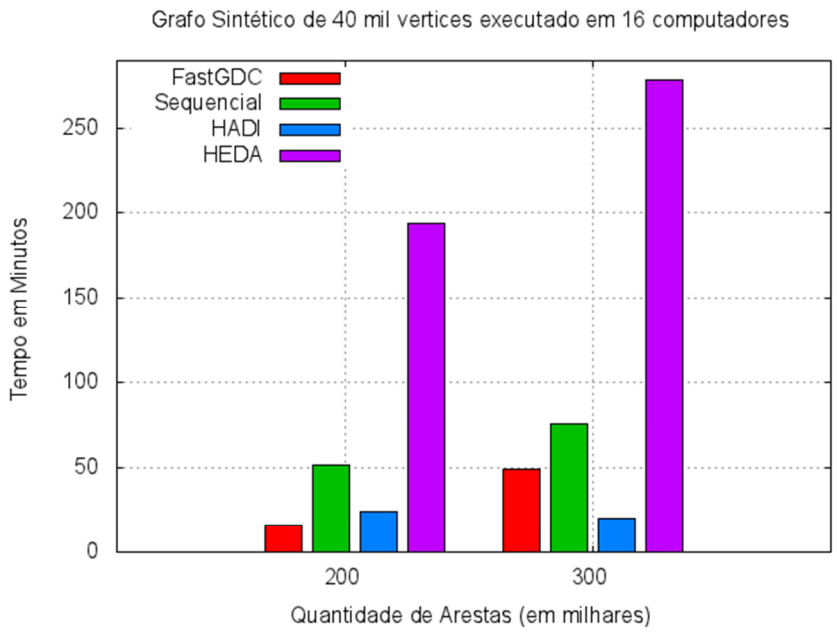


Figura 32. Tempos de execução para grafos com 40 mil vértices.

A Figura 33 mostra o *speedup* absoluto calculado a partir dos resultados da Tabela 14. Tanto o HADI quanto FastGDC produzem *speedup* maior do que 1, pois apenas eles conseguem melhorar o tempo produzido pelo algoritmo sequencial.

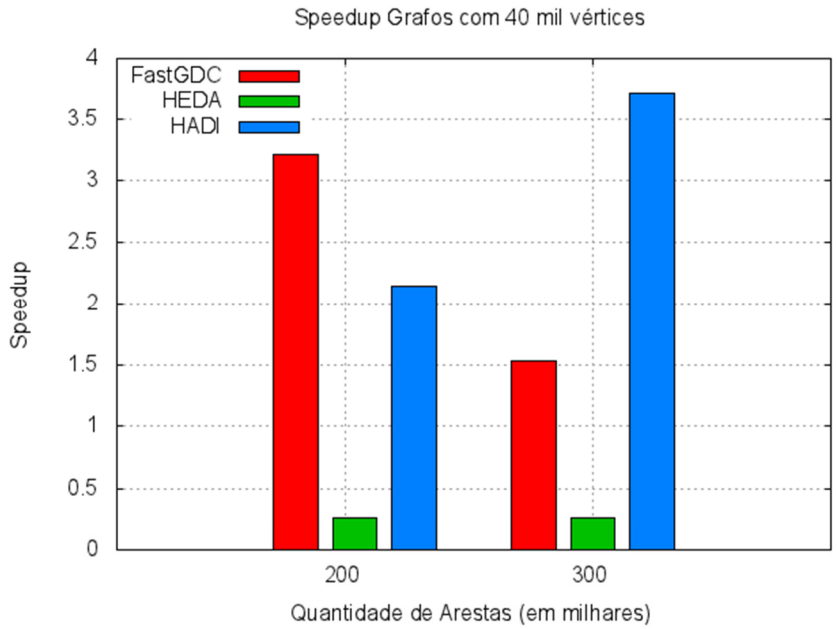


Figura 33. Speedup para grafos de 40 mil vértices.

f) Grafos com 50 mil vértices

Esta subseção apresenta o tempo de execução e *speedup* para um grafo com 50 mil vértices e 200 mil arestas. Este é o maior, dentre os grafos sintéticos gerados, neste trabalho.

Tabela 15. Tempos de execução para um grafo com 50 mil vértices.

Qtd.Arestas	Sequencial	HEDA	HADI	FastGDC
200	107,33	291,47	26,23	41,59

Assim como acontece com os outros grafos de grande porte, o HADI apresenta o menor tempo de execução, decorrente do fato de não ter que percorrer o grafo inteiro para produzir seus resultados. O FastGDC, no entanto, consegue produzir resultados exatos em um tempo significativamente menor do que o algoritmo Sequencial. O HEDA confirma a tendência de produzir resultados piores, chegando a gastar quase 3 vezes mais do que o algoritmo Sequencial para concluir seu processamento. A Figura 34 mostra os resultados da Tabela 15.

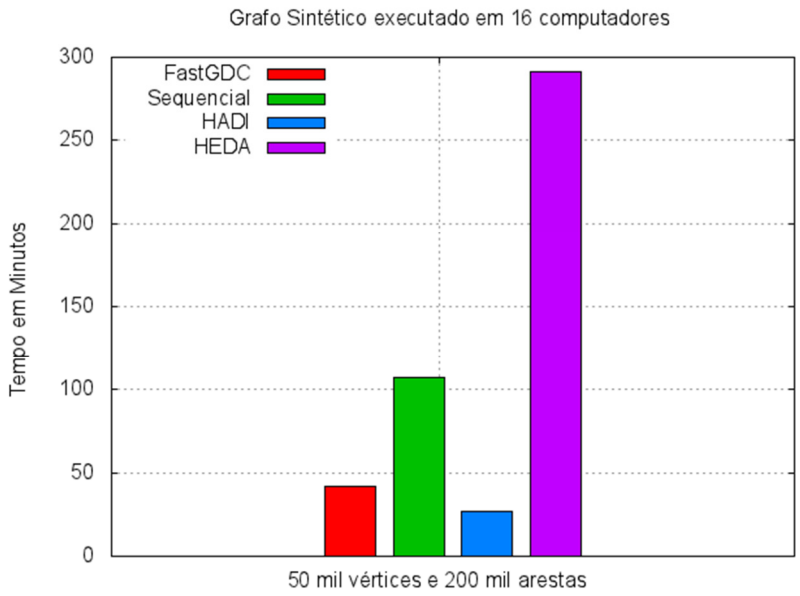


Figura 34. Tempo de execução para um grafo de 50 mil vértices.

A Figura 35 mostra o speedup absoluto para o grafo de 50 mil vértices e 200 mil arestas calculado de acordo com os resultados da Tabela 15.

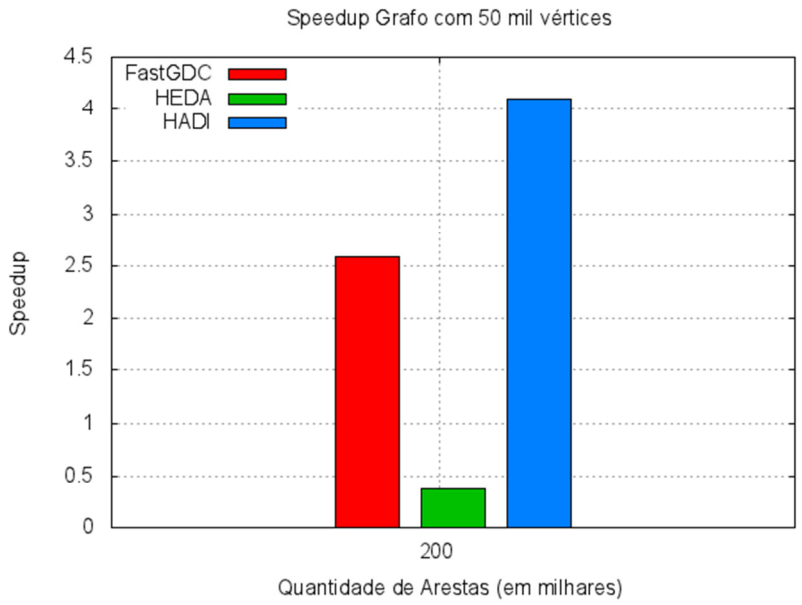


Figura 35. Speedup para um grafo com 50 mil vértices.

6.1.2. Grafos reais

Nesta subseção são apresentados os resultados comparativos obtidos ao processar alguns grafos reais utilizando os algoritmos Sequencial, HEDA, HADI e FastGDC.

6.1.2.1. Grafos do *dataset* da Universidade de Stanford

Os grafos utilizados neste trabalho pertencem à chamada "Collaboration Network", e representam redes de colaboração entre autores de artigos sobre temas de pesquisa como astrofísica e relatividade. As características de cada grafo são mostradas na Tabela 7 na página 51.

A Tabela 16 apresenta os tempos de execução em minutos de cada algoritmo.

Tabela 16. Tempos de execução dataset da Universidade de Standford.

Grafos	Sequencial	HEDA	HADI	FastGDC
ca-HepTh	2,20	16,66	39,40	1,13
ca-HepPh	9,16	25,11	26,66	3,30
ca-CondMat	21,71	67,49	29,68	7,26
email-Enron	47,61	213,73	24,07	8,47
ca-AstroPh	30,02	56,20	27,37	7,69

Os grafos acima compartilham uma característica que impactou consideravelmente os tempos de execução do HADI. Apesar de ter um número de vértices menor, o rótulo de cada vértice é um número entre 0 e 67.747, o que implicou em mais esforço computacional para o HADI produzir seus resultados. Assim, pela primeira vez o HEDA teve tempos de execução menores que o HADI para os grafos ca-HepTh e ca-HepPh. Para os grafos email-Enron e ca-AstroPh, maiores que os anteriores, o impacto do rótulo do vértice foi menor e o HADI se comportou melhor que o HEDA, tal como seria esperado. No entanto, dentre os algoritmos testados, apenas o FastGDC conseguiu produzir tempos melhores do que o algoritmo Sequencial. A Figura 36 permite visualizar os resultados mostrados na Tabela 16.

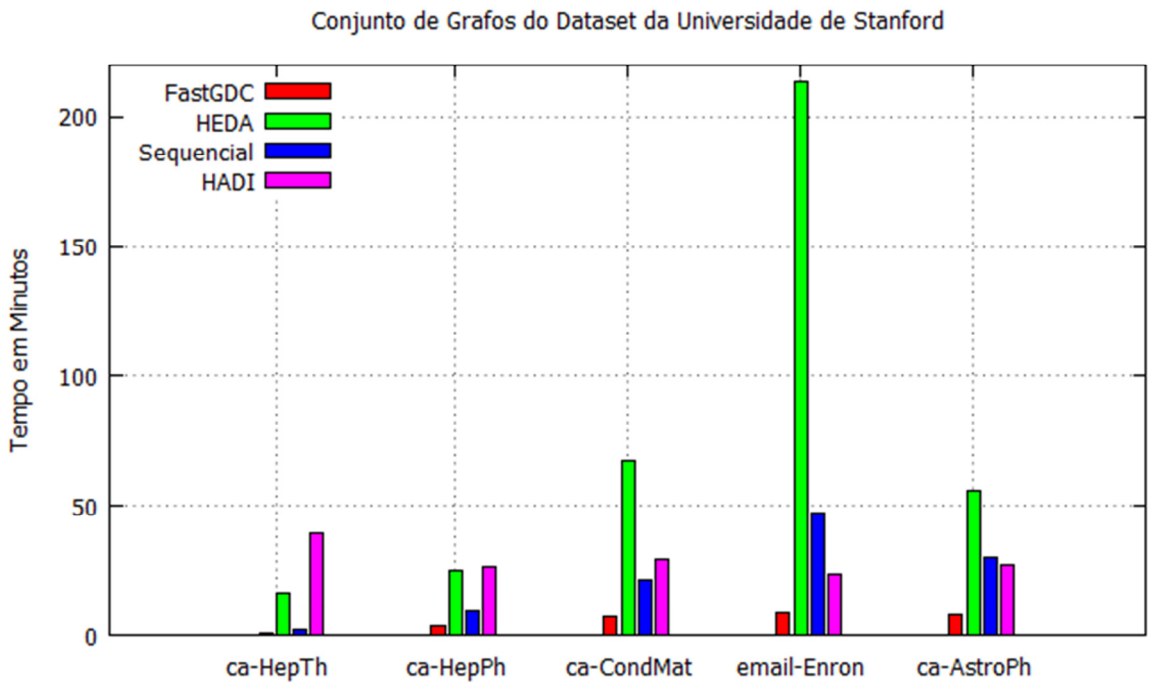


Figura 36. Tempos de execução para os grafos do dataset de Standford.

A Figura 37, que mostra o *speedup* absoluto para os algoritmos FastGDC, HEDA e HADI, e comprova que o algoritmo FastGDC obteve os melhores resultados.

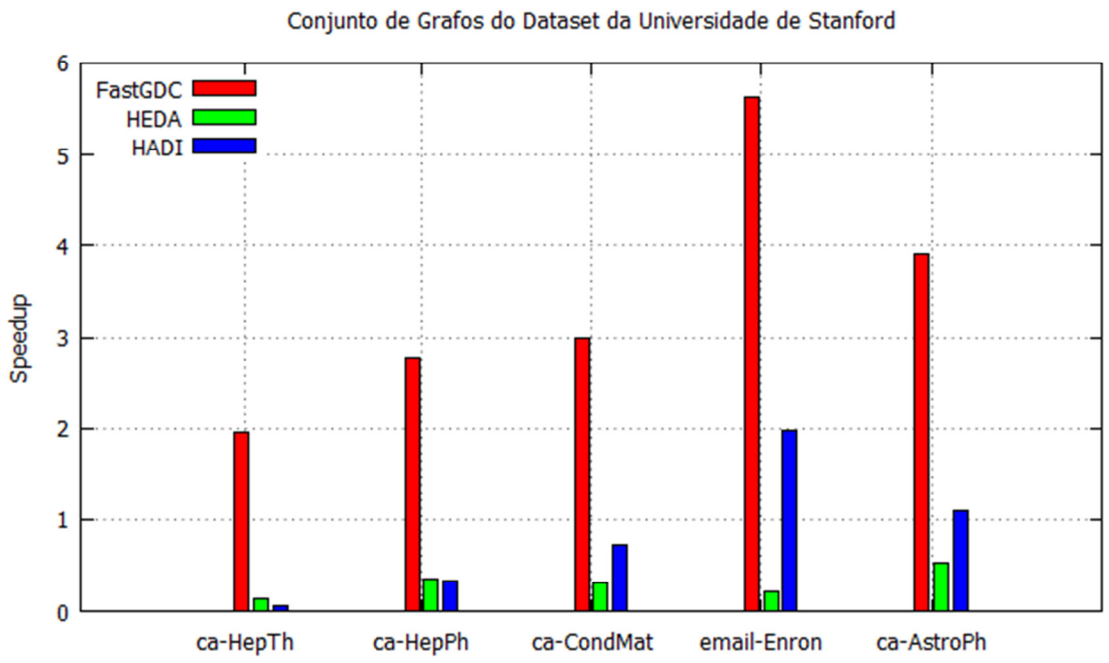


Figura 37. Speedup para os grafos do dataset da Universidade de Standford.

6.1.2.2. Grafos representando a topologia da Internet

Os grafos testados nesta subseção representam à topologia da Internet entre os meses de março e julho de 2013, descrevendo essencialmente a topologia de interconexão entre roteadores. Os detalhes sobre quantidade de vértices e arestas de cada um foram apresentados na Tabela 8 na página 52. Para os testes realizados com os algoritmos FastGDC, HEDA e HADI, foram utilizados 18 (dezoito) computadores. Os resultados dos tempos de execução para cada combinação de algoritmo e grafos são mostrados na Tabela 17 e visualizados na Figura 38.

Tabela 17. Tempos de execução para os grafos reais (internet)

meses	Sequencial	HEDA	HADI	FastGDC
março	88.12	381.40	21.99	27.56
abril	87.23	390.55	20.58	26.75
maio	88.35	395.10	20.41	30.23
junho	79.58	383.06	22.41	25.92
julho	90.47	406.03	22.47	26.28

Tal como aconteceu com os grafos sintéticos de maior tamanho, neste caso o HADI apresenta os melhores tempos de execução, enquanto o FastGDC oferece resultados exatos ao custo de um tempo de execução ligeiramente superior. O algoritmo

Sequencial obteve resultados dentro do esperado, e o HEDA novamente apresentou tempos de execução bastante piores.

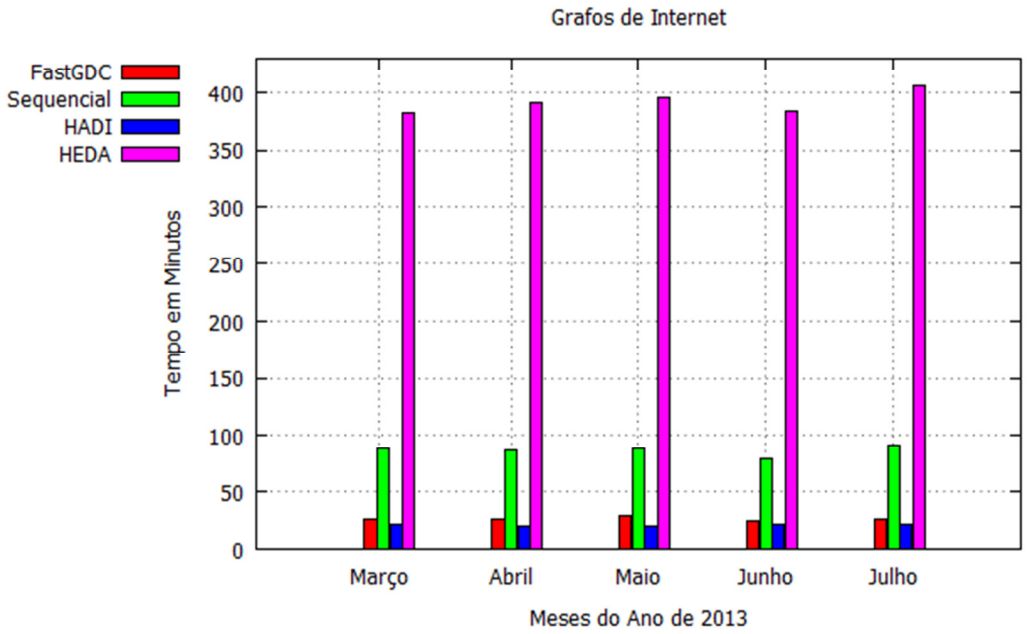


Figura 38. Tempos de execução para os grafos que descrevem a topologia da Internet.

Com as informações apresentadas na Tabela 17 foi possível calcular o *speedup* absoluto para este conjunto de grafos, conforme mostrado na Figura 39.

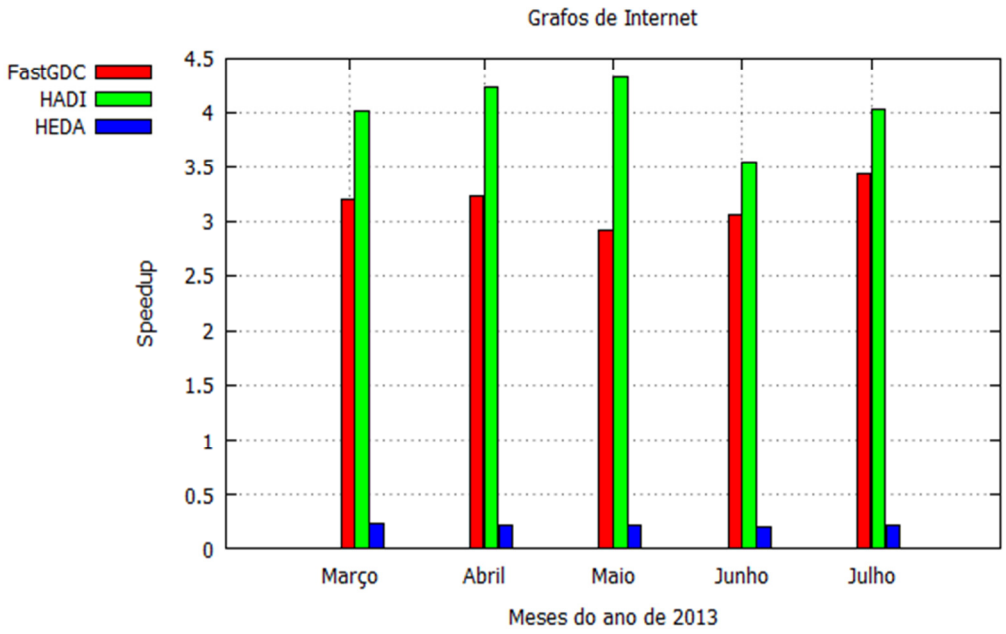


Figura 39. Speedup para os grafos que representam a topologia da Internet.

Para este conjunto de dados, o algoritmo FastGDC apresentou desempenho superior aos algoritmos que oferecem resultados exatos: o Sequencial e o HEDA. O algoritmo HADI, que calcula resultados aproximados, obteve melhores resultados.

6.2. Escalabilidade

Nesta seção será analisado o comportamento dos algoritmos HEDA, HADI e FastGDC na medida em que novos recursos computacionais são disponibilizados para o processamento dos grafos. Em um cenário ideal, o tempo de execução de cada algoritmo deveria cair de forma proporcional à quantidade de recursos novos disponibilizados. No entanto, pelo menos três fatores impedem que isso aconteça.

O primeiro destes fatores é a sobrecarga de gerenciamento imposta pelos novos recursos computacionais disponibilizados. Este problema é particularmente evidente nos modelos MapReduce e BSP, que contêm etapas exclusivamente dedicadas à comunicação entre os computadores que participam do processamento e que, naturalmente, tendem a demorar mais na medida em que novos computadores são adicionados.

O segundo fator diz respeito ao tamanho do problema. Para problemas com número pequeno de vértices e arestas, a adição de novos recursos computacionais pode ser desnecessária e até prejudicial, pois o eventual ganho em capacidade de processamento trazido pelos novos recursos é cancelado pelo aumento da sobrecarga na comunicação.

Por fim, o terceiro fator envolve as características intrínsecas do algoritmo sendo executado. Por exemplo, o FastGDC calcula os resultados executando uma quantidade de *supersteps* que depende apenas das características do grafo, e não da infraestrutura utilizada. Um *superstep* compreende etapas de computação, comunicação e sincronização. Destas, apenas a etapa de computação fica mais rápida na medida em que novos computadores são adicionados pois, presumivelmente, cada nó participante terá menos dados para processar. Já as etapas de comunicação e sincronização tendem a demorar mais na medida em que a infraestrutura cresce, compensando e eventualmente até cancelando o ganho obtido na etapa de computação.

A seguir são mostrados os resultados do processamento de 4 grafos com 10 mil vértices e 100 mil, 200 mil, 400 mil e 600 mil arestas. Para cada um destes grafos os recursos computacionais foram adicionados de maneira gradativa, utilizando 2, 4, 8, 12 e 16 computadores.

6.2.1. Grafo com 10 mil vértices e 100 mil arestas

A Figura 40 mostra o *speedup* absoluto para um grafo com 10 mil vértices e 100 mil arestas. Nessa figura, é possível observar que todos os algoritmos ficaram distantes do *speedup* linear. O FastGDC teve melhor desempenho, quando comparados com os outros algoritmos, em todas as configurações testadas.

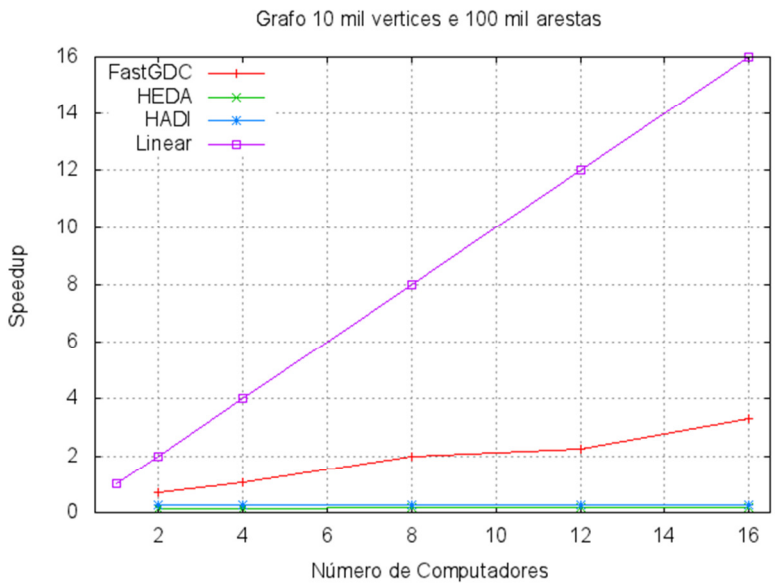


Figura 40. Speedup para um grafo com 10 mil vértices e 100 mil arestas.

Os resultados mostrados na Figura 40 confirmam que o FastGDC foi o único dos algoritmos testados que conseguiu se beneficiar dos novos recursos computacionais. Na medida em que novos computadores foram adicionados o tempo de processamento do FastGDC diminuiu e, portanto, seu *speedup* aumentou, enquanto o HEDA e HADI permaneceram com *speedups* praticamente inalterados. Estes resultados confirmam que, para grafos deste tamanho processados usando o modelo MapReduce, os potenciais

ganhos trazidos por novos recursos computacionais são cancelados pelo aumento na sobrecarga de comunicação e gerenciamento.

O cálculo da eficiência também ajuda a avaliar a capacidade de cada algoritmo de aproveitar novos recursos computacionais. O valor da eficiência normalmente diminui na medida em que novos computadores são adicionados, já que o aumento da sobrecarga de comunicação implica em que cada computador passa mais tempo aguardando enviar ou receber informações e menos tempo fazendo trabalho útil.

A Figura 41 mostra a eficiência calculada para cada algoritmo em cada um dos cenários testados. Como era de se esperar, a eficiência do HADI e do HEDA cai significativamente na medida em que novos computadores são adicionados, devido a que o *speedup* destas soluções é praticamente constante em todas as configurações testadas. A eficiência do FastGDC também diminui na medida em que a quantidade de computadores aumenta, mas em um ritmo menor.

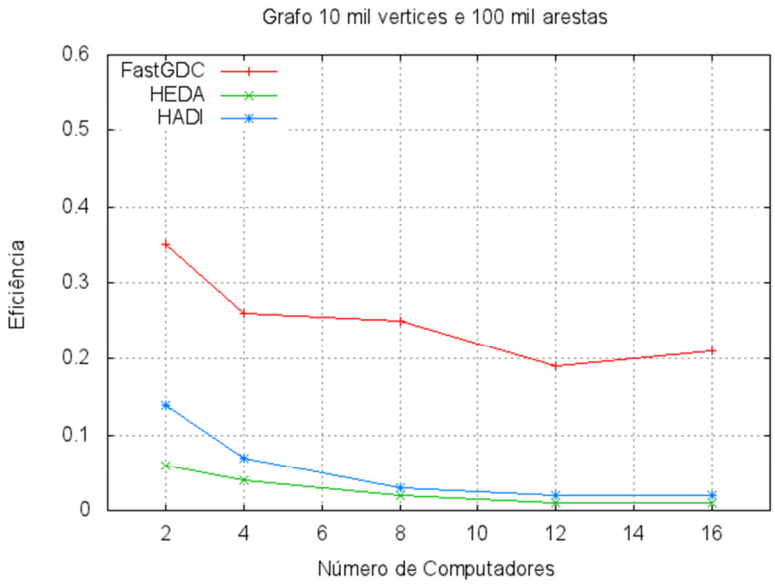


Figura 41. Eficiência para um grafo com 10 mil vértices e 100 mil arestas.

6.2.2. Grafo com 10 mil vértices e 200 mil arestas

A Figura 42 mostra os resultados de *speedup* absoluto para os algoritmos FastGDC, HEDA e HADI ao processar um grafo sintético com 10 mil vértices e 200 mil arestas. Para este grafo, o algoritmo FastGDC apresentou falhas relacionadas ao

gerenciamento de memória do *framework* Hama quando executado utilizando 2 computadores.

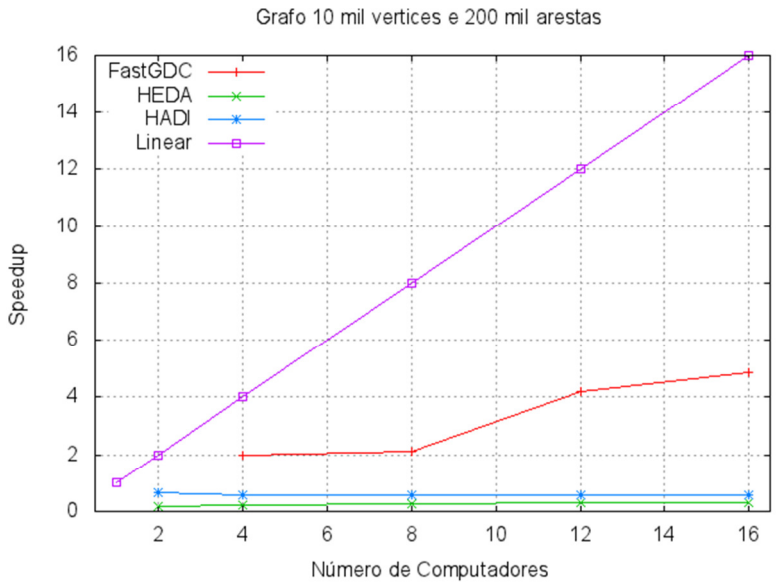


Figura 42. Speedup para um grafo com 10 mil vértices e 200 mil arestas.

Assim como na Figura 40, neste caso apenas o FastGDC conseguiu melhorar seu *speedup* na medida em que novos computadores foram adicionados. Tanto o HADI quanto o HEDA mantiveram valores praticamente constantes em todos os casos, com o HADI apresentando resultados ligeiramente melhores do que o HEDA.

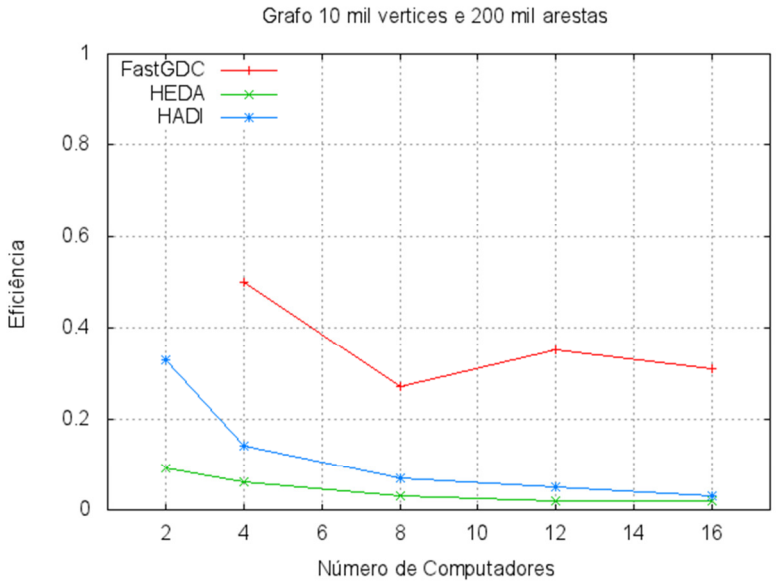


Figura 43. Eficiência para um grafo com 10 mil vértices e 200 mil arestas.

6.2.3. Grafo com 10 mil vértices e 400 mil arestas

A Figura 44 mostra o *speedup* absoluto para os algoritmos FastGDC, HEDA e HADI ao processar um grafo sintético de 10 mil vértices e 400 mil arestas. A execução do algoritmo FastGDC utilizando 2 computadores, novamente apresentou falhas. Estas falhas estão relacionadas ao gerenciamento de memória do *framework* Hama, e estão associadas às mensagens trocadas entre os nós de processamento.

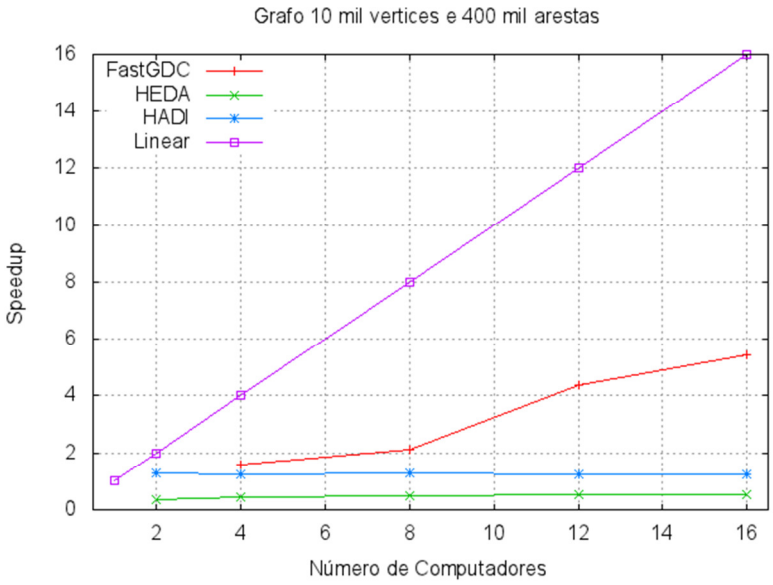


Figura 44. Speedup para um grafo com 10 mil vértices e 400 mil arestas.

Os resultados mostrados na Figura 44 mostram que, mesmo com um grafo maior, o HEDA e o HADI não conseguem melhorar significativamente seu desempenho na medida em que novos recursos computacionais são adicionados. O FastGDC, a exemplo do que fez nos grafos anteriores, consegue aproveitar os novos recursos para melhorar seus resultados.

A Figura 45 mostra os resultados de eficiência para cada um dos cenários testados. Assim como nos casos anteriores, a eficiência do HEDA e do HADI diminui consideravelmente na medida em que novos computadores são adicionados, uma consequência direta do fato do *speedup* se manter praticamente constante. O FastGDC, no entanto, consegue manter o valor da eficiência dentro do mesmo patamar

independentemente da quantidade de computadores utilizados, o que indica que consegue aproveitar efetivamente os novos recursos disponibilizados.

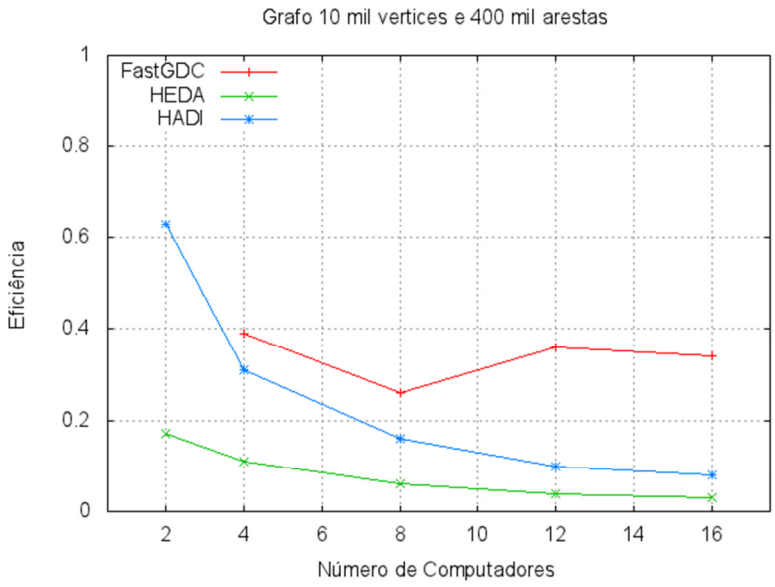


Figura 45. Eficiência para um grafo com 10 mil vértices e 400 mil arestas.

6.2.4. Grafo com 10 mil vértices e 600 mil arestas

A Figura 46 mostra o *speedup* absoluto considerando um grafo sintético com 10 mil vértices e 600 mil arestas processado pelos algoritmos FastGDC, HEDA e HADI. Novamente, a execução do algoritmo FastGDC em 2 computadores falhou.

A execução do algoritmo HADI utilizando 2 computadores apresentou *speedup* superlinear devido ao aumento na quantidade de memória disponível no sistema. Para um grafo desse tamanho, a solução sequencial sofre de *trashing*, um fenômeno em que páginas da memória virtual são constantemente escritas e lidas no disco rígido, o que impacta significativamente no seu tempo de execução. A partir de dois computadores este problema diminui, o que explica o *speedup* superlinear.

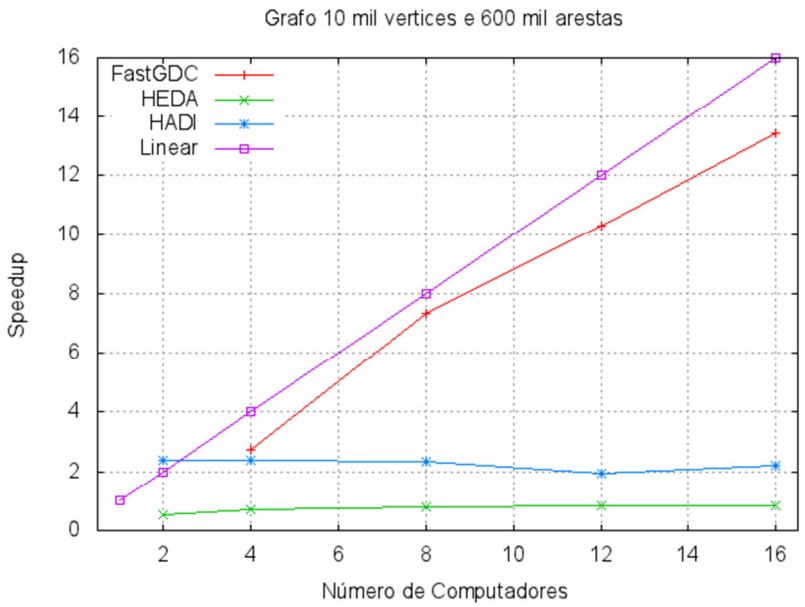


Figura 46. Speedup para um grafo com 10 mil vértices e 600 mil arestas.

Ainda na Figura 46 é possível observar que o *speedup* do algoritmo FastGDC nas execuções com 4, 8, 12 e 16 computadores ficou bastante próximo da linearidade. Isso porque com um grafo maior – 10 mil vértices e 600 mil arestas –, é possível aproveitar melhor os recursos computacionais adicionados. Um grafo com maior número de arestas tem a tendência de ter diâmetro menor, o que implica que o algoritmo FastGDC é executado em um número menor de *supersteps*. O HADI e o HEDA mantiveram *speedups* praticamente constantes, a exemplo do que aconteceu nos grafos anteriores. Cabe ressaltar que o HEDA apresentou *speedup* menor do que 1 em todos os testes realizados, o que implica que, pelo menos nos cenários testados, sua utilização nunca é vantajosa.

A Figura 47 mostra a eficiência calculada para cada um dos experimentos realizados com o grafo de 10 mil vértices e 600 mil arestas.

O HEDA e o HADI continuam apresentando valores decrescentes para a eficiência na medida em que o número de computadores aumenta, decorrentes do fato do *speedup* ser praticamente constante independentemente da quantidade de computadores utilizados. O FastGDC obteve eficiência máxima com 8 computadores, e valores similares para 12 e 16 computadores. O valor menor de eficiência obtido para 4

computadores é ainda reflexo do mesmo problema que impediu a execução do FastGDC em dois computadores e que deve ser sanado em futuras versões do Hama.

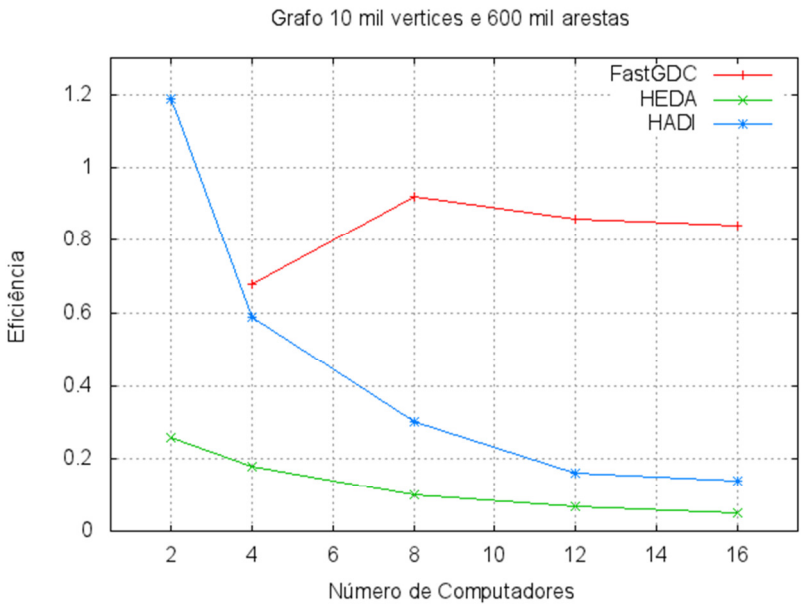


Figura 47. Eficiência para um grafo com 10 mil vértices e 600 mil arestas.

6.3. Armazenamento de vértices do grafo em disco

O *framework* Hama pode ser configurado para despejar no HDFS o conteúdo que ultrapassa o limite da memória física de cada nó do *cluster* e, ao menos em teoria, conseguir processar grafos maiores. Esta configuração envolve alterações diretas no código fonte da plataforma, em particular relacionadas ao uso das classes *DiskVerticesInfo* e *DiskQueue*. Nesta seção são apresentados os resultados obtidos usando o armazenamento em disco durante a execução dos testes e os comparativos com o processamento dos mesmos grafos usando apenas memória.

Apesar de que em teoria usar o armazenamento em disco deveria permitir que o Hama processasse grafos bem grandes não foi esse o resultado constatado na prática. O *framework* Hama ainda apresentou problemas, principalmente em relação ao número de mensagens trocadas entre os vértices do grafo, que o impediram de processar com sucesso grafos maiores dos que apresentados nesta seção. No entanto, este *framework*

encontra-se em pleno desenvolvimento e o fórum de discussões mantido pelos desenvolvedores indica que este problema será resolvido em breve.

Desta forma, procurou-se realizar os experimentos respeitando as limitações presentes no *framework*. Foram utilizados um conjunto com grafos sintéticos de 10 mil vértices e quantidade variada de arestas. Este teste foi realizado adicionando o número de computadores para que fosse possível observar se haveria alteração significativa no tempo de execução.

A Figura 48 mostra os resultados do processamento de um grafo com 10 mil vértices e 100 mil arestas. Como é de se esperar, os tempos de execução usando o disco para armazenar vértices são maiores, e a maior diferença ocorre com dois computadores, quando a quantidade de memória RAM agregada ainda é baixa. A legenda Ativo e Inativo, indica sobre a utilização ou não dos parâmetros DiskVerticesInfo e DiskQueue.

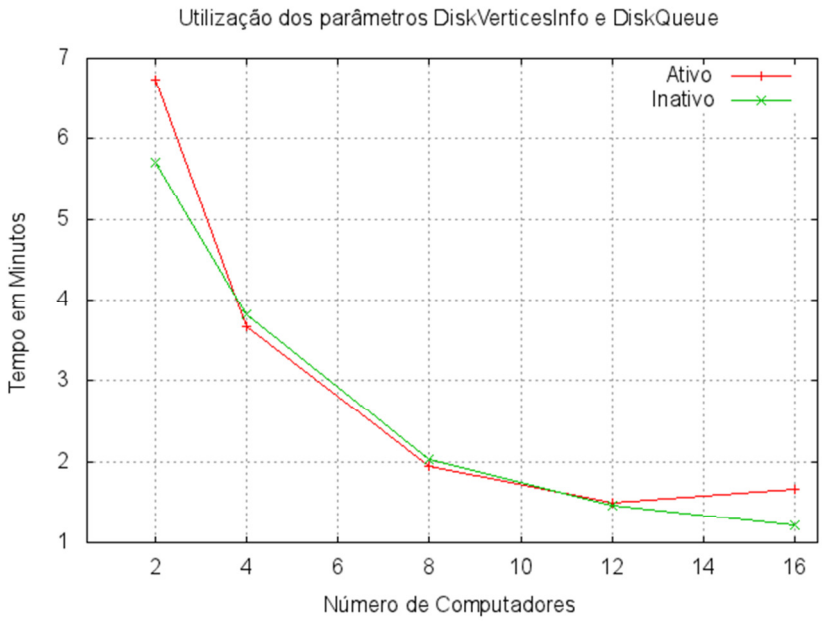


Figura 48. Grafo com 10 mil vértices e 100 mil arestas com e sem despejo em disco.

Quando o número de arestas aumentou de 100 mil para 200 mil, essa diferença ficou mais evidente, como mostrado na Figura 49.

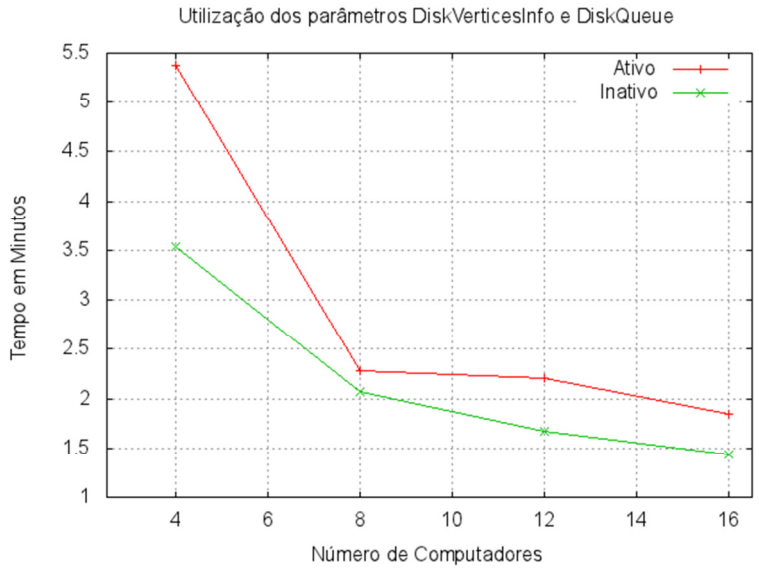


Figura 49. Grafo com 10 mil vértices e 200 mil arestas com e sem despejo em disco.

Na Figura 50 a diferença entre a utilização ou não dos parâmetros para despejo em disco fica mais bem evidenciada, devido a que o grafo utilizado tem 10 mil vértices e 600 mil arestas, ou seja, uma quantidade 6 vezes maior de arestas.

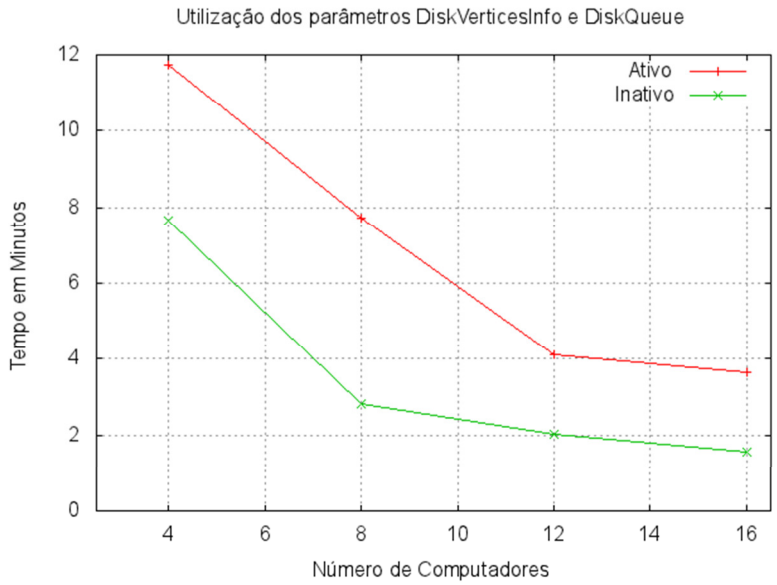


Figura 50. Grafo com 10 mil vértices e 600 mil arestas com e sem despejo em disco.

Como era de se esperar os resultados obtidos com o despejo em disco ativado são piores do que aqueles que utilizam apenas memória. A vantagem desta configuração,

que seria a possibilidade de processar grafos maiores, não foi constatada nesta pesquisa produto de outros problemas decorrentes da imaturidade da plataforma Hama.

6.4. Análise dos resultados

Os resultados mostrados nesta seção apontam para a superioridade do algoritmo FastGDC, que usa o modelo BSP, quando comparado ao HEDA e o HADI, que usam o modelo MapReduce. Para todos os grafos apresentados o FastGDC apresentou melhor desempenho que o algoritmo Sequencial e que o HEDA, lembrando que apenas esses três oferecem soluções exatas ao cálculo de centralidade em grafos. O FastGDC foi mais rápido que o HADI, que apenas oferece soluções aproximadas, para grafos sintéticos com até 30 mil vértices e 200 mil arestas.

Para os grafos do *dataset* da Universidade de Stanford, o FastGDC teve melhor desempenho que o HEDA e o HADI – mesmo o HADI não oferecendo solução exata. Para os grafos que representam a topologia da Internet o FastGDC manteve seu melhor desempenho apenas frente ao HEDA. O algoritmo HADI teve o melhor desempenho dentre todos os algoritmos testados para este tipo de grafos.

A análise de escalabilidade constatou que o HEDA e o HADI não conseguem aproveitar efetivamente novos recursos computacionais na medida em que eles são disponibilizados. Tanto um quanto o outro mantiveram valores de *speedup* praticamente constantes independentemente da quantidade de computadores utilizados, um claro indicativo de que o benefício de mais poder computacional estava sendo cancelado pelo aumento na sobrecarga de comunicação e gerenciamento imposta pelo modelo MapReduce. O FastGDC se mostrou escalável em todos os casos, particularmente para grafos grandes, decorrente da menor sobrecarga de comunicação imposta pelo modelo BSP. Para um grafo com 10 mil vértices e 600 mil arestas o algoritmo FastGDC apresentou *speedup* próximo da linearidade, e eficiência próximo a 1 (um), valores que permitem afirmar que o algoritmo FastGDC é escalável.

Embora os testes apresentados nesta seção sejam limitados, pelo número de computadores e consequentemente pelo tamanho dos grafos, eles validam a hipótese de que o modelo BSP é mais apropriado do que o modelo MapReduce para trabalhar com

grafos. Os algoritmos para grafos, e em particular os que calculam medidas de centralidade, são caracterizados por uma alta taxa de troca de informações entre os vértices. Pelo menos nas implementações mais conhecidas do MapReduce, como o Hadoop, esta troca de informações é feita através de leituras e escritas em um sistema de arquivos distribuído, acessível por todos os nós de processamento. Este mecanismo, quando comparado à troca de mensagens utilizada pelo modelo BSP, introduz uma sobrecarga significativa em cada iteração do algoritmo. Acredita-se ser esta a razão fundamental para as diferenças de tempo de execução constatadas em todos os testes.

Capítulo 7. Conclusões e trabalhos futuros

Este trabalho comparou os modelos BSP e MapReduce no cálculo de medidas de centralidade em grafos grandes. Para representar o modelo BSP os autores desenvolveram um algoritmo chamado FastGDC. Os resultados obtidos com este algoritmo foram comparados com os algoritmos HEDA (Nascimento & Murta, 2012) e HADI (Kang, Tsourakakis, Appel, Faloutsos, & Leskovec, 2011), duas soluções com propósitos similares e que seguem o modelo MapReduce. O FastGDC foi executado usando o Apache Hama, uma implementação conhecida do modelo BSP, enquanto o HEDA e o HADI foram testados utilizando o Apache Hadoop.

Os resultados apresentados neste trabalho indicam que o FastGDC é mais rápido que o HEDA em todos os grafos testados, e mais rápido que o algoritmo HADI em grafos sintéticos com até 30 mil vértices. Para os grafos do dataset de Stanford o FastGDC é melhor que todos os outros algoritmos. A análise destes resultados, e das características dos modelos BSP e MapReduce, indicam que o principal responsável por esta diferença é o mecanismo utilizado por ambos os modelos para comunicar resultados intermediários entre uma etapa e outra. No caso do Hadoop, esta comunicação se dá através da leitura e escrita de arquivos em um sistema de arquivos distribuído chamado HDFS e fornecido como parte do ecossistema do Hadoop. Já o Hama utiliza mecanismos de troca de mensagens para o mesmo fim.

Para o processamento de grafos com maior número de vértices e arestas enfrentaram-se problemas com a maturidade do Hama, que acusou falhas de gerenciamento de memória quando o tamanho do grafo aumentou. Outros *softwares* que implementam o modelo BSP, como o GPS (Salihoglu & Widom, 2012) e o Giraph (Apache Incubator Giraph, 2012), encontram-se em um estágio ainda mais incipiente de desenvolvimento. Os autores têm trabalhado diretamente com os desenvolvedores do Hama para tentar sanar estes problemas melhorando a qualidade da ferramenta.

Dentre os trabalhos futuros planejados estão comparar os modelos BSP e MapReeduce utilizando outros problemas que envolvam grafos (e.g. ancestral comum

mais baixo, componentes conexos, etc), assim como a implementação do algoritmo HADI usando o modelo BSP.

Referências

Yahoo Launches World's Largest Hadoop Production Application. (2008, Fevereiro 10). Retrieved from Developer Yahoo: http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html

Apache Hama Installation Guide. (2012). Retrieved from Apache.org: https://issues.apache.org/jira/secure/attachment/12528220/ApacheHamaInstallationGuid

Apache Incubator Giraph. (2012). Retrieved from Apache Giraph: http://incubator.apache.org/giraph/

Stanford Large Nertwork Dataset Collection. (2012). Retrieved from Stanford.edu: http://snap.stanford.edu/data/

Apache Hama. (2013). Retrieved from Apache Hama: https://issues.apache.org/jira/secure/attachment/12528219/ApacheHamaDesign.pdf

Cloudera Ask Bigger Questions. (2013, Fevereiro 01). Retrieved from Cloudera: http://www.cloudera.com/content/cloudera/en/home.html

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

MapR Technologies. (2013, 02 10). Retrieved from MapR Technologies: http://www.mapr.com/

Aho, A., Hopcroft, J., & Ullman, J. (1973). On finding lowest common ancestors in trees. *Proc. 5th ACM Symp. Theory of Computing (STOC)*, pp. 253–265.

Alba, E. (2002). Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters 82.1*, 7-13.

Anderson, D. P., Werthimer, D., Cobb, J., Korpela, E., & Lebofsky, M. (2000). SETI@home: Internet Distributed Computing for SETI. *Bioastronomy 99: A New Era in Bioastronomy*, p. 511. Retrieved from Search for Extraterrestrial Intelligence: http://setiathome.berkeley.edu/

Anderson, R. J., & Miller, G. L. (1990). A simple randomized parallel algorithm for list-ranking. *Information Processing Letters. v. 33, n. 5*, 269-273.

Bisseling, R. H. (2004). *A structured approach using BSP and MPI.* Londres: OXFORD University Press.

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

Cáceres, E. N., & Nasu, E. N. (2003). A BSP/CGM Algorithm for Computing Euler Tours in Graphs. *In Proccedings of the 15th Symposium on Computer Architecture and High Performance Computing*.

Cáceres, E. N., Dehne, F., Ferreira, A., Flocchini, P., Rieping, I., Roncato, A., . . . Song, S. W. (1997). Efficient parallel graph algorithms for coarse grained multicomputers and BSP. *24th International Colloquium on Automata, Languages, and Programing* (pp. 390-400). Lecture Notes in Computer Science.

Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., . . . Zdonik, S. (2002). Monitoring streams: a new class of data management applications. 215-226.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to Algorithms.* Cambridge: Mit Press.

Coulouris, G., Dollimore, J., & Kindberg, T. (2007). *Distributed Systems: Concepts and Design.* Porto Alegre: Bookman.

Cugola, G., & Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.44*, 1-62.

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

Eager, D. L., Zahorjan, J., & Lazowska, E. D. (1989). Speedup versus efficiency in parallel systems. *Computers, IEEE Transactions on 38(3)*, 408-423.

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

Etzion, O., & Niblett, P. (2011). *Event processing in action.* Stanford: Manning Publications Co.

Flynn, M. J. (1972). Some Computer organizations and their effectiveness. *IEEE Trans. Comput. 21*, 948-960.

Freeman, L. C. (1977). A Set of Measures of Centrality Based on Betweenness. *Sociometry v.40 n.1*, 35-41.

Frey, E. (2006, Abril 06). *Mapreduce Bash Script*. Retrieved from Blog Last Fm: http://blog.last.fm/2009/04/06/mapreduce-bash-script

Gustafson, J. L. (1992). The consequences of fixed time performance measurement. *Proceedings of the Twenty-Fifth Hawaii International Conference Vol. 2. IEEE*, 113-124.

Hage, P., & Harary, F. (1995). Eccentricity and centrality in networks. *Social Networks*, Volume 17, Issue 1.

Jain, S., & Mallozzi, M. (2010). *Parallel Heuristics for TSP on MapReduce.* Brow University Data-Intensive Scalable Computing. Technical Report.

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)*, 8.

Kleinberg, J., & Tardos, E. (2006). *Algorithm Design.* Boston: Addison Wesley.

Kopytoff, V. G. (2013, 02 05). *Hortonworks.* Retrieved from Hortonworks: http://hortonworks.com/

Kumar, V. (2002). *Introduction to Parallel Computing.* Boston: Addison-Wesley Longman Publishing Co.

Lam, C. (2010). *Hadoop in Action.* Manning Publications.

Luckham, D. C. (2002). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Boston: Addison-Wesley Longman Publishing Co.

Luckham, D. C. (2006, Agosto 1). *What's the Difference Between ESP and CEP?* Retrieved from Complex Event Processing & Real Time Intelligence: http://www.complexevents.com/2006/08/01/what%E2%80%99s-the-difference-between-esp-and-cep/

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel: A System for Large-Scale Graph Processing. *SIGMOD '10 Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135-146.

Mewawalla, C. (2012). Big Data. *Global investment themes: telecoms, media and technology*, 3-19.

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.

Nascimento, J. P., & Murta, C. D. (2011). Um algoritmo paralelo para o cálculo de centralidade em grafos grandes. (Doctoral dissertation, Master's thesis, PPGMMC, CEFET–MG).

Nieminem, J. (1974). On the centrality in a graph . *Scandinavian Journal of Psychology v15 n.1*, 332-336.

Noll, M. G. (2012, 07 20). *Tutorial Multi Node Cluster.* Retrieved from Michael-noll.com: www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/

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

Patterson, D. A., & Hennessy, J. L. (2005). *Organização e Projeto de Computadores.* Rio de Janeiro: Campus.

Pettey, C., & Goasduff, L. (2011, Junho 27). *Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data.* Retrieved from Gartner: http://www.gartner.com/newsroom/id/1731916

Russell, S., & Norvig, P. (2004). *Inteligência Artificial.* São Paulo: Campus.

Sabidussi, G. (1966). The centrality index of a graph. *Psuchometrika v31*, 581-603.

Salihoglu, S., & Widom, J. (2012). *GPS: A Graph Processing System.* Stanford InfoLab.

Schmidt, S. (2009). *The Holumbus Framework: Distributed computing with MapReduce in Haskel.* Wedel: FHWEDEL.

Stonebraker, M., Çetintemel, U., & Zdonik, S. (2005, 12 4). The 8 requirements of real-time stream processing. *SIGMOD Rec. 34,4*, pp. 42-47.

Tiskin, A. (2001). All-Pairs Shortest Paths Computation in the BSP Model. *In Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, 178-189.

Tomassini, M. (2005). *Spatially Structured Evolutionary Algorithms.* Berlin Heidelberg: Springer.

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

Wasserkrug, S., Gal , A., Etzion, O., & Turchin, Y. (2008). Complex Event Processing Over Uncertain Data. *Proceedings of the second international conference on Distributed event-based systems* (pp. 253-264). New York: ACM.

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

Apêndice A

Passos para execução do algoritmo FastGDC

Para executar o algoritmo FastGDC é necessário dispor de um ambiente de *cluster* que tenha sido configurado utilizando o sistema de arquivos HDFS, e o *framework* Hama. O formato de grafo aceito pelo FastGDC é: #vertice_id tab vértice_adj;vértice_adj;vértice_adj. Desta forma, com o ambiente configurado, é possível enumerar os passos para a execução do algoritmo:

1) O primeiro passo, é iniciar o servidor com o HDFS e carregar o grafo a ser utilizado.

```
hduser@mestre:~$ sh start-dfs.sh //inicia o HDFS
hduser@mestre:~$ hadoop fs -put grafo_utilizado /pasta_hdfs //carrega o grafo
```

2) Depois, é necessário iniciar o *cluster* com o *framework* Hama:

```
hduser@mestre:~$ sh start-bspd.sh //inicia o Hama
```

3) Após, com a linha abaixo, o algoritmo FastGDC é executado.

```
hduser@mestre:~$ time hama jar fastgdc.jar tamanho_vetor_distâncias
/caminho_grafo_hdfs /caminho_saida_hdfs número_partes_trabalho limite_superstep
```

Na linha de comando apresentada no passo 3, foi utilizado o comando time para medir o tempo de execução. O parâmetro tamanho_vetor_distâncias, esta relacionado ao tamanho do vetor utilizado por cada nó do grafo para armazenar as distâncias em relação aos outros vértices do grafo. O parâmetro caminho_grafo_hdfs, diz respeito à localização do grafo no HDFS, que será executado. O parâmetro caminho_saida_hdfs, esta relacionado ao caminho no HDFS que será armazenado o arquivo com o resultado. O parâmetro número_partes_trabalho, indica em quantas partes o trabalho será divido, para esse parâmetro é necessário considerar o número de nós no *cluster*, cada nó é configurado para assumir apenas 3 trabalhos em paralelo por execução. Por fim, o parâmetro limite_superstep indica qual será o limite superior de *supersteps* a serem executados.

Anexos

Nesta seção, são apresentadas tabelas com todos os resultados encontrados para os algoritmos HEDA, HADI, FastGDC e Sequencial.

1. Tabela de resultados algoritmo Sequencial

GRAFOS SINTÉTICOS

5 mil vértices	diâmetro	raio	Média de tempo em minutos
20mil arestas	13	11	0,33
100mil arestas	5	4	1,35
200mil arestas	4	3	2,60
400mil arestas	3	3	5,12
600mil arestas	3	3	7,70

10 mil vértices	diâmetro	raio	Média de tempo em minutos
100mil arestas	7	5	4,03
200mil arestas	5	4	7,00
300mil arestas	4	4	9,93
400mil arestas	5	3	13,37
500mil arestas	4	3	17,12
600mil arestas	4	3	20,83

20 mil vértices	diâmetro	raio	Média de tempo em minutos
200mil arestas	7	6	19,33
300mil arestas	6	5	27,18
400mil arestas	5	4	34,88
500mil arestas	5	4	43,28

30 mil vértices	diâmetro	raio	Média de tempo em minutos
200mil arestas	10	7	40,60
300mil arestas	8	6	48,62
400mil arestas	7	5	62,37
500mil arestas	6	5	76,90

40 mil vértices	diâmetro	raio	Média de tempo em minutos
200mil arestas	13	11	51,42
300mil arestas	11	7	75,37
400mil arestas	8	6	96,28

50 mil vértices	diâmetro	raio	Média de tempo em minutos
200mil arestas	13	11	107,55

DATASET STANFORD

Média do tempo sequencial em minutos

cahepth	2,2
cahepph	9,16
astroph	30,02
condmat	21,71
emailenron	47,61

GRAFOS INTERNET

Média do tempo sequencial em minutos

março	88,12
abril	87,23
maio	88,35
junho	79,58
julho	90,47

2. Tabela de resultados algoritmo HEDA

GRAFOS SINTÉTICOS

5 mil vértices	**tempos em minutos**			**média**	**Desvio Padrão**	***Speedup***
100 mil arestas	12,29	12,28	12,29	12,29	0,01	0,11
200 mil arestas	13,05	12,49	12,59	12,71	0,30	0,20
400 mil arestas	13,35	13,31	13,41	13,36	0,05	0,38
600 mil arestas	13,54	13,48	13,57	13,53	0,05	0,57

10 mil vértices	**tempos em minutos**			**média**	**Desvio Padrão**	***Speedup***
100 mil arestas	22,41	22,55	22,5	22,49	0,07	0,18
200 mil arestas	24	24,07	24,01	24,03	0,04	0,29
300 mil arestas	26,05	26,03	26,03	26,04	0,01	0,38
400 mil arestas	25,24	25,11	25,15	25,17	0,07	0,53
500 mil arestas	24,54	24,51	24,45	24,50	0,05	0,70
600 mil arestas	24,39	24,5	24,51	24,47	0,07	0,85

20 mil vértices	**tempos em minutos**			**média**	**Desvio Padrão**	***Speedup***
200 mil arestas	68,4	68,21	68,49	68,37	0,14	0,28
300 mil arestas	72,35	73,24	72,45	72,68	0,49	0,37
400 mil arestas	78,53	78,7	78,29	78,51	0,21	0,44
500 mil arestas	81,17	81,56	81,2	81,31	0,22	0,53

30 mil vértices	tempos em minutos			média	Desvio Padrão	*Speedup*
200 mil arestas	139,9	141,44	139,58	140,31	0,99	0,29
300 mil arestas	168,52	169,35	170,46	169,44	0,97	0,29

40 mil vértices	tempos em minutos			média	Desvio Padrão	*Speedup*
200 mil arestas	193,32	194,11	195,11	194,18	0,90	0,26
300 mil arestas	279,2	277,8	279,46	278,82	0,89	0,27

50 mil vértices	tempos em minutos			média	Desvio Padrão	*Speedup*
200 mil arestas	276,4	298,48	299,53	291,47	13,06	0,37

DATASET STANFORD

grafo	vértices	arestas	Tamanho vetor	tempos em minutos			Média	Desvio Padrão	*Speedup*
cahepth	9877	51971	68747	16,63	16,63	16,70	16,66	0,04	0,13
cahepph	12008	237010	89209	25,25	24,95	25,13	25,11	0,15	0,36
condmat	23133	186936	108300	67,65	67,65	67,17	67,49	0,28	0,32
emailenron	36692	367662	36694	214,50	214,33	212,37	213,73	1,19	0,22
astroph	18772	396160	133280	56,63	56,17	56,20	56,33	0,26	0,53

GRAFOS INTERNET

arestas	vértices	mês	tempos em minutos			Media	Desvio Padrão	*Speedup*
336992	44447	março	379,08	381,25	383,93	381,42	2,43	0,23
341470	44729	abril	393,02	389,32	389,62	390,65	2,06	0,22
350626	44961	maio	389,53	396,13	399,28	394,98	4,98	0,22
305278	44819	junho	382,48	382,03	383,83	382,78	0,94	0,21
353768	44457	julho	402,47	411,52	403,48	405,82	4,96	0,22

3. Tabela de resultados algoritmo HADI

GRAFOS SINTÉTICOS

				Diâmetro				
5 mil vértices	tempos em minutos			HADI	Exato	média	Desvio Padrão	*Speedup*
100mil arestas	10,58	12,52	12,57	4	5	11,89	1,13	0,11
200mil arestas	10,62	10,62	10,60	4	4	10,61	0,01	0,25
400mil arestas	8,72	8,68	8,68	3	3	8,69	0,02	0,59
600mil arestas	8,77	8,72	8,67	3	3	8,72	0,05	0,88

				Diâmetro				
10 mil vértices	**tempos em minutos**			HADI	Exato	**média**	**Desvio Padrão**	***Speedup***
100mil arestas	14,48	14,42	14,42	6	7	14,44	0,04	0,28
200mil arestas	12,48	12,57	12,47	5	5	12,51	0,05	0,56
300mil arestas	10,65	10,65	10,70	4	4	10,67	0,03	0,93
400mil arestas	11,00	10,72	10,82	4	5	10,84	0,14	1,23
500mil arestas	10,80	10,60	10,68	4	4	10,69	0,10	1,60
600mil arestas	8,78	10,75	8,75	3	4	9,43	1,15	2,21

				Diâmetro				
20 mil vértices	**tempos em minutos**			HADI	Exato	**média**	**Desvio Padrão**	***Speedup***
200mil arestas	16,35	16,37	16,42	7	7	16,38	0,03	1,18
300mil arestas	14,45	12,57	14,47	6	6	13,83	1,09	1,97
400mil arestas	12,62	12,62	12,62	5	5	12,62	0,00	2,76
500mil arestas	12,78	13,05	12,77	5	5	12,87	0,16	3,36

				Diâmetro				
30 mil vértices	**tempos em minutos**			HADI	Exato	**média**	**Desvio Padrão**	***Speedup***
200mil arestas	20,23	20,27	20,37	9	10	20,29	0,07	2,00
300mil arestas	16,73	16,40	18,33	7	8	17,16	1,03	2,83
				Diâmetro				
40 mil vértices	**tempos em minutos**			HADI	Exato	**média**	**Desvio Padrão**	***Speedup***
200mil arestas	24,00	24,05	24,00	11	13	24,02	0,03	2,14
300mil arestas	20,35	20,32	20,30	9	11	20,32	0,03	3,71
				Diâmetro				
50 mil vértices	**tempos em minutos**			HADI	Exato	**média**	**Desvio Padrão**	***Speedup***
200mil arestas	26,25	26,25	26,18	12	13	26,23	0,04	4,10

DATASET STANFORD

grafo	**vértices**	**arestas**	**tamanho vetor**	**tempos em minutos**			**media**	**desvio padrão**	***Speed up***
cahepth	9877	51971	68747	35,48	48,62	34,10	39,40	8,01	0,06
cahepph	12008	237010	89209	26,12	27,47	26,38	26,66	0,71	0,34
condmat	23133	186936	108300	33,25	27,85	27,95	29,68	3,09	0,73
emailenron	36692	367662	36694	24,02	24,07	24,13	24,07	0,06	1,98
astroph	18772	396160	133280	27,97	27,95	26,18	27,37	1,02	1,10

GRAFOS INTERNET

arestas	vértices	mês	tempos em minutos			media	desvio padrão	*Speedup*
336992	44447	março	20,90	22,55	22,53	21,99	0,95	4,01
341470	44729	abril	18,78	20,48	22,48	20,58	1,85	4,24
350626	44961	maio	20,35	20,50	20,37	20,41	0,08	4,33
305278	44819	junho	22,28	22,57	22,38	22,41	0,14	3,55
353768	44457	julho	22,37	22,52	22,53	22,47	0,09	4,03

4. Tabela de resultados algoritmo FastGDC

GRAFOS SINTÉTICOS

5 mil vértices	tempos em minutos			media	Desvio Padrão	*Speedup*
100mil arestas	0,78	0,70	0,70	0,73	0,05	1,85
200mil arestas	0,67	0,65	0,68	0,67	0,02	3,90
400mil arestas	1,13	1,22	1,23	1,19	0,05	4,28
600mil arestas	2,25	1,87	2,05	2,06	0,19	3,75

10 mil vértices	tempos em minutos			media	Desvio Padrão	*Speedup*
100mil arestas	1,23	1,18	1,22	1,21	0,03	3,33
200mil arestas	1,68	1,33	1,28	1,43	0,22	4,88
300mil arestas	2,12	2,08	2,18	2,13	0,05	4,67
400mil arestas	2,52	2,38	2,50	2,47	0,07	5,42
500mil arestas	1,37	1,38	1,58	1,44	0,12	11,85
600mil arestas	1,53	1,57	1,55	1,55	0,02	13,44

20 mil vértices	tempos em minutos			media	Desvio Padrão	*Speedup*
200mil arestas	6,22	6,72	6,35	6,43	0,26	3,01
300mil arestas	8,50	9,48	8,28	8,76	0,64	3,10
400mil arestas	8,50	8,95	8,78	8,74	0,23	3,99
500mil arestas	10,05	8,43	9,20	9,23	0,81	4,69

30 mil vértices	tempos em minutos			media	Desvio Padrão	*Speedup*
200mil arestas	14,05	14,38	14,48	14,31	0,23	2,84
300mil arestas	24,82	21,43	26,55	24,27	2,60	2,00

40 mil vértices	tempos em minutos			media	Desvio Padrão	*Speedup*
200mil arestas	16,38	16,27	15,38	16,01	0,55	3,21
300mil arestas	48,78	48,45	49,12	48,78	0,33	1,54

50 mil vértices	tempos em minutos			media	Desvio Padrão	*Speedup*
200mil arestas	41,53	41,48	41,75	41,59	0,14	2,59

DATASET STANFORD

grafo	vértices	arestas	Tamanho vetor	tempos em minutos			média	desvio padrão	*Speedup*
cahepth	9877	51971	68747	1,13	1,18	1,07	1,13	0,06	1,95
cahepph	12008	237010	89209	3,3	3,12	3,48	3,30	0,18	2,78
condmat	23133	186936	108300	7,25	7,17	7,35	7,26	0,09	2,99
emailenron	36692	367662	36694	8,31	9,02	8,08	8,47	0,49	5,62
astroph	18772	396160	133280	7,3	7,44	8,34	7,69	0,56	3,90

GRAFOS INTERNET

arestas	vértices	mês	tempos em minutos			média	desvio padrão	*Speedup*
336992	44447	março	27,58	27,08	28,02	27,56	0,47	3,20
341470	44729	abril	26,58	27,71	26,59	26,96	0,65	3,24
350626	44961	maio	30,21	30,32	30,17	30,23	0,08	2,92
305278	44819	junho	25,42	26,33	26,2	25,98	0,49	3,06
353768	44457	julho	26,26	26,17	26,41	26,28	0,12	3,44

5. Resultados *speedup* variando o número de computadores

a) Tempo sequencial

10 mil vértices	
100 mil arestas	4,03
200 mil arestas	7
400 mil arestas	13,37
600 mil arestas	20,83

b) *Speedup* algoritmo HADI

	Computadores				
10 mil vértices	**2**	**4**	**8**	**12**	**16**
100 mil arestas	0,28	0,28	0,28	0,28	0,28
200 mil arestas	0,67	0,56	0,56	0,56	0,56
400 mil arestas	1,27	1,25	1,27	1,25	1,23
600 mil arestas	2,39	2,38	2,36	1,93	2,21

c) *Speedup* algoritmo HEDA

	Computadores				
10 mil vértices	**2**	**4**	**8**	**12**	**16**
100 mil arestas	0,12	0,15	0,17	0,18	0,18
200 mil arestas	0,19	0,24	0,28	0,29	0,29
400 mil arestas	0,34	0,45	0,50	0,53	0,53
600 mil arestas	0,53	0,72	0,79	0,84	0,84

d) *Speedup* algoritmo FastGDC

	Computadores				
10 mil vértices	**2**	**4**	**8**	**12**	**16**
100 mil arestas	0,71	1,06	1,98	2,24	3,31
200 mil arestas	-	1,98	2,14	4,20	4,88
400 mil arestas	-	1,57	2,10	4,36	5,42
600 mil arestas	-	2,72	7,35	10,33	13,44

6. Resultados eficiência variando o número de computadores

a) Eficiência algoritmo HADI

	Computadores				
10 mil vértices	**2**	**4**	**8**	**12**	**16**
100 mil arestas	0,14	0,07	0,03	0,02	0,02
200 mil arestas	0,33	0,14	0,07	0,05	0,03
400 mil arestas	0,63	0,31	0,16	0,10	0,08
600 mil arestas	1,19	0,59	0,30	0,16	0,14

b) Eficiência algoritmo HEDA

	Computadores				
10 mil vértices	**2**	**4**	**8**	**12**	**16**
100 mil arestas	0,06	0,04	0,02	0,01	0,01
200 mil arestas	0,09	0,06	0,03	0,02	0,02
400 mil arestas	0,17	0,11	0,06	0,04	0,03
600 mil arestas	0,26	0,18	0,10	0,07	0,05

c) Eficiência algoritmo FastGDC

	Computadores				
10 mil vértices	**2**	**4**	**8**	**12**	**16**
100 mil arestas	0,35	0,26	0,25	0,19	0,21
200 mil arestas	-	0,50	0,27	0,35	0,31
400 mil arestas	-	0,39	0,26	0,36	0,34
600 mil arestas	-	0,68	0,92	0,86	0,84

7. Algoritmo Sequencial

```
package faccamp.graph;

import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import java.util.SortedMap;
import java.util.TreeMap;
/**
 * Grafo representado por uma lista de adjacencia.
 * Reference: Introduction to Algorithms - CLRS.
* @since: 14/11/2012
 */
public class Grafo {
	private static ArrayList<Vertice> data = new ArrayList<Vertice>();
	private static Map<Vertice, HashSet<Vertice>> map;

	/**
	 * Construtor inicializa a listaAdjacencia e carrega o grafo
	 */
	public Grafo() {
		map = new HashMap<Vertice, HashSet<Vertice>>();
		carregaEdges();
	}

	/**
	 * Este método conecta um Edge ou outro
	 *
```

```
 */
private void conectaEdges(Vertice vertexA, Vertice vertexB) {
	criaArestas(vertexA, vertexB);
	criaArestas(vertexB, vertexA);
}

/**
 * Método que cria aresta entre o 'EdgeA' com o 'EdgeB'. Se 'EdgeA'
 * já existe na listaAdjacencia, pega a lista de adjacencia deste Edge e
 * adiciona o 'EdgeB'. Se não existir cria uma nova lista e adiciona o
 * 'EdgeB' à listaAdjacencia'.
 */
private void criaArestas(Vertice n1, Vertice n2) {
	if (map.containsKey(n1)) {
		map.get(n1).add(n2);
	} else {
		HashSet<Vertice> list = new HashSet<Vertice>();
		list.add(n2);
		map.put(n1, list);
		data.add(n1);
	}
}

/**
 * Método retorna a excentricidade de um Edge informado no parametro
 * Ele encontra o menor caminho utilizando o método BFS
 * e(v) = max {d(u,v)}.
 */

public static Integer menorCaminho(int Edge) {
  Vertice input = findEdgeByid(Edge);
  SortedMap<Vertice, Integer> distance = new TreeMap<Vertice, Integer>();
  SortedMap<Vertice, Boolean> visited = new TreeMap<Vertice, Boolean>();
	Set<Vertice> keys = map.keySet();
	for (Vertice u : keys) {
		if (u != input) {
			distance.put(u, 0);
			visited.put(u, false);
		}
	}

	distance.put(input, 0);
	visited.put(input, true);

	UniqueQueue que = new UniqueQueue();
	//Queue<Edge> que = new LinkedList<Edge>();
	que.add(input);
```

```
        while (!que.isEmpty()) {
                Vertice u = que.remove();
                HashSet<Vertice> adj_u = map.get(u);

                if (adj_u != null) {
                        for (Vertice v : adj_u) {
                                if (!visited.get(v)) {

                                        distance.put(v, distance.get(u) + 1);

                                        visited.put(v, true);
                                        que.add(v);
                                }
                        }
                }
        }

        Collection<Integer> eMax = distance.values();
        return Collections.max(eMax);
}

/**
 * Cria um arraylist com todos os vertices
 * e um Hashmap com todos os vertices adjacentes.
 */

private void carregaEdges() {
        try {
                FileInputStream fstream = new FileInputStream(
                                "/home/administrador/grafo.txt");
                DataInputStream in = new DataInputStream(fstream);
                BufferedReader br = new BufferedReader(new
                InputStreamReader(in));
                String strLine;
                while ((strLine = br.readLine()) != null) {
                        //System.out.println(strLine);
                        String[] aresta = strLine.split("\\t");
                        int source = Integer.parseInt(aresta[0]);
                        int dest = Integer.parseInt(aresta[1]);
                        conectaEdges(new Vertice(source), new Vertice(dest));
                }
                in.close();
        } catch (Exception e) {
                System.err.println("Error: " + e.getMessage());
        }
}
```

```
	/**
	 * Método retorna o vertices pelo id.
	 */
	private static Vertice findEdgeByid(int id) {
		for (Vertice Edges : data) {
			if (Edges.getVertice() == id) {
				return Edges;
			}
		}
		return null;
	}
	/**
	 * Método executa o calculo da excentricade do grafo
	 *
	 */
	public static void execExcentricidade(){
		ArrayList<Integer> allExcentricidade = new ArrayList<Integer>();
		for (Vertice edges : data) {
			allExcentricidade.add(menorCaminho(edges.getVertice()));

		}
		Excentricidade E(G) = max {d(u,v)}.
		System.out.println("Excentricidade:");
		System.out.println(Collections.max(allExcentricidade));
		Raio(G) = min(e(x))
		System.out.println("Raio:");
		System.out.println(Collections.min(allExcentricidade));
		Diametro(G) = max(e(x))
		System.out.println("Diametro:");
		System.out.println(Collections.max(allExcentricidade));
	}

}

package faccamp.graph;

public class ExcentricidadeMain {
	public static void main(String[] args) {
		Grafo g = new Grafo();
		Grafo.execExcentricidade();
		}
}
```

8. Comprovante de submissão de artigo

CSBC 2013 (WPerformanc | JEMS: : Paper (Francisco S

https://jems.sbc.org.br/PaperShow.cgi?m=114059

#114059: *Cálculo de Medidas de Centralidade em Grafos Grandes*

Authors:	81009 Francisco Sanches Banhos Filho (Bacharel em Ciencia da Computacao) 133042 Eduardo Javier Huerta Yero (Faccamp) Send a message to these authors using your personal email client: 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 Processing (BSP), have been considered in recent literature to be more appropriate for this task. This paper proposes a BSP-based algorithm named FastGDC to calculate centrality measures in large graphs and compares its results with HEDA (Hadoop-based Exact Diameter Algorithm), an algorithm based on the MapReduce model. Preliminary results indicate that FastGDC is indeed faster than HEDA for the graphs tested.
Topics:	Avaliação de Desempenho em Sistemas Distribuídos
Conference and **track**:	*CSBC 2013 (WPerformance)* / **Trilha Principal**
Paper identifier:	581066567
Notes:	
Printing problems:	
Status:	

Uploaded files:

Description	Upload deadline	Allowed file types	Max size	Upload	Show	Size	Created	Delete
Paper manuscript	Apr 24, 2013 - 09:00 AM (BRT)	pdf	unlimited			0.2 MB	May 10, 2013 - 02:27 AM	
Versão Final	Jun 10, 2013 - 11:55 PM (BRT)	pdf	unlimited		no files uploaded			
Copyright	Jun 10, 2013 - 11:55 PM (BRT)	pdf	unlimited		no files uploaded			

Type	Reviewer	Assigned (history)	Assigned by	Confirmed	Reminded	Due	Completed
review		2013-04-24 11:10:20	2013-04-24 11:10:20	2013-05-15 12:44:44	2013-05-13 09:45:05	2013-05-28	2013-05-15 12:54:20
Qual é o seu parecer final sobre o artigo?: .		*Qual é a relevância do artigo para o WPerformance?:* .	*Qual é o grau de originalidade do artigo?:* .	*Avalie o artigo quanto a sua qualidade técnica:* .	*Avalie o artigo quanto a qualidade da apresentação (escrita, organização, clareza, referências, figuras e tabelas, etc):* .		
4: aceitação forte		2: relevante	3: médio	3: média	3: muito bom		

Enumere os pontos fortes do artigo (uma ou duas frases por ponto): .

Artigo bem escrito, com introdução que contextualiza bem o trabalho. Aplicação avaliada é atual e relevante.

9. Artigo Submetido

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 Processing (BSP), have been considered in recent literature to be more appropriate for this task. This paper proposes a BSP-based algorithm named FastGDC to calculate centrality measures in large graphs and compares its results with HEDA (Hadoop-based Exact Diameter Algorithm), an algorithm based on the MapReduce model. Preliminary results indicate that FastGDC is indeed faster than HEDA for the graphs tested.*

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 Processing (BSP), tem sido sugerido como mais apropriado para resolver problemas em grafos. Este trabalho propõe um algoritmo baseado em BSP chamado FastGDC para calcular medidas de centralidade em grafos grandes e compara seus resultados com o HEDA (Hadoop-based Exact Diamenter Algorithm), um algoritmo baseado no modelo MapReduce. Resultados preliminares apontam que o desempenho do FastGDC é superior ao do HEDA para os grafos testados.*

1. Introdução

Durante a última década a quantidade de dados disponível digitalmente tem aumentado de forma significativa. A explosão no uso das redes sociais e de sistemas para compartilhamento de músicas e vídeos, além da consolidação do comércio eletrônico, são os maiores responsáveis por este aumento. A maior disponibilidade de dados coletados através de logs de servidores, sensores e satélites, dentre outras fontes de dados digitais, completam este cenário.

Em 2012, aproximadamente 2.5 exabytes são criados diariamente, e estima-se que a quantidade total de dados armazenados atinja 2.7 zetabytes [Mewawalla 2012]. Este crescimento esta se dando de forma tão acelerada que 90% dos dados armazenados hoje foram criados nos últimos dois anos [Mewawalla 2012]. Para nomear este fenômeno – a geração, armazenamento, processamento e análise de grandes massas de dados – tem-se utilizado o termo Big Data [Pettey and Goasduff 2011].

A empresa Gartner em [Pettey and Goasduff 2011] define os desafios e as oportunidades associados a Big Data em três dimensões: volume, onde a complexidade

esta associada à quantidade, tanto em dados tradicionais quanto em novos tipos de dados; velocidade, onde se busca analisar os dados com a mesma velocidade em que são produzidos; e variedade, relacionado ao processamento de diferentes tipos de informações (e.g. tabular, hierárquica, e-mails, fotos, áudio, etc.).

Para processar estas grandes massas de dados tem-se utilizado técnicas como *Complex Event Processing* [Wasserkrug et al. 2008], *Stream Processing* [Gummaraju and Rosenblum 2005], MapReduce [Dean and Ghemawat 2008] e *Bulk-synchronous parallel* (BSP) [Valiant 1989]. O ponto em comum entre todas elas é a preocupação por resolver de forma inovadora um ou vários dos "V" definidos pela Gartner. *Complex Event Processing* e *Stream Processing* focam em soluções em tempo real, enquanto BSP e MapReduce tem como alvo problemas com grandes volumes de dados. O presente trabalho abordará MapReduce e BSP.

Dentre os problemas normalmente resolvidos usando técnicas de Big Data destacam-se aqueles associados a grafos [Bondy and Murty 2008]. 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 normalmente utilizadas para Big Data. No entanto, os grafos apresentam desafios únicos que fazem com que a aplicabilidade destas técnicas não seja tão natural: por exemplo, há comumente uma interdependência forte entre os nós do grafo, o que dificulta a paralelização da maioria dos algoritmos que tratam com eles. Estes problemas magnificam-se quando se trata de grafos com milhões de nós e arestas.

Este trabalho terá como foco o cálculo de medidas de centralidade em grafos grandes. Cálculos de centralidade envolvem o calculo da distância entre todos os vértices do grafo. Para cada vértice, a maior destas distâncias é sua excentricidade. A maior excentricidade dentre todos os vértices é o diâmetro do grafo, e a menor, o raio [Hage and Harary 1995]. Estas medidas de centralidade são essenciais para muitos dos problemas que tratam com grafos, como por exemplo aqueles que precisam determinar pontos de distribuição para uma rede determinada (e.g. localização de centrais telefônicas). No entanto, para grafos grandes tem-se usado medidas aproximadas para substituí-las por ser impraticável o cálculo das medidas exatas em um tempo razoável [Kang et al. 2011]. Hoje, com as técnicas de Big Data, é possível analisar grafos com milhões, e até mesmo bilhões de vértices, oferecendo soluções exatas.

Algumas soluções exatas para o problema de centralidade em grafos grandes já foram implementadas. Em [Nascimento and Murta 2012] utiliza-se o modelo de programação distribuída MapReduce para calcular a excentricidade de todos os nós do grafo. Apesar de apresentar resultados promissores, alguns artigos na literatura mais recente apontam o modelo BSP como sendo mais apropriado para tratar com grafos [Pace 2012]. Neste trabalho será explorada esta hipótese utilizando o modelo BSP para calcular a excentricidade de todos os nós de um grafo e comparar seus resultados com o algoritmo proposto em [Nascimento and Murta 2012].

O restante deste trabalho está organizado como a seguir. A Seção 2 apresenta os trabalhos relacionados. Os modelos *Bulk-synchronous parallel* e MapReduce são apresentados na Seção 3. O algoritmo proposto é apresentado na Seção 4. A Seção 5 relaciona o *dataset* utilizado e os testes realizados. A Seção 6 trata da análise dos resultados. Na Seção 7 são apresentadas as conclusões e os trabalhos futuros.

2. Trabalhos Relacionados

Hage e Harary definem três tipos de centralidade em [Hage and Harary 1995): centralidade de grau (*Degree Centrality*), centralidade de proximidade (*Closeness Centrality*) e centralidade de intermediação (*Betweenness Centrality*). Estes modelos são comumente aplicados em estudos que envolvem redes sociais. Os autores propuseram uma nova medida de centralidade baseada no cálculo da excentricidade de cada vértice, que eles chamaram de *Efficiency Centrality*. Na teoria dos grafos esta medida é conhecida apenas como centralidade.

Para calcular a centralidade de um grafo é necessário utilizar alguma estratégia que resolva o problema de caminhos mais curtos de todos os pares (*All-Pair Shortest Path*). Os algoritmos de Dijkstra e Bellman-Ford se aplicados a cada vértice do grafo resolve o problema de maneira sequencial [Cormen et al. 2001; Kleinberg and Tardos 2006].

O restante desta seção apresenta dois algoritmos que usam MapReduce para calcular medidas de centralidade em grafos grandes: o HADI e o HEDA.

2.1. HADI (Hadoop based DIameter estimator)

O HADI (*Hadoop based DIameter estimator*) é um algoritmo que calcula o raio e o diâmetro de grafos grandes utilizando técnicas de aproximação. O algoritmo é implementado seguindo o modelo MapReduce e executado usando Hadoop. Ele possui uma versão mais simples chamada HADI-*naive* e uma versão mais performática chamada HADI-*plain*, que possui melhoras na fase Reduce garantindo um melhor desempenho [Kang et al. 2011].

O HADI usa o algoritmo de Flajolet-Martin para estimar o diâmetro efetivo do grafo, que é a quantidade mínima de saltos necessária para conectar pelo menos 90% dos nós do grafo. Este algoritmo consegue diminuir a complexidade de tempo e espaço necessárias para o cálculo exato ao custo de fornecer apenas informações aproximadas. A complexidade de tempo do HADI para um grafo de diâmetro d com n nós e m arestas processado usando M máquinas é de $O(\frac{d(m+n)}{M}\log\frac{m+n}{M})$, e a complexidade de espaço de $O((m+n)\log n)$.

2.2. HEDA (Hadoop-based Exact Diameter Algorithm)

Como alternativa ao HADI, Nascimento e Murta propuseram o HEDA (*Hadoop-based Exact Diameter Algorithm*) [Nascimento and Murta 2012]. Diferente do HADI, o HEDA determina o diâmetro exato a partir do cálculo do caminho mais curto entre todos os pares de nós do grafo. Este problema, chamado de *All Pairs Shortest Path*, é implementado em [Nascimento and Murta 2012] através da busca em largura partindo de cada um dos nós do grafo.

A complexidade de espaço do algoritmo HEDA para um grafo com n nós e m arestas é de $O(n^2 + (n+m))$, correspondente ao tamanho do arquivo de distâncias e o arquivo inicial de arestas. Já a complexidade de tempo é $O(\frac{n(n+m)}{M})$ para um grafo processado usando M computadores.

3. Modelo BSP x Modelo MapReduce

Nesta seção apresentamos brevemente os modelos MapReduce e BSP e fazemos algumas considerações sobre suas capacidades relativas para resolver alguns tipos de problemas.

3.1. MapReduce

MapReduce é um modelo de programação criado pelo Google para processar conjuntos grandes de dados [Dean and Ghemawat 2008]. O modelo consiste em duas funções, *map* e *reduce*, que combinadas permitem dividir os dados a serem processados entre um conjunto potencialmente grande de nós de processamento e posteriormente gerar o resultado final a partir dos resultados parciais de cada nó. A função *map* recebe um conjunto de pares chave/valor de entrada e produz um conjunto de pares chave/valor intermediários. A implementação do modelo MapReduce é então encarregada de agrupar todos os valores intermediários que têm a mesma chave e entregá-los à função *reduce*, que produz o conjunto final de pares chave/valor.

A principal vantagem do modelo MapReduce é que permite ao programador se concentrar no problema específico a ser resolvido, enquanto os detalhes de distribuição de carga, tolerância a falhas e processamento de resultados intermediários são responsabilidade da implementação do modelo. Um problema que comumente é apresentado como primeiro exemplo de aplicação MapReduce é o de contar a quantidade de vezes que cada palavra aparece em um conjunto de documentos potencialmente grande. O pseudo-código da função *map* é mostrado no Algoritmo 1 e da função *reduce* no Algoritmo 2 [Dean and Ghemawat 2008].

Algoritmo 1. Função Map

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
EmitIntermediate(w, "1");
```

Algoritmo 2. Função Reduce

```
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
result += ParseInt(v);
Emit(AsString(result));
```

A função *map* emite cada palavra com um contador associado, que contém o valor "1" nesta implementação mais simples. A função *reduce* soma os contadores de todas as entradas geradas para cada palavra.

3.1.1. Apache Hadoop

O framework Apache Hadoop desenvolvido pela *Apache Software Foudation* é o mais conhecido dentre os frameworks de livre utilização que implementam o modelo MapReduce. Escrito em Java, o Hadoop é utilizado para a execução de aplicativos em grandes aglomerados de computadores. Grandes empresas, como Yahoo, utilizam o Hadoop na implementação de partes do motor de busca [Yahoo Launches World's Largest Hadoop Production Application 2008]. De modo geral, Hadoop é utilizado

como solução para problemas que envolvam *data warehousing*, inteligência de negócio, aplicações analíticas e mídias sociais.

O Hadoop é hoje uma coleção de subprojetos correlatos que juntos fornecem uma infraestrutura completa para computação distribuída em grande escala. Embora o Hadoop seja mais conhecido por MapReduce e seu sistema de arquivos distribuídos (HDFS - *Hadoop Distributed File System*), os outros subprojetos fornecem serviços complementares. Na Tabela 1 são apresentados os subprojetos que compõem o Apache Hadoop.

Tabela 1. Subprojetos que compõem o Apache Hadoop [White 2009]

<table>
<tr><td>Pig</td><td>Chukwa</td><td>Hive</td><td>Hbase</td></tr>
<tr><td>MapReduce</td><td colspan="2">HDFS</td><td>Zookeeper</td></tr>
<tr><td colspan="2">Core</td><td colspan="2">Avro</td></tr>
</table>

Segue uma breve descrição de cada subprojeto.

- Core: conjunto de componentes e interfaces para sistemas de arquivos distribuídos e I/O (*Input/Output*) universal;
- Avro: sistema de serialização de dados eficiente com armazenamento de dados persistente;
- MapReduce: responsável pelo processamento distribuído de dados e ambiente de execução que roda em grandes aglomerados de máquinas;
- HDFS: sistema de arquivos distribuído que funciona em grandes aglomerados de máquinas;
- Pig: linguagem de fluxo de dados e ambiente de execução para explorar conjuntos de dados muito grandes;
- HBase: banco de dados distribuído orientado a coluna;
- ZooKeeper: serviço distribuído de administração altamente disponível, fornecendo primitivas como bloqueios distribuídos;
- Hive: sistema para *data warehouse* distribuído que gerencia os dados armazenados no HDFS e fornece uma linguagem de consulta baseada em SQL;
- Chukwa: sistema para análise de dados distribuídos.

Na próxima seção descrevemos o modelo BSP e o Apache Hama, uma das implementações mais conhecidas deste modelo.

3.2. BSP

O modelo *Bulk Synchronous Parallel* (BSP), proposto por Valient em 1989 [Valiant 1989], define um computador abstrato que age como uma ponte entre hardware e software. Este tipo de modelo, conhecido como *bridging models*, se caracterizam por conter elementos de arquitetura de computadores e de arquitetura de software. O exemplo mais conhecido deste tipo de modelos é a máquina de von Neumann, que teve papel essencial no desenvolvimento da computação ao permitir que desenvolvedores de hardware e software pudessem trabalhar de forma relativamente isolada focando apenas nas características do modelo descrito.

Um computador BSP consiste de uma coleção de processadores, cada um com memória privada, conectados por uma rede de comunicação. Um algoritmo BSP consiste de uma sequência de *supersteps*. Um *superstep* compreende três componentes, explicados a seguir.

- **Computação concorrente**: cada processador participante executa processos independentes e assíncronos que usam apenas dados armazenados nas suas respectivas memórias locais.

- **Comunicação**: os processos trocam informações entre eles usando primitivas unilaterais *put* e *get*, que acessam e modificam posições de memória de outros processadores.

- **Sincronização global**: quando um processo atinge este ponto ele aguarda por que todos os outros processos concluam suas ações de comunicação.

A Figura 1 apresenta o fluxo de um algoritmo de BSP. Nela é possível observar um algoritmo BSP com cinco *supersteps* executados em cinco processadores. As linhas verticais indicam a computação local de cada processador; as setas indicam a comunicação com outro processador. As linhas marcadas com a palavra *comp* indicam *supersteps* de computação. As linhas marcadas com a palavra *comm* indicam *supersteps* de comunicação. No primeiro *superstep* de comunicação o processador P(0) envia dados para todos os outros processadores. Cada *superstep* é terminado por uma sincronização global.

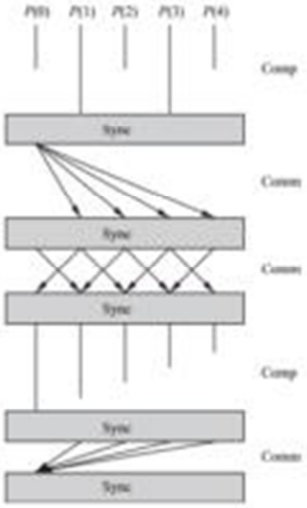


Figura 1. Passos de um Algoritmo BSP [Bisseling 2004]

A próxima seção apresenta um framework do modelo BSP.

3.2.1. Apache Hama

Hama [Apache Hama 2013] é um *framework* para computação distribuída baseado no modelo BSP que tem seu funcionamento inspirado no Pregel [Malewicz et al. 2010]. O Hama é mantido pela *Apache Software Foundation* e, portanto, usufrui de parte do ecossistema de projetos conforme mostrado na Tabela 1. Em particular, o projeto *Hadoop Distributed File System* (HDFS) é extensivamente utilizado para armazenar os dados a serem processados e os resultados gerados pelos aplicativos Hama.

A Figura 2 mostra o fluxo de execução de um aplicativo Hama.

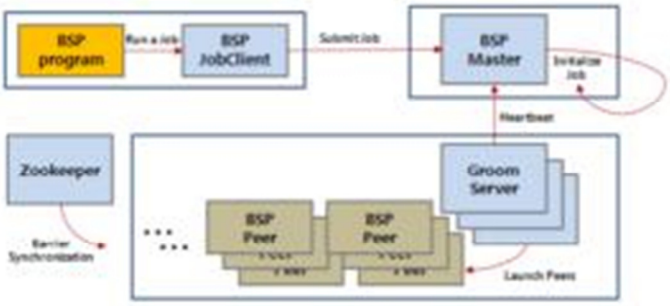


Figura 2. Arquitetura Hama [Apache Hama 2013]

A solicitação para iniciar a execução de um aplicativo em Hama é feita pelo programa do usuário através de uma cópia local do *BSP JobClient*, cuja função é a de contatar o *BSP Master*, submeter o pedido de execução e aguardar pelos resultados. Ao receber um pedido o *BSP Master* contata os *Groom Servers* e aloca as tarefas a serem executadas por cada um. Cada *Groom Server* é responsável por pelo menos uma instância de *BSP Peer*, que é encarregada de executar de fato o código próprio do aplicativo. Os *Groom Servers* contatam periodicamente o *BSP Master* através de um *heartbeat* para informar seu estado, que inclui a quantidade de tarefas sendo executadas, quantidade de memória livre, entre outras informações necessárias para que o *BSP Master* possa executar suas funções. Por fim, o *Zookeeper* é encarregado de implementar as primitivas de sincronização de barreira que fazem parte do modelo BSP.

3.3 Considerações

O modelo de processamento Bulk Synchronous Parallel proposto por Valient [Valiant, 1989] é um modelo bastante conhecido. Ele tem como característica principal a troca de mensagens entre os pares para informar status, gerenciar e sincronizar o processamento. Este modelo se mostrou eficiente na implementação de diversos algoritmos paralelos [Cáceres et al. 1997]. Já o modelo MapReduce proposto por Dean e Ghemawat [Dean and Ghemawat 2008] tem seu funcionamento baseado na troca de informações através de um sistema de arquivo distribuído (e.g. *Hadoop Distributed File System*, ou HDFS). O modelo MapReduce tem revolucionado a maneira com que a análise de grandes massas de dados vinham sendo feitas por apresentar um modelo escalável e tolerante a falhas.

Embora os dois modelos difiram na maneira com que trata as informações, eles também possuem algumas semelhanças. Pace propõe dois teoremas que provam que os dois modelos são equivalentes para algumas classes de problemas [Pace 2012]. Dentre seus resultados destaca-se a conclusão de que existem alguns problemas para os quais o modelo MapReduce não apresenta resultados satisfatórios.

Pace analisou os problemas de *Sorting*, *Dense Matrix Multiplication* e *Breadth First Search* sobre a eficiência ou não de uma implementação utilizando o modelo MapReduce. Ele reforça que os teoremas propostos por ele provam que o problema *Breadth First Search* não poderá ser implementado de maneira eficiente utilizando o modelo MapReduce.

Outro indicativo que o modelo MapReduce não escala muito bem para todos os problemas é mostrado por Jain et. al. em [Jain and Mallozzi 2010]. Neste artigo, os

autores demostram uma implementação do problema do Caixeiro Viajante mal sucedida utilizando MapReduce.

Baseado nestes resultados prévios, o presente trabalho apresenta uma solução para o cálculo de medidas de centralidade em grafos grandes utilizando o Modelo BSP e compara os resultados com aqueles obtidos pelas implementações que utilizam o Modelo MapReduce.

4. Algoritmo Proposto

O algoritmo proposto foi denominado FastGDC (*Fast Graph Diameter Calculator*). Ele tem como objetivo efetuar o cálculo exato de medidas de centralidade em grafos, principalmente diâmetro e raio. A aplicação FastGDC utiliza o modelo *Bulk Synchronous Parallel* (BSP) na manipulação de grafos grandes. Dentre os frameworks que implementam este modelo destacam-se o GPS [Salihoglu and Widom 2012], Giraph [Apache Incubator Giraph 2012], e o Apache HAMA. Este último acabou sendo escolhido para os testes por aparentar estar em um estágio mais avançado de desenvolvimento.

A seguir mostramos o algoritmo utilizado para calcular as medidas de centralidade de um grafo usando o modelo BSP.

Algoritmo 3. Método Compute ()

Entrada: Grafo, Tamanho do vetor de distâncias

Saída: Arquivo contendo as excentricidades de todos os vértices

```
1 Método compute
2 // First superstep. Create and initialize distance_vector.
3 // It will contain "0" for the current node, "1" for neighbours
4 // of the current node and "MAX_INTEGER" for all other nodes
5   if (superstep == 0)
6           distance_vector = new Vector(NODE_COUNT);
7           for each index in distance_vector
8                   if (index == current_node)
9                           distance_vector[index] = 0;
10                  else if (isNeighbour(index, current_node))
11                          distance_vector[index] = 1;
12                  else
13                          distance_vector[index] = MAX_INTEGER;
14                  endif
15          endfor
16  else
18           // shouldHalt will indicate whether this node made any changes in this
19          // superstep. Otherwise, this node will vote to halt the processing.
20          // When all nodes vote to halt, the processing stops.
21          shouldHalt = true;
22          // Now we process every message received looking for path to nodes
23          // that are shorter than the ones we have.
24          for each message in messages_received_from_neighbouring_nodes
25                  received_distance_vector = getContents(message);
26                  // we got the distance vector from a neighbouring node. Now we analyze
27                  // every entry looking for better paths
28                  for each index in received_distance_vector
29                          current_entry = distance_vector[index];
30                          remote_entry = received_distance_vector[index];
31                          if (remote_entry < current_entry)
32                                  // a better path has been found. Now we update the distance vector
33                                  // and the eccentricity of this node. We also set shouldHalt to false,
34                                  // since the current node was updated.
35                                  shouldHalt = false;
36                                  distance_vector[index] = received_distance_vector[index];
37                                  if (getEccentricity(current_node) < remote_entry)
```

```
38                                  setEccentricity(current_node, remote_entry);
39                              endif
40                      endif
41              endfor
42          endfor
43
44          // If nothing was updated, vote to halt processing.
45          if (shoulHalt)
46              voteToHalt(current_node);
47          else
48              // Now we start the communication phase. We will prepare a distance vector to be sent
49              // to neighbouring nodes which contains the current distance values for this nodes plus 1.
50              // In other words, it contains the distance from the neighbouring nodes to any other
51              // node when using the current node as the first step of the path.
52              distance_vector_to_send = distance_vector
53              for each index in distance_vector_to_send
54                      distance_vector_to_send[index] = distance_vector[index]+1;
55              endfor
56
57              // Send the calculated vector to neighbouring nodes.
58              for each node in neighbour_nodes
59                      send(node, distance_vector_to_send);
60              endfor
61          endif
62      endif
63 end Método Compute
```

O Algoritmo 3 é executado para cada nó do grafo. Ele começa no *superstep* 0, que é dedicado apenas a fazer as inicializações necessárias para o processamento. Em particular, um vetor de distâncias é criado e inicializado contendo 0 na entrada correspondente ao nó atual (chamado de *current_node* no algoritmo), 1 para todos os vizinhos do nó atual e o valor máximo possível para números inteiros em todas as outras entradas.

A partir deste ponto começa uma sequência de *supersteps* que contém 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 um vetor de distâncias que reflete o tamanho do caminho desde o nó atual até qualquer outro nó através do nó vizinho. Como cada vizinho envia uma mensagem similar, o nó atual é informado do comprimento de todos os caminhos possíveis até qualquer outro nó. Inicialmente muitos destes valores ainda não foram calculados e, portanto, o processo precisa ser repetido até que não haja novas atualizações.

Caso não tenha havido nenhuma atualização no vetor de distâncias do nó atual após processar todas as mensagens o nó atual emite seu voto para que o processamento termine. Caso contrário, o ele cria um vetor de distâncias que reflete o comprimento do caminho desde um nó vizinho até qualquer outro nó passando pelo nó atual e o envia a todos seus nós vizinhos. Este vetor é em essência o vetor de distâncias do nó atual incrementado em 1 em cada entrada.

Quando o algoritmo termina cada nó contém sua excentricidade no grafo. O menor valor de excentricidade é o raio do grafo e o maior valor seu diâmetro.

Em termos de complexidade computacional para um grafo com n nós e m arestas processados usando M computadores, o FastGDC é equivalente ao HEDA ($O(\frac{n(n+m)}{M})$), já que precisa executar uma busca em largura partindo de cada nó do grafo. A complexidade de espaço, no entanto, é menor, visto que cada nó armazena

apenas dois vetores de distâncias que contém uma entrada para cada vértice do grafo, o que gera uma complexidade de $O(n^2)$.

5. Testes

Para avaliar o algoritmo proposto neste trabalho utilizamos dois grafos sintéticos com 10 mil e 15 mil nós respectivamente, além de um grafo conhecido como "High Energy Physics Theory", que contém 9877 nós. Todos eles foram processados utilizando o FastGDC e o HEDA.

O ambiente de testes foi configurado com 14 computadores com processador AMD Phenom II com 4 núcleos de processamento, 4 Gigabytes de memória e 250 Gigabytes de espaço em disco. Para comunicação foi utilizado um switch padrão Ethernet 10/100 Mbps.

Cada um dos 14 computadores recebeu o Sistema Operacional Ubuntu 12.10 versão Desktop, a Java *Virtual Machine* versão 1.6 e o Apache Hadoop na versão 1.0.4. Por se tratar de um framework em desenvolvimento o HAMA foi instalado em sua versão *Trunk* 0.6.0.

O Hadoop foi configurado de acordo com as informações disponibilizadas por Noll [Noll 2012] e os parâmetros recomendados para execução do HEDA por Nascimento e Murta em [Nascimento and Murta 2012]. Dos 14 computadores, 2 deles foram escalados para executar os serviços Zookeeper (sincronização) e Datanode (sistema de arquivos distribuído HDFS). O HAMA foi configurado seguindo um documento disponibilizado pelo desenvolvedor do framework em [Apache Hama Installation Guide 2012]. Uma série de testes iniciais foram feitos para determinar os parâmetros que produziam os melhores resultados de desempenho.

Na próxima subseção apresentamos os resultados parciais dos experimentos.

5.1. Resultados Experimentais HEDA X FastGDC

A seguir mostramos os resultados comparativos obtidos nos testes com 3 grafos: um grafo sintético de 15 mil vértices, o grafo de "High Energy Physics Theory" de 9877 vértices e por fim um grafo sintético de 10 mil vértices. Os tempos de execução utilizados nos comparativos correspondem à media de 3 tempos de execução obtidos em testes independentes.

5.2. Grafo sintético 15 mil vértices

Este teste consiste em calcular o diâmetro e o raio de um grafo sintético de 15 mil vértices gerados utilizando o modelo de Erdös-Rényi [Erdos & Renyi 1959]. Os tempos de execução dos Algoritmos FastGDC e HEDA podem ser visualizados na Tabela 2.

Tabela 2. Tempo de execução FastGDC Grafo 15 mil vertices

Algoritmos	Tempo de execução			Média
FastGDC	51m26s	51m21s	51m11s	**51m19s**
HEDA	73m33s	76m10s	75m8s	**74m57s**

A diferença entre as soluções foi de 23m38s e pode ser melhor visualizada na Figura 3.

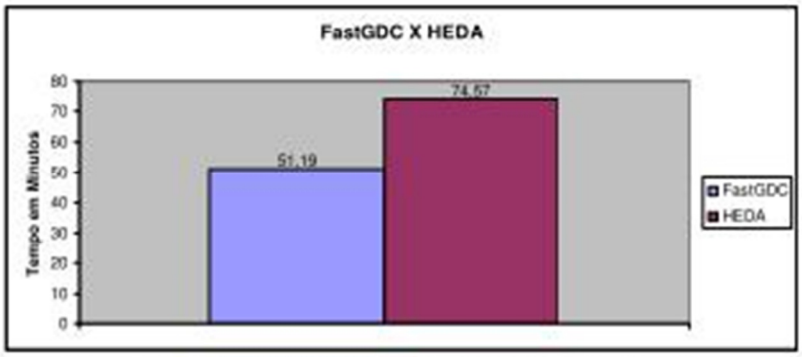


Figura 3. Gráfico FastGDC X HEDA Grafo 15 mil vértices

5.3. Grafo High Energy Physics Theory

Neste teste foi utilizado um grafo com 9877 vértices disponibilizado pela Universidade de Stanford, e que está disponível em [Stanford Large Nertwork Dataset Collection 2012]. Os tempos de execução, juntamente com a média de tempo gasto por cada Algoritmo – HEDA e FastGDC – são mostrados na Tabela 3.

Tabela 3. Tempo de execução FastGDC Grafo Dataset Stanford

Algoritmos	Tempo de Execução			Média
FastGDC	32m37s	32m45s	34m37s	**33m20s**
HEDA	44m53s	45m54s	41m47s	**44m11s**

A diferença de tempo de execução das soluções foi de 11m31s e pode ser melhor visualizada de acordo com a Figura 4.

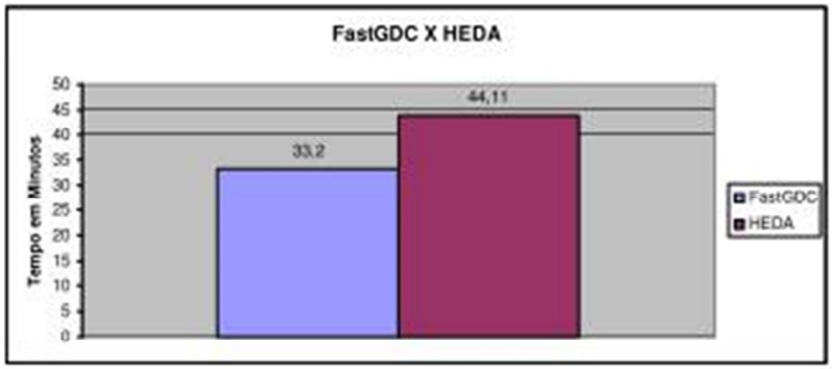


Figura 4. Gráfico FastGDC X HEDA Grafo 9877 vértices

5.4 Grafo sintético de 10 mil vértices adicionando computadores

Para este teste foi utilizado um grafo sintético de 10 mil vértices, executando primeiramente em um conjunto menor de computadores e depois adicionando computadores um a um. O Hadoop pôde ser testado a partir de 3 computadores, enquanto o Hama exigiu pelo menos 6 para conseguir executar o processamento sem falhas.

O resultado da comparação é apresentado na Figura 5.

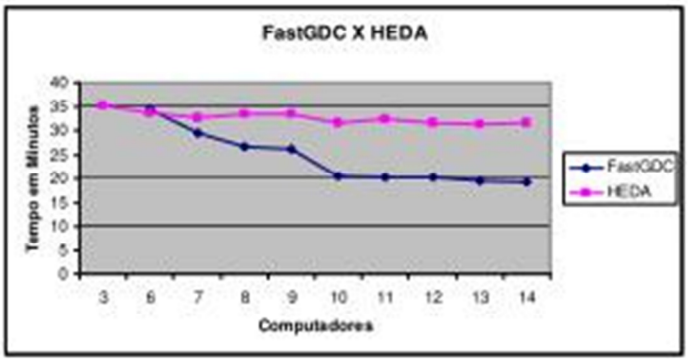


Figura 5. Comparação FastGDC X HEDA variando o número de computadores

6. Análise dos resultados

Os resultados mostrados na seção apontam para a superioridade do modelo BSP para o cálculo de medidas de centralidade em grafos grandes quando comparado ao MapReduce. Para o grafo de 15 mil vértices mostrado na Figura 3 a redução de tempo é da ordem de 30%, ao passo que para o grafo da Figura 4, com 9877 vértices, a redução é de 25%. Estes resultados parecem indicar que o BSP aumenta a vantagem sobre o MapReduce na medida em que o tamanho do grafo aumenta, um resultado bastante promissor.

A Figura 5 mostra que ambos os modelos estabilizaram seu tempo de execução a partir de um determinado número de computadores, indicando que o tamanho do grafo sendo processado (10 mil vértices) não foi suficiente para usufruir das contribuições feitas pelo poder computacional sendo adicionado ao sistema. Em outras palavras, os ganhos potenciais de desempenho trazidos pelos computadores adicionados foram amortizados pela sobrecarga extra para gerenciá-los. No entanto, cabe destacar que o Hama não mostrou sinais de saturação e se comportou da mesma forma que o Hadoop, que implementa um modelo reconhecidamente escalável.

Embora reconhecidamente limitados, tanto pelo tamanho reduzido dos grafos quanto pela quantidade de computadores utilizados, estes testes parecem validar a hipótese de que o BSP é mais apropriado do que o MapReduce para trabalhar com grafos grandes. Os algoritmos de grafos, e em particular os que calculam medidas de centralidade, são caracterizados por uma alta taxa de troca de informações entre os vértices. Pelo menos nas implementações mais conhecidas do MapReduce, como o Hadoop, esta troca de informações é feita através de leituras e escritas em um sistema de arquivos distribuído acessível por todos os nós de processamento. Este mecanismo, quando comparado à troca de mensagens utilizada pelo modelo BSP, introduz uma sobrecarga significativa em cada iteração do algoritmo. Acreditamos ser esta a razão fundamental para as diferenças de tempo de execução constatadas nos testes.

7. Conclusões e trabalhos futuros

Este trabalho apresentou o algoritmo FastGDC para cálculo de medidas de centralidade em grafos grandes. Este algoritmo é baseado no modelo BSP, que separa claramente as etapas de computação, comunicação e sincronização para melhorar sua escalabilidade. Os resultados obtidos foram comparados com o HEDA, um algoritmo com propósito similar que segue o modelo MapReduce. O FastGDC foi executado usando o Apache

Hama, uma implementação conhecida do modelo BSP, enquanto o HEDA foi testado utilizando o Apache Hadoop, e seguindo as sugestões de configuração especificadas originalmente pelos seus autores em [Nascimento and Murta 2012].

Os resultados preliminares apresentados neste trabalho indicam que o FastGDC é entre 25 e 30% mais rápido que o HEDA. A análise destes resultados e das características do BSP e o MapReduce parecem indicar que o principal responsável por esta diferença é o mecanismo utilizado por ambos os modelos para comunicar resultados intermediários entre uma etapa e outra. No caso do Hadoop, esta comunicação de dá através da leitura e escrita de arquivos em um sistema de arquivos distribuído chamado HDFS e fornecido como parte do ecossistema do Hadoop. Já o Hama utiliza mecanismos de troca de mensagens para o mesmo fim.

Dentre os trabalhos futuros planejados para esta pesquisa está o processamento de grafos com mais vértices usando um número maior de nós computacionais. Neste sentido, os autores têm enfrentado problemas com a maturidade do Hama, que acusou problemas de gerenciamento de memória quando o tamanho do grafo aumentou. Outros softwares que implementam o modelo BSP, como o GPS [Salihoglu and Widom 2012] e o Giraph [Apache Incubator Giraph 2012], encontram-se em um estágio ainda mais incipiente de desenvolvimento. Os autores estão trabalhando diretamente com os desenvolvedores do Hama para tentar sanar estes problemas e conseguir conduzir testes mais pesados.

Por fim, cabe destacar outros esforços de pesquisa sendo conduzidos dentro do grupo, como a utilização de bancos de dados maciçamente paralelos para resolver os problemas de centralidade e comparar seus tempos de execução com os obtidos pelo FastGDC e o HEDA.

7. Referências

Yahoo Launches World's Largest Hadoop Production Application. (10 de Fevereiro de 2008). Fonte: Developer Yahoo: http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html

Apache Hama Installation Guide. (2012). Fonte: Apache.org: https: //issues.apache.org/jira/secure/attachment/12528220/ApacheHamaInstallationGuid

Apache Incubator Giraph. (2012). Fonte: Apache Giraph: http://incubator.apache.org/giraph/

Stanford Large Nertwork Dataset Collection. (2012). Fonte: Stanford.edu: http://snap.stanford.edu/data/

Apache Hama. (2013). Fonte: Apache Hama: https://issues.apache.org/jira/secure/attachment/12528219/ApacheHamaDesign.pdf

Bisseling, R. H. (2004). *A structured approach using BSP and MPI*. Londres: OXFORD University Press.

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

Cáceres, E. N., Dehne, F., Ferreira, A., Flocchini, P., Rieping, I., Roncato, A.,Song, S. W. (1997). Efficient parallel graph algorithms for coarse grained multicomputers and

BSP. *24th International Colloquium on Automata, Languages, and Programing* (pp. 390-400). Lecture Notes in Computer Science.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to Algorithms.* Cambridge: Mit Press.

Dean, J., & Ghemawat, S. (2008). *ACM Digital Library*, 107-113.

Erdos, P., & Renyi, A. (1959). On random graphs I. *Publ. Math. Debrecen.*

Gummaraju, J., & Rosenblum, M. (2005). Stream Programming on General-Purpose. *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)* (pp. 343-354). Washington: IEEE Computer Society.

Hage, P., & Harary, F. (1995). Eccentricity and centrality in networks. *Social Networks*, Volume 17, Issue 1.

Jain, S., & Mallozzi, M. (2010). *Parallel Heuristics for TSP on MapReduce.* Brow University Data-Intensive Scalable Computing.

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.

Kleinberg, J., & Tardos, E. (2006). *Algorithm Design.* Boston: Addison Wesley.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel: A System for Large-Scale Graph Processing. *SIGMOD '10 Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135-146.

Mewawalla, C. (2012). Big Data. *Global investment themes: telecoms, media and technology*, 3-19.

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.

Noll, M. G. (20 de 07 de 2012). *Tutorial Multi Node Cluster.* Fonte: Michael-noll.com: www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/

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

Pettey, C., & Goasduff, L. (27 de Junho de 2011). *Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data.* Fonte: Gartner: http://www.gartner.com/newsroom/id/1731916

Salihoglu, S., & Widom, J. (2012). *GPS: A Graph Processing System.* Stanford InfoLab.

Valiant, L. G. (1989). Bulk-synchronous parallel computers. *In Parallel Processing and Artificial Intelligence*, 15-22.

Wasserkrug, S., Gal , A., Etzion, O., & Turchin, Y. (2008). Complex Event Processing Over Uncertain Data. *Proceedings of the second international conference on Distributed event-based systems* (pp. 253-264). New York: ACM.

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