Trabalho de Conclusão de Curso

Modelagem Assíncrona do Page Rank

Matheus Inacio Batisa Santos
matheus.inacio@laccan.ufal.br

Orientador:

Prof. Dr. André Luiz Lins de Aquino

Maceió, Agosto de 2021

Matheus Inacio Batisa Santos

Modelagem Assíncrona do Page Rank

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientador:

Prof. Dr. André Luiz Lins de Aquino

Maceió, Agosto de 2021

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico

Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

S237m Santos, Matheus Inacio Batisa.
Modelagem assíncrona do *Page Rank* / Matheus Inacio Batisa Santos. – 2021.
22 f. : il.

Orientador: André Luiz Lins de Aquino.
Monografia (Trabalho de conclusão de curso em Ciência da Computação) –Universidade Federal de Alagoas, Instituto de Computação. Maceió, 2021.

Bibliografia: f. 19-22.

1. *Pagerank* (Algoritmo). 2. Sistemas distribuídos. 3. Assíncrona. I. Título.

CDU: 004.421

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Instituto de Computação - IC
Campus A. C. Simões - Av. Lourival de Melo Mota, BL 12
Tabuleiro do Martins, Maceió/AL - CEP: 57.072-970
Telefone: (082) 3214-1401

Trabalho de Conclusão de Curso – TCC

Formulário de Avaliação

Curso: _Ciências da Computação_

Nome do Aluno

M	A	T	H	E	U	S		I	N	Á	C	I	O						
B	A	T	I	S	T	A		S	A	N	T	O	S						

Nº de Matrícula

1	3	2	1	4	0	0	3					-	

Título do TCC (Tema)

Modelagem Assíncrona do Page Rank

Banca Examinadora

Andre Luiz Lins Aquino Nome do Orientador	Assinatura
Leonardo Viana Pereira Nome do Professor	Assinatura
Geymerson dos Santos Ramos Nome do Professor	Assinatura

Data da Defesa	Nota Obtida
23 / 08 / 2021	9,5 (nove e meio)

Observações

Realizar as sugestões de texto identificadas pela banca.

Coordenador do Curso
De Acordo

Assinatura

Agradecimentos

É chegada a hora de por um fim nessa etapa da vida iniciada em 2013, de muito aprendizado acadêmico e pessoal, onde pude me encontrar.

Minha gratidão aos meus pais, ao professor André Aquino que mudou a trajetória da minha vida e carreira na universidade, abrindo as portas do LaCCAN e me apresentando para o mundo da pesquisa, também quero agradecer em especial ao Geymerson que me convidou a conhecer o LaCCAN.

Quero agradecer a todos que fazem parte do LaCCAN, Toni, Daise e Thamires pelas conversas e descontração diária. Demétrios, Hyuri, Randy, Bruno, David, Israel, Daniel, Cristopher, Charles, Fabricio, Eduardo, Alvino, Pedro e todos dos "Labs" 15, 16 e 17 por toda parceria, conversas e amizade. A todos os professores, em especial André Aquino (Alla), por acreditar em mim nos momentos que eu não acreditava, por toda confiança, amizade.

Resumo

Neste trabalho, apresentamos uma solução distribuída exata para o problema de classificação dos vértices mais populares de um grafo. Nos últimos anos, a quantidade de links WWW tem se tornado cada vez maior, então algoritmos sequenciais ou apenas paralelos não são mais viáveis, pois são executados em apenas uma máquina. Neste artigo, redesenhamos o algoritmo *PageRank*, proposto pelos fundadores do Google, para que funcione de forma eficiente em *clusters* de máquinas. Trazendo uma proposta inovadora em relação a literatura com a modelagem assíncrona. A solução usa o middleware Java Cá & Lá (JCL) e foi testada em um dos líderes do setor - o Apache GraphX. Os resultados usando instâncias aleatórias e reais demonstraram que nossa solução, chamada JCL *PageRank*, foi $10x$ mais rápida que o GraphX. Realizamos experimentos com máquinas de poder computacional diferentes, desde as configurações mais simples à configurações mais robustaz de *clusters* diferentes e os resultados de tempo de execução foram semelhantes. Ambas as soluções JCL e GraphX calculam os valores *PageRank* para todos os vértices de um grafo, portanto, sem aproximações. O cálculo personalizado *PageRank* também é executado pelas soluções GraphX e JCL *PageRank*.

Palavras-chave: Pagerank, sistemas distribuídos, assíncrona.

Abstract

In this work, we present an exact distributed solution for the problem of ranking the most popular vertices of a graph. In recent years, the amount of WWW links has become increasing, so sequential or just parallel algorithms are no longer feasible as they run on just one machine. In this work, we redesign the *PageRank* algorithm, proposed by Google's founders, so that it runs efficiently under machine clusters. Bringing an innovative proposal in relation to the literature with asynchronous modeling. The solution uses the Java Cá & Lá (JCL) middleware and it has been tested against one of the industry leaders - the Apache GraphX. The results using synthetic and real instances demonstrated that our solution, named JCL *PageRank*, was 10x faster than GraphX. We have performed experiments with small and big machines in two different cluster configurations and the runtime results were similar. Both JCL and GraphX solutions calculate the *PageRank* values for all vertices of a graph, thus without approximations. Personalized *PageRank* calculus is also performed by both GraphX and JCL *PageRank* solutions.

Keywords: Pagerank, distributed systems, asynchronous.

Sumário

Lista de Figuras

Lista de Tabelas

1

Introdução

1.1 Motivação

A *World Wide Web* (WWW) é uma grande coleção de conteúdo composta por textos, imagens, vídeos e áudio que duplica em volume a cada intervalo de seis a dez meses (Barsagade, 2003). Devido a tamanho crescimento, é natural que se desenvolvam algoritmos para extrair informação desse acervo de itens e suas relações. Um dos maiores desafios de tais algoritmos é encontrar relevância de páginas web. A abordagem mais conhecida para tal problema faz uso dos diversos *links* que cada página embute, ou seja, o grafo de links que se forma a partir das diversas páginas existentes na Web é percorrido de diferentes formas para se obter os conteúdos mais relevantes. O método mais conhecido é o *Page Rank* (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 em uma cadeia de Markov, de modo que, navega-se sucessivamente entre as arestas do grafo de forma aleatória. Em um cenário real, pode haver links entre duas páginas que se apontam ou situações mais complexas como ciclos. Para solucionar tal situação ao navegar pelo grafo, normalmente os algoritmos efetuam saltos, indo, por exemplo, para páginas subsequentes de forma aleatória. O algoritmo proposto por Page et al. (1999) considera no cálculo de percurso do grafo uma variável que contabiliza visitas feitas a um vértice.

Dado o atual volume de páginas na *Web*, é impraticável calcular o *Page Rank* de todos os vértices de um grafo de forma sequencial ou paralelamente em uma única máquina. Encontrar novos algoritmos que permitam o percurso do grafo concorrentemente é essencial para ganhos de desempenho, e o desafio se encontra em propor soluções sem perdas de acurácia, escalavelmente, e com sincronização simples mediante o excesso de troca de mensagens entre nós de um *cluster*.

No contexto de *Big Data*, o algoritmo de *Page Rank* permite a identificação de relações em grandes volumes de dados não estruturados através de representação em estruturas de grafo,

demasiadamente complexas para verificação humana. Os grafos podem tornar perceptíveis as características cada vez mais diversificadas dos dados, enriquecidas pela heterogeneidade das redes, em um cenário onde o 5G ganha força juntamente com advento da Internet das Coisas.

O objetivo deste trabalho é propor uma implementação distribuída para o algoritmo de cálculo do *Page Rank*. Em resultados preliminares, implementamos um modelo parcialmente assíncrono, em que cada *host* pode calcular o resultado de forma independente. A proposta utilizará o *middleware* Java Cá&Lá (de Souza Cimino et al., 2019), trabalho anterior em conjunto com UFOP, que disponibiliza uma interface de programação amigável, eficiente, e escalável, além de permitir execução de algoritmos na linguagem de programação Java, uma grande facilidade em termos de portabilidade de aplicação, sem necessidade de refatoração de código. Para fins de comparação com nossa solução, utilizaremos o atual líder de mercado - Apache GraphX. É esperado que a solução apresente resultados consistentes ou melhores em acurácia, escaláveis, ao consumo otimizado de recursos computacionais distribuídos.

1.2 Trabalhos Relacionados

Em Bahmani et al. (2012), são apresentados dois algoritmos para lidar com atualizações incrementais de grafos: *Proportional Probing* e *Priority Probing*. O primeiro usa a ideia de que vértices com valores de *PageRank* maiores devem ser investigados com mais frequência, pois podem impactar outros valores de vértices. A seleção dos vértices a serem explorados ocorre em paralelo, portanto mais de um caminho pode ser verificado a cada iteração do algoritmo. Incrementalmente, o *Proportional Probing* tenta reduzir o espaço de busca ao obter os k vértices mais promissores de um gráfico. O segundo algoritmo detecta vértices com valores *PageRank* proporcionais ao valor atual de todo o gráfico obtido até o momento atual e usando qualquer medida de sumarização. A seleção paralela de caminhos é mantida para acelerar múltiplas travessias. A principal limitação de ambas as ideias é o vetor de valores *PageRank* que deve ser armazenado localmente para fornecer informações para o processo de decisão.

O trabalho de Guo et al. (2017) apresenta uma solução distribuída para o cálculo do *Personalized Page Rank* (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 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.

Em Bahmani et al. (2011), é apresentado um algoritmo para o cálculo do PPR de grafos massivos, conhecido a priori, utilizando *Map Reduce* Dean and 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 et al. (2011) é considerado um dos mais eficientes utilizando *Map Reduce* frente outras Lin and Schatz (2010); Kang et al. (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 com a equação 1.1, resultando no número de 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.

$$O(log(n)/\varepsilon) \tag{1.1}$$

Em Lofgren et al. (2014), é apresentada uma solução paralela, similar à solução apresentada em Bahmani et al. (2011); Sarma et al. (2013), portanto não necessita de pré-processamento. O algoritmo *Frontier-Aided Significance 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 et al. (2010); Sarma et al. (2013) ou esquema algébrico linear Andersen et al. (2007); Lofgren and 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.

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 et al. (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 trinta e duas reais máquinas no *cluster* conseguiu ser três vezes mais rápido do que o experimento com apenas oito máquinas, entretanto quando se usou sessenta e quatro reais máquinas o *speedup* foi apenas de $3.5x$, ou seja, apenas ligeiramente superior à configuração com trinta e duas 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.

No trabalho de Chen et al. (2020) temos uma demonstração clara da importância do estudo sobre algoritmos como Pagerank, podem ser vistos como clássicos, porem estão sempre ganhando espaço, principalmente como a era da informação onde temos os grafos como uma das estruturas de representação de dados bem poderosas.

Neste trabalho é usado *Distributed Backward Search Algorithm* (DBSA) é uma versão do BSA Andersen et al. (2008) para um PPR, com objetivo de obter uma métrica de similaridade e importância para dos atributos usando peso das arestas. A estrutura dos grafo segue uma

topologia estrela, onde o nó central representa uma entidade e nós adjacentes seus atributos.

A solução proposta por Hou et al. (2021), apresenta uma proposta de algoritmo massivo paralelo para *Personalized PageRank* usando de monte-carlo combinado com algoritmos de *push*, com objetivo de reduzir o custo computacional, trazendo um *framework* amigável e com menor requerimento de iterações. Nesta solução a quantidade de iterações é limitada por

$$O(log\frac{n^2 log(n)}{\varepsilon^2 m}) \tag{1.2}$$

sob uma carga de

$$O(m/p) \tag{1.3}$$

onde m é o número arestas do gráfico de entrada, p é o número de executores e ε é um parâmetro de erro definido pelo usuário.

1.3 Objetivo

O objetivo deste projeto foi propor um nova implementação para o algoritmo *Page Rank* de forma distribuída e assíncrona. A proposta utilizou o *middleware* Java Cá&Lá (de Souza Cimino et al., 2019), trabalho anterior em conjunto com UFOP, que disponibiliza uma interface de programação amigável, eficiente, e escalável, além de permitir execução de algoritmos na linguagem de programação Java, uma grande facilidade em termos de portabilidade de aplicação, sem necessidade de refatoração de código.

Para fins de comparação com nossa solução, utilizaremos o atual líder de mercado - Apache GraphX. É esperado que a solução apresente resultados consistentes ou melhores em acurácia, escaláveis, ao consumo otimizado de recursos computacionais distribuídos.

1.4 Contribuições

Neste trabalho, desenvolvemos um *Personalized PageRank* originado de estudo de caso para o middleware já citado *JCL*, apresentando uma arquitetura de fácil implementação, também, para alcançar o modelagem distribuída e por sua vez evoluiu para uma versão assíncrona.

Onde na atualidade temos cada vez mais dados, se faz necessário a constante busca por um algoritmo cada vez mais eficientes no processamento de dados, sendo grafo uma representação muito relevante de dados, temos aqui o objetivo principal de trazer uma contribuição para eficiência e possibilitando uso de *clusters* heterogêneos. Durante os experimentos, confrontando o nosso *baseline* com a implementação do *Pagerank* encontrada no Spark, nossa proposta de PPR se mostrou até $10x$ mais rápida no processamento de um mesmo grafo, se

comparado com a implementação encontrada no Spark.

1.5 Estrutura do texto

A estrutura do texto segue como capítulo 2 apresentando as tecnologias e conceitos envolvidos no trabalho apresentado, no capítulo 3 apresentamos nossa proposta, capítulo 4 apresenta os resultados e por fim, o capítulo 5 contém a conclusão e as considerações finais.

2

Tecnologias e Conceitos

2.1 Java Cá&Lá

O Java Cá & Lá (JCL) é um middleware orientado à tarefas e *Distributed Shared Memory* (DSM) (Almeida et al., 2019; de Souza Cimino et al., 2019). Ele também permite a integração com *brokers Message Queuing Telemetry Transport* 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 (de Souza Cimino et al., 2019).

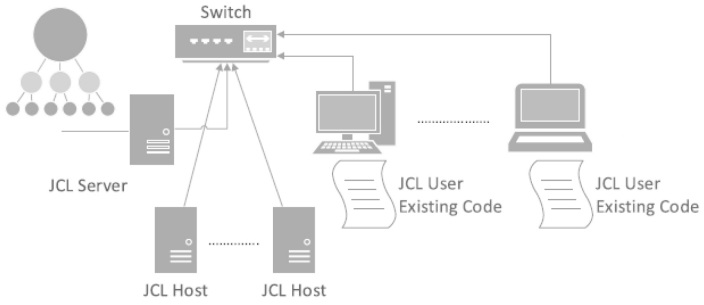


Figura 2.1: Aquitetura JCL (Almeida et al., 2019)

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, 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, entidade com IPs inválidos podem ser integrado por meio de um *gateway*

ou usando somente o *Media Access Control* (MAC), 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, 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, tomando como critério a *Hash* gerada pelo objeto Java. 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.

2.1.1 Host

O JCL Host tem duas funções básicas: armazenar os objetos enviados pelo usuário ou por outro *host* e invocar métodos de classe previamente registrados. Sua arquitetura permite que o componente *Host* determine dinamicamente o número de *threads* (*workers*) executando os métodos de classe do usuário final. Por padrão, o aplicativo é configurado para usar o número total de núcleos disponíveis no *Host*, ou seja, se o computador tiver quatro núcleos, serão criados quatro *workers*. No entanto, o usuário pode criar quantos trabalhadores forem necessários simplesmente definindo a propriedade que atribui o número de *threads* a serem criados. A justificativa para tal recurso é a possibilidade de combinar métodos vinculados à CPU com execuções de métodos vinculados a I/O, exigindo que o sistema operacional os programe, o que aumenta o número de trocas de contexto, porém tal carga de trabalho extra costuma compensar, uma vez que há é uma possibilidade de pré-buscar execuções de métodos vinculados à CPU enquanto espera pelos resultados de I/O.

2.1.2 Hash

Dentro do JCL é possível encontrar estruturas de chave-valor, onde podemos associar uma chave à um valor. Neste componente, assim como as versões do JCL, o *Lambari* representa a implementação local.

Similar ao Lambari JCL Hash, temos um estrutura chave-valor, que se diferencia por ser visível globalmente dentro do *cluster*.

2.2 Pagerank

Para a abordagem do problema, é essencial a compreensão de técnicas do estado da arte para *Page Rank*, sistemas distribuídos e grafos, logo, as atividades iniciais do trabalho devem contemplar uma revisão da literatura, estudando resultados e comportamento de aplicações que utilizam *Page Rank* principalmente em modelagens distribuídas. Dentro dos experimentos programados serão considerados dez grafos aleatórios, assim como grafos formados a partir das paginas da Wikipedia e Orkut, disponibilizados por Leskovec and Krevl (2014). No quesito de implementação e reprodução, o *middleware* Java Cá&Lá será utilizado. O mesmo oferece todo ferramental necessário para execução das mais diversa variações do algoritmo, assim como execução das modelagem, tanto distribuída quanto em multi-core de forma amigável.

Será considerada a definição apresentada por Berkhin (2005) para formalizar a ideia contida no *Page Rank* e explicar mais detalhadamente a alta dependência no uso dos vértices do grafo para se obter as páginas mais relevantes. De modo geral, $G = (P, L)$ é um grafo direcionado com vértices P, que são páginas HTML da Web, e arestas direcionadas L, que são *links* contidos nas páginas. Sendo u uma página da web existente em P, e B_u o conjunto de páginas que referenciam u (arestas de chegada), n_u o seu número de links, o modelo de navegação aleatória utilizado pelo *Page Rank* pode ser definido por

$$f(u) = \sum_{d \in B_u} \frac{f(d)}{n_d} \tag{2.1}$$

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 locais de saída com probabilidade c e salta para algum $d \in P$ com probabilidade $(1-c)$, portanto

$$f(u) = (1-c) + c \sum_{d \in B_u} \frac{f(d)}{n_d} \tag{2.2}$$

para primeira iteração do algoritmo, tomamos

$$f(u) = 1 \tag{2.3}$$

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). Os valores $0,85$ até $0,9$ são usados na prática. 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.

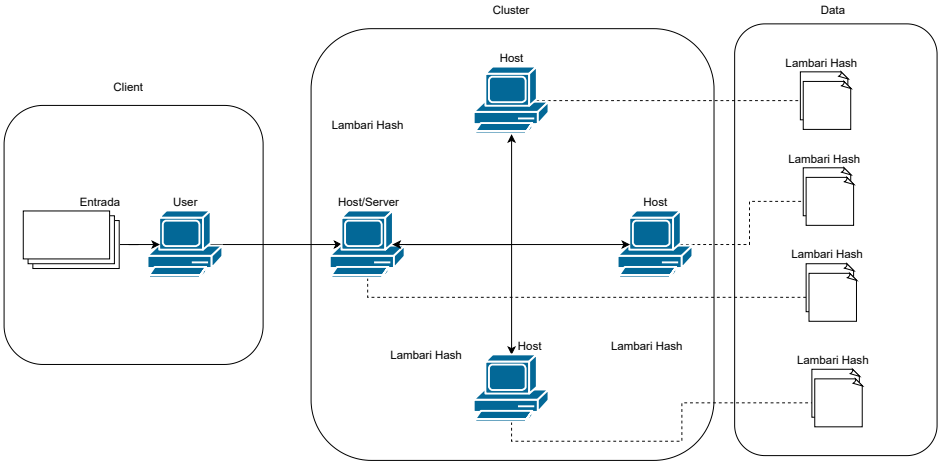
Algoritmos distribuídos para calcular o *Page Rank* de grafos massivos se tornam uma das alternativas de solução mais promissoras, contudo estes podem se tornar extremamente ineficientes. Em linhas gerais, a cada página visitada se necessita dos resultados de etapas anteriores, além da necessidade de uma constante sincronização e várias trocas de mensagens entre nós de um *cluster*.

3

Nossa proposta

Neste trabalho trazemos uma modelagem do algoritmo *pagerank*, onde a principal contribuição é uma arquitetura distribuída, com o objetivo fazer uma implementação de tal forma que seja possível eliminar barreiras de sincronização ou semáforos a cada iteração do algoritmo. Nossa arquitetura pode ser representada por três camadas: *Client*, *Cluster*, *Data*.

Figura 3.1: Arquitetura



3.1 Separação

Na figura 3.1 podemos ilustrar como foi pensada a modelagem, na mesma usamos dois *hosts*, mas isso é meramente ilustrativo, podemos ter n *hosts*. O fluxo de execução vem do *client* por meio da componente *User*, onde os vértices são separados em bloco e divididos entre os *hosts*. Cada *host* h armazena duas *Lambari JCL Hash*, uma contendo matriz de adjacência dos quais h é responsável, e outra com os valores de *pagerank*, tanto para os vértices de sua responsabilidade, quanto para seus vizinhos.

O *client* é componente com menor carga de responsabilidade, visto que depois de receber os dados, dividir e enviar aos *hosts*, não tem participação nas etapas posteriores.

Na camada *data* temos a *Lambari JCL Hash* criada, onde cada *host* possui duas *Lambari JCL Hash*, onde a primeira armazena os valores de pagerank, enquanto a segunda é a lista de adjacência dos vértices que compõem sua parcela de dados.

Nas etapas iniciais o *client* recebe um conjunto de dados G, referentes à construção do grafo, o conjunto G é separo em k fatias, sendo k correspondente ao número de *hosts* disponíveis, onde os vértices não se repetem entre as fatias de dados. A partir daí são construídas as *hashs* compostas da lista de adjacência e tabela de *pagerank*, que deve ser armazenada localmente em cada respectivo *host*.

3.2 Atualização

Nesta modelagem um ponto importante é a comunicação. Como a proposta é trazer uma estrutura de dados que suporte a comunicação assíncrona entre *hosts* na etapa de atualização cada *host* envia para os demais somente os vértices dos quais ele é responsável. Dessa forma temos uma redução no tamanho dos pacotes trocados pelos *hosts* na fase atualização, menor e por sua vez o processamento é mais rápido. Como cada *host* envia unicamente novos valores de pagerank dos seus vértices e cada vértice possui um único *host* responsável, não temos uma condição de disputa no momento de atualização dos valores de pagerank.

4

Resultados e Discussões

Para fins experimentais, neste trabalho foi elaborado um cenário comparativo usando a modelagem proposta em contrapartida com a implementação trazida pelo GraphX, para meio avaliativo adotamos o tempo de processamento.

4.1 Experimento

Tabela 4.1: Tamanho dos grafos

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 mi
6	100 mil	100 mil
7	100 mil	1 mi
8	100 mil	10 mi
9	1 mi	1 mi
10	1 mi	10 mi
11	1 mi	100 mi

Neste experimento geramos doze grafos sintéticos de forma aleatória, seus respectivos tamanhos podem ser encontrados na tabela 4.1 e dois grafos reais baseados respectivamente na rede social Orkut e na página web wikipedia. Para composição do cenário foram usados duas máquinas virtuais (VM).

Configuração das VMs

- 32 Cores;
- 64 RAM;
- Sistema Ubuntu

Cada experimento foi repetido dez vezes, onde consideramos a média de tempo como métrica, esse valores podem ser encontrados na tabela 4.2.

4.2 Tempo de execução

Tabela 4.2: Tempo de processamento do grafo

	JCL-20	**JCL-21.1**	**JCL-21.2**	**GraphX**
0	1,312	1,234	1,82	15
1	0,622	0,811	2,12	21
2	1,249	1,022	2,46	22
3	0,968	0,695	2,81	16
4	1,828	1,285	3,20	24
5	8,364	6,271	4,28	29
6	3,307	2,663	4,99	19
7	8,624	7,297	7,11	49
8	57,512	46,396	17,62	80
9	24,279	12,052	17,78	60
10	83,280	72,240	29,57	312
11	-1	-1	202,64	-1
orkut	-1	-1	356,47	-1

Na tabela 4.2, temos os tempo de processamento para *baseline* GraphX, e evolução da proposta. Para qualquer comparação usaremos a *JCL-21.2* versão mais recente, os valores da primeira coluna fazem referencia aos grafos usados no experimento, valores numéricos correspondem aos grafos aleatórios. Podemos encontrar algumas celular da tabela 4.2 como -1, esses são os casos que a modelagem não consegue concluir o experimento.

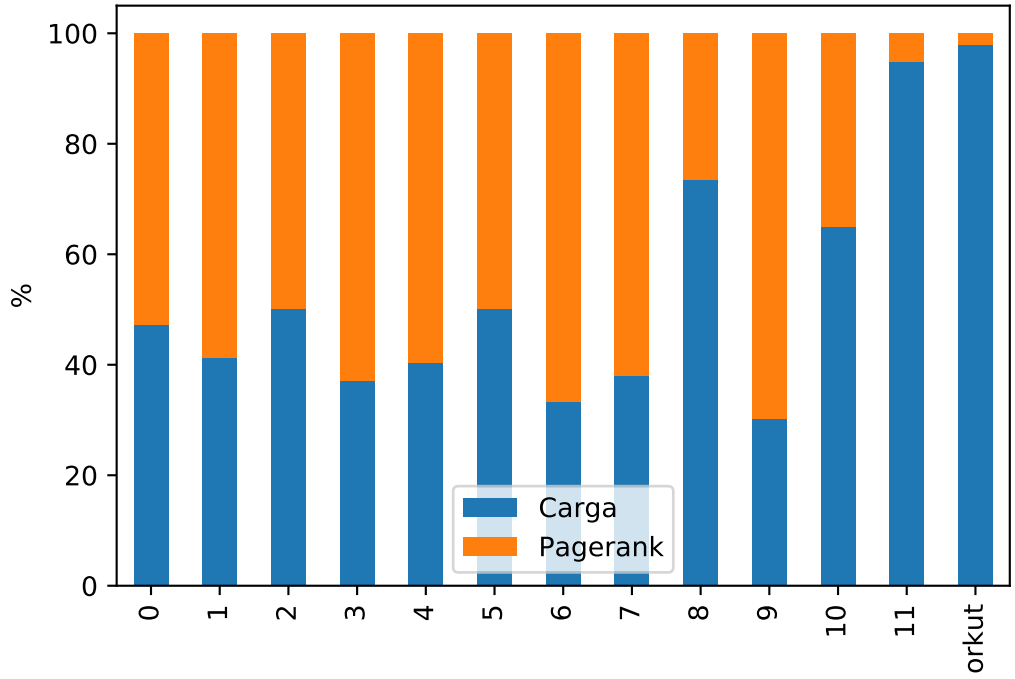
Na figura 4.3, podemos vê a diferença entre tempo de processamento para os experimento que nosso *baseline* apresenta resultados. Já na figura 4.2 temos os resultados para os experimentos que somente a modelagem proposta apresenta resultados, apresentando um outlier para base do orkut, o que é justificado principalmente pelo tamanho do grafo, 3072441 vértices e 234370166 arestas, tornando cada vez mais sensível à problemas de rede.

Uma observação sobre o *GraphX*, durante o experimento fiz registro do uso de memoria, porém o *GraphX* registra uso de toda memoria disponível independente do tamanho da base, assim tomamos decisão de desconsiderar uso de memoria. Outro ponto interessante é *GraphX* apresentou muitos problemas durante execução do experimento 10, onde repetidas vezes não

concluía processo, consultando documentação cheguei ao ponto que a comunicação entre os *hosts* usa protocolo de comunicação UDP.

Durante experimento, foi verificado o resultado dos cálculos do *pagerank* separadamente.

Figura 4.1: Porcentagem de tempo médio de cada componente do JCL PageRank para as diferentes instâncias



4.2.1 Avaliação de tempo por etapa

As Figuras 4.1 mostram a porcentagem de cada etapa referente ao tempo de execução total do algoritmo.Se analisarmos os casos maiores, podemos observar principalmente o curto computacional na etapa de "Carga"cresce principalmente quando aumentamos o números de arrestas, o que faz sentido dado que quanto mais arrestas, mais vértices de um outro *host* vamos encontrar, impactando diretamente a comunicação entre os *hosts*.

Figura 4.2: Tempo de processamento do grafo com JCL

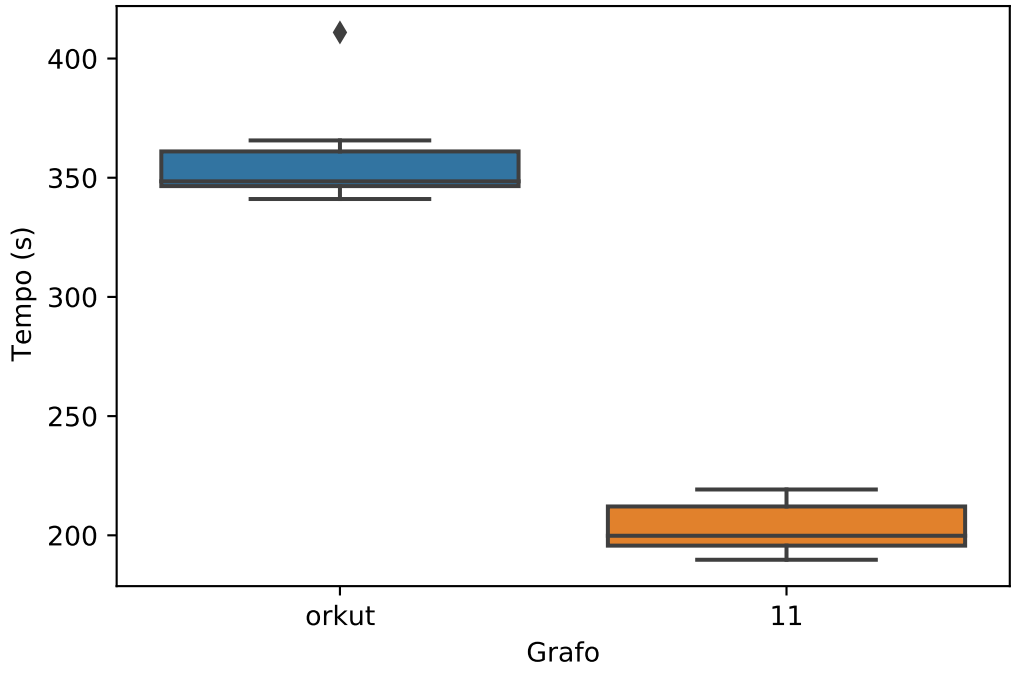
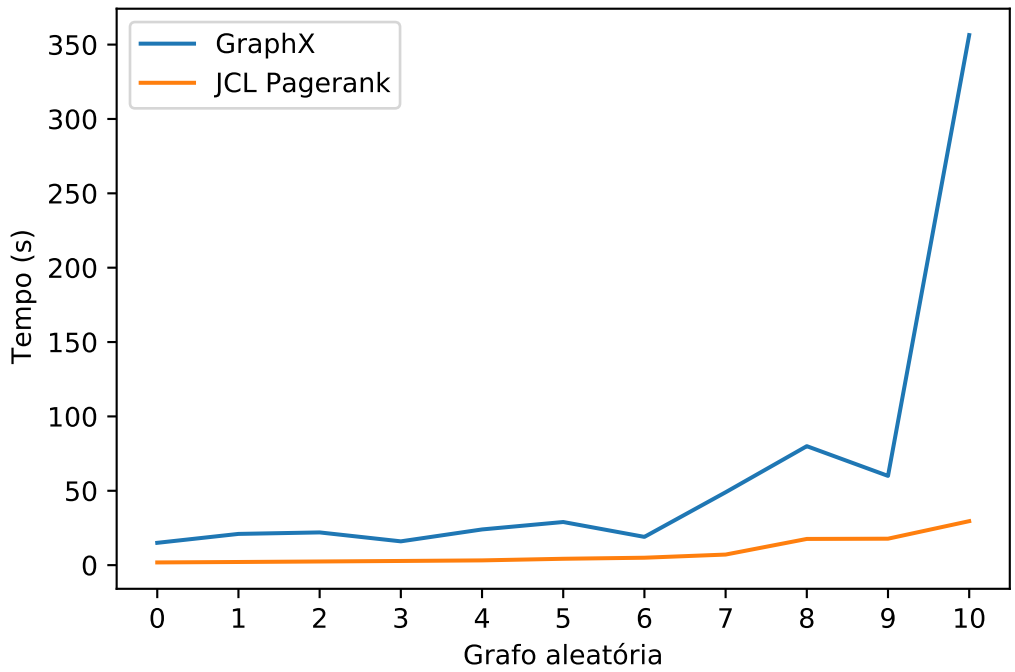


Figura 4.3: Tempo de processamento do grafo



5

Considerações Finais

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.

Sobre este trabalho podemos concluir que um *pagerank* assíncrono é possível, onde obtivermos resultados promissores em contra partida com o estado da arte *Spark*, mas ainda exitemos possibilidades a se explorar com o *Spark*, um exemplo é implementação do *JCL pagerank* dentro do *Spark*, assim como trazer implementação do *pagerank* do *Spark* para o Java cá&lá, e fazendo uma avaliação dentro do mesmo *middleware*.

Em trabalhos futuros seria interessante, levar esta modelagem para outros cenários, avaliar viabilidade do uso para processamentos de redes complexas, explorar problemas maiores e principalmente trazer *cluster* com maior capacidade de processamento. Vale citar que seria interessante fazer um estudo mais completo sobre impacto da fase de carregamento sobre o tempo geral, assim como dados de memoria *Spark* não fornece essa informação separadamente, outro *baseline* interessante seria o Hou et al. (2021).

Referências bibliográficas

André Luís Barroso Almeida, Saul Emanuel Delabrida Silva, Antonio C Nazare Jr, and Joubert de Castro Lima. Jcl: A high performance computing java middleware. In *ICEIS (1)*, pages 379–390, 2016.

André Luís Barroso Almeida, Leonardo de Souza Cimino, José Estevão Eugênio de Resende, Lucas Henrique Moreira Silva, Samuel Queiroz Souza Rocha, Guilherme Aparecido Gregorio, Gustavo Silva Paiva, Saul Delabrida, Haroldo Gambini Santos, Marco Antonio Moreira de Carvalho, et al. A general-purpose distributed computing java middleware. *Concurrency and Computation: Practice and Experience*, 31(7):e4967, 2019.

Reid Andersen, Christian Borgs, Jennifer Chayes, John Hopcraft, Vahab S Mirrokni, and Shang-Hua Teng. Local computation of pagerank contributions. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 150–165. Springer, 2007.

Reid Andersen, Christian Borgs, Jennifer Chayes, John Hopcroft, Vahab Mirrokni, and Shang-Hua Teng. Local computation of pagerank contributions. *Internet Mathematics*, 5 (1-2):23–45, 2008.

Konstantin Avrachenkov, Nelly Litvak, Danil Nemirovsky, and Natalia Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM Journal on Numerical Analysis*, 45(2):890–904, 2007.

Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized pagerank. *Proceedings of the VLDB Endowment*, 4(3):173–184, 2010.

Bahman Bahmani, Kaushik Chakrabarti, and Dong Xin. Fast personalized pagerank on mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 973–984. ACM, 2011.

Bahman Bahmani, Ravi Kumar, Mohammad Mahdian, and Eli Upfal. Pagerank on an evolving graph. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 24–32. ACM, 2012.

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

Pavel Berkhin. A survey on pagerank computing. *Internet Mathematics*, 2(1):73–120, 2005.

Lu Chen, Yunjun Gao, Xingrui Huang, Christian S Jensen, and Bolong Zheng. Efficiently distributed clustering algorithms on star-schema heterogeneous graphs. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

Leonardo de Souza Cimino, José Estevão Eugênio de Resende, Lucas Henrique Moreira Silva, Samuel Queiroz Souza Rocha, Matheus de Oliveira Correia, Guilherme Souza Monteiro, Gabriel Natã de Souza Fernandes, Renan da Silva Moreira, Junior Guilherme de Silva, Matheus Inácio Batista Santos, Andre Luiz Lins Aquino, André Luís Barroso Almeida, and Joubert de Castro Lima. A middleware solution for integrating and exploring iot and hpc capabilities. *Software: Practice and Experience*, 49(4):584–616, 2019. **DOI** 10.1002/spe.2630. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2630.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 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)*, pages 17–30, 2012.

Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613, 2014.

Tao Guo, Xin Cao, Gao Cong, Jiaheng Lu, and Xuemin Lin. Distributed algorithms on exact personalized pagerank. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 479–494. ACM, 2017.

Taher Haveliwala, Sepandar Kamvar, Dan Klein, Chris Manning, and Gene Golub. Computing pagerank using power extrapolation. Technical report, Stanford, 2003.

Guanhao Hou, Xingguang Chen, Sibo Wang, and Zhewei Wei. Massively parallel algorithms for personalized pagerank. *PROCEEDINGS OF THE VLDB ENDOWMENT*, 14(9):1668–1680, 2021.

U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: mining peta-scale graphs. *Knowledge and information systems*, 27(2):303–325, 2011.

Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, pages 78–85. ACM, 2010.

Peter Lofgren and Ashish Goel. Personalized pagerank to a target node. *arXiv preprint arXiv:1304.4658*, 2013.

Peter A Lofgren, Siddhartha Banerjee, Ashish Goel, and C Seshadhri. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1436–1445. ACM, 2014.

Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

Fatemeh Rahimian, Amir H Payberah, Sarunas Girdzijauskas, and Seif Haridi. Distributed vertex-cut partitioning. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 186–200. Springer, 2014.

Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

Atish Das Sarma, Anisur Rahaman Molla, Gopal Pandurangan, and Eli Upfal. Fast distributed pagerank computation. In *International Conference on Distributed Computing and Networking*, pages 11–26. Springer, 2013.

Zhewei Wei, Xiaodong He, Xiaokui Xiao, Sibo Wang, Shuo Shang, and Ji-Rong Wen. Topppr: top-k personalized pagerank queries with precision guarantees on large graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 441–456. ACM, 2018.

Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.