UNIVERSIDADE FEDERAL DE OURO PRETO

DEPARTAMENTO DE COMPUTAÇÃO

Matheus de Oliveira Correia

JCL PAGE RANK: UMA SOLUÇÃO DISTRIBUÍDA PARA GRAFOS MASSIVOS

Ouro Preto, MG
2019

Matheus de Oliveira Correia

UNIVERSIDADE FEDERAL DE OURO PRETO

DEPARTAMENTO DE COMPUTAÇÃO

Monografia II apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Joubert de Castro Lima

Coorientador: André Luis Barroso Almeida

Ouro Preto, MG
2019

C817j Correia, Matheus de Oliveira.
JCL Page Rank [manuscrito]: uma solução distribuída para grafos massivos / Matheus de Oliveira Correia. - 2019.

29f.: il.: grafs; tabs.

Orientador: Prof. Dr. Joubert de Castro Lima.
Coorientador: Prof. MSc. André Luis Barroso Almeida.

Monografia (Graduação). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Departamento de Computação.

1. Teoria dos grafos. 2. Sistemas operacionais distribuídos (Computadores) . 3. Java (Linguagem de programação de computador). I. Lima, Joubert de Castro. II. Almeida, André Luis Barroso. III. Universidade Federal de Ouro Preto. IV. Titulo.

CDU: 004.451

Catalogação: ficha.sisbin@ufop.edu.br

Matheus de Oliveira Correia

JCL PAGE RANK: UMA SOLUÇÃO DISTRIBUÍDA PARA GRAFOS MASSIVOS

Monografia II apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau em Bacharel em Ciência da Computação.

Aprovada em Ouro Preto, 11 de dezembro de 2019.

Joubert de Castro Lima
Universidade Federal de Ouro Preto
Orientador

André Luis Barroso Almeida
Instituto Federal de Minas Gerais
Coorientador

Prof. Dr. Marco Antonio Moreira de Carvalho
Universidade Federal de Ouro Preto - UFOP
Examinador

Prof. Dr. Túlio Ângelo Machado Toffolo
Universidade Federal de Ouro Preto - UFOP
Examinador

Dedico esse trabalho aos meus pais, Luciano e Sandra, por me propiciarem a melhor educação que podiam.

Agradecimentos

Agradeço primeiro a minha mãe e ao meu pai por me apoiarem nos momentos difíceis dessa caminhada e me incentivarem sempre a estudar.

Agradeço ao meu orientador Dr. Joubert Castro Lima, pela paciência, apoio e dedicação em me ensinar ao longo de 3 anos. Sua ajuda nesse trabalho é imensa, seja com ideias e sugestões ou até não deixando eu desanimar diante aos obstáculos encontrados. Agradeço a ele imensamente por contribuir para minha formação ética e profissional. Sendo uma pessoa ao qual tenho grande admiração, além de ser um profissional exemplar em que me inspiro.

Agradeço a todos os amigos de infância e aos demais que fiz durante essa caminhada de UFV e UFOP, em especial para todos que passaram pelo HPC-lab, que tiveram a paciência de me ajudar e ensinar.

Resumo

Neste trabalho, é apresentada uma solução distribuída para o problema de elencar os vértices mais populares de um grafo. Tal problema é de fundamental importância nas diversas redes sociais, por exemplo, recorrentemente pesquisamos páginas segundo algum critério de importância. Nos últimos anos os grafos têm aumentado de tamanho à medida que a WWW (World Wide Web) cresce, portanto algoritmos sequenciais ou apenas paralelos não são mais praticáveis, pois executam em apenas uma máquina. Neste trabalho, optamos por redesenhar o algoritmo *PageRank*, proposto pelos fundadores do Google, para que este seja executado eficientemente sob *clusters* de máquinas. A solução utiliza o *middleware* Java Cá&Lá (JCL) e foi testada contra um dos líderes de mercado - o Apache GraphX. Os resultados mostram que a solução, denominada JCL Page Rank 2.0, é mais veloz que o GraphX em nove das instâncias utilizadas, no cenário analisado.

Palavras-chave: PageRank, grafos, GraphX, sistemas distribuídos, Java Cá&Lá.

Abstract

In this paper, we present a distributed solution to the problem of listing the most popular vertices of a graph. This problem is of fundamental importance in various social networks, for example, we recurrently search pages according to some criterion of importance. In recent years graphs have increased in size as the WWW (World Wide Web) grows, so sequential or just parallel algorithms are no longer feasible as they run on just one machine. In this paper, we chose to redesign the PageRank algorithm proposed by Google's founders so that it runs efficiently under machine clusters. The solution uses Java Cá Lá (JCL) middleware and has been tested against one of the industry leaders - Apache GraphX. The results show that the solution, called JCL Page Rank 2.0, is faster than GraphX for nine instances used in the analyzed scenario.

Keywords: PageRank, graphs, GraphX, distributed systems, Java Cá&Lá.

Lista de Ilustrações

Lista de Tabelas

Sumário

1 Introdução

A WWW (World Wide Web) é uma grande coleção de textos, links, imagens, vídeos, áudios e conteúdos multimídia que duplica seu tamanho a cada seis ou dez meses (BARSAGADE, 2003). Devido a tamanho crescimento, é natural que se passe a desenvolver algoritmos cujo objetivo é extrair informação à partir de tal acervo de itens e suas relações. Um dos maiores desafios de tais algoritmos é encontrar a relevância das páginas Web e a abordagem mais conhecida para tal problema faz uso dos diversos links que cada página possui, ou seja, o grafo de links que se forma à partir das diversas páginas existentes na Web é percorrido de diferentes formas para se obter os conteúdos mais relevantes. O método mais conhecido se chama *Page Rank* e foi apresentado em 1999 pelos fundadores do Google, Larry Page e Sergey Brin (PAGE et al., 1999).

A definição de *Page Rank* tem sua base intuitiva em uma navegação aleatória pelo grafo, podendo ser vista como uma distribuição estacionária de uma cadeia de Markov (KEMENY; SNELL, 1976), ou seja, se navega aleatoriamente em um grafo, caminhando sucessivamente por suas arestas. Entretanto, em um cenário real na WWW há chances do algoritmo entrar em ciclos de páginas, pois pode haver links entre duas páginas que se apontam ou situações mais complexas onde ciclos maiores de repetições ocorrem, causando *loops* infinito. Para solucionar tal situação ao navegar pelo grafo, normalmente os algoritmos efetuam saltos, indo, por exemplo, para páginas subsequentes, igualmente de forma aleatória. Pensando nisto, o algoritmo proposto por Page et al. (1999) coloca no cálculo de percurso do grafo uma variável que contabiliza visitas feitas a um vértice, o que evita tal cenário.

Devido ao crescimento do número de páginas Web, atualmente é impraticável calcular o *Page Rank* de todos os vértices de um grafo de forma sequencial ou mesmo paralelamente em uma única máquina. Entretanto, a solução de Page et al. (1999) requer alta dependabilidade no percurso do grafo, podendo haver cenários onde todos os vértices dependem de todos os demais no cálculo do *Page Rank*. Diante disto, encontrar novos algoritmos que permitam o percurso do grafo concorrentemente se torna essencial para melhora no desempenho, contudo tais algoritmos podem trazer perdas de acurácia no resultado da solução ou mesmo podem não escalar devido a problemas de sincronização e excesso de troca de mensagens entre nós de um *cluster*.

Para formalizar a ideia contida no *Page Rank*, e assim explicar mais detalhadamente a alta dependabilidade no uso dos vértices do grafo para se obter as páginas mais relevantes, utilizamos a definição apresentada em Berkhin (2005). Seja G=(P,L) um grafo direcionado com vértices P que são páginas HTML da Web e arestas direcionadas L que são links contidos nas páginas. Seja u uma página da web existente em P, sendo B_u o conjunto de páginas que referenciam u, ou seja, arestas de chegada, e F_u o conjunto de páginas que U referencia, ou seja, arestas de saída. Considere também $N_u = |F_u|$ como sendo o número de links de U. O modelo de navegação

aleatória utilizado pelo *Page Rank* pode ser definido como $PR(u) = \sum_{d \in B_u} PR(d)/N_d$, onde d significa uma página existente no conjunto de páginas B_u que referencia u. O algoritmo ao navegar pelo grafo segue uma das arestas de saída locais com probabilidade C e salta para algum $d \in P$ com probabilidade (1 - C), portanto $PR(u) = (1 - C) + C \sum_{d \in B_u} PR(d)/N_d$. Na literatura, existem diferentes maneiras para escolher o intervalo C, variando de 0,85 (PAGE et al., 1999) a 0,99 (HAVELIWALA et al., 2003). Segundo tais autores, os valores 0,85 até 0,9 são usados na prática.

É evidente a dependência no cálculo de relevância de uma página u, obtido por meio de $PR(u)$, com as demais páginas, pois no cálculo se utiliza B_u. Isto se repete para todas as páginas de P no grafo G, portanto o algoritmo *Page Rank* tradicional é naturalmente sequencial. Quando se considera grafos massivos a implementação tradicional se torna lenta e muitas vezes inviável, uma vez que as pontuações das páginas ocorrem à partir de todo o grafo. Algoritmos distribuídos para calcular o *Page Rank* de grafos massivos se tornam uma das alternativas de solução mais promissoras, contudo estes podem se tornar extremamente ineficientes. Em linhas gerais, a cada página visitada se necessita dos resultados de etapas anteriores, além da necessidade de uma constante sincronização e várias trocas de mensagens entre nós de um *cluster*. O problema de alocação de vértices e arcos no *cluster* é outro desafio que pode se tornar ainda mais complexo à medida que o grafo sofre atualizações (FUJIWARA et al., 2013).

1.1 Objetivo

O objetivo deste trabalho é a implementação e teste de um algoritmo distribuído para o cálculo do *Page Rank* de todos os vértices de um grafo composto por links Web. Para atingirmos tal objetivo, utilizamos o *middleware* Java Cá&Lá (ALMEIDA et al., 2019; CIMINO et al., 2019) para construir tal algoritmo, pois este se mostra eficiente, escalável e simples de programar, além de ser portável para inúmeras plataformas e permitir a execução de algoritmos já existentes sem qualquer refatoração de código. A solução, denominada JCL Page Rank, foi testada contra um dos líderes de mercado - o Apache GraphX. Os resultados mostram que o JCL Page Rank é mais veloz que o GraphX na maioria das instâncias utilizadas.

1.2 Organização do Trabalho

O restante do trabalho está organizado da seguinte forma: no Capítulo 2, há a revisão da bibliografia com os algoritmos paralelos e distribuídos que efetuam o cálculo do *PageRank* a partir de grafos massivos. Os trabalhos foram classificados de acordo com a estratégia de particionamento para varreduras do grafo, precisamente trabalhos que adotam o esquema algébrico linear, a estratégia *score update* e, por fim, Monte Carlo. No Capítulo 3, apresentamos o algoritmo JCL PageRank, incluindo seus pontos fortes e limitações. No Capítulo 4, são apresentadas a

configuração do ambiente de testes, os cenários de testes e os resultados obtidos. Por fim, no Capítulo 5 as conclusões e os desdobramentos futuros são detalhados.

2 Fundamentação Teórica e Revisão Bibliográfica

Na primeira seção deste capítulo há o detalhamento dos fundamentos teóricos e explicações prévias que servem para orientação, análise e interpretação deste trabalho. Nas demais seções há o detalhamento do estado da arte em algoritmos, *frameworks* e *middlewares* paralelos ou distribuídos para o cálculo do *Page Rank* à partir de grafos massivos. Os trabalhos sequenciais, tais como Haveliwala (1999), Arasu et al. (), Chen, Gan e Suel (2002), Jeh e Widom (2003), Broder et al. (2006), Andersen et al. (2007), Fujiwara et al. (2013), Beamer, Asanović e Patterson (2017), estão fora do escopo deste trabalho, com isto não são descritos neste capítulo.

2.1 Fundamentação Teórica

Nesta seção explicamos as estratégias para o cálculo do *Page Rank* e a arquitetura do *middleware* Java Cá & Lá - JCL.

2.1.1 *Page Rank*

Após o primeiro algoritmo para o cálculo do *Page Rank* ter sido apresentado há quase vinte anos atrás (PAGE et al., 1999), muitas variações do mesmo foram propostas, dentre elas a estratégia chamada *Personalized Page Rank* (PPR), onde o cálculo ocorre à partir de um subconjunto de vértices e arcos do grafo, havendo, normalmente, redução do espaço de busca.

Segundo Fujiwara et al. (2013), existem três estratégias para o cálculo do *Page Rank* e também do PPR: i) esquema algébrico linear (FUJIWARA et al., 2013; GUO et al., 2017), ii) *score update* (BAHMANI et al., 2012; ANDERSEN et al., 2007) e iii) Monte Carlo (BAHMANI; CHAKRABARTI; XIN, 2011; SARMA et al., 2013; LOFGREN et al., 2014; WEI et al., 2018). A primeira tentativa de classificação das possíveis soluções para o cálculo do *Page Rank* foi feita por Bahmani, Chakrabarti e Xin (2011) e nela os autores optaram por definir *score update* como apenas uma variação do esquema algébrico linear. Neste trabalho, utilizamos a classificação apresentada em Fujiwara et al. (2013).

O esquema algébrico linear foi proposto por Kamvar, Haveliwala e Golub (2004). Este aplica técnicas de otimização de matriz para reduzir os custos de varredura do espaço de busca, onde o cálculo da pontuação de um vértice do grafo é suspenso assim que ele começa a convergir, ou seja, quando o cálculo do *Page Rank* de um vértice u começa a ser feito à partir de outros resultados não *default* do *Page Rank* de outros vértices associados a u a convergência aumenta. Como limitação de tal esquema, o mesmo não é estacionário, ou seja, os valores de *Page Rank*

produzidos são irregulares; outra limitação é que o grafo deve ser conhecido por completo a priori.

Na estratégia *score update* a ideia é que uma mudança incremental no grafo cause maior impacto apenas nas pontuações locais do *Page Rank*, ou seja, apenas os vértices que são impactados numa mudança são efetivamente considerados no cálculo, portanto a aleatoriedade que permitia saltos e explorações com igual probabilidade de todo o grafo não acontece. O trabalho seminal nesta temática data de 2004 (CHIEN et al., 2004) e a maior limitação de tal estratégia é o fato dos grafos nem sempre mudarem de maneira incremental, portanto não há suporte para cenários onde o grafo não é conhecido a priori.

A estratégia Monte Carlo pré-computa varreduras aleatórias pelo grafo e as armazena, portanto para calcular a pontuação aproximada de um vértice u, há o cálculo do número total de vezes que u foi visitado pelas varreduras aleatórias e, normalmente, concorrentes pelo grafo. Infelizmente, há alto custo de escrita e leitura dos caminhos aleatórios gerados com a estratégia Monte Carlo, incluindo nas suas principais variações (AVRACHENKOV et al., 2007; BAHMANI; CHOWDHURY; GOEL, 2010). De uma forma geral, o número de caminhos aleatórios utilizado pode piorar a eficiência ou a qualidade do resultado do *Page Rank*. Assim como o esquema algébrico linear, qualquer alteração no grafo faz com que novas varreduras aleatórias sejam geradas.

2.1.2 Java Cá & Lá - JCL

O Java Cá & Lá (JCL) é um *middleware* orientado à tarefas e memória compartilhada distribuída (DSM) (ALMEIDA et al., 2019; CIMINO et al., 2019). Ele também permite a integração com *brokers* MQTT, portanto ele consegue se inscrever a tópicos, submeter notificações diversas e as receber. De uma forma geral, o JCL é o primeiro *middleware* a integrar as tecnologias IoT e HPC (CIMINO et al., 2019).

A Figura 2.1 ilustra a arquitetura do JCL em um *cluster*, sendo o JCL composto pelos componentes: JCL_Server, JCL_Host, JCL_SuperPeer e JCL_User (ALMEIDA et al., 2016). Em conjunto, os componentes são capazes de armazenar e processar objetos Java em ambiente distribuído ou mesmo sensoriar ambientes e processar os dados de forma distribuída em múltiplos *clusters*. O JCL_Server é responsável por gerenciar os demais componentes da arquitetura, contudo sem virar o gargalo, pois delega permissões aos JCL_Hosts e JCL_SuperPeers, sendo os JCL_Hosts os responsáveis pelos serviços de armazenamento, sensoriamento, atuação ou processamento. Já os JCL_SuperPeers são responsáveis por interligar múltiplos *clusters*, incluindo os que possuem IPs inválidos, portanto o JCL_SuperPeer serve de ponte entre duas redes.

O JCL_Host é a fachada ou API do JCL, portanto cabe ao programador incluir tal componente em sua aplicação. É nele que os métodos são invocados para execuções remotas ou para instanciação de variáveis ou para criação de mapas distribuídos. Os serviços de sensoriamento,

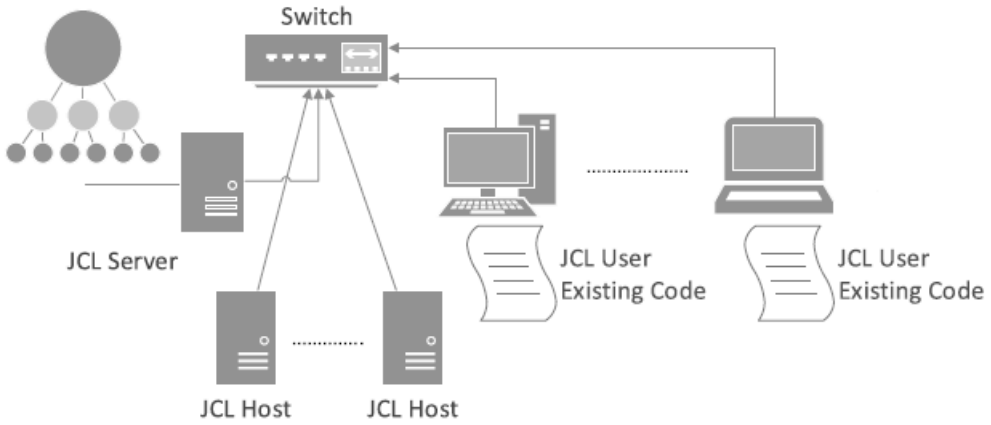


Figura 2.1 – Implantação do JCL (ALMEIDA et al., 2019)

sensibilidade de contexto e atuação em ambientes remotos também podem ser feitos via API e com o JCL_User. Cabe ao *kernel* JCL decidir onde os objetos serão armazenados e onde as tarefas serão executadas. O programador também pode optar por executar uma tarefa numa máquina específica do *cluster*. Os sensores podem ser configurados remotamente e programaticamente, assim como a adição de contextos diversos. O fato do JCL ter sido desenhado inicialmente para HPC e depois ter incorporado serviços IoT o permitiu oferecer ao programador os modelos de programação DSM e baseado em tarefas para o desenvolvimento de aplicações IoT, ou seja, no JCL não há apenas o modelo de eventos usando o padrão *publish/subscribe* para desenvolver IoT.

Atualmente, o JCL implementa a obtenção explícita dos objetos e dos resultados das tarefas, assim como dos pares chave-valor do mapa JCL. Cabe ao JCL_User encontrar onde está armazenado o objeto que corresponde à variável, à entrada do mapa ou ao resultado de uma determinada tarefa, ou seja, encontrar um JCL_Host no *cluster* e obter uma cópia do objeto para a máquina onde o JCL_User está em execução, mesmo que já haja uma cópia nesta máquina sem qualquer nova modificação. O mesmo ocorre quando o acesso é bloqueado a outros JCL_Users em outras máquinas do *cluster*, portanto concorrentes.

2.2 Trabalhos Relacionados usando esquema algébrico linear ou *score update*

Em Bahmani et al. (2012), são apresentados dois algoritmos para o cálculo do *Page Rank* de um grafo, onde alterações incrementais no grafo impactam apenas nas pontuações locais do *Page Rank*. O primeiro algoritmo, denominado *Proportional Probing*, baseia-se na intuição de que os vértices com valores mais altos precisam ser investigados com mais frequência, uma vez que eles podem ter maior impacto nos valores de *Page Rank* de outros vértices. Em linhas gerais, a cada iteração à partir de um vértice u, qualquer outro vértice do grafo que possui relação com u é escolhido de acordo com uma probabilidade proporcional aos valores do vetor de *Page Rank* do grafo até o momento. Existe uma escolha de quais vértices serão processados em paralelo,

com isto há como verificar mais de um vértice por iteração. Como consequência, ao final de cada iteração à partir de u, um novo vetor de *Page Rank* é obtido e desta forma, incrementalmente, tenta-se reduzir o espaço de busca na obtenção dos top-k vértices mais promissores de um grafo. O segundo algoritmo, denominado *Priority Probing*, detecta vértices com frequências proporcionais ao vetor de valores *Page Rank* do grafo corrente. A escolha de vértices concorrentemente ocorre também no segundo algoritmo. Manter o vetor de valores *Page Rank* de um grafo massivo com bilhões de vértices sob um *cluster* é um desafio.

O trabalho de Guo et al. (2017) apresenta uma solução distribuída para o cálculo do PPR de forma exata, baseando-se no particionamento do grafo com base em premissas, tais como tempo para o cálculo do *Page Rank*, o custo de comunicação e o custo computacional de cada máquina do cluster. Dois algoritmos são apresentados para o particionamento do grafo no cluster: *Graph Partition based Algorithm*, denominado por GPA, e *Hierarchical Graph Partition based Algorithm*, denominado HGPA. O algoritmo GPA procura por vértices que sejam vértices de articulação, ou seja, que ligam diferentes subgrafos e particionam o grafo em diversos subgrafos, onde o cálculo do *Page Rank* é feito em paralelo em cada um dos subgrafos. É possível recursivamente aplicar o GPA dentro do subgrafo, ou seja, particionar o subgrafo em subgrafos de nível inferior. Já o HGPA cria uma hierarquia entre os subgrafos, colocando-os em uma estrutura de árvore. A cada novo subgrafo gerado ele é considerado um nível abaixo do grafo que o gerou. O cálculo do PPR é realizado usando o esquema algébrico linear, onde subgrafos no mesmo nível hierárquico podem ser processados concorrentemente e segundo estratégia *bottom-up*, ou seja, o processamento ocorre dos subgrafos que formam as folhas da árvore para sua raiz, com isto ao chegar a raiz há todas as informações necessárias ao cálculo do PPR à partir de um subconjunto de vértices. Os vértices pontes são usados quando algum subgrafo de hierarquia superior necessita de informações de subgrafos hierarquicamente inferiores. Uma limitação no trabalho ocorre com grafos altamente conectados, pois o particionamento usando vértices que atuam como vértices de articulação se torna custoso e bastante complexo. Outra limitação é não haver descrição de suporte a grafos dinâmicos que não são conhecidos a priori. O custo computacional, precisamente processamento e armazenamento, do particionamento e da hierarquização de subgrafos sempre precisa ser considerado, sendo em alguns cenários uma limitação da solução distribuída.

2.3 Trabalhos Relacionados que adotam Monte Carlo

Em Bahmani, Chakrabarti e Xin (2011), é apresentado um algoritmo para o cálculo do PPR de grafos massivos, conhecido a priori, utilizando *Map Reduce* (DEAN; GHEMAWAT, 2008) e Monte Carlo. É uma proposta distribuída e paralela para o cálculo não só dos top-k vértices do grafo como também o cálculo do valor de PPR de todos os vértices. O *middleware Map Reduce* é utilizado para processar grandes quantidades de dados de forma paralela e distribuída, portanto fica responsável pela distribuição de tarefas, armazenamento e tolerância a falhas. O algoritmo proposto mescla os caminhos aleatórios gerados pelo método de Monte Carlo com o

intuito de produzir um único caminho aleatório partindo de cada vértice existente do grafo, desta forma o tempo de leitura dos caminhos gerados diminui. Posteriormente à etapa de mescla, é contado quantas vezes cada vértice é visitado, similar às demais soluções *Page Rank* que utilizam Monte Carlo. O algoritmo apresentado em Bahmani, Chakrabarti e Xin (2011) é considerado um dos mais eficientes utilizando *Map Reduce* frente outras (LIN; SCHATZ, 2010; KANG; TSOURAKAKIS; FALOUTSOS, 2011).

Sarma et al. (2013) apresenta uma implementação para um cálculo do *Page Rank* de forma distribuída em grafos direcionados ou não direcionados. O algoritmo é baseado em Monte Carlo para calcular a distribuição do *Page Rank*, sendo denominado *Basic Page Rank Algorithm* (BPRA). O BPRA realiza o cálculo com precisão em $O(log(n)/\epsilon)$ iterações, onde n é a quantidade de vértices existentes no grafo. A lógica dessa implementação consiste em caminhar por cada vértice do grafo concorrentemente, ou seja, em cada etapa um vértice vizinho, também chamado vértice de saída, é escolhido de maneira aleatória. Cada vértice armazena a quantidade de vezes que é visitado, portanto mecanismos de sincronização são considerados. O algoritmo leva em conta que cada vértice armazena a identificação dos seus vizinhos, dessa maneira o grafo pode estar distribuído segundo inúmeras alternativas de particionamento, todas tendo como foco o controle da comunicação e sincronização entre as máquinas do *cluster*. Assume-se que a comunicação ocorre em intervalos síncronos, ou seja, os vértices podem efetuar computações diversas e todas serão aguardadas ou sincronizadas ao final de um intervalo síncrono, garantindo a correta computação no intervalo de execuções subsequente. Até mesmo o envio de mensagens entre vértices é sincronizado ao final de um intervalo síncrono, entretanto cada vértice pode enviar apenas uma mensagem por arco do grafo.

Em Lofgren et al. (2014), é apresentada uma solução paralela, similar à solução apresentada em Bahmani, Chakrabarti e Xin (2011), Sarma et al. (2013), portanto não necessita de pré-processamento. O algoritmo *Frontier-Aided Signicance Thresholding for Personalized Page Rank* (FAST-PPR) baseia-se numa técnica de busca bidirecional para estimativa do PPR. De uma forma geral, para cada vértice vizinho escolhido aleatoriamente há outro vértice, denominado alvo, com isto, à partir de ambos vértices gera-se um caminho de forma concorrente. Note que não apenas a escolha de vértices vizinhos pode ser feita concorrentemente, como também a geração de um caminho do PPR pode ser feita concorrentemente. FAST-PPR promete ser 20 vezes mais rápido que as demais abordagens existentes no estado da arte para estratégias que utilizem Monte Carlo (AVRACHENKOV et al., 2007; BAHMANI; CHOWDHURY; GOEL, 2010; SARMA et al., 2013) ou esquema algébrico linear (ANDERSEN et al., 2007; LOFGREN; GOEL, 2013).

O TopPPR, apresentado em Wei et al. (2018), combina três técnicas para percorrer o grafo de um vértice a outro (*Forward Search, Random Walk Sampling e Backward Search*), em conjunto com o método Monte Carlo para calcular os top-k vértices. O TopPPR primeiro realiza a técnica *Forward Search*, onde descobre um conjunto de vértices que podem levar ao vértice destino. À partir de tal conjunto, a técnica *Random Walk Sampling* entra em ação, permitindo

chegar ao vértice destino com a exploração de múltiplos caminhos concorrentemente. Utiliza-se a técnica *Backward Search* em alguns vértices, resultando em um conjunto de possíveis top-k vértices que vão ter seus valores de PPR refinados iterativamente. O principal desafio do TopPPR é que as escolhas de vértices realizadas antes do cálculo do PPR, precisamente as realizadas pelas técnicas *Forward Search* e *Backward Search*, devem ser precisas, pois caso contrário há perdas de top-k vértices reais.

2.4 Trabalhos Relacionados sem detalhes de qual estratégia utilizam

GraphX, apresentado por Xin et al. (2013) e Gonzalez et al. (2014), é um dos principais *frameworks* de processamento de grafos construído em cima do *middleware* Apache Spark (ZAHARIA et al., 2010), sendo o Spark um sistema de processamento distribuído e amplamente utilizado mundialmente. A solução GraphX, de maneira iterativa, consegue transformar informações de vértices em informações de vértices e arcos adjacentes. Há adoção de um modelo de execução concorrente chamado *bulk synchronous*, onde cada vértice pode ser processado concorrentemente aos demais. O processamento concorrente segue um pipeline, denominado GAS (GONZALEZ et al., 2012), composto pelos seguintes passos: *Gather*, *Apply*, e *Scatter*; desta forma, os vértices pedem informações de seus vizinhos, as processa e depois publicam novos valores a serem consumidos por outros vértices do grafo. Como não há envio de mensagens e sim consumo de informações de vértices, há como introduzir cortes de vértices, chamado *vertex-cut partitioning* (RAHIMIAN et al., 2014), há como introduzir a iteração entre arcos (ROY; MIHAILOVIC; ZWAENEPOEL, 2013), e reduzir a movimentação de dados. A solução proíbe comunicação direta entre vértices não adjacentes, portanto padrões de comunicação mais aleatórios não são permitidos. Segundo os autores, GraphX consegue processar grafos com bilhões de vértices e trilhões de arcos. O experimento *Page Rank* com 32 máquinas no *cluster* conseguiu ser 3 vezes mais rápido do que o experimento com apenas 8 máquinas, entretanto quando se usou 64 máquinas o *speedup* foi apenas de 3.5x, ou seja, apenas ligeiramente superior à configuração com 32 máquinas, mas melhor do que 3.2x de *speedup* obtido pelo concorrente PowerGraph (GONZALEZ et al., 2012), o que realça o desafio em distribuir o cálculo do *Page Rank* em *clusters* de grande porte.

Pregel (MALEWICZ et al., 2010) é uma solução distribuída desenhada especificamente para grafos que possuem como modelo de programação para as suas aplicações a troca de mensagens explícita, muito diferente de outras soluções, tal como GraphX, que cria uma abstração de espaço distribuído e compartilhado de memória, o que muitas vezes simplifica o desenvolvimento de aplicações. Os vértices enviam mensagens a outros vértices após calcularem seus valores de *Page Rank*, usando para isto os arcos de saída do vértice sendo processado. Esta estratégia pode limitar a adoção de otimizações diversas que permitem reduzir o espaço de busca da solução e

evitar processar vértices não promissores. O Pregel também adota o modelo de execução concorrente chamado *bulk synchronous*, onde cada vértice pode ser processado concorrentemente aos demais, contudo sempre obedecendo a barreiras de sincronização entre cada estágio do pipeline, ou seja, há sincronização após cada *bulk synchronous* que implementa um estágio do pipe, assim como ocorre com PowerGraph e GraphX. Infelizmente, nenhum experimento com o algoritmo *Page Rank* foi relatado em Malewicz et al. (2010), existindo apenas um exemplo de como o implementar na solução proposta.

PowerGraph (GONZALEZ et al., 2012) é outra solução para processamento em grafos com o algoritmo de *Page Rank* implementado internamente. Tanto PowerGraph quanto GraphX utilizam abstrações e modelos de execução muito similares, ou seja, a ideia de vértices adjacentes e o modelo de execução GAS, apresentando no trabalho PowerGraph, são idênticos. O que os diferencia é que o GraphX é implementando sob o *middleware* Spark, ou seja, em Java e obedecendo as primitivas de programação *Map/Reduce*, e o PowerGraph é implementado em C++ e sem qualquer *middleware* como suporte. Segundo os autores de GraphX, a API do *framework* PowerGraph não é intuitiva.

2.5 JCL Page Rank e os trabalhos correlatos

O presente trabalho foi implementado em duas versões. Em uma, denominada JCL Page Rank 1.0, foi utilizado o sistema de memória distribuído e compartilhado do JCL, assim como o modelo de programação baseado em tarefas assíncronas. O conjunto de vértices e arcos foram armazenados no *cluster* transparentemente pelo JCL e as tarefas, que representavam o cálculo de *Page Rank* dos vértices, percorrem os vértices vizinhos, conforme o algoritmo sequencial de 1999 (PAGE et al., 1999). Esta versão demonstrou ser simples de programar, sendo muito similar a uma versão sequencial ou multi-thread, contudo esta ficou cerca de 200 vezes mais lenta do que a solução GraphX.

Já na segunda versão, denominada JCL Page Rank 2.0, a estratégia GAS (GONZALEZ et al., 2012) também foi utilizada, portanto múltiplos vértices são processados concorrentemente. Além disto, há o particionamento dos vértices do grafo no *cluster* utilizando as premissas do JCL (ALMEIDA et al., 2019), ou seja, usando o *hashcode* do vértice. Segundo os experimentos do JCL, para vértices com nomes iguais a u_i, onde i>0, a distribuição de vértices no *cluster* fica bem igualitária. Até mesmo para nomes aleatórios os resultados não foram desanimadores. Para cada vértice se armazena cópias de seus vértices de saída e entrada, evitando comunicações desnecessárias entre máquinas do *cluster*. Ao invés da memória distribuída e compartilhada, foram utilizadas as várias memórias locais e a versão paralela do JCL em conjunto com sua versão distribuída, tentando absorver as simplificações de programação de ambas. Após cada iteração do *Page Rank* numa determinada máquina do *cluster*, as publicações de atualização de valores dos vértices daquela máquina precisam ocorrer, pois afinal tais vértices podem ser

vizinhos de vértices armazenados em outras máquinas do *cluster*. Tais atualizações de valores de *Page Rank* ocorrem concorrentemente, fazendo com que todo o *cluster* se atualize para a próxima iteração do algoritmo.

Foi feito a ideia de publicar os novos valores de *Page Rank* de um determinado vértice tão logo este sofra atualização via programação JCL, precisamente utilizando tarefas remotas assíncronas, e tal publicação de atualização é feita em todo *cluster* mais eficientemente que o GraphX para a maioria das instância de grafos utilizadas nos experimentos. Os detalhes arquiteturais das duas implementações feitas neste trabalho são realizados no próximo capítulo. Detalhes de experimentação são apresentados no Capítulo 4.

2.6 Avaliações comparativas dos trabalhos

Segundo Wei et al. (2018), existem três tipos diferentes de consultas em um grafo para o cálculo do PPR, sejam elas: i) ponto-a-ponto ii) fonte única e iii) top-k. Consultas ponto-a-ponto solicitam o valor de *Page Rank* de um determinado vértice u em relação aos vértices de origem informados, ou seja, o valor de u é calculado à partir dos vértices que estão entre u e os vértices de origem. Consultas fonte única (WANG et al., 2017) solicitam o valor de *Page Rank* de cada vértice em relação a um vértice de origem e um parâmetro k. Este tipo de consulta retorna os vértices top-k juntamente com seus respectivos valores de *Page Rank*. Consultas top-k solicitam os k vértices cujos valores de *Page Rank* são os maiores do grafo.

Nesta seção, classificamos os trabalhos encontrados de acordo com a estratégia adotada, seja Monte Carlo ou esquema algébrico linear ou *score update*, e também de acordo com o tipo de consulta suportada, seja top-k, ponto-a-ponto ou fonte única. Também é avaliado se o trabalho calcula PPR ou apenas o *Page Rank* tradicional à partir de todo o grafo, o que invalida suporte a diferentes tipos de consulta. Por fim, o requisito solução paralela ou distribuída é verificado.

A Tabela 2.1 apresenta uma comparação dos trabalhos relacionados, resumindo os requisitos implementados. Cada requisito pode ser satisfeito (✓), não ter indicação encontrada sobre (?) ou não atende ao requisito (-).

Segundo Sarma et al. (2013), há poucos trabalhos com algoritmos comprovadamente completos e totalmente distribuídos para o cálculo do *Page Rank*, o que é possível perceber analisando a Tabela 2.1. Percebe-se também que a grande maioria do trabalhos foca na estratégia Monte Carlo por esta permitir a introdução de concorrência mais facilmente. Os trabalhos Monte Carlo suportam apenas consultas top-k, pois estes calculam o valor aproximado de *Page Rank*. Nos trabalhos analisados sobre PPR não há soluções com suporte à consultas ponto-a-ponto. Há apenas dois trabalhos utilizando o esquema algébrico linear. De uma forma geral, estratégias Monte Carlo possibilitam diferentes tipos de consultas, conforme ilustra a Tabela 2.1.

Tabela 2.1 – Tabela comparativa de soluções concorrentes para o cálculo do Page Rank

Solução	ALorSU	MC	PPRs	UFq	TKq	P2Pq	D	P
JCL Page Rank 1.0	✓	-	-	✓	✓	-	✓	✓
JCL Page Rank 2.0	✓	-	✓	✓	✓	-	✓	-
Guo et al. (2017)	✓	-	✓	✓	✓	-	✓	✓
Bahmani et al. (2012)	✓	-	✓	✓	✓	-	-	✓
Bahmani, Chakrabarti e Xin (2011)	-	✓	✓	-	✓	-	✓	✓
Sarma et al. (2013)	-	✓	✓	-	✓	-	✓	✓
Lofgren et al. (2014)	-	✓	✓	-	✓	-	-	✓
Wei et al. (2018)	-	✓	✓	-	✓	-	-	✓
Xin et al. (2013)	✓	-	?	✓	✓	✓	✓	-
Gonzalez et al. (2012)	?	?	?	?	?	?	✓	-
Malewicz et al. (2010)	?	?	?	?	?	?	✓	-

ALorSU: Algébrico linear ou Score Update
MC: Monte Carlo
PPRs: Suporte para PPR
UFq: Consulta fonte única
TKq: Consulta top-k
P2Pq: Consulta ponto-a-ponto
D: Solução distribuída
P: Solução paralela

3 Desenvolvimento

O algoritmo JCL Page Rank permite o cálculo de valores de *Page Rank* de todos os vértices do grafo, portanto se assemelha a estratégia denominada esquema algébrico linear, permitindo com isto consultas top-k, como também os valores do vetor de *Page Rank* de todos os vértices. O JCL Page Rank 2.0 é arquiteturalmente organizado em dois componentes o primeiro é responsável pela varredura da entrada, indexação e particionamento do grafo; o segundo é responsável pelo cálculo do valor de *Page Rank* para cada vértice, assim como a publicação remota e assíncrona de suas atualizações nas mais diversas máquinas de um *cluster*. Neste capítulo, há menção ao JCL Page Rank 1.0 com intuito de simplicidade de codificação em ambientes distribuídos e não com intuito de eficiência.

3.1 JCL Page Rank 1.0

A primeira versão do JCL Page Rank visa calcular o valor de Page Rank de todos os vértices de grafos massivos, entretanto esta versão foca apenas no armazenamento de forma distribuída, usando para isto uma JCL-Hash-Map. A premissa é que não é viável armazenar tal grafo em apenas uma máquina comum de um cluster. Enquanto o armazenamento é realizado de maneira distribuída, o cálculo dos valores de *Page Rank* é realizado de maneira iterativa, assim como apresentado em (PAGE et al., 1999), portanto em uma máquina do *cluster*. A opção por utilizar a maneira iterativa do cálculo do *Page Rank* se deve a sua exatidão, facilidade de implementação no JCL e precisão que o modo iterativo fornece.

O grafo pode ser facilmente carregado à partir de uma das máquinas do *cluster* ou de todos eles. Basicamente, há uma instância do mapa distribuído e chamado JCL-Hash-Map Page Rank, onde as chaves são os vértices e os valores são os seus vizinhos. Uma única varredura da instância do grafo e este já se encontra no *cluster*. O particionamento do grafo é feito transparentemente pelo JCL e usando *hashcodes* dos objetos vértices ou chaves do mapa e o número de membros do *cluster* - $hash(u_i)\ mod(cs)$, onde $hash$ é a função *hashcode*, mod é a função módulo, u_i é um vértice qualquer e cs é o tamanho do *cluster*. O JCL permite escritas concorrentes nos mapas distribuídos e para isto basta acessar cada par chave-valor usando as primitivas *acquire* e *release*.

Após a instância do grafo ter sido armazenada no *cluster*, o Algoritmo 1 recebe um número de iterações, que serve para calibrar a precisão do cálculo do *Page Rank* de um vértice e o tempo total do algoritmo. No algoritmo, o *default* são 10 iterações, conforme proposto por Haveliwala (1999). Para cada vértice u do grafo G se obtém os arcos $inEdges$ que estão ligados a u. À partir de cada arco da lista de arcos $inEdges$ se obtém um vértice que está ligado a u e, consequentemente, seu valor de *Page Rank*. A variável local "vizinhos" acumula o valor de *Page Rank* de u até que os arcos em $inEdges$ tenham sido completamente analisados. Por fim, o vetor

de *Page Rank* do grafo é atualizado no *cluster* usando a variável global JCL $PRatual[u]$, com isto todas as tarefas em execução no *cluster* acessam tal variável transparentemente via JCL.

Algoritmo 1: Page Rank 1.0

Entrada: Recebe a quantidade de iterações $Iter$ e um grafo G em formato da JCL-Hash-Map.

Saída: Vetor com o valor de Page Rank para cada vértice do grafo PR.

```
1  início
2      Cria um vetor PR
3      Cria um vetor PRatual
4      para 0 até Iter faça
5          para cada u_i em G faça
6              inEdges ←G.get(u_i)
7              para cada vertice em inEdges faça
8                  numLinks = G.get(vertice).size
9                  vizinhos = (PR[vertice]/numLinks) + vizinhos
10             fim
11             PRatual[u_i] = (0.15 + 0.85 * vizinhos)
12         fim
13         PR <- PRatual
14     fim
15     retorna PR
16 fim
```

3.2 JCL Page Rank 2.0

A segunda versão do JCL Page Rank também realiza o cálculo do valor de *Page Rank* de todos os vértices do grafo, entretanto esta versão realiza esta tarefa concorrentemente e de forma distribuída. Outra diferença crucial frente a primeira versão é não utilizar o mapa distribuído JCL e sim vários mapas JCL, sendo um por cada JCL_Host ou máquina do *cluster*.

3.2.1 Componente Load

No JCL Page Rank 2.0 há apenas JCL-Hash-Maps locais, precisamente dois mapas locais para cada JCL_Host, sendo um mapa responsável por armazenar os vértices e seus vizinhos e outro para auxiliar com o armazenamento do valor do *Page Rank* de cada um dos vizinhos e o número de arestas que saem dos mesmos.

O componente Load realiza a leitura da instância do grafo ou uma partição da instância de um grafo em uma máquina do *cluster*. Ambos mapas são montados com apenas uma varredura da instância do grafo. Como em um *cluster* há vários JCL_Hosts, o serviço de particionamento é feito pelo componente Load.

A função de particionamento dos vértices é a mesma feita pelo JCL (ALMEIDA et al., 2019) para variáveis e chaves de mapas, ou seja, se calcula a função *hash* do vértice e depois se calcula o módulo de tal valor pelo número de JCL_Hosts no *cluster*. Não há garantia de distribuição uniforme no *cluster*, mas os experimentos com objetos com nomes obedecendo uma ordem (Ex. u_i onde i>0) ou plenamente aleatórios mostraram resultados com boa distribuição em (ALMEIDA et al., 2019). Ao final da varredura feita pelo componente Load há dois mapas por JCL_Host do *cluster* segundo a estratégia de partição. O particionamento garante que não há um vértice do mapa principal em dois JCL_Hosts em um *cluster*, mas há cópias de algumas informações de vértices vizinhos a estes, armazenados no mapa auxiliar, para reduzir as comunicações em rede.

São dois mapas criados pelo componente Load e por JCL_Host. Após todos os mapas carregados, há o envio destes mapas pela rede para os demais máquinas do *cluster*, sempre obedecendo a estratégia de particionamento explicada anteriormente. Com os mapas em cada JCL_Host, a segunda fase do algoritmo começa, ou seja, o cálculo do *Page Rank* ocorre. Note que os mapas servem também como *buffers*, pois evitam o envio pela rede de pares *key-value* individualmente. Optamos por agrupar e enviar os dois mapas via JCL em apenas uma tarefa assíncrona por JCL_Host.

Mais detalhadamente, o mapa principal em cada JCL_Host é composto por chaves do tipo String que armazenam os nomes dos vértices do grafo. Os valores são do tipo Neighbors, onde tal classe reúne informações para um vértice u_i, tais como a lista com os nomes dos vértices que chegam em u_i, um inteiro responsável por guardar o número de arestas que saem de u_i e um float responsável por armazenar o valor de *Page Rank* de u_i. O mapa auxiliar é construído a partir do mapa principal. Nele, se armazena todos os vértices presentes na lista da classe Neighbors como chaves de tal mapa. Os valores do mapa armazenam dois números cada: o valor de *Page Rank* de cada chave e o número de arestas que saem dos vértices presentes em Neighbors.

A Figura 3.1 ilustra um cenário onde a instância do grafo reside em uma das máquinas do *cluster*. Este cenário ocorre nos experimentos (Capítulo 4), pois no GraphX não há o conceito de particionamento das instâncias de um grafo. No exemplo, o grafo é percorrido, o mapa principal é preenchido, após tal preenchimento se monta o mapa auxiliar e, por fim, submetem-se ambos os mapas para o *cluster*. Monta-se um par de mapas por JCL_Host usando o esquema de particionamento usado pelo JCL e já explicado. Neste exemplo, basta a fase dois do algoritmo JCL Page Rank iniciar, pois todo o grafo já foi percorrido. Já no caso de múltiplas partições de uma instância de um grafo que estão armazenadas em múltiplas máquinas de um *cluster*, após envio aos JCL_Hosts, há necessidade de reunir os dois mapas recebidos de cada componente Load instalado no *cluster*. Em suma, cada JCL_Host pode receber dois mapas por componente Load do *cluster* e estes precisam ser reunidos em um par final de mapas locais por JCL_Host.

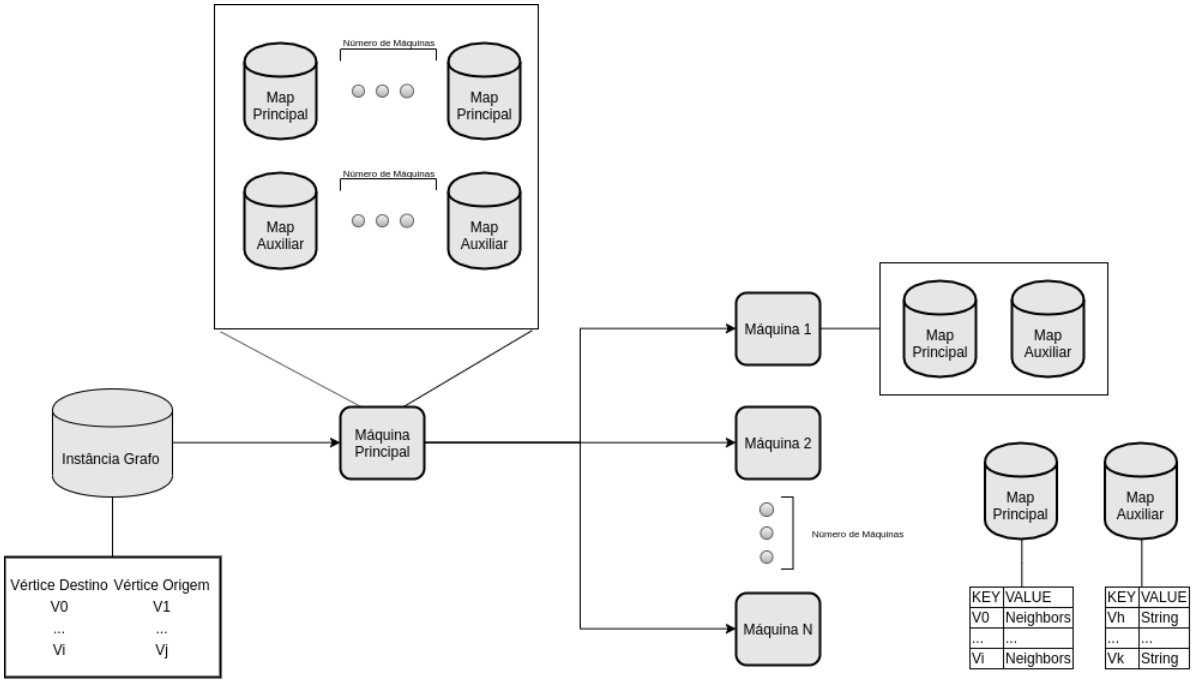


Figura 3.1 – Exemplo de uma instância de um grafo em apenas uma máquina do cluster

3.2.2 Componente Cálculo de Page Rank

Após o componente Load ser chamado e executado, é possível executar o componente Cálculo do Page Rank. O componente Cálculo Page Rank calcula o valor de *Page Rank* para cada vértice existente no mapa principal e propaga tais valores para os vizinhos destes vértices que residem em outros JCL_Hosts do *cluster*. Para que esta publicação de valores de *Page Rank* aconteça de maneira correta e eficiente, foi utilizada a técnica GAS (*Gather, Apply, Scatter*) (GONZALEZ et al., 2012), também utilizada pelas soluções GraphX, Pregel e PowerGraph, ou seja, três das ferramentas para processamento de grafos massivos mais utilizadas mundialmente. A técnica GAS possui três etapas, onde a primeira assume que informações de vértices vizinhos são requisitadas, a segunda etapa assume que algum processamento é realizado com tais informações e, por fim, existe a etapa de sincronização onde cada vértice comunica a seus vizinhos os resultados do processamento.

O Algoritmo 2 apresenta o pseudocódigo do Componente Cálculo de Page Rank. Inicialmente, são solicitadas informações dos vértices vizinhos aos vértices que se encontram no mapa principal num determinado JCL_Host (linhas 8, 9 e 10). Tais vértices foram particionados anteriormente de forma tal a não haver redundância no *cluster*. As informações necessárias para o grupo de vértices do mapa principal encontram-se no mapa auxiliar, pois indicam o valor de *Page Rank* e quantas arestas saem de um determinado vértice vizinho (linhas 13 e 14).

A segunda etapa do componente consiste no cálculo do *Page Rank*, assim como é feito pelo método tradicional (PAGE et al., 1999) (linha 16), porém há como paralelizar tal cálculo com os variados JCL_Hosts de um *cluster*. Note que o algoritmo não precisa se comunicar com

Algoritmo 2: JCL Page Rank 2.0

```
   Entrada: Recebe a quantidade de iterações Iter.
1  início
2      //Acessar os mapas principal e auxiliar, para solicitar as informacoes necessarias
3      Acesso a map principal localGraph
4      Acesso a map auxiliar localGraphNeighbors
5      para 0 até Iter faça
6          //Solicita informacoes e calcula PageRank
7          para cada u_i em localGraph faça
8              inEdges ←localGraph.get(u_i)
9              vizinhos = 0;
10             se inEdges não é vazio então
11                 para cada u_j em inEdges faça
12                     inf ←localGraphNeighbors.get(u_j).split(":")
13                     prvalue = inf[0]
14                     links = inf[1]
15                     vizinhos = (prvalue/links) + vizinhos
16                     V_j.PR = 0.15 + (0.85 * vizinhos)
17                 fim
18                 localGraph.put(u_i, inEdges)
19             fim
20         fim
21         //Agrupamento dos valores de page rank que são publicados
22         Cria uma lista keys
23         para cada u_i em localGraph faça
24             inEdges ←localGraph.get(u_i)
25             keys.add(inEdges )
26         fim
27         //Chamada da funcao que serve para publicar os valores de page rank dos vertices
             de localGraph
28         Chama jcl para executar a funcao ("RemoteUpdates", keys)
29         Cria barreira de sincronizacao para aguardar as execucoes de "RemoteUpdates"
30     fim
31 fim
```

outras máquinas do *cluster* para realizar o cálculo do *Page Rank* de um determinado vértice do mapa principal e isto ocorre graças ao particionamento feito na etapa anterior. Há replicação de dados dos vizinhos com o mapa auxiliar (variável $localGraphNeighbors$), mas isto traz redução do uso da rede de dados, entretanto este mapa deve ser atualizado para que os valores de *Page Rank* estejam corretos. Mais a frente será explicado como isto ocorre.

Por fim, é realizada a etapa de sincronização, onde os valores de *Page Rank* são enviados para todas as máquinas do *cluster* (linhas 28 e 29), almejando atualizar os mapas auxiliares em tais máquinas. As três etapas do GAS são repetidas até que o número de iterações, informado como parâmetro para o componente, seja atingido. Assim como na versão 1.0 do JCL Page Rank,

o algoritmo na versão 2.0 utiliza 10 iterações, conforme proposto por Haveliwala (1999).

Algoritmo 3: RemoteUpdates

Entrada: Recebe uma lista com os vértices para serem atualizados $keys$

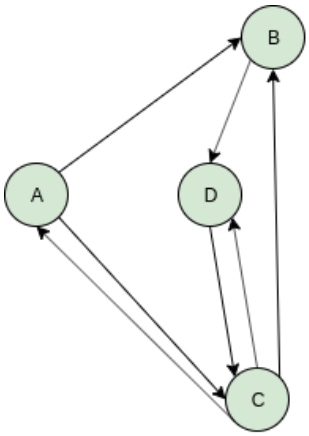
1 **início**
2 Acesso a map auxiliar $localGraphNeighbors$
3 **para** *cada* u_i *em* $keys$ **faça**
4 **se** $localGraphNeighbors$.*get(*u_i*) não é vazio* **então**
5 inf ←localGraphNeighbors.get(u$_i$).split(":")
6 $localGraphNeighbors$.put(u_i, $u_i.PR$+":"+inf[1])
7 **fim**
8 **fim**
9 **fim**

O Algoritmo 3 detalha o processo de atualização do mapa auxiliar em um determinado JCL_Host do *cluster*. Note que todas as tarefas que estejam executando o Algoritmo 2 solicitam a execução de mais k tarefas do tipo *RemoteUpdates*, onde k é tamanho do *cluster*, para que todos os vértices vizinhos sejam atualizados antes que a iteração prossiga até o limite superior (Ex. 10 iterações). O Algoritmo 3 recebe como parâmetro a lista de valores de *Page Rank* atualizados e precisa propagar tais atualizações no mapa auxiliar onde os vértices vizinhos residem. Para cada vértice recebido se valida sua existência no mapa auxiliar (variável $localGraphNeighbors$) e, caso positivo, o valor de *Page Rank* é atualizado (linhas 5 e 6). Ao final da varredura da variável *keys*, há um mapa auxiliar devidamente atualizado segundo os vértices contidos no mapa principal de outro JCL_Host do *cluster* e fruto da execução do Algoritmo 2, precisamente da linha 28 de tal algoritmo.

3.2.3 Funcionamento do JCL Page Rank 2.0

Para ajudar a compreensão do JCL Page Rank 2.0, há o exemplo da Figura 3.2. O algoritmo após executar o Componente Load gera os mapas 3.2 (a), (b), (c) e (d) a partir de um grafo com quatro vértices e sete arestas. O *cluster* no exemplo possui apenas duas máquinas, portanto no mapa principal da máquina 1 há os vértices a e b e a quais vértices estes se relacionam. Já na máquina 2 do *cluster* há os vértices c e d e suas relações. Inicialmente, o mapa auxiliar possui apenas o valor de *Page Rank* da iteração 0, portanto o valor *default* 1.0 é atribuído. Além do *Page Rank*, há o número de arestas que saem dos vértices vizinhos de a e b, ou seja, os vértices a e c possuem 2 e 3 arestas de saída, respectivamente. Na máquina 2 do *cluster* o mapa auxiliar contém todos os vértices do grafo e na iteração 0 os valores de *Page Rank* e as arestas de saída são igualmente preenchidos.

Na máquina 1 do *cluster* o componente Cálculo de Page Rank realiza os cálculos dos valores de *Page Rank* dos vértices a e b e para isto ele utiliza as infromações dos vizinhos armazenados no mapa auxiliar c e d. Uma vez calculados tais valores, o componente já pode



Key	Value
a	c
b	a,c

(a) Mapa Principal máquina 1

Vértice	Iteração 0	Iteração 1	Iteração 2
a	1.0:2	0.43:2	0.54:2
c	1.0:3	1.42:3	1.41:3

(b) Mapa Auxiliar máquina 1

Key	Value
c	a,d
d	b,c

(c) Mapa Principal máquina 2

Vértice	Iteração 0	Iteração 1	Iteração 2
a	1.0:2	0.43:2	0.54:2
b	1.0:1	0.85:1	0.72:1
c	1.0:3	1.42:3	1.41:3
d	1.0:1	1,28:1	1.27:1

(d) Mapa Auxiliar máquina 2

Figura 3.2 – Exemplo JCL Page Rank 2.0

publicar os novos valores nas máquinas 1 e 2, portanto os mapas auxiliares são acessados e as atualizações ocorrem periodicamente. O mesmo pipeline de execução do componente Cálculo de Page Rank é feito na máquina 2 do *cluster*. No mapa principal se armazena o valor de *Page Rank* dos vértices e quantas arestas chegam a tal vértice, portanto o vértice a, por exemplo, tem seu valor de *Page Rank* atualizado tanto no mapa principal quanto no mapa auxiliar, pois este é vizinho do vértice b.

No exemplo da Figura 3.2, para calcularmos o valor de *Page Rank* do vértice b primeiro temos que saber quais vértices referenciam b. Para isto, verificamos o mapa principal da máquina 1 que possui b, obtendo os vértices a e c. Com os vértices vizinhos podemos começar o cálculo da seguinte forma: $PR(b) = 0,15 + 0,85 * (PR(a)/links(a) + PR(c)/links(c))$, onde $PR(a)$ é o valor de *Page Rank* de a na iteração anterior à atual e $links(a)$ é a quantidade de vértices que a referencia. O mesmo vale para $PR(c)$ e $links(c)$. Tanto para a quanto para c as informações de PR e de $links$ são obtidas com o mapa auxiliar. Ao realizarmos o cálculo na $iteração1$ obtém-se, por exemplo, 0.85, portanto basta adotar a mesma lógica para cada vértice do grafo. Apenas quando todos os vértices tiverem seus valores de *Page Rank* atualizados é que uma iteração termina.

4 Experimentos e Avaliações

Neste capítulo, apresentamos os cenários comparativos para avaliar o desempenho do algoritmo JCL Page Rank 1.0 e JCL Page Rank 2.0 frente ao líder do mercado GraphX. A configuração do *cluster*, assim como o detalhamento das instâncias dos grafos utilizados nos experimentos, são feitos na próxima seção. Em seguida, os resultados experimentais são apresentados e, por fim, a discussão sobre os resultados é realizada.

4.1 Configuração dos experimentos

O ambiente experimental consiste em um *cluster* heterogêneo com 5 máquinas com variada capacidade de processamento e armazenamento em memória principal. Os dispositivos são interconectados com um *switch* Ethernet gigabit. A configuração de cada máquina é detalhada na Tabela 4.1, organizados da melhor para o pior máquina em termos de poder de processamento.

Cada experimento foi repetido 5 vezes para cada uma das instâncias de um grafo. Os experimentos foram conduzidos em um *cluster* descrito na Tabela 4.1. As instâncias utilizadas foram obtidas juntamente com o GraphX e suas descrições encontram-se na Tabela 4.2. Os valores inseridos nos gráficos das Figuras 4.1, 4.2 e 4.3 são a média dos resultados dos 5 experimentos; o desvio padrão é informado no gráfico como barras de erro.

Tabela 4.1 – Máquinas utilizadas durante os experimentos e as respectivas configurações

Máquina	S.O	Modelo CPU	CPU Cores	RAM
1	Ubuntu 16.04	Intel Core i5-2500 @ 3.30GHz	4	32GB
2	Ubuntu 16.04	Intel Core i5-2400 @ 3.10GHz	4	8GB
3	Ubuntu 16.04	Intel Xeon E5405 @ 2.0GHz	8	16GB
4	Ubuntu 16.04	Intel Xeon @ 3.0GHz	4	16GB
5	Ubuntu 16.04	Intel Xeon E5310 @ 1.60GHz	4	8GB

4.2 Avaliações e experimentos

Primeiro, avaliamos a escalabilidade das duas versões do JCL Page Rank para as diferentes instâncias utilizadas. Posteriormente, comparamos essas versões do JCL Page Rank contra o algoritmo de *Page Rank* implementado com o GraphX (XIN et al., 2013). Optamos por avaliar nossos algoritmos contra o GraphX por este ser um dos líderes do mercado e representar uma enorme comunidade de desenvolvedores. O GraphX é uma das soluções mais utilizadas fora do âmbito acadêmico, segundo Sahu et al. (2019). Foi utilizada a versão sem tolerância a falhas do GraphX por esta ser a mais veloz e mais simples de ser instalada. Mensurar o impacto do serviço

Tabela 4.2 – Diferentes instâncias utilizadas nos experimentos

Instância	Num. Vértices	Num. Arestas
0	1 mil	1 mil
1	1 mil	10 mil
2	1 mil	100 mil
3	10 mil	10 mil
4	10 mil	100 mil
5	10 mil	1 milhão
6	100 mil	100 mil
7	100 mil	1 milhão
8	100 mil	10 milhões
9	1 milhão	10 milhões

de tolerância a falhas ao executar o algoritmo de *Page Rank* no GraphX é parte dos trabalhos futuros. Há uma instância de um grafo massivo que foi adotada por Xin et al. (2013), entretanto esta não foi utilizada neste trabalho, pois o *cluster* com apenas 5 máquinas não a comportou. Esta e outras instâncias são encontradas nos trabalhos Boldi e Vigna (2004), Boldi et al. (2011), mas não as utilizamos pelo mesmo motivo de saturação de infraestrutura.

4.2.1 Gargalos do JCL Page Rank

Nesta seção, analisamos o comportamento do JCL Page Rank 1.0 e JCL Page Rank 2.0 para as diferentes instâncias. Foram avaliadas separadamente cada etapa de cada algoritmo com o intuito de descobrir possíveis gargalos durante a execução do mesmo. As Figuras 4.1 e 4.2 mostram a porcentagem de cada etapa referente ao tempo de execução total do algoritmo.

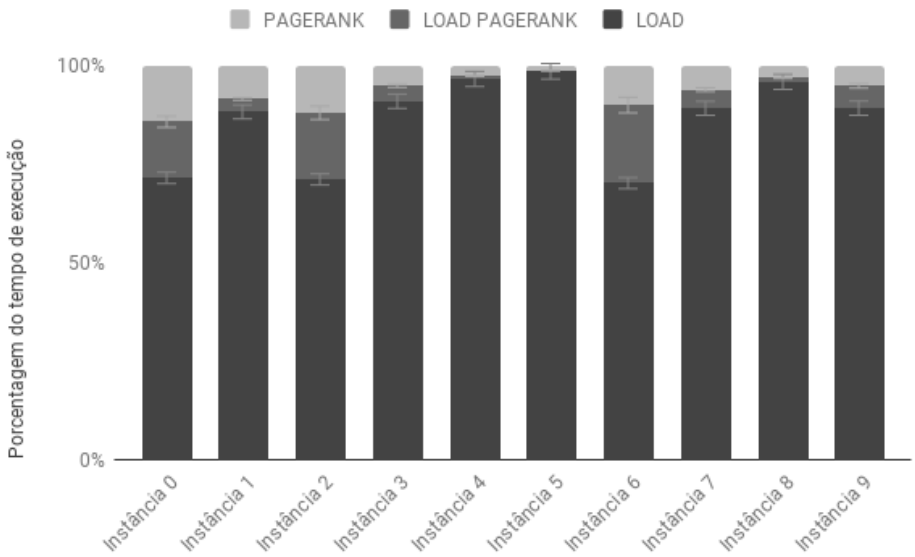


Figura 4.1 – Porcentagem de tempo de cada componente do JCL PageRank 1.0 para as diferentes instâncias

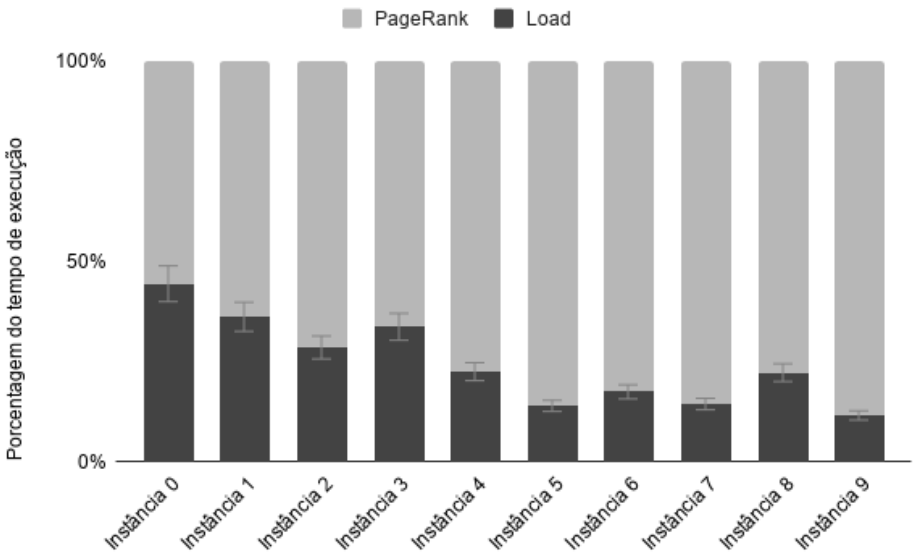


Figura 4.2 – Porcentagem de tempo de cada componente do JCL PageRank 2.0 para as diferentes instâncias

Podemos notar na Figura 4.1 que a implementação utilizada no JCL Page Rank 1.0 é ineficiente, pois necessita muitas trocas de mensagens para a construção da JCL-Hash-Map distribuída, conforme demonstra a porcentagem do tempo na execução do componente "Load"dessa versão. Note que o componente Load é divido em dois outros (Load e Load Page Rank), pois há a necessidade de distribuir os dados do grafo entre os JCL_Hosts inicialmente (Load) para depois, concorrentemente, preencher a JCL-Hash-Map distribuída (Load Page Rank).

No JCL Page Rank 2.0 (Figura 4.2) há apenas a necessidade de enviar as partições do grafo para os JCL_Hosts destinos, portanto não há estruturas distribuídas e nem duas fases de Load. É possível perceber que o gargalo na etapa de Load da versão 1.0 foi solucionado na versão 2.0, dessa maneira a maior parte do tempo de execução da nova versão é do Componente de Cálculo de Page Rank. Ao detalhar os tempos de tal componente, perceberemos que o gargalo é o tempo de publicação dos valores de *Page Rank* nos mapas auxiliares de todas as máquinas do *cluster*, conforme já esperado.

4.2.2 JCL Page Rank versus GraphX

Neste experimento, comparamos o tempo de execução das duas versões do JCL Page Rank contra a solução GraphX. Para o experimento foram utilizadas as 5 máquinas do *cluster*, detalhados na Tabela 4.1. As instâncias da Tabela 4.2 foram novamente utilizadas para tal comparação nos diferentes cenários do experimento.

Nos experimentos fica claro a diferença entre os desenhos arquiteturais das soluções JCL Page Rank 1.0 e as demais, chegando a ser mais de 100x mais lenta. Frente ao GraphX, o JCL Page Rank 2.0 mostrou-se também eficiente, sendo mais rápido em até 12 vezes. Apenas para

a instância 8 o GraphX obteve desempenho aproximadamente 2 vezes melhor que versão JCL Page Rank 2.0. Isto se deve em parte ao fato da instância possuir vértices com muitas arestas e isto gera mapas auxiliares que aumentam significativamente a etapa de publicação de valores de *Page Rank*, ou seja, tais mapas aumentam a latência. A Figura 3.2 ilustra um exemplo onde a máquina 2 do *cluster* possui um mapa auxiliar com todos os vértices do grafo. Como trabalho futuro há inúmeras alternativas de compactar tal mapa, reduzindo não somente seu tempo de envio pela rede, como também o tempo de indexação.

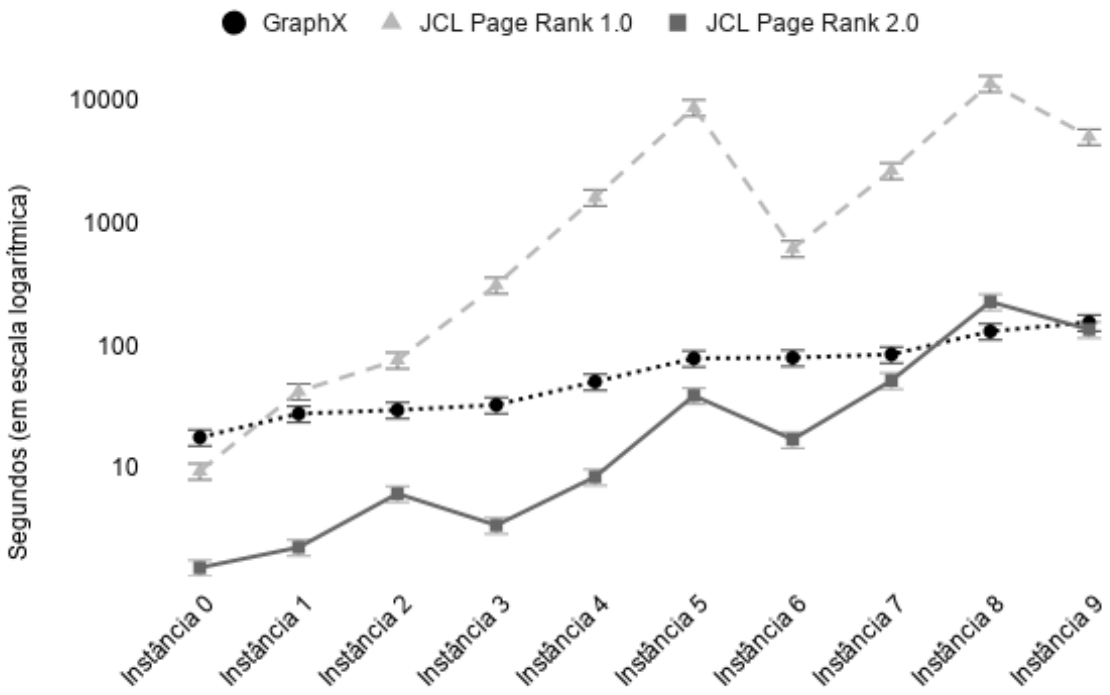


Figura 4.3 – JCL Page Rank versus GraphX

5 Conclusão

A WWW e as redes sociais para os mais variados serviços se popularizam cada dia mais. A utilidade de algoritmos que buscam ranquear páginas ou quaisquer itens segundo sua popularidade, mais precisamente entre vértices de um grafo, é cada vez maior. Um dos desafios de tais algoritmos é conseguir escalar para grandes instâncias de grafos, ou seja, para grafos massivos.

Neste sentido, algoritmos distribuídos são uma das poucas alternativas de solução para se elencar um conjunto de vértices mais populares num grafo massivo de entidades diversas (fotos, textos, vídeos, áudios, mapas, etc.) e suas relações. Neste trabalho, apresentamos dois algoritmos distribuídos para o cálculo dos valores de *Page Rank* de todos os vértices de um grafo. Foram realizados testes comparativos com a solução Apache GraphX, um dos líderes de mercado, possuindo enorme comunidade de usuários e também desenvolvedores. Os resultados mostraram que uma das soluções deste trabalho, denominada JCL Page Rank 2.0, consegue ser mais veloz que o GraphX em nove instâncias de grafos distribuídas em conjunto com o *software* GraphX.

A solução JCL Page Rank 2.0 é mais uma alternativa que utiliza a estratégia GAS, apresentada por Gonzalez et al. (2012), para garantir concorrência no cálculo dos valores de *Page Rank*. O particionamento do grafo no *cluster* ocorre de forma rápida, pois exige apenas o número de máquinas existentes no *cluster* e a função *hash* do vértice. Há dois mapas locais que evitam inúmeras comunicações ao realizar os cálculos de *Page Rank*, entretanto o tamanho de uma delas pode tornar a solução ineficiente. Cabe as soluções JCL Page Rank futuras compactar o mapa auxiliar de diferentes formas com o intuito de acelerar sua montagem, assim como seu envio pela rede.

Outra estratégia de particionamento do grafo poderia ser usar o grau dos vértices. Tal estratégia pode agrupar vértices e reduzir as comunicações, contudo é sempre importante realçar as limitações desta estratégia no que se refere ao dinamismo do grafo, ou seja, suas atualizações e como estas afetam as relações entre vértices, consequentemente afetam o particionamento, causando realocações de vértices em máquinas do *cluster*. Redes sociais e muitos outros casos de estudo possuem grafos dinâmicos. A estratégia de particionamento usada no JCL Page Rank 2.0 é a mesma do JCL e esta não sofre alterações se as relações do grafo mudam, pois não se baseia nesta propriedade.

Os experimentos foram conduzidos em *cluster* com apenas 5 máquinas e todas heterogêneas. A instância massiva com bilhões de arestas e centenas de milhões de vértices não conseguiu ser armazenada em RAM em tal *cluster* por nenhuma solução testada. Cabe aos trabalhos futuros testar com as instâncias de grafos massivos apresentadas por Boldi e Vigna (2004), Boldi et al. (2011). A utilização de nuvens públicas com 32 ou mesmo 64 máquinas formando um *cluster*

homogêneo será muito relevante, pois se assemelhará ao ambiente de teste do GraphX (XIN et al., 2013).

Neste trabalho há algoritmos para cálculo do valor de *Page Rank* de todos os vértices de um grafo e a resolução deste problema já garante solução para o PPR, ou seja, para a versão personalizada do *Page Rank* que funciona entre dois vértices específicos. Mesmo já resolvendo o PPR, há necessidade de experimentos que apresentem os resultados do JCL Page Rank 2.0 em diferentes cenários. Por fim, não foram testados os tipos de consulta suportados pela solução deste trabalho, tal como consultas top-k. Tais experimentos não são complexos, pois já são suportados, todavia auxiliarão ao reforçar a qualidade dos resultados do JCL Page Rank 2.0.

Referências

ALMEIDA, A. L. B. et al. A general-purpose distributed computing java middleware. *Concurrency and Computation: Practice and Experience*, v. 31, n. 7, p. e4967, 2019. E4967 cpe.4967. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4967>.

ALMEIDA, A. L. B. et al. Jcl: A high performance computing java middleware. In: *ICEIS (1)*. [S.l.: s.n.], 2016. p. 379–390.

ANDERSEN, R. et al. Local computation of pagerank contributions. In: SPRINGER. *International Workshop on Algorithms and Models for the Web-Graph*. [S.l.], 2007. p. 150–165.

ARASU, A. et al. Pagerank computation and the structure of the web: Experiments and algorithms. In: . [S.l.: s.n.].

AVRACHENKOV, K. et al. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM Journal on Numerical Analysis*, SIAM, v. 45, n. 2, p. 890–904, 2007.

BAHMANI, B.; CHAKRABARTI, K.; XIN, D. Fast personalized pagerank on mapreduce. In: ACM. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. [S.l.], 2011. p. 973–984.

BAHMANI, B.; CHOWDHURY, A.; GOEL, A. Fast incremental and personalized pagerank. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 4, n. 3, p. 173–184, 2010.

BAHMANI, B. et al. Pagerank on an evolving graph. In: ACM. *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. [S.l.], 2012. p. 24–32.

BARSAGADE, N. Web usage mining and pattern discovery: A survey paper. *Computer Science and Engineering Dept., CSE Tech Report*, v. 8331, 2003.

BEAMER, S.; ASANOVIĆ, K.; PATTERSON, D. Reducing pagerank communication via propagation blocking. In: IEEE. *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. [S.l.], 2017. p. 820–831.

BERKHIN, P. A survey on pagerank computing. *Internet Mathematics*, Taylor & Francis, v. 2, n. 1, p. 73–120, 2005.

BOLDI, P. et al. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: ACM. *Proceedings of the 20th international conference on World wide web*. [S.l.], 2011. p. 587–596.

BOLDI, P.; VIGNA, S. The webgraph framework i: compression techniques. In: ACM. *Proceedings of the 13th international conference on World Wide Web*. [S.l.], 2004. p. 595–602.

BRODER, A. Z. et al. Efficient pagerank approximation via graph aggregation. *Information Retrieval*, Springer, v. 9, n. 2, p. 123–138, 2006.

CHEN, Y.-Y.; GAN, Q.; SUEL, T. I/o-efficient techniques for computing pagerank. In: ACM. *Proceedings of the eleventh international conference on Information and knowledge management*. [S.l.], 2002. p. 549–557.

CHIEN, S. et al. Link evolution: Analysis and algorithms. *Internet mathematics*, Taylor & Francis, v. 1, n. 3, p. 277–304, 2004.

CIMINO, L. de S. et al. A middleware solution for integrating and exploring iot and hpc capabilities. *Software: Practice and Experience*, v. 49, n. 4, p. 584–616, 2019. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2630>.

DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, ACM, v. 51, n. 1, p. 107–113, 2008.

FUJIWARA, Y. et al. Fast and exact top-k algorithm for pagerank. In: *Twenty-Seventh AAAI Conference on Artificial Intelligence*. [S.l.: s.n.], 2013.

GONZALEZ, J. E. et al. Powergraph: Distributed graph-parallel computation on natural graphs. In: *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. [S.l.: s.n.], 2012. p. 17–30.

GONZALEZ, J. E. et al. Graphx: Graph processing in a distributed dataflow framework. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. [S.l.: s.n.], 2014. p. 599–613.

GUO, T. et al. Distributed algorithms on exact personalized pagerank. In: ACM. *Proceedings of the 2017 ACM International Conference on Management of Data*. [S.l.], 2017. p. 479–494.

HAVELIWALA, T. *Efficient computation of PageRank*. [S.l.], 1999.

HAVELIWALA, T. et al. *Computing PageRank using power extrapolation*. [S.l.], 2003.

JEH, G.; WIDOM, J. Scaling personalized web search. In: ACM. *Proceedings of the 12th international conference on World Wide Web*. [S.l.], 2003. p. 271–279.

KAMVAR, S.; HAVELIWALA, T.; GOLUB, G. Adaptive methods for the computation of pagerank. *Linear Algebra and its Applications*, Elsevier, v. 386, p. 51–65, 2004.

KANG, U.; TSOURAKAKIS, C. E.; FALOUTSOS, C. Pegasus: mining peta-scale graphs. *Knowledge and information systems*, Springer, v. 27, n. 2, p. 303–325, 2011.

KEMENY, J. G.; SNELL, J. L. *Markov Chains*. [S.l.]: Springer-Verlag, New York, 1976.

LIN, J.; SCHATZ, M. Design patterns for efficient graph algorithms in mapreduce. In: ACM. *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*. [S.l.], 2010. p. 78–85.

LOFGREN, P.; GOEL, A. Personalized pagerank to a target node. *arXiv preprint arXiv:1304.4658*, 2013.

LOFGREN, P. A. et al. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In: ACM. *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. [S.l.], 2014. p. 1436–1445.

MALEWICZ, G. et al. Pregel: a system for large-scale graph processing. In: ACM. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. [S.l.], 2010. p. 135–146.

PAGE, L. et al. *The PageRank citation ranking: Bringing order to the web.* [S.l.], 1999.

RAHIMIAN, F. et al. Distributed vertex-cut partitioning. In: SPRINGER. *IFIP International Conference on Distributed Applications and Interoperable Systems*. [S.l.], 2014. p. 186–200.

ROY, A.; MIHAILOVIC, I.; ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In: ACM. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. [S.l.], 2013. p. 472–488.

SAHU, S. et al. *The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey*. 2019.

SARMA, A. D. et al. Fast distributed pagerank computation. In: SPRINGER. *International Conference on Distributed Computing and Networking*. [S.l.], 2013. p. 11–26.

WANG, S. et al. Fora: Simple and effective approximate single-source personalized pagerank. In: ACM. *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. [S.l.], 2017. p. 505–514.

WEI, Z. et al. Topppr: top-k personalized pagerank queries with precision guarantees on large graphs. In: ACM. *Proceedings of the 2018 International Conference on Management of Data*. [S.l.], 2018. p. 441–456.

XIN, R. S. et al. Graphx: A resilient distributed graph system on spark. In: ACM. *First International Workshop on Graph Data Management Experiences and Systems*. [S.l.], 2013. p. 2.

ZAHARIA, M. et al. Spark: Cluster computing with working sets. *HotCloud*, v. 10, n. 10-10, p. 95, 2010.