Daniel Presser

GREFT: UMA ARQUITETURA PARA PROCESSAMENTO DISTRIBUÍDO DE GRAFOS DE LARGA ESCALA TOLERANTE A FALTAS

Dissertação submetido ao Programa de Pós Graduação em Ciência da Computação para a obtenção do Grau de Mestre em Ciência da Computação.
Orientador: Prof. Lau Cheuk Lung, Dr.
Universidade Federal de Santa Catarina

Florianópolis

2016

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Presser, Daniel
Greft: Uma arquitetura para processamento distribuído de grafos de larga escala tolerante a faltas / Daniel Presser ; orientador, Lau Cheuk Lung - Florianópolis, SC, 2016.
87 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Ciência da Computação.

Inclui referências

1. Ciência da Computação. 2. Tolerância a Faltas. 3. Sistemas Distribuídos. 4. Processamento de Grafos. I. Lung, Lau Cheuk. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Daniel Presser

GREFT: UMA ARQUITETURA PARA PROCESSAMENTO DISTRIBUÍDO DE GRAFOS DE LARGA ESCALA TOLERANTE A FALTAS

Este Dissertação foi julgado aprovado para a obtenção do Título de "Mestre em Ciência da Computação", e aprovado em sua forma final pelo Programa de Pós Graduação em Ciência da Computação.

Florianópolis, 11 de fevereiro 2016.

Profª. Carina Friedrich Dorneles, Drª.
Universidade Federal de Santa Catarina
Coordenadora

Prof. Lau Cheuk Lung, Dr.
Universidade Federal de Santa Catarina
Orientador

Banca Examinadora:

Profª. Luciana Rech, Drª.
Universidade Federal de Santa Catarina
Presidente

Prof. Altair Olivo Santin, Dr.
Pontifícia Universidade Católica do Paraná

Prof. Frank Augusto Siqueira, Dr.
Universidade Federal de Santa Catarina

Prof. Ronaldo dos Santos Mello, Dr.
Universidade Federal de Santa Catarina

À memória do meu pai.

AGRADECIMENTOS

Agradeço ao meu orientador, professor Lau Cheuk Lung, não só por ter aceito e guiado meu trabalho, mas também pela sua compreensão frente às dificuldades que enfrentei. Sem isso, certamente este trabalho não seria finalizado. Agradeço também à professora Luciana Rech que, mesmo passando por um momento delicado, encontrou tempo para me auxiliar na conclusão e apresentação deste trabalho. Meus agradecimentos também ao Prof. Miguel Pupo Correia, pelas importantes contribuições e coautoria dos artigos, e aos colegas do LaPesD, pela companhia nestes dois anos.

Agradeço também a toda minha família pelo apoio e incentivo nesta jornada. Apesar destes últimos dois anos terem sido de realizações, também foi um período de muita dor, pois tivemos que lidar com a perda do meu pai, Rainer Francisco Presser. Foi dele que tomei gosto pela busca do conhecimento. Ele e minha mãe, Marinês Presser, sempre me apoiaram e deram todas as condições e incentivos para que eu pudesse escolher o melhor caminho para minha formação. Faltam palavras pra descrever minha gratidão por isso. Agradeço também à Nadi, minha tia e principal incentivadora para que realizasse o mestrado. Finalmente, agradeço à Alexandra, minha esposa, cuja companhia dá sentido a tudo isso.

RESUMO

Grafos são usados para modelar um grande número de problemas reais em áreas como aprendizado de máquina e mineração de dados. O crescimento das bases de dados destas áreas tem levado à criação de uma variedade de sistemas distribuídos para processamento de grafos muito grandes, dentre os quais se destaca o Pregel, da Google. Embora esses sistemas costumem ser tolerantes a faltas de parada, a literatura sugere que eles também estão suscetíveis a faltas arbitrárias acidentais. Neste trabalho é apresentado Greft, uma arquitetura para processamento distribuído de grafos de larga escala capaz de lidar com essas faltas, baseado no Graph Processing System (GPS), uma implementação de código aberto do Pregel. São apresentados também resultados experimentais do protótipo obtidos na Amazon Web Services (AWS), onde demonstra-se que este algoritmo usa o dobro de recursos do original, em vez de 3 ou 4 vezes, como é comum em modelos tolerantes a faltas Bizantinas. Com isso, seu custo torna-se aceitável para aplicações críticas que requerem esse nível de tolerância a faltas.

Palavras-chave: Tolerância a Faltas; Sistemas Distribuídos; Processamento de Grafos

ABSTRACT

Graphs are used to model a large number of real problems in areas such as machine learning and data mining. The increasing dataset sizes has led to the creation of various distributed large scale graph processing systems, among which Google's Pregel stands out. Although these systems usually tolerate crash faults, literature suggests they are vulnerable to accidental arbitrary faults as well. In this dissertation we present the architecture, algorithms and a prototype of such system that can tolerate this kind of fault, based on Graph Processing System (GPS), an open source implementation of Pregel. Experimental results of the prototype in Amazon Web Services (AWS) are presented, showing that it uses twice the resources of the original implementation, instead of 3 or 4 times as usual in Byzantine fault-tolerant systems. This cost is acceptable for critical applications that require this level of fault tolerance.

Keywords: Fault Tolerance; Distributed Systems; Graph Processing

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

GPS	Graph Processing System
AWS	Amazon Web Services
BSP	Bulk Synchronous Parallel
HDFS	Hadoop Distributed File System
GAS	Gather-Apply-Scatter
RDD	*Resilient Distributed Dataset*
DAG	*Directed Acyclic Graph*
RDG	*Resilient Distributed Graph*
API	*Application Programming Interface*
TSL	Trinity Specification Language
TFS	Trinity File System
EC2	Elastic Compute Cloud
SSD	*Solid State Disk*
SSSP	*Single Source Shortest Path*
WCC	*Weakly Connected Components*

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Grafos são extensivamente usados na modelagem e solução de problemas reais em áreas como redes, aprendizado de máquina e mineração de dados, com aplicações que incluem algoritmos para recomendação de produtos (MIRZA; KELLER; RAMAKRISHNAN, 2003), detecção de fraudes (EBERLE; GRAVES; HOLDER, 2010) e análise de redes de terroristas por parte de agências de inteligência (LATORA; MARCHIORI, 2004). A emergência das redes sociais nos últimos anos ampliou as áreas de pesquisa fortemente ligadas ao estudo de grafos. Elas inclusive introduziram o conceito de grafos ao grande público, através de ferramentas como a *Graph Search* do Facebook (SENGUPTA, 2013).

Algoritmos e modelos de processamento em grafos também enfrentam os desafios impostos pelo crescente tamanho e complexidade das bases de dados nos últimos anos. Isto tem chamado a atenção dos pesquisadores para modelos de processamento capazes de lidar com grafos muito grandes de maneira paralela e distribuída. Vários modelos têm sido propostos, como Pregel (MALEWICZ et al., 2010), PowerGraph (GONZALEZ et al., 2012), Trinity (SHAO; WANG; LI, 2013) e Graphx (XIN et al., 2013). Dentre estes, destaca-se o Pregel, desenvolvido pela Google. Trata-se de um modelo que pode ser utilizado em *clusters* com milhares de máquinas e é capaz de processar grafos da ordem de bilhões de vértices. Pregel é baseado no Bulk Synchronous Parallel (BSP) (VALIANT, 1990), um modelo de computação paralela e distribuída que se mostrou adequado para o processamento de uma grande variedade de algoritmos em grafos, sendo suportado por diversos sistemas de processamento de grafos de larga escala (GONZALEZ et al., 2012; XIN et al., 2013; SHAO; WANG; LI, 2013). A implementação do Pregel é proprietária da Google, mas existem outros sistemas baseados nele e de código aberto, como o Apache Giraph (AVERY, 2011) e o Graph Processing System (GPS) (SALIHOGLU; WIDOM, 2013).

Por ser tão escalável, Pregel foi desenhado para tolerar faltas por meio do uso de *checkpoints*. De tempos em tempos, o estado de cada processo é armazenado num sistema de arquivos distribuído, para que fique disponível mesmo que o processo falhe. Caso algum problema seja detectado, os processos restauram seu estado a partir do último *checkpoint* registrado. Se alguma das máquinas não estiver mais disponível, seu *checkpoint* é distribuído para as máquinas restantes. Após isso, o

sistema recomeça o processamento do ponto onde parou.

Em sistemas de larga escala, falhas em máquinas ou discos são relativamente comuns, então tolerá-las é fundamental. Entretanto, há problemas conhecidos e mais sutis, que podem afetar a corretude do resultado de um processamento sem causar sua interrupção (SCHROEDER; GIBSON, 2007). Há vários anos, Driscoll et al. (2003) já demonstrava a existência de faltas arbitrárias ou Bizantinas em vários sistemas reais. Mais recentemente, um caso de bit invertido em mensagens trocadas entre os servidores do Amazon S3[1] se propagou e causou instabilidade no sistema. Só foi possível resolver o problema com um reinício total, que resultou em um período de indisponibilidade (Amazon S3 Team, 2008). Um estudo conduzido por 2 anos e meio nos servidores da Google evidenciou que mais de 8% de seus módulos de memória apresentaram falhas em algum momento (SCHROEDER; PINHEIRO; WEBER, 2009). Outro estudo da Microsoft mostrou que erros em processadores também são frequentes (NIGHTINGALE; DOUCEUR; ORGOVAN, 2011). Os mecanismos de tolerância a faltas do Pregel e outros sistemas de processamento distribuído de grafos atuais não são capazes de lidar com falhas arbitrárias acidentais (AVIZIENIS et al., 2004), que podem ser causadas por esse tipo de erro em memórias ou processadores.

Para lidar com faltas de natureza arbitrária, normalmente usa-se a replicação do processamento para mascarar seus efeitos. Um exemplo de técnica genérica é o algoritmo de replicação de máquinas de estado conhecido por *Practical Byzantine Fault Tolerance* (CASTRO; LISKOV, 2002). No entanto, por ser voltada para aplicações cliente-servidor, esta técnica é inadequada para processamento distribuído de grafos. Além disso, são necessárias $3f + 1$ réplicas para tolerar f faltas, o que torna seu custo proibitivo no contexto de larga escala. Outra técnica mais eficiente é a que foi utilizada para tornar o MapReduce (DEAN; GHEMAWAT, 2008) tolerante a faltas Bizantinas (COSTA et al., 2011). Neste modelo, que usa $f + 1$ réplicas, cada tarefa *map* ou *reduce* é replicada e seus resultados são comparados. No caso de divergência, a tarefa é executada novamente até que se alcance uma maioria de resultados iguais. Entretanto, Kajdanowicz, Kazienko e Indyk (2014) mostraram que o MapReduce é até 10 vezes mais lento que modelos como o do Pregel para processamento de grafos, o que inviabiliza seu uso.

[1] Amazon S3 é um serviço de armazenagem de arquivos distribuído e tolerante a faltas.

1.2 PROPOSTA DO TRABALHO

Neste trabalho é proposto o *Greft*, uma arquitetura para processamento distribuído de grafos de larga escala capaz de lidar com faltas arbitrárias acidentais de maneira eficiente. Greft é baseado no Pregel e segue o modelo de computação do BSP, uma popular abstração para processamento paralelo e distribuído usada em diversos outros sistemas de processamento de grafos.

No Greft, cada vértice do grafo é replicado em máquinas diferentes para que seja possível comparar a evolução de seu estado durante o processamento. Com o uso de diversas técnicas foi possível reduzir o número de réplicas necessárias para $f + 1$, para tolerar f réplicas faltosas. Como o sistema é destinado ao processamento de grafos de larga escala, a verificação da evolução do estado dos vértices é otimizada usando-se apenas resumos criptográficos dos estados de todos os vértices de cada réplica, reduzindo enormemente o volume de dados trafegados entre os processos.

É introduzida uma nova técnica de gerenciamento dos *checkpoints*, que dispensa o uso de um sistema de arquivos distribuídos nesta tarefa. Valendo-se da replicação e de resumos criptográficos para validar a integridade dos arquivos, os processos podem armazenar os *checkpoints* em disco local sem comprometer as garantias do sistema. O sistema de arquivos distribuído é usado apenas para armazenar as entradas e os resultados da computação.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

Elaborar uma arquitetura para um sistema de processamento distribuído de grafos de larga escala tolerante a faltas arbitrárias acidentais de maneira eficiente, que utiliza replicação dos vértices do grafo em $f + 1$ réplicas para tolerar até f réplicas faltosas.

1.3.2 Objetivos Específicos

Com vistas a alcançar o objetivo geral, foram definidos os seguintes objetivos específicos:

1. Avaliar o estado da arte em modelos de processamento distribuído de grafos, incluindo os mecanismos de tolerância a faltas em tais modelos, através de levantamento bibliográfico;

2. Propor uma arquitetura, um modelo de sistema e um algoritmo de processamento distribuído de grafos capaz de tolerar faltas arbitrárias acidentais de maneira eficiente através da replicação dos vértices do grafo;

3. Otimizar o armazenamento de *checkpoints* para recuperação de faltas, dispensando o uso de sistema de arquivos distribuído;

4. Implementar um protótipo do modelo e realizar uma avaliação experimental com o uso de grafos reais;

1.4 CONTRIBUIÇÕES

As principais contribuições deste trabalho são:

1. A criação de um algoritmo tolerante a faltas arbitrárias acidentais para processamento distribuído de grafos;

2. A implementação de um protótipo deste algoritmo com base no GPS, uma implementação do Pregel, incluindo otimizações no gerenciamento dos *checkpoints*;

3. Uma avaliação experimental detalhada do funcionamento e desempenho do protótipo na Amazon Web Services (AWS), utilizando um grafo real para os testes.

1.5 ORGANIZAÇÃO DO TEXTO

Neste capítulo são descritas as motivações para realização do trabalho e os objetivos desta dissertação. O restante do documento está organizado da seguinte maneira:

- **Capítulo 2 - Fundamentação Teórica**: Os conceitos sobre sistemas distribuídos e tolerância a faltas, Teoria de Grafos e o modelo Bulk Synchronous Parallel (BSP) são descritos neste capítulo com o objetivo de sustentar o trabalho proposto;

- **Capítulo 3 - Trabalhos Relacionados**: Neste capítulo são listados os trabalhos correlatos ao tema de processamento distribuído de grafos de larga escala, destacando os mecanismos de tolerância a faltas em tais sistemas. É apresentada uma revisão da literatura e o estado da arte no qual a proposta está inserida;

- **Capítulo 4 - Greft**: Neste caítulo são apresentados o modelo, a arquitetura, os algoritmos e a implementação do protótipo do *Greft*, um sistema de processamento distribuído de grafos de larga escala capaz de tolerar faltas arbitrárias acidentais baseado na replicação dos vértices do grafo. Neste capítulo também é descrito o novo mecanismo de armazenamento de *checkpoints* utilizado na recuperação de faltas;

- **Capítulo 5 - Avaliação Experimental**: apresenta os resultados de uma avaliação experimental do protótipo descrito no capítulo anterior. Também é apresentada uma comparação quantitativa com o modelo em que o Greft foi baseado, onde é demonstrado que o Greft usa o dobro dos recursos do modelo original;

- **Capítulo 6 - Conclusões e Trabalhos Futuros**: avalia os resultados obtidos com a realização deste trabalho em relação aos objetivos iniciais e traz sugestões de temas que ainda podem ser explorados em tolerância a faltas de processamento distribuído de grafos.

2 FUNDAMENTAÇÃO TEÓRICA

Para facilitar a compreensão desta dissertação, neste capítulo serão expostos: conceitos básicos de sistemas distribuídos, incluindo suas principais características e modelos de faltas; aspectos da Teoria de Grafos relacionados a este trabalho; e uma descrição do Bulk Synchronous Parallel (BSP), um modelo de processamento paralelo e distribuído utilizado em diversos sistemas de processamento de grafos.

2.1 SISTEMAS DISTRIBUÍDOS

Sistema distribuído é aquele em que componentes de *hardware* e *software* estão interconectados por uma rede e se comunicam e coordenam suas ações apenas através da troca de mensagens (COULOURIS; DOLLIMORE; KINDBERG, 2005). Apesar de simples, essa definição é ampla o suficiente para englobar desde pequenos sistemas compostos por poucas máquinas numa rede local a sistemas com milhares de componentes distribuídos geograficamente e conectados pela Internet.

A partir desta definição é possível inferir as principais características de sistemas distribuídos, que devem ser consideradas no seu desenvolvimento. Segundo Coulouris, Dollimore e Kindberg (2005), as principais características são:

- **Concorrência**: é da natureza de sistemas distribuídos que tarefas sejam executadas concorrentemente em máquinas separadas. Se bem explorada, essa característica permite ao sistema aumentar sua capacidade de processamento adicionando-se mais componentes. Para isso, a coordenação da execução e o compartilhamento de recursos entre os componentes são aspectos que precisam ser tratados adequadamente pelo sistema.

- **Ausência de relógio global**: como os componentes do sistema só podem se comunicar através de troca de mensagens, a precisão com que é possível sincronizar seus relógios é limitada. Normalmente os tempos de comunicação têm variações desconhecidas, e os próprios relógios das máquinas apresentam diferenças na contagem do tempo. Como a coordenação das ações no sistema normalmente é baseada numa noção de tempo compartilhada entre os componentes, é necessário considerar como implementar a sincronia no sistema.

- **Falhas independentes**: em sistemas distribuídos, há várias formas como seus componentes podem falhar. Falhas na rede podem causar o isolamento de máquinas, e estas podem não ter como detectar se o restante do sistema parou de funcionar ou está apenas muito lento. Da mesma forma, uma falha numa máquina que a faça parar de funcionar pode não ser observada imediatamente pelo restante do sistema. Partes do sistema podem falhar independentemente de outras, e isso precisa ser levado em consideração na implementação.

Para facilitar a compreensão de um sistema distribuído, normalmente são usadas abstrações para descrever seus componentes. Os **processos** são responsáveis pela execução dos algoritmos do sistema distribuído, e podem representar um computador, um processador de um computador ou mesmo um *thread* de execução dentro de um processador. Já os ***links*** são os canais de comunicação que os processos usam para trocar mensagens, representando a rede que interconecta os computadores onde os processos são executados (CACHIN; GUERRAOUI; RODRIGUES, 2011).

Embora úteis, essas abstrações por si só não são capazes de descrever toda variedade possível de sistemas distribuídos. Por isso é necessário estendê-las incluindo suposições sobre seu funcionamento. Essas suposições incluem, por exemplo, as garantias dadas pelos canais de comunicação quanto à entrega das mensagens, garantias de sincronia (ou sua ausência) e os tipos de faltas que podem ocorrer nos diversos componentes. O conjunto dessas abstrações e suposições dá origem ao **modelo do sistema** (CACHIN; GUERRAOUI; RODRIGUES, 2011), que é uma descrição mais completa das características e garantias do sistema.

2.1.1 Modelos de Sincronia

Os modelos de sincronia descrevem as garantias necessárias ou oferecidas por um sistema distribuído no que se refere aos tempos de transmissão de mensagens, execução de código e flutuações nos relógios[1] de seus componentes. Há três grandes modelos:

- **Sistemas distribuídos síncronos**: nestes sistemas há limites superiores e inferiores bem definidos de tempo necessário para

[1] A taxa de flutuação dos relógios locais é a taxa em que os relógios locais se distanciam de um relógio de referência devido às limitações na precisão da contagem de tempo inerentes aos circuitos usados.

executar cada passo de um processo. As mensagens são transmitidas dentro de intervalos conhecidos de tempo e a taxa de flutuação dos relógios locais dos componentes do sistema tem limites conhecidos (HADZILACOS; TOUEG, 1994).

- **Sistemas distribuídos assíncronos**: nestes sistemas não há nenhuma garantia de tempo definida. O processamento e a transmissão de mensagens podem levar tempos arbitrariamente longos para concluir e não há qualquer garantia quanto à taxa de flutuação dos relógios locais (COULOURIS; DOLLIMORE; KINDBERG, 2005).

- **Sistemas distribuídos de sincronia parcial**: nestes sistemas assume-se que exista um limite superior aos tempos de comunicação e processamento, mas que é desconhecido no início do sistema; ou que exista um limite conhecido mas que seja garantido apenas eventualmente (DWORK; LYNCH; STOCKMEYER, 1988).

As garantias de um sistema distribuído síncrono são muito difíceis de serem obtidas na prática, por isso é incomum que sejam utilizadas (COULOURIS; DOLLIMORE; KINDBERG, 2005). Já os modelos assíncronos e de sincromia parcial, embora tornem o desenvolvimento de aplicações mais complexo, são mais realistas. Suas garantias são compatíveis, por exemplo, com o que se observa na Internet.

2.1.2 Tolerância a Faltas

Falhas em componentes de *hardware* e *software* são comuns em sistemas computacionais. Falhas em componentes de sistemas distribuídos trazem dificuldades adicionais pela característica de independência entre elas, já explorada na caracterização de sistemas distribuídos. Neste contexto, tanto os processos quanto os *links* podem falhar de várias maneiras diferentes. Quando se trata de sistemas de larga escala, esse problema fica mais evidente, pois quanto maior o número de componentes do sistema, maiores as chances de algum deles falhar.

É importante destacar a nomenclatura usada neste trabalho para se referir aos estágios de um sistema no que tange à tolerância a faltas. Conforme Avizienis et al. (2004), o sistema é dito **correto** quando cumpre sua especificação. Uma **falha** ocorre quando sua especificação não é respeitada. Um **erro** é a transição do sistema a um estado que pode levar a uma falha. Uma **falta** é a causa de um erro. Dessa forma,

pode-se dizer que uma falta leva a um erro, que pode levar o sistema a uma falha. É possível, portanto, tolerar apenas faltas, uma vez que a falha é a expressão de um erro que se propagou pelo sistema e o fez desviar de sua especificação.

As faltas que podem ocorrer num sistema distribuído são agrupadas em modelos que seguem uma hierarquia, visível na figura 1. As características dos modelos são:

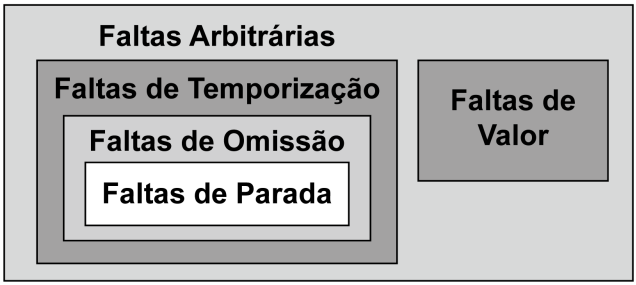


Figura 1 – Classificação de Faltas.

- **Falta de parada (*crash*)**: ocorre quando um processo que estava operando corretamente para de funcionar prematuramente e não responde a mais nenhuma mensagem de outros componentes (TANENBAUM; STEEN, 2007). Neste modelo, os processos restantes podem ou não ser capazes de detectar tal falta. Em sistemas síncronos, essa detecção é mais simples, já que há limites conhecidos de tempo que um componente pode levar para responder. É seguro assumir que um processo parou quando ele não responde dentro desse limite. Em sistemas assíncronos ou de sincronia parcial essa detecção não é possível. Pode-se usar mecanismos de *timeout* para assumir que um processo falhou, mas não é possível ter certeza, uma vez que o processo pode apenas estar sobrecarregado e demorando mais que o normal para responder às mensagens (COULOURIS; DOLLIMORE; KINDBERG, 2005).

- **Falta de omissão**: ocorre quando um processo não responde a uma requisição. Normalmente esse tipo de falta é causado pela perda de mensagens por causa de problemas nos canais de comunicação ou por *buffer overflows* não tratados adequadamente pelo sistema (TANENBAUM; STEEN, 2007).

- **Falta de temporização**: ocorre quando um processo não res-

peita as especificações de tempo de resposta do sistema. Normalmente estas faltas são causadas por problemas de desempenho num componente que o fazem enviar mensagens com atraso, embora também seja possível ocorrer quando um componente envia mensagens antes do esperado pelo destinatário (TANENBAUM; STEEN, 2007).

- **Falta de valor**: trata-se de um tipo de falta mais sério, e ocorre quando um processo do sistema envia um valor errado como resposta a uma requisição (TANENBAUM; STEEN, 2007).

- **Faltas arbitrárias**: também conhecidas por Bizantinas, é o modelo mais complexo de faltas. Nele os processos e canais apresentam comportamentos arbitrários, podendo omitir ou enviar mensagens com valores arbitrários, desrespeitar definições de tempo e pular para estados incorretos. Estas faltas podem ter causas acidentais, como erros em memórias ou processadores, ou serem maliciosas. No caso de faltas maliciosas, processos faltosos podem agir de maneira combinada para enganar o sistema ou atrasar ao máximo sua execução (TANENBAUM; STEEN, 2007).

Há diversas técnicas para se lidar com faltas em sistemas distribuídos. Dentre elas, é possível destacar as seguintes:

- **Detecção de faltas**: algumas faltas podem ser detectadas e podem ser tratadas antes de se propagarem pelo sistema. Um exemplo é o uso de resumos criptográficos para detectar a corrupção de dados em um arquivo ou mensagem (COULOURIS; DOLLIMORE; KINDBERG, 2005).

- **Mascaramento de faltas**: faltas detectáveis podem ser tratadas de maneira a minimizar ou anular seu impacto, como solicitar o reenvio de mensagens cujo resumo criptográfico é inválido (COULOURIS; DOLLIMORE; KINDBERG, 2005). Para faltas mais complexas pode-se usar a replicação: cria-se duas ou mais cópias de um componente de forma que se um falhar outro possa assumir seu lugar, ou ainda que seja possível detectar um falta comparando o estado das réplicas (TANENBAUM; STEEN, 2007).

- **Recuperação de faltas**: neste caso, o sistema precisa ser desenhado para que seu estado possa ser recuperado após a falha de um componente (TANENBAUM; STEEN, 2007). Normalmente usa-se um mecanismo de recuperação em retrocesso (*rollback*),

em que o sistema volta a um estado anterior à falta. Uma forma de implementar essa técnica é o uso de *checkpoints*: de tempos em tempos o estado do sistema é armazenado e, quando ocorre uma falta, o estado é restaurado de forma que o sistema possa continuar em funcionamento.

2.2 TEORIA DE GRAFOS

Nos últimos anos, a Teoria de Grafos se estabeleceu como uma importante ferramenta matemática numa grande variedade de áreas, como pesquisa em química, genética, linguística, engenharia elétrica, geografia, sociologia e arquitetura. Ao mesmo tempo, também se firmou como uma disciplina própria da matemática (WEST et al., 2001). Como trata-se de um conjunto grande de conhecimentos, nesta seção são explorados apenas os elementos da teoria que têm relação direta com este trabalho.

2.2.1 Definição de Grafos

Muitos problemas do mundo real podem ser descritos através de diagramas compostos de pontos ligados uns aos outros por linhas. Grafos são assim chamados justamente por poderem ser representados graficamente[2], e essa representação gráfica permite visualizar muitas de suas propriedades (BONDY; MURTY, 1976).

Formalmente, um grafo G consiste de um conjunto finito não vazio $V(G)$ de elementos chamados **vértices** e um conjunto finito $E(G)$ de pares não ordenados de elementos de $V(G)$ chamados **arestas**. Uma aresta $[v, w]$ conecta os vértices v e w e normalmente é abreviado como vw (WEST et al., 2001). A figura 2 apresenta um exemplo de grafo cujo conjunto de vértices $V(G)$ é $[u, z, v, w]$ e o conjunto de arestas $E(G)$ é $[uv, uz, zv, vw]$.

Além das definições de vértice e aresta, outros termos são usados para descrever os detalhes de um grafo. Os vértices v e w de um grafo G são ditos **adjacentes** se existe uma aresta d ligando os dois. Esta aresta, por sua vez, é dita **incidente** aos vértices. Da mesma forma, as arestas c e d são ditas adjacentes se têm um vértice v em comum (WEST et al., 2001). Estas relações podem ser observadas na figura 2. Já o **grau** de um vértice v de um grafo G é o número de arestas incidentes

[2] A palavra *graph* em inglês pode ser traduzida como *grafo* ou *gráfico*.

em v (WEST et al., 2001). Na figura 2 o vértice v tem grau 3, já que há 3 arestas (a, c, d) ligando este vértice a outros.

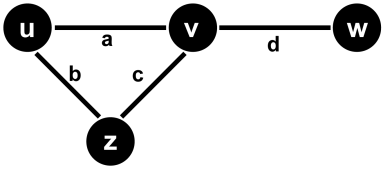


Figura 2 – Um grafo G.

É possível dividir um grafo G em partes, chamadas **subgrafos**. Um subgrafo H de um grafo G é composto por uma lista de vértices $V(H) \subseteq V(G)$ e por uma lista de arestas $E(H) \subseteq E(G)$. A figura 3 representa o subgrafo H extraído do grafo G (figura 2).

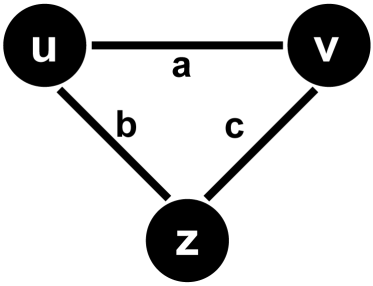


Figura 3 – Um subgrafo H do grafo G.

2.2.2 Grafos Direcionados

A definição dada anteriormente trata de grafos simples. Uma outra categoria de interesse para este trabalho é a de grafos direcionados, ou dígrafos. Nos dígrafos é assumido que arestas que ligam vértices o fazem em uma direção definida. Portanto, a definição do conjunto de arestas $E(DG)$ de um dígrafo DG precisa ser reformulada para: um conjunto finito não vazio de pares **ordenados** de elementos do conjunto de vértices $V(DG)$. Uma aresta que liga os vértices w a v de DG é representada como (w, v), e significa que a ligação vai de w a v, e não o contrário (WEST et al., 2001). A figura 4 representa um dígrafo DG

cujo conjunto de vértices $V(DG)$ é $[u, z, v, w]$ e o conjunto de arestas $E(DG)$ é $[(u, z), (u, v), (v, z), (w, v)]$. As setas das arestas indicam sua direção. Uma aresta c é dita incidente apenas ao seu vértice de destino. Da mesma forma o grau de um vértice é medido pelo número de arestas que o tem como destino.

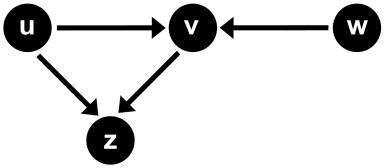
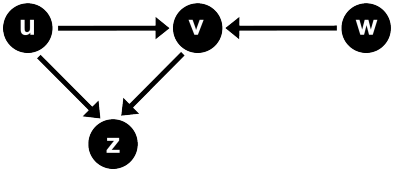


Figura 4 – Um dígrafo DG

2.2.3 Estruturas de Dados para Representação de Grafos

Há várias estruturas de dados que podem ser usadas para representar grafos. As mais comuns são as matrizes de adjacências e listas de adjacências. Uma matriz de adjacência M é a matriz $m \times m$ cuja ij-ésima entrada é 1 se o vértice i é adjacente ao j, e 0 caso contrário (WEST et al., 2001). Um grafo direcionado de exemplo e sua respectiva matriz de adjacências são exibidos na figura 5.



	u	v	w	z
u	0	0	0	0
v	1	0	1	0
w	0	0	0	0
z	1	1	0	0

Figura 5 – Um grafo e sua representação numa matriz de adjacências.

Uma lista de adjacências pode ser definida como um dicionário cujas chaves são os vértices de um grafo e os valores são listas contendo os vértices que lhe são adjacentes. A figura 6 ilustra um dígrafo e sua representação como uma lista de adjacências.

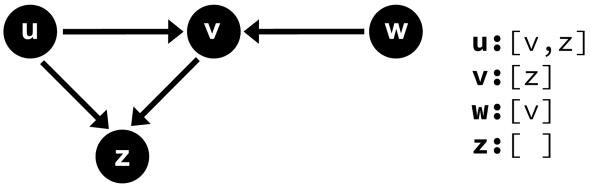


Figura 6 – Um grafo e sua representação numa lista de adjacências.

A diferença entre as duas estruturas de dados está nas complexidades de espaço utilizado para sua armazenagem e de tempo necessário para busca de um vértice. Assumindo V como o número de vértices e E como número de arestas de um grafo, na matriz de adjacências o espaço necessário é da ordem $O(V^2)$, enquanto o tempo de busca de uma aresta é $O(1)$. No caso da lista de adjacências, tanto o espaço necessário quanto o tempo de busca no pior caso são da ordem $O(V + E)$.

2.3 BULK SYNCHRONOUS PARALLEL

Bulk Synchronous Parallel (BSP) (VALIANT, 1990) é um modelo de programação de propósito geral para computação paralela. Inicialmente foi imaginado como um modelo em um nível abaixo das linguagens de programação, de maneira que compiladores pudessem automaticamente produzir código paralelizável seguindo seus princípios com um *overhead* previsível. Entretanto, também observou-se que é possível criar código diretamente compatível com o modelo de maneira mais eficiente para muitas aplicações. Atualmente, ele é usado em diversas áreas que se beneficiam de paralelismo, como computação de álgebra linear, inteligência artificial (SEO et al., 2010) e processamento de grafos (MALEWICZ et al., 2010).

O BSP é caracterizado pela divisão da computação em passos, chamados ***supersteps***, que possam ser executados em paralelo, mas que tem seu início sincronizado (i.e., cada *superstep* só começa quando o anterior terminar em todos os processos). Durante a execução destes passos, cada processo acessa apenas uma memória local para realizar a computação. Quando todos os processos finalizam um *superstep* é iniciado um período de troca de mensagens para movimentação dos dados computados até então, de forma a atualizar as memórias locais de cada processo. Após todos os processos finalizarem o envio e recebimento de mensagens, o processamento do próximo *superstep* é iniciado

e essa sequencia se repete até o final completo da computação. Essa dinâmica é demonstrada na figura 7. Nela é possível ver que primeiro a computação usando memória local é executada, e que, embora leve tempos diferentes para terminar em cada processo, apenas ao final se inicia a fase de troca de mensagens. Após isso há a sincronização para início de um novo *superstep*.

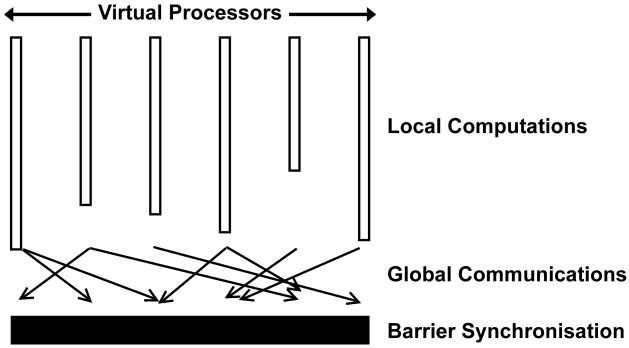


Figura 7 – Um *superstep* (HILL; MCCOLL; SKILLICORN, 1997).

Este modelo é flexível o suficiente para implementar uma grande variedade de algoritmos, além de tornar sua implementação mais simples, já que o programador não precisa se preocupar com condições de corrida ou *dead-locks*, que são problemas comuns em computação paralela. Entretanto, é necessário uma maior preocupação em agrupar a computação a ser realizada pelo algoritmo em lotes, que serão executados em cada *superstep*. Além disso, é preciso distribuir o processamento de uma forma que todos os processos levem aproximadamente o mesmo tempo para finalizar cada *superstep*, pois um processo mais lento compromete toda a computação.

Embora formalmente a troca de mensagens devesse acontecer apenas após a finalização da execução da computação local em todos os processos, as implementações de *frameworks* BSP podem otimizar essa dinâmica de acordo com a necessidade. Pode ser possível aos processos enviar mensagens a qualquer momento da sua computação local, desde que essas mensagens sejam recebidas pelo destinatário após o final do *superstep* corrente. Dessa forma, é possível ao *framework* acumular mensagens para enviá-las de uma só vez, ou enviar as mensagens conforme há recursos disponíveis para tal. Isso permite uma melhor utilização dos recursos, inclusive utilizando tempo que um processo

que tenha terminado sua computação mais rápido ficaria ocioso (HILL; MCCOLL; SKILLICORN, 1997).

Esse mecanismo de troca de mensagens garante paralelismo total aos processos durante cada *superstep*. Isso também permite outras otimizações ao *framework*, como gerenciar o número de processos criados independentemente do número de processadores disponíveis, de forma a reduzir a quantidade de trabalho executado por cada processo. Com isso, é possível distribuir melhor o trabalho entre os processadores disponíveis, executando os processos conforme houver processadores disponíveis, já que os processos podem ser executados tanto paralelamente quanto sequencialmente sem prejudicar a computação.

2.4 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentadas as definições teóricas que embasam o desenvolvimento deste trabalho. A compreensão das características e problemas encontrados em sistemas distribuídos, como tolerância a faltas e mecanismos de sincronização, são fundamentais para o desenvolvimento de um sistema desenhado para lidar com processamento de larga escala, como é o Greft. Estas características, bem como o modelo do sistema, são explorados nas seções 4.1, 4.2 e 4.3 do capítulo de descrição do sistema.

O domínio de aplicação do Greft é delimitado especificamente ao processamento de grafos. Portanto, é necessário descrever quais são as propriedades de grafos interessantes a este trabalho, bem como estabelecer o vocabulário utilizado ao descrever estas propriedades. As definições de grafos descritas aqui são utilizadas nos capítulos 4 e 5.

O modelo BSP é uma abstração usada ou suportada pela maioria dos modelos de processamento de grafos disponíveis na literatura. No capítulo 3 são apresentados diversos sistemas de processamento distribuído de grafos baseados em tal modelo (MALEWICZ et al., 2010; AVERY, 2011; SEO et al., 2010; SALIHOGLU; WIDOM, 2013; GONZALEZ et al., 2012; SHAO; WANG; LI, 2013; XIN et al., 2013; WANG et al., 2014). O próprio Greft é baseado no BSP. Portanto, sua compreensão é fundamental para o desenvolvimento deste trabalho. Além do capítulo 3, o BSP e suas características também são mencionados nos capítulos 4 e 5.

3 TRABALHOS RELACIONADOS

Uma vasta literatura sobre tolerância a faltas Bizantinas ou arbitrárias existe, com modelos propostos desde a década de 1980 (LAMPORT; SHOSTAK; PEASE, 1982). Replicação de máquinas de estado é uma técnica genérica para lidar com faltas, que já demonstrou poder ser implementada de forma eficiente mesmo tolerando faltas Bizantinas (CASTRO; LISKOV, 2002). Depois desse trabalho, vários algoritmos eficientes apareceram, como a biblioteca UpRight (CLEMENT et al., 2009) e a EBAWA (VERONESE et al., 2010). Modelos para processamento paralelo de larga escala, como o *MapReduce*, também contam com propostas para tolerar faltas Bizantinas, como em Costa et al. (2011) e Stephen e Eugster (2013). Entretanto, como já discutido, estes modelos não são adequados ao processamento distribuído de grafos, principalmente pelo custo envolvido nestas técnicas.

Neste capítulo são descritas as principais características de sistemas de processamento distribuído de grafos encontrados na literatura, com atenção especial aos mecanismos de tolerância a faltas que cada um propõe. Inicialmente são listados os trabalhos baseados no modelo BSP, descrito na seção 2.3 deste trabalho. Após, são listados modelos com abstrações diferentes, mas que também trazem contribuições ao tema.

3.1 PREGEL

Pregel (MALEWICZ et al., 2010) é um modelo de desenvolvimento e um *framework* que suporta este modelo para processamento distribuído de grafos de larga escala. É o primeiro a usar um modelo de computação baseado no BSP. Suas contribuições com base nesse modelo são usadas por vários outros trabalhos posteriores. Trata-se de um software proprietário da Google, então não está disponível para uso fora da empresa.

Neste modelo, um grafo direcionado e uma função a ser executada nos vértices desse grafo são recebidas como entrada para a computação. Cada vértice do grafo possui uma identificação única, um valor, um estado (ativo ou inativo), uma lista de adjacências com ou sem pesos e uma fila de mensagens recebidas. Os vértices são distribuídos entre as máquinas de um *cluster* para realizar o processamento. Cada máquina geralmente processa muitos vértices do grafo.

Pregel é composto por dois tipos de processos: um *master* e vários *workers*. Cada *worker* recebe uma partição (um subgrafo) do grafo de entrada e é responsável por executar a função definida pelo usuário nos vértices dessa partição. Já o *master* é o processo responsável pelo particionamento e distribuição das partições do grafo entre os *workers* e a coordenação da execução dos *supersteps*.

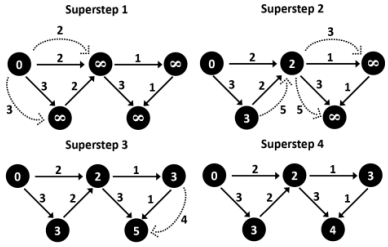
Os programas são expressos como uma sequência de iterações, onde uma função definida pelo usuário é executada em cada vértice do grafo de maneira paralela, com o processamento dividido em *supersteps*, seguindo o padrão estabelecido pelo BSP. A função define o comportamento de um único vértice v, que recebe como parâmetro, dentro do *superstep* S. Nesta função, o vértice v pode ler as mensagens enviadas por outros vértices no *superstep* $S-1$, modificar seu valor e estado, modificar suas arestas e pesos e enviar mensagens a outros vértices. Estas mensagens só serão recebidas pelo vértice de destino no *superstep* $S+1$.

No primeiro *superstep* todos os vértices iniciam com estado ativo e, durante a execução, a função definida pelo usuário pode alterar o estado para inativo. Caso um vértice se torne inativo, a função definida pelo usuário não é executada sobre ele nos próximos *supersteps*. Entretanto, caso o vértice receba uma mensagem, ele é automaticamente reativado e volta a participar do *superstep*. A computação continua enquanto houver vértices ativos ou mensagens em trânsito. Ao terminar, cada *worker* salva sua partição do grafo para um arquivo de saída. O conjunto desses arquivos é considerado o resultado da computação. É importante ressaltar que, embora o Pregel possua esse conceito de barreira de sincronização (originário do BSP), o sistema em si é assíncrono: não há limites estabelecidos nos tempos de comunicação ou de processamento em seus componentes.

A figura 8 ilustra esse modelo com a execução em um grafo do algoritmo *Single Source Shortest Path* (MALEWICZ et al., 2010). Este algoritmo pressupõe um grafo cujas arestas tem pesos e, assumindo esses pesos como uma distância entre os vértices, calcula o menor caminho de um vértice de origem a todos os outros vértices do grafo. Assume-se que todos os vértices iniciam com estado ativo e valor $+\infty$. No primeiro *superstep*, o vértice de origem inicia seu valor com zero, envia uma mensagem para cada um de seus vizinhos somando seu valor atual ao peso da aresta que os conecta e se inativa. O algoritmo também é executado no restante dos vértices, já que no início da computação todos estão ativos. Entretanto, eles apenas se inativam, uma vez que não tem mensagens a receber (as mensagens enviadas pelo vértice origem

serão recebidas apenas no próximo *superstep*).

No segundo *superstep*, a função é executada apenas nos dois vértices adjacentes ao de origem, pois foram ativados automaticamente ao receber as mensagens enviadas no *superstep* anterior. Cada um dos dois vértices atualiza seu valor com o que foi recebido nas mensagens e envia mensagens para seus adjacentes. Esse processo continua até que não existam mais mensagens enviadas e todos os vértices estejam inativos. Nesse ponto, cada vértice terá como valor a menor distância dele ao de origem.



Single Source Shortest Path

```
1: minDist ← origem(this.id)  0 : +∞
2: for msg in this.mensagens do
3:     minDist ← min(msg.dist, minDist)
4: end for
5: if minDist < this.valor then
6:     this.valor ←minDist
7:     for aresta in this.arestas do
8:         valorMsg ← this.valor + aresta.peso
9:         enviar(aresta.id, valorMsg)
10:    end for
11: end if
12: this.ativo ← FALSE
```

Figura 8 – Execução do SSP no Pregel.

Como a transmissão de mensagens enviadas pelos vértices entre as máquinas causa um impacto grande no desempenho do sistema, é possível reduzir o número de mensagens trocadas entre máquinas com o uso de um *Combiner*. Trata-se de uma classe especial que tem por objetivo combinar em uma só as mensagens que tenham o mesmo destino. Com isso, mesmo que vários vértices de uma máquina produzam mensagens para um vértice v em outra máquina, apenas uma mensagem que resulta da combinação é de fato transmitida pela rede para este vértice. Há também a possibilidade de criação de *Aggregators*, que funcionam como objetos globais a todos os processos. Cada processo pode adicionar dados a um *Aggregator* durante um *superstep*. No próximo *superstep*, uma função de agregação é executada sobre esses dados (por exemplo: SUM, MAX, MIN) e o novo valor fica disponível para todos os processos.

Para tolerar faltas, o Pregel usa um mecanismo de *checkpoints*. De tempos em tempos o estado de cada processo é salvo num local persistente (como um sistema de arquivos distribuído). O processo *master* faz checagens periódicas do estado de cada *worker* e, caso um processo pare de responder a essas mensagens ele é considerado faltoso e é assu-

mido que o estado desse processo foi perdido. O *master* então orienta os processos restantes a recarregarem os dados do último *checkpoint* disponível e redistribui as partições sob responsabilidade do processo faltoso entre os processos restantes. O processamento recomeça a partir do ponto onde o *checkpoint* foi feito em todos os processos.

3.2 APACHE GIRAPH

O Apache Giraph (AVERY, 2011) é um sistema de processamento distribuído de grafos de código aberto baseado no Pregel. Giraph implementa as mesmas abstrações disponíveis no Pregel, como a computação por *supersteps*, *Aggregators* e *Combinators*.

O sistema é implementado como um *job* do Hadoop (WHITE, 2012), um popular *framework* de computação paralela e distribuída de propósito geral. Com isso, é possível reutilizar a infraestrutura já montada para o Hadoop que várias empresas possuem, ou mesmo usar serviços de nuvem como o Amazon Elastic MapReduce (Amazon Web Services, 2015).

A arquitetura do Giraph é bastante semelhante à do Pregel. Ele conta com processos *workers* e *masters*, que têm as mesmas funções dos correspondentes no Pregel. Entretanto, no Giraph, o *master* é implementado sobre o Apache Zookeeper (HUNT et al., 2010), um serviço de coordenação de processos replicado, capaz de tolerar faltas de parada. Além disso, são suportados diversos conectores para recuperação e armazenamento do grafo de entrada, *checkpoints* e resultados finais da computação.

Da mesma forma que Pregel, Giraph tolera faltas de parada dos *workers* através do armazenamento de *checkpoints* de tempos em tempos. O intervalo de *supersteps* para criação dos *checkpoints* é definido pelo usuário e os arquivos propriamente ditos são armazenados no Hadoop Distributed File System (HDFS) (SHVACHKO et al., 2010), um sistema de arquivos distribuído que é parte do Apache Hadoop, otimizado para o armazenamento de arquivos de grandes dimensões e capaz de tolerar faltas de parada. Desta forma, caso um *worker* falhe, seu *checkpoint* fica replicado em outras máquinas e pode ser recuperado para a computação recomeçar.

3.3 APACHE HAMA

O Apache Hama (SEO et al., 2010) é um *framework* para processamento de larga escala baseado no BSP, construído sobre o Apache Hadoop (WHITE, 2012). Trata-se de um *framework* mais amplo, que também suporta computação de matrizes e redes neurais além de processamento de grafos. Seu pacote de processamento de grafos é baseado no Pregel e implementa as mesmas abstrações. Já para o processamento de matrizes, é usada uma abstração do MapReduce.

Assim como Pregel e Giraph, Hama tolera faltas de parada através de *checkpoints*. O intervalo de *supersteps* para a criação de *checkpoints* é configurado pelo usuário e, da mesma forma que no Giraph, os arquivos são salvos no HDFS. Com isso, no caso de falha de um processo, é possível recuperar seu *checkpoint* de uma das réplicas do HDFS.

3.4 GRAPH PROCESSING SYSTEM

O Graph Processing System (GPS) (SALIHOGLU; WIDOM, 2013) é uma implementação de código aberto de um modelo baseado no Pregel. Ele traz como contribuições melhorias na partição do grafo entre as máquinas e uma interface para computações globais. As contribuições no particionamento têm por objetivo manter vértices que se comunicam muito nas mesmas máquinas, reduzindo a quantidade de mensagens trocadas na rede. Já a interface para computações globais facilita a implementação de algoritmos que seriam muito complicados de implementar com uma visão centrada aos vértices - como alguns algoritmos de clusterização.

A arquitetura do GPS conta com um processo coordenador, chamado *GPSMaster* que gerencia a execução dos *supersteps* nos processos, chamados de *GPSWorker*. O HDFS é usado para armazenar e distribuir os arquivos de entrada, os *checkpoints*, e os resultados finais da computação. Nos *checkpoints* são armazenados todos os dados dos vértices atribuídos ao processo, incluindo a fila de mensagens recebidas para o *superstep* em que o *checkpoint* foi gerado. Para troca de mensagens, o GPS utiliza Apache MINA, uma biblioteca de comunicação assíncrona de alto desempenho. Essa arquitetura pode ser vista na Figura 9.

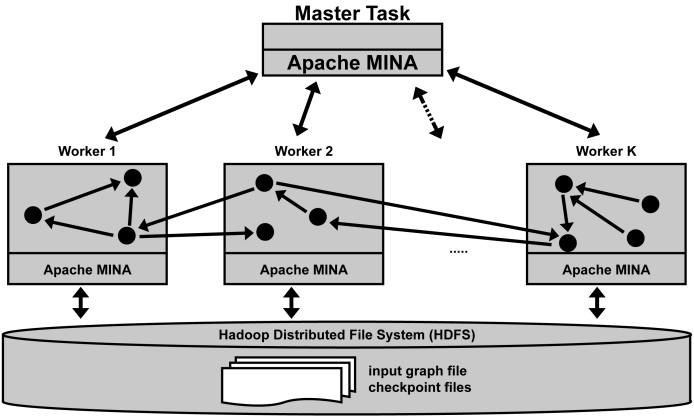


Figura 9 – Arquitetura do GPS (SALIHOGLU; WIDOM, 2013).

Assim como outros sistemas baseados no Pregel, GPS tolera faltas de parada nos processos *workers*. O processo coordenador é o responsável por detectar eventuais falhas através de troca de mensagens periódicas. Caso um *GPSWorker* falhe, o coordenador solicita aos processos restantes que reiniciem o processamento a partir do último *checkpoint* armazenado. Os vértices que estavam sob a responsabilidade do processo faltoso são obtidos de uma réplica do HDFS e redistribuídos entre os processos restantes para que a computação possa ser reiniciada.

3.5 POWERGRAPH

O PowerGraph (GONZALEZ et al., 2012) é um sistema de processamento distribuído otimizado para grafos ditos naturais. Grafos naturais são caracterizados por seguirem lei de potência na relação entre vértices e arestas, ou seja: há poucos vértices com muitas arestas e muitos vértices com poucas arestas. Estes grafos são chamados de naturais pois essa é uma característica observada com frequência em grafos modelados a partir de dados do mundo real. Grafos com essas características podem trazer problemas aos modelos baseados no Pregel, pois neles a execução da função definida pelo usuário em um vértice ocorre em apenas uma máquina. Caso os vértices com grau muito alto não sejam adequadamente distribuídos entre as máquinas, o processamento pode ficar desbalanceado, com algumas máquinas demorando mais que outras para computar um *superstep*.

Para lidar com esse tipo de grafo, o PowerGraph introduz uma abstração de programação com uma interface mais complexa para o programador. Enquanto nas implementações do Pregel o usuário define uma única função a ser executada pelo sistema em cada vértice do grafo em cada *superstep*, o PowerGraph introduz o modelo *Gather-Apply-Scatter* (GAS). Este modelo representa as três fases conceituais da execução de um algoritmo num vértice de um grafo. Primeiramente, os valores de vértices vizinhos ou das arestas do vértice são obtidos (GATHER). Depois, algum processamento é realizado com esses valores que resulta na aplicação de um novo valor ao vértice atual, seus vizinhos ou suas arestas (APPLY). Finalmente, os novos valores são distribuídos aos vértices vizinhos (SCATTER). Um programa no PowerGraph precisa implementar essas três funções, e ainda uma função de agregação SUM que é executada após a GATHER. A estrutura básica de tais programas está representada na figura 10.

```
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather(D_u, D_(u,v), D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^new
  // Run on scatter_nbrs(u)
  scatter(D_u^new, D_(u,v), D_v) → (D_(u,v)^new, Accum)
}
```

Figura 10 – Funções que um programa PowerGraph precisa implementar (GONZALEZ et al., 2012).

Essa abstração permite ao PowerGraph manter o código orientado a vértices, que é comum em outros sistemas, e também distribuir entre máquinas o processamento de um único vértice. As funções GATHER e SUM são executadas como passos de *Map* e *Reduce*, respectivamente, num modelo de MapReduce (DEAN; GHEMAWAT, 2008). Isso quer dizer que sua execução pode ocorrer em paralelo para todos os vizinhos do vértice V que está sendo processado. Entretanto, para que isso seja possível, é necessário que a função SUM definida pelo usuário seja associativa e comutativa. Após as funções GATHER e SUM serem executadas, é invocada a função APPLY sobre V, depois, a função SCATTER para todos os vizinhos de V, também em paralelo.

O PowerGraph tem dois modos de execução: síncrono e assíncrono. O modo síncrono é baseado no BSP e muito semelhante ao do Pregel. Nele, as funções GAS são executadas em sequência, e a execução de

cada uma é chamada de *minor-step*. Um *superstep* é caracterizado pela execução de todos os *minor-steps* em todos os vértices do grafo. Um novo *superstep* só acontece após a finalização do anterior. Já no modo assíncrono, a execução é realizada conforme os recursos de processamento e de rede estão disponíveis. As alterações realizadas nas funções APPLY e SCATTER ficam imediatamente visíveis para os vizinhos do vértice e não existe barreira de sincronização entre *minor-steps*. Este modo é útil para algoritmos que precisam convergir a um valor, já que pode acelerar esse processo. Entretanto, os resultados podem variar de acordo com os recursos disponíveis para execução. Vale ressaltar que os termos *síncrono* e *assíncrono* neste contexto referem-se à sincronia na execução das funções, não em limites de tempo para comunicação ou processamento, como é comum em sistemas distribuídos.

Para tolerar faltas, o PowerGraph também usa um mecanismo de *checkpoints*, que é adaptado para suportar o modo assíncrono. No modo síncrono, como no Pregel, os *checkpoints* são criados entre os *supersteps*, em intervalos configuráveis. Já no modo assíncrono, a execução das funções GAS é suspensa durante a criação do *checkpoint*, de modo a se obter um *checkpoint* consistente. Os arquivos são salvos em um sistema de arquivos distribuídos de maneira que fiquem disponíveis mesmo que uma máquina falhe. No caso de falha, as máquinas restantes restauram seu estado a partir do último *checkpoint* armazenado. Apenas faltas de parada são toleradas por esse mecanismo.

3.6 GRAPHX

O GraphX (XIN et al., 2013) é um sistema de processamento distribuído de grafos baseado no Spark (ZAHARIA et al., 2010). Spark é um sistema de processamento paralelo de propósito geral, com objetivos semelhantes ao do MapReduce (DEAN; GHEMAWAT, 2008). Entretanto, o Spark conta com uma abstração que suporta a computação de tarefas modeladas como dígrafos acíclicos (*Directed Acyclic Graphs* - DAGs) em vez do modelo com apenas as funções *Map* e *Reduce*. Além disso, também conta com uma abstração de armazenamento em memória chamada *Resilient Distributed Dataset* (RDD). O RDD conta com diversos mecanismos de particionamento e distribuição entre os nós do sistema, e é sobre ele que a computação de uma tarefa Spark é executada. Em geral, uma computação no Spark é um conjunto de tarefas modeladas como um DAG, onde cada tarefa recebe um RDD como entrada e produz um novo RDD como saída. Porém, da mesma forma que

outros sistemas de processamento distribuído de propósito geral, essas abstrações não são adequadas para processamento de grafos.

Para processar grafos de maneira eficiente, as abstrações do Spark foram adaptadas, dando origem ao GraphX. Mantendo a lógica do Spark, as tarefas do GraphX podem ser encaradas como transformações em um grafo, onde cada operação recebe um grafo de entrada e produz um novo grafo como saída. Para isso, uma nova abstração de armazenamento foi construída, chamada *Resilient Distributed Graph* (RDG). O grafo é armazenado no RDG como um conjunto de tabelas do RDD, utilizando um particionamento baseado em arestas. As arestas do grafo são distribuídas entre as máquinas do sistema, de forma que cada aresta esteja em apenas uma máquina, e os vértices possam estar em mais de uma máquina. Esse mecanismo é semelhante ao do PowerGraph e facilita o processamento de vértices com grau muito alto, comuns em grafos ditos naturais. Para armazenar o grafo são usadas três tabelas do RDD: uma tabela particionada que contém as arestas do grafo (*Edge Table*), outra que contém os valores dos vértices (*Vertex Data Table*) e uma que mapeia os *ID's* de cada vértice às partições na tabela de arestas onde estão suas arestas (*Vertex Map*). Essa estrutura pode ser vista na Figura 11.

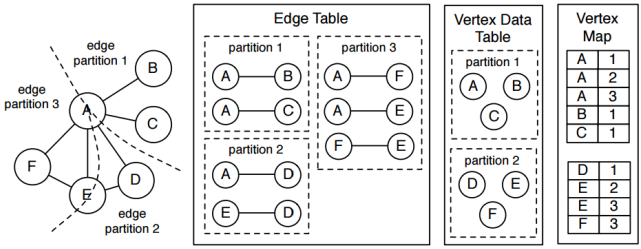


Figura 11 – Estrutura de dados do RDG (XIN et al., 2013).

O RDG tem uma interface própria para implementação de código de processamento dos grafos. Entretanto, por uma questão de compatibilidade e para demonstrar a flexibilidade desse modelo, é possível implementar as mesmas abstrações do Pregel e do PowerGraph sobre a interface do RDG.

A tolerância a faltas do GraphX é baseada nos mecanismos do RDD, sobre o qual o RDG é construído. O RDD tem um mecanismo de armazenamento do histórico das operações realizadas em cada *dataset*, semelhante a um mecanismo de *checkpoints*. Este histórico de operações pode ser usado para restauração do estado do sistema no caso de faltas.

A recuperação consiste em executar novamente essas operações sobre os dados originais até chegar ao estado que o sistema estava no momento da falta. Este mecanismo suporta apenas faltas de parada.

3.7 TRINITY

Trinity (SHAO; WANG; LI, 2013) é um sistema distribuído de processamento de grafos baseado numa nuvem de memória compartilhada. O sistema é composto por uma infraestrutura de armazenamento baseada num modelo de memória compartilhada distribuída globalmente endereçável e um *framework* para execução do processamento. Este modelo permite implementação de aplicações OLTP[1], além das OLAP[2] comuns a outros sistemas.

A arquitetura de Trinity pode ser vista na Figura 12. O sistema é composto por três tipos de processos em um *cluster*: *slaves*, *proxies* e *clients*. *Slaves* têm duas funções: armazenar os dados do grafo e executar o código definido pelo usuário. *Proxies* funcionam como um intermediário entre os *clients* e *slaves*, gerenciando a troca de mensagens entre estes componentes, incluindo agregação de mensagens para reduzir o tráfego na rede. Já os *clients* são os componentes que permitem a interação do usuário com um *cluster* Trinity, através de um conjunto de *Application Programming Interfaces* (APIs) disponibilizadas pelo sistema.

A nuvem de memória é implementada nos *slaves* como um armazenamento de chave-valor. As chaves são identificadores globais de 64 bits e os valores podem ser conjuntos de dados de qualquer formato e tamanho. A memória é dividida em *trunks*, sendo que cada *slave* normalmente fica responsável por mais de um *trunk*. São usadas funções de *hash* para, a partir de uma chave, localizar o *trunk* onde ela está armazenada. Há também uma tabela de endereçamento global mantida pelo sistema, que permite localizar a máquina onde está cada *trunk*. Uma vez localizado o *trunk*, usa-se mais uma função de *hash* para localizar a posição do valor naquele *trunk*. Estas funções de *hash* são implementadas de maneira a permitir a inclusão ou remoção de máquinas ao *cluster*.

[1] *OnLine Transaction Processing.*
[2] *OnLine Analytical Processing.*

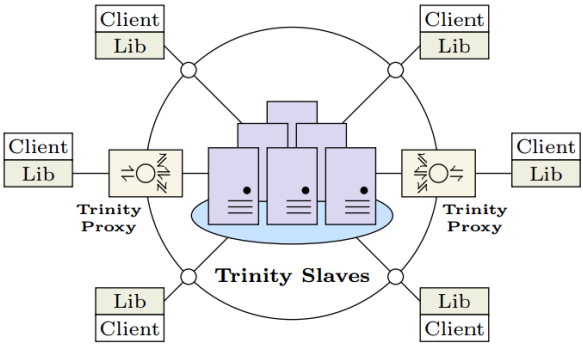


Figura 12 – Arquitetura de um *cluster* Trinity (SHAO; WANG; LI, 2013).

Para computação do grafo, Trinity oferece uma abstração baseada no BSP - igual à do Pregel, e uma segunda abstração também semelhante ao Pregel mas mais restritiva. A diferença desta segunda abstração é que, em cada *superstep*, só é permitida a comunicação com um conjunto fixo de vértices - normalmente os adjacentes ao vértice em questão. Embora esse modelo seja mais restrito, é incomum que vértices se comuniquem com outros vértices que não sejam adjacentes nos algoritmos normalmente usados nestes sistemas. Essa limitação torna os padrões de comunicação previsíveis e permite que o sistema otimize esse processo. Também é suportada uma abstração assíncrona, onde não há a barreira de sincronização dos *supersteps* e o código é executado conforme há recursos disponíveis.

Trinity também possui uma linguagem de especificação, chamada Trinity Specification Language (TSL). O objetivo desta linguagem é mapear para objetos o conjunto de dados armazenado como valor de cada chave. Desta forma, o programador lida com objetos em vez de *arrays* de *bytes* ou outros tipos primitivos, facilitando o desenvolvimento das aplicações.

Para tolerar faltas, Trinity implementa um mecanismo de *checkpoints* quando executado no modo síncrono (BSP), que são gerados em intervalos de *supersteps* e armazenados num sistema de arquivos distribuídos chamado Trinity File System (TFS). Desta forma, os arquivos ficam replicados e disponíveis mesmo que a máquina que os criou falhe. No caso do modo assíncrono, o *framework* introduz interrupções no processamento para que seja possível armazenar os dados de cada máquina de maneira consistente. Além dos *checkpoints*, Trinity também precisa restaurar a tabela de endereçamento global após

a falha de um componente. Para isso, essa tabela é replicada entre os componentes do sistema, sendo que um deles é eleito líder e fica responsável por mantê-la atualizada. No caso de uma falha, o líder recompõe essa tabela de acordo com a nova configuração do sistema e a distribui para as réplicas. Caso o líder falhe, é iniciada uma eleição e, após a escolha de um novo líder, a tabela é restaurada. Com a tabela de endereçamento global e os *checkpoints* restaurados, o sistema pode voltar a processar do ponto onde parou. Com este mecanismo, Trinity tolera apenas faltas de parada.

3.8 SURFER

Surfer (CHEN et al., 2010) é um *framework* para processamento de grafos de larga escala desenhado para ser executado em ambientes de nuvem. Sua interface de programação conta com duas primitivas: *MapReduce* e *Propagation*. A primitiva *MapReduce* é utilizada para realizar o processamento nos vértices de forma paralela, seguindo o modelo *MapReduce* original. Já a *Propagation* é responsável pela distribuição dos resultados das computações *MapReduce* entre os vértices através das arestas do grafo. Além disso, Surfer conta com uma interface gráfica onde o usuário pode construir a lógica de seu programa usando blocos pré-programados disponibilizados pelo *framework*.

Uma computação em Surfer é composta por um conjunto de operações *MapReduce* e *Propagation*. Estas operações são modeladas como um DAG, otimizadas de acordo com regras pré-estabelecidas e transformadas num plano de execução. A arquitetura de Surfer conta com um processo responsável por gerenciar a execução de um plano nas máquinas do cluster. Ao receber um plano, este processo distribui as tarefas de acordo com as máquinas disponíveis e controla a execução de cada tarefa. Para tolerar faltas, este processo gerenciador verifica o estado das máquinas do sistema e, caso alguma falhe, a tarefa que estava sob sua responsabilidade é executada novamente em outra máquina. Com este mecanismo, Surfer é capaz de tolerar apenas faltas de parada.

3.9 IMITATOR

Imitator (WANG et al., 2014), baseado no Pregel, propõe a replicação de vértices em memória para tolerar faltas de parada, dispensando o uso de *checkpoints*. Para isso, Imitator se utiliza de uma

técnica existente em alguns sistemas de processamento de grafos, como Apache Hama, que consiste em replicar vértices com grau muito alto em processos diferentes para tornar o acesso a eles mais rápido nestes processos. Estendendo essa técnica, Imitator cria réplicas de todos os vértices do grafo. No caso de falha de um processo, essas réplicas são redistribuídas entre os processos restantes para dar continuidade ao processamento.

Para manter as réplicas de cada vértice, *Imitator* define uma das réplicas como sendo a *master* e usa os intervalos entre os *supersteps* para atualizar o estado das outras réplicas, chamadas *mirror*, através de troca de mensagens. Esse mecanismo é baseado no utilizado pelo próprio Hama para manter o estado dos vértices replicados com o objetivo de melhorar o desempenho do sistema. A Figura 13 mostra esse mecanismo. Ela assume que já existiriam vértices do grafo de exemplo (*sample graph*) que seriam replicados por questão de desempenho (*COMP replica*) e mostra a replicação do restante dos vértices cujo objetivo é tolerar faltas (*FT replica*).

Além dos dados normais do vértice, tanto os *masters* quanto os *mirrors* armazenam as máquinas onde estão suas réplicas. Dessa forma, caso uma máquina falhe, os vértices nas máquinas restantes sabem quais devem ser restaurados com base nessa informação. A restauração se dá com a transmissão dos vértices que estavam na máquina faltosa para as que restaram no sistema. Alternativamente, se existir uma máquina ociosa disponível, os vértices podem ser transmitidos para ela, evitando sobrecarga. Esse mecanismo é capaz de tolerar apenas faltas de parada.

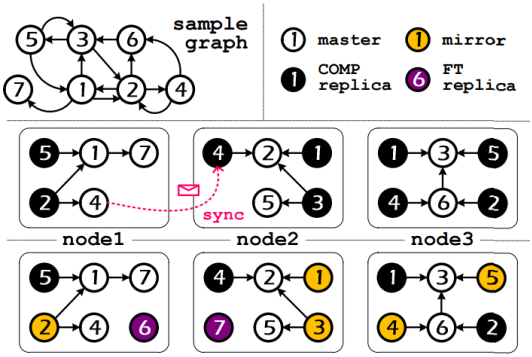


Figura 13 – Replicação de vértices no Imitator (WANG et al., 2014).

3.10 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados os principais trabalhos relacionados ao problema de processamento distribuído de grafos, com foco nos mecanismos de tolerância a faltas apresentados por cada sistema. Um resumo das características de cada modelo é exibido na tabela 1.

Tabela 1 – Trabalhos relacionados e suas características

	Mecanismo de T.F.	**Tipo de falta**	**Réplicas**
Pregel	Recuperação em retrocesso	parada	-
Giraph	Recuperação em retrocesso	parada	-
Hama	Recuperação em retrocesso	parada	-
GPS	Recuperação em retrocesso	parada	-
PowerGraph	Recuperação em retrocesso	parada	-
Trinity	Recuperação em retrocesso	parada	-
GraphX	RDD	parada	-
Surfer	Recuperação em retrocesso	parada	-
Imitator	Replicação	parada	$f+1$
Greft	*Replicação*	*arbitrárias acidentais*	$f+1$

Observa-se que todos os sistemas listados nesta seção tem preocupação com tolerância a faltas. Entretanto, todos estão limitados a modelos de faltas de parada. Além disso, exceto por Imitator, os sistemas usam algum mecanismo de *checkpoint* para recuperação após uma falta. Para que um *checkpoint* fique disponível no caso de parada da máquina onde foi gerado, ele precisa ser armazenado em algum tipo de sistema de arquivos distribuídos. Greft, que será apresentado no próximo capítulo, é o primeiro sistema capaz de lidar com um modelo mais amplo de faltas, incluindo faltas arbitrárias acidentais. Para isso, utiliza replicação do processamento, além de otimizar o armazenamento dos *checkpoints* dispensando o uso de um sistema de arquivos distribuído nesta tarefa.

4 GREFT

Neste capítulo é apresentado o Greft[1], um sistema de processamento de grafos tolerante a faltas arbitrárias acidentais baseado no Pregel (MALEWICZ et al., 2010). Inicialmente são apresentadas a arquitetura e o modelo do sistema, depois os algoritmos e as técnicas usadas.

4.1 DEFINIÇÕES E PREMISSAS DO SISTEMA

Greft é baseado no Pregel, portanto, segue o modelo BSP. Este modelo é o mais utilizado dentre os sistemas de processamento distribuído de grafos existentes na literatura. Assim como outros sistemas, Greft tem como função a realização de tarefas analíticas sobre grafos muito grandes de maneira distribuída e tolerante a faltas. O cliente solicita a execução de uma tarefa e aguarda sua finalização antes de realizar uma nova requisição. Assume-se, portanto, que não há necessidade de ordenação das requisições. Essa é uma característica comum a muitos sistemas de processamento de grafos, como Pregel (MALEWICZ et al., 2010), PowerGraph (GONZALEZ et al., 2012) e GraphX (XIN et al., 2013).

Diferentemente de outros sistemas existentes até o momento, Greft é capaz de garantir que uma computação termine e tenha um resultado correto mesmo na presença de faltas arbitrárias acidentais (AVIZIENIS et al., 2004), utilizando a replicação do processamento para este fim.

Uma requisição Greft é composta por um grafo direcionado de entrada $\mathcal{G}$, uma função definida pelo usuário $\mathcal{F}$ e um conjunto de argumentos opcionais $\mathcal{A}$, que podem ser usados pela função $\mathcal{F}$. Seguindo o modelo do Pregel (MALEWICZ et al., 2010), o grafo $\mathcal{G}$ é uma lista de vértices, sendo que cada vértice é composto por:

- Uma identificação única (ID);
- Um valor cujo tipo é definido pelo usuário;
- Um estado (ativo ou inativo);

[1] O nome foi criado com base numa composição livre dos termos *Graph*, *Replication* e *Fault Tolerance*

- Uma lista de vértices adjacentes com um peso opcional, representado por um valor cujo tipo é definido pelo usuário;
- Uma lista de mensagens recebidas.

A execução da tarefa consiste primeiramente no particionamento do grafo $\mathcal{G}$ em subgrafos, que são distribuídos entre os processos do sistema. O grafo $\mathcal{G}$ é particionado de forma que cada subgrafo possa ser enviado para dois ou mais processos, garantindo assim a replicação da computação.

Após isso, cada processo realiza a execução da função $\mathcal{F}$ sobre os vértices do subgrafo de $\mathcal{G}$ que lhe foi atribuído em iterações chamadas *supersteps*. Os *supersteps* continuam enquanto houver vértices ativos e mensagens pendentes de entrega. Como cada conjunto de réplicas possui os mesmos vértices, é possível comparar seu estado durante o processamento para identificar eventuais estados arbitrários. A arquitetura e os algoritmos do sistema são detalhados nas seções abaixo.

4.2 ARQUITETURA

O sistema é composto de um conjunto de processos distribuídos. Há dois tipos de processos específicos do Greft: GMaster e GWorker. Além destes, o sistema também utiliza o HDFS para receber os grafos de entrada e armazenar os resultados da computação. Os detalhes dos processos são descritos abaixo:

- **GMaster**: é o processo coordenador do Greft. Ele é responsável por atender às requisições dos clientes, distribuir os vértices do grafo de entrada entre os GWorkers e gerenciar a execução dos *supersteps*. O GMaster também é quem define quando *checkpoints* devem ser criados ou restaurados e é o responsável por lidar com as falhas em outros processos de maneira a manter o sistema funcionando. Tipicamente há apenas um GMaster num ambiente de execução Greft.
- **GWorker**: são os processos responsáveis por realizar o processamento propriamente dito no sistema. Eles recebem uma partição do grafo de entrada, realizam a computação de cada *superstep* e armazenam seu estado ou restauram *checkpoints* quando solicitado pelo GMaster. O número de GWorkers num ambiente Greft é variável, e é o componente que garante a escalabilidade do sistema. Seu número pode crescer para atender uma requisição com

um grafo muito grande ou para terminar o processamento mais rápido, e também pode diminuir no caso de requisições menores.

- **HDFS**: é o sistema de arquivos distribuídos utilizado no Greft. É utilizado apenas para receber o grafo de entrada e para armazenar os resultados da computação. Ele é utilizado no começo da computação, antes de iniciar os *supersteps*, quando cada GWorker carrega sua partição do grafo para memória. Depois disso, é usado novamente só ao final de todos os *supersteps*, quando cada GWorker escreve seu resultado em arquivos. A forma como os arquivos de entrada ficam distribuídos no *cluster* HDFS pode ser otimizada utilizando as ferramentas de particionamento do grafo disponibilizadas pelo GPS.

A figura 14 apresenta uma visão de alto nível da arquitetura do Greft e como seria uma distribuição típica em servidores de um *cluster*, incluindo a distribuição de um grafo de exemplo entre os GWorkers. É importante ressaltar que os componentes do HDFS podem ser distribuídos de maneiras diferentes - não é necessário que o *NameNode* resida na mesma máquina do *GMaster*, por exemplo.

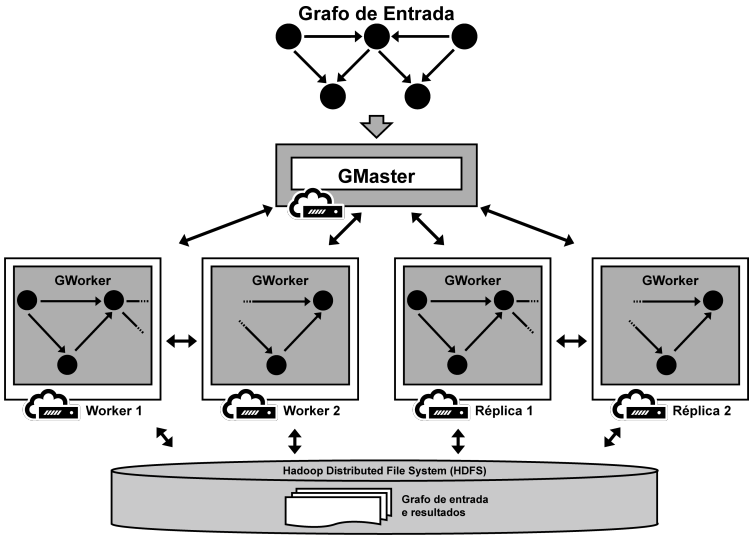


Figura 14 – Arquitetura do Greft

4.3 MODELO DO SISTEMA

Um processo é correto se ele segue os algoritmos estabelecidos. Caso contrário, trata-se de um processo faltoso. Os processos são executados em servidores de um *datacenter*. O modelo de faltas é o de faltas arbitrárias acidentais, permanentes ou transientes (AVIZIENIS et al., 2004). Assume-se que um GWorker pode falhar por parada ou produzir resultados arbitrários, mas não de maneira maliciosa. Já os clientes são tidos como sempre corretos, visto que são os interessados no resultado da computação. O GMaster também é assumido como sempre correto, do mesmo modo que no GPS e no Pregel, que não possuem mecanismos para tratar falhas no coordenador. Embora tenha sido usada a versão oficial, o HDFS também é assumido como tolerante a faltas arbitrárias, pois já existe uma versão tolerante a faltas Bizantinas utilizando a biblioteca *UpRight* (CLEMENT et al., 2009).

O sistema é assíncrono, pois não há nenhum pressuposto quanto a limites em tempo de processamento ou comunicação. Há apenas um tempo máximo de espera na checagem de estado feita pelo GMaster nos GWorkers, que define se um GWorker é suspeito de ter falhado por parada ou não, numa implementação simples de uma abstração de detector de falhas (CACHIN; GUERRAOUI; RODRIGUES, 2011). Também é assumido que os processos são conectados por canais confiáveis, onde mensagens não são perdidas, duplicadas ou corrompidas - como é provido pelo TCP/IP. Finalmente, também assume-se a existência de funções resumo criptográfico (*hash*) resistentes à colisão, isto é, que seja inviável encontrar duas entradas que resultem na mesma saída. O SHA-1 é um exemplo de tal função.

O algoritmo é parametrizado com f. Normalmente este parâmetro indica a quantidade de processos que podem falhar arbitrariamente sem comprometer o funcionamento do sistema. No caso do Greft, o parâmetro tem um significado diferente: assumindo um conjunto $\{V_1, V_2, \cdots, V_n\}$ de réplicas de um vértice V, f é o número máximo de faltas que podem ocorrer nas réplicas de forma que o estado do vértice esteja igual entre elas após a execução de um (ou conjunto de) *supersteps*. Outros parâmetros do algoritmo são descritos na seção 4.4.

4.4 ALGORITMOS

Nesta seção são descritos os algoritmos do Greft: um para o GMaster e outro para os GWorkers. Eles são detalhados a seguir.

4.4.1 GMaster

O algoritmo 1 é o executado pelo GMaster. Como já descrito na seção 4.1, o Greft responde a apenas uma requisição por vez. Dessa forma, o algoritmo do GMaster é executado uma vez para cada tarefa. Os seguintes parâmetros precisam ser informados para a execução:

- $\mathcal{W}$: conjunto de GWorkers disponíveis para a execução da tarefa;
- $\mathcal{G}$: grafo de entrada;
- $\mathcal{N}$: máquinas que estão disponíveis para substituir GWorkers faltosos durante a execução da tarefa;
- f: número de faltas toleradas, conforme definido na seção 4.3;
- f_{max}: indica o número máximo de vezes que um conjunto de réplicas pode apresentar divergências antes de ser removido do sistema;
- *spc*: define o intervalo de *supersteps* entre cada *checkpoint*.

Inicialmente, o algoritmo do GMaster realiza o particionamento do grafo de entrada $\mathcal{G}$ em $\mathcal{P}$ partições. O número de partições é definido pela quantidade de GWorkers disponíveis em $\mathcal{W}$ e pelo valor parametrizado em f, de forma que exista uma partição para cada conjunto de $f + 1$ GWorkers disponíveis em $\mathcal{W}$ (linha 2). Depois, cada partição p é associada a $f + 1$ GWorkers de $\mathcal{W}$, que passam a ser réplicas. Cada conjunto de réplicas é então armazenado na variável $\mathcal{R}$ (linha 4). Estas réplicas deverão seguir a mesma sequência de estados durante a computação da tarefa, já que operam sobre a mesma partição do grafo. Comparando seus estados é possível detectar valores arbitrários. Nesse mesmo ponto é inicializada a variável $\mathcal{F}$, que é usada para controlar o número de inconsistências detectadas em cada conjunto de réplicas em $\mathcal{R}$.

A principal parte do algoritmo é o laço de *supersteps*. Nele, o GMaster define quais as próximas operações que os GWorkers devem executar e envia mensagens para eles com os comandos adequados. Cada mensagem enviada pelo GMaster pode conter um ou mais comandos, e cada comando conta com parâmetros adicionais necessários à sua execução. A lista das mensagens que o GMaster pode enviar estão detalhadas na tabela 2.

Cada iteração do laço de *supersteps* consiste no envio de uma mensagem contendo um comando START_SUPERSTEP e possivelmente

Tabela 2 – Comandos enviados em mensagens do GMaster para GWorkers

Command	**Instrui GWorkers para:**
PARTITION	Carregar uma partição do grafo de entrada
START_SUPERSTEP	Iniciar um novo superstep
CREATE_CHECKPOINT	Salvar o estado atual num *checkpoint*
RESTORE_CHECKPOINT	Substituir o estado atual pelo do *checkpoint*
RESTORE_REPLICAS	Carregar parte de um checkpoint de outro GWorker

comandos para criar ou restaurar *checkpoints*. Após enviar a mensagem para todos os GWorkers, GMaster aguarda que respondam ou que sejam marcados como suspeitos de *crash*. Um GWorker é marcado como suspeito de ter parado quando ele deixa de enviar as mensagens de *heartbeat* para GMaster. Este mecanismo não está descrito nos algoritmos pois é bem simples: de tempos em tempos cada GWorker envia para GMaster uma mensagem com informações sobre o andamento da computação do *superstep* atual.

Para detectar um valor arbitrário nos estados dos vértices do grafo, GMaster precisa de um meio de comparar o estado de cada vértice do grafo com suas $f+1$ réplicas distribuídas entre os GWorkers. Entretanto, seria muito custoso transmitir o estado de todos os vértices de todas as réplicas ao GMaster para que ele pudesse compará-los, uma vez que se assume que o grafo de entrada é muito grande. Em vez disso, Greft usa resumos criptográficos dos estados de cada réplica. Após processar um *superstep*, cada GWorker computa um resumo criptográfico dos estados de todos os vértices sob sua responsabilidade e inclui esse resumo na mensagem enviada ao GMaster. Para computar o resumo criptográfico, os vértices cada GWorker são ordenados numericamente pelo seu identificador e é aplicada, sequencialmente, uma função de *hash* (SHA-1) sobre seus estados. Como os GWorkers estão agrupados em conjuntos de $f+1$ réplicas que processam a mesma partição do grafo, cada vértice de cada partição deve estar no mesmo estado em todas as $f+1$ réplicas após um *superstep*. Disso se conclui que, se o resumo criptográfico é computado numa ordem definida em cada réplica, os resultados de todas as $f+1$ réplicas de cada conjunto serão iguais. Comparando os resumos criptográficos de cada conjunto de réplicas, Greft pode detectar se alguma das réplicas produziu algum valor arbitrário durante o processamento, uma vez que a função de resumo criptográfico resistente à colisões garante que os resumos serão diferentes neste caso. Quem define o que é o estado de cada vértice é a função definida pelo usuário: pode ser apenas o valor do vértice, ou os pesos

das suas arestas, ou ambos.

No caso de *faltas transientes* - que acontecem, mas depois desaparecem - não há necessidade de remover processos e redistribuir os vértices. Bastaria executar novamente o processamento para tentar chegar a uma maioria de $f + 1$ resultados iguais para cada vértice do grafo. Entretanto, uma *falta permanente* (por exemplo, um bit de memória travado em um valor) pode fazer que um conjunto de GWorkers nunca dê resultados iguais. Para este caso, é definido o parâmetro f_{max} como o limite de faltas observadas em um conjunto de réplicas a partir do qual tal conjunto é removido do sistema. Já quando um processo falha por parada (*crash*), ele entra na lista de suspeitos por não enviar as mensagens periódicas de estado (*heartbeats*) ao GMaster. Neste caso, o conjunto de réplicas também é removido do sistema.

Quando o GMaster remove um conjunto de réplicas, primeiramente ele verifica se existem máquinas adicionais que possam ser usadas para substituí-las. Se existirem, a partição dos processos removidos é enviada a elas através do comando *PARTITION* e elas são adicionadas ao processamento. Se não houver mais máquinas à disposição, a informação dos *checkpoints* dos processos removidos é incluída no comando *RESTORE_REPLICAS* e enviado aos GWorkers. Cada conjunto de GWorkers restante, por sua vez, restaura uma partição do arquivo de *checkpoint* do conjunto removido (ver função *restoreCheckpointPartition* do algoritmo 2), de forma que os vértices presentes no arquivo sejam igualmente distribuídos entre eles e sejam replicados. Em ambos os casos, o GMaster também envia o comando *RESTORE_CHECKPOINT* para que os processos restaurem o seu último *checkpoint* e recomecem o processamento.

Com a introdução das réplicas de cada processo não há mais necessidade de armazenar os *checkpoints* num sistema de arquivos distribuídos. Os *checkpoints* passam a ser armazenados apenas no disco local de cada GWorker e, no caso de falha de uma máquina, seu estado pode ser restaurado a partir do último *checkpoint* armazenado em uma de suas réplicas (função *restoreFromLocalOrReplica* no algoritmo 2). Esta técnica também é usada na restauração do estado de processos removidos por atingirem f_max ou que falharam por parada. Dessa forma, o sistema de arquivos distribuídos é usado apenas para armazenar os arquivos de entrada e os resultados da computação. Além disso, sempre que um *checkpoint* é gerado por um GWorker, um resumo criptográfico do arquivo é enviado ao GMaster. Com isso, quando for necessário restaurar *checkpoints*, o GMaster devolve estes resumos para os GWorkers, que podem verificar a integridade de seus arquivos e so-

licitar o arquivo de sua réplica no caso de uma divergência.

Algoritmo 1 GMaster

Require: $\mathcal{W}$: set of workers, $\mathcal{G}$: input graph, $\mathcal{N}$: available nodes

1: **function** GMaster(f, f_{max}, *spc*)
2: $\mathcal{P} \leftarrow$ partition $\mathcal{G}$ into $\lfloor |\mathcal{W}|/(f+1) \rfloor$ subgraphs
3: **for all** $p \in \mathcal{P}$ **do**
4: $\mathcal{R}[p] \leftarrow \{w \in \mathcal{W} \mid w_1 ... w_{f+1}\}$
5: $\mathcal{W} \leftarrow \mathcal{W} - \mathcal{R}[p]$
6: $\mathcal{F}[p] \leftarrow 0$
7: msg $\leftarrow$ empty message
8: msg.addCmd(PARTITION, $\mathcal{R}$)
9: superstep $\leftarrow 1$
10: **while** $\mathcal{G}.active()$ **do**
11: msg.addCmd(START_SUPERSTEP, superstep)
12: **if** superstep *mod* spc == 0 **then**
13: msg.addCmd(CREATE_CHECKPOINT, superstep)
14: sendToAll($\mathcal{R}$, msg)
15: msg $\leftarrow$ empty message
16: superstep $\leftarrow$ superstep + 1
17: $\forall r \in \mathcal{R}$ wait for *responses* or *suspicions*
18: **for all** $p \in \mathcal{P}$ **do**
19: **if** $(\exists w \in \mathcal{R}[p] \mid suspicions[w] = True)$ **or** $(\exists w, w' \in \mathcal{R}[p] \mid resp[w].hash \neq resp[w'].hash)$ **then**
20: checkpoint $\leftarrow$ lastCheckpoint()
21: superstep $\leftarrow$ checkpoint.superstep
22: msg.addCmd(RESTORE_CHECKPOINT, checkpoint)
23: $\mathcal{F}[p] \leftarrow \mathcal{F}[p] + 1$
24: **if** $(\mathcal{F}[p] > f_{max})$ **or** $(\exists w \in \mathcal{R}[p] \mid suspicions[w] = True)$ **then**
25: removed $\leftarrow \mathcal{R}[p]$
26: $\mathcal{R} \leftarrow \mathcal{R} - \{removed\}$
27: **if** $|\mathcal{N}| > f + 1$ **then**
28: $\mathcal{R}[p] \leftarrow \{w \in \mathcal{N} \mid w_1 ... w_{f+1}\}$
29: $\mathcal{N} \leftarrow \mathcal{N} - \mathcal{R}[p]$
30: $\mathcal{F}[p] \leftarrow 0$
31: msg.addCmd(PARTITION, $\mathcal{R}[p]$)
32: **else**
33: msg.addCmd(RESTORE_REPLICS, removed)
34: **end function**

4.4.2 GWorker

O algoritmo 2 é o executado pelos GWorkers. Trata-se de um algoritmo mais simples, que apenas espera mensagens do GMaster e executa os comandos recebidos na ordem descrita no algoritmo. É importante lembrar que uma mensagem enviada pelo GMaster pode conter mais de um comando. Portanto, mais de uma operação pode ser realizada em cada laço. O algoritmo continua sua execução enquanto o GMaster estiver ativo e, ao final, apenas um dos GWorkers de cada conjunto de réplicas escreve seu resultado para o sistema de arquivos distribuído.

Algoritmo 2 GWorker

```
 1: function GWorker(master)
 2:     while master.active() do
 3:         msg ← waitMessage(master)
 4:         resp ← empty response
 5:         if msg.PARTITION then
 6:             partition ← msg.partition()
 7:         if msg.CREATE_CHECKPOINT then
 8:             checkpoint ← partition.createCheckpoint()
 9:             resp.addCheckpointHash(checkpoint.computeHash())
10:         if msg.RESTORE_CHECKPOINT then
11:             partition.restoreFromLocalOrReplica(msg.superstep,
    msg.checkpoint.hashes)
12:             if msg.RESTORE_REPLICAS then
13:                 partition.restoreCheckpointPartition(msg.replicas,
    msg.checkpoint.hashes)
14:         if msg.START_SUPERSTEP then
15:             for vertex ∈ partition do
16:                 vertex.execute(msg.superstep)
17:             resp.addSuperstepHash(partition.computeHash())
18:         send(resp, master)
19:     if not partition.isReplica then
20:         partition.writeResult()
21: end function
```

4.5 CORRETUDE

Nesta seção são descritas as características dos algoritmos do Greft que garantem que, mesmo na presença de um número limitado de faltas, a computação termina em algum momento e seu resultado é correto.

Inicialmente, são descritos os seguintes Axiomas dos algoritmos:

Axioma 1. Existe ao menos $f + 1$ réplicas de cada vértice do grafo de entrada para uma computação, distribuídos de tal forma que cada conjunto de $f + 1$ GWorkers seja responsável pelos mesmos vértices.

Axioma 2. Existe ao menos uma réplica correta de cada vértice do grafo durante a computação de cada *superstep*.

Axioma 3. É improvável que ocorram faltas em f réplicas de um vértice de forma que seu estado fique igual nestas réplicas.

Axioma 4. Um *superstep* só inicia após o término do anterior em todos os GWorkers.

Axioma 5. Uma computação leva um número finito de *supersteps* para terminar.

Axioma 6. Os *checkpoints* são criados pelos GWorkers no início de um *superstep*, antes de realizar a computação propriamente dita deste *superstep*.

Axioma 7. GWorkers enviam periodicamente ao GMaster mensagens com o andamento da computação do *superstep* atual.

Axioma 8. GMaster mantém uma lista de GWorkers suspeitos de terem parado de funcionar durante cada *superstep*.

Axioma 9. Ao final de um *superstep*, cada GWorker envia um resumo criptográfico dos estados dos vértices sob sua responsabilidade para o GMaster.

A partir destes Axiomas são apresentados os seguintes Lemas e Teoremas:

Lema 1. Um GWorker que tenha falhado por parada será incluído, em algum momento, na lista de GWorkers suspeitos de parada mantida pelo GMaster.

Prova: De acordo com o Axioma 7, GMaster é capaz de conhecer o andamento do processamento de um *superstep* em cada GWorker. Caso um GWorker pare de funcionar, ele também não enviará mais estas mensagens de andamento. Caso um GWorker não envie estas mensagens por um determinado número de períodos, GMaster pode assumir que tal GWorker falhou e incluí-lo na lista de suspeitos.

Lema 2. GMaster é capaz de determinar quando um *superstep* terminou.

Prova: De acordo com o Axioma 9, GMaster pode facilmente observar a finalização de um *superstep* em GWorkers que estão funcionando e enviaram a mensagem correspondente. Já no caso de GWorkers que tenham parado de funcionar, de acordo com o Lema 1, eles serão incluídos na lista de suspeitos em algum momento. Quando todos

os GWorkers tiverem enviado a mensagem de finalização ou forem incluídos na lista de suspeitos, GMaster pode determinar que o *superstep* atual terminou.

Lema 3: GMaster é capaz de determinar se ocorreram faltas durante a execução de um *superstep*.

Prova: De acordo com o Lema 2, GMaster é capaz de determinar quando a execução de um *superstep* terminou em todos os GWorkers. Os Axiomas 1, 2 e 3 garantem que haverá $f + 1$ réplicas de cada vértice do grafo de entrada, das quais ao menos uma será correta e nenhuma das f restantes, quando faltosas, terão um estado igual. Desta forma, as réplicas de um vértice só serão corretas se estiverem com o mesmo estado. Como esses vértices estão agrupados em conjuntos de $f + 1$ GWorkers, os resumos criptográficos calculados por eles ao final de cada *superstep* (Axioma 9) serão iguais se e somente se todos os vértices sob sua responsabilidade estejam corretos. Assumindo um resumo nulo para GWorkers presentes na lista de suspeitos, GMaster pode determinar que ocorreu alguma falta no processamento de um *superstep* quando um conjunto de réplicas apresentar diferenças em seus resumos criptográficos.

Lema 4: Um novo *superstep* só inicia se o sistema estiver num estado correto.

Prova: De acordo com o Axioma 4, um novo *superstep* só inicia quando o anterior terminar. O Lema 2 garante que GMaster é capaz de detectar quando um *superstep* terminou e o Lema 3 garante que GMaster é capaz de determinar se ocorreram faltas na execução do *superstep* anterior. Com isso, é possível ao GMaster iniciar um novo *superstep* apenas quando o anterior terminou e o sistema está num estado correto.

Lema 5: Um *checkpoint* sempre contém um estado correto de um GWorker.

Prova: Os Axiomas 4 e 6 garantem que um *checkpoint* é criado no intervalo após a finalização de um *superstep* em todos os GWorkers, mas antes do início da computação do próximo *superstep* em cada GWorker. Como o Lema 4 garante que um *superstep* só inicia se o sistema estiver correto, é garantido que um *checkpoint* criado neste momento contém um estado correto de cada GWorker.

Lema 6. A restauração de um *checkpoint* coloca o sistema em um estado correto.

Prova: De acordo com o Lema 5, cada *checkpoint* contém um estado correto de um GWorker. É garantido, portanto, que a restauração de todos os *checkpoints* recoloca o sistema num estado correto. Essa

garantia permanece verdadeira mesmo nos casos em que réplicas são removidas do sistema e seus *checkpoints* são distribuídos entre outros GWorkers. Portanto, após restaurar um *checkpoint* é possível iniciar um novo *superstep* e preservar as garantias do Lema 4.

Teorema 1. Uma computação termina em algum momento.

Prova: O Axioma 5 garante que haverá um número finito de *supersteps* a executar para que a computação termine. Já o Axioma 4 diz que um novo *superstep* só pode iniciar após o anterior terminar. O Lema 2 garante que, mesmo na presença de faltas, é possível determinar quando um *superstep* terminou, permitindo ao sistema avançar ao próximo. Em conjunto com o Lema 6, que garante que é possível recolocar o sistema num estado correto caso algum problema ocorra, o Lema 2 garante que novos *supersteps* podem iniciar. Com isso, garante-se que *supersteps* iniciem e terminem, de forma que o sistema é capaz de avançar e, em algum momento, executar todos os *supersteps* necessários para completar a computação.

Teorema 2. O resultado de uma computação será sempre correto.

Prova: Para chegar a um resultado correto, o sistema precisa se manter num estado correto até o fim da execução dos *supersteps*. O Axioma 5 garante que há um limite de *supersteps* a executar e o Lema 3 garante que o sistema é capaz de detectar faltas que comprometam o estado do sistema. Em conjunto com o Lema 6, que garante que é possível recolocar o sistema num estado correto após um problema, garante-se que é possível manter seu estado correto até o fim da computação.

4.6 PROTÓTIPO

O protótipo do Greft foi implementado sobre a versão inicial do GPS disponibilizada pelos autores.[2] O GPS foi escrito em Java, então são descritas as alterações realizadas por classes. Nenhuma alteração foi realizada no HDFS, dado que já existe uma versão tolerante a faltas Bizantinas. Além da implementação dos algoritmos descritos anteriormente, também foi necessária a implementação do armazenamento e restauração de *checkpoints*, uma vez que os mecanismos de tolerância a faltas não foram disponibilizados junto com a versão inicial do GPS.

A classe *GMaster* foi criada a partir da *GPSMaster*, original-

[2] Obtida em 23/09/2014, no endereço https://subversion.assembla.com/svn/phd-projects/gps/trunk/

mente disponível no GPS. Nesta classe foram incluídos os parâmetros f, f_max e *spc*. A rotina de particionamento do grafo foi alterada para replicar $f+1$ vezes os vértices do arquivo de entrada em GWorkers diferentes. A mensagem de início de um *superstep* foi alterada para incluir os comandos de criação e restauração de *checkpoint*. No laço de controle dos *supersteps*, após o recebimento de todas as mensagens de finalização do *superstep*, foi incluída a comparação dos resumos criptográficos recebidos dos processos. Caso diferenças sejam encontradas, é sinalizada a restauração do último *checkpoint* na mensagem que será enviada para iniciar o próximo *superstep*. Resumos dos arquivos de *checkpoint* de cada processo e sua réplica são incluídos nessa mensagem. Adicionalmente, caso um conjunto de processos tenha ultrapassado o número f_max de diferenças detectadas ou algum processo tenha falhado por parada, eles são removidos. Se existirem máquinas disponíveis, elas são usadas para restaurar os vértices dos processos que foram removidos. Senão, informações sobre os *checkpoints* dos processos faltosos são incluídas nas mensagens enviadas aos processos restantes.

A classe *GWorker* foi criada a partir da *GPSWorker* original. O laço de controle de *supersteps* foi alterado com a inclusão das rotinas de criação e restauração de *checkpoints* e de geração de resumo criptográfico do estado dos vértices. Ao receber a mensagem de início de um *superstep*, o GWorker gera ou restaura um *checkpoint* conforme definido na mensagem. Já a geração do resumo criptográfico é feita após o processamento de todos os vértices num *superstep*. O algoritmo SHA-1 é executado sobre o estado de todos os vértices, que estão ordenados numericamente pelo seu identificador. Isso garante que um processo e sua réplica cheguem ao mesmo resumo caso o estado de seus vértices esteja igual. Esse resumo e o de um eventual *checkpoint* são adicionados à mensagem de finalização de *superstep* enviada ao GMaster.

Para as rotinas de criação e restauração de *checkpoints*, foram criadas as classes *CheckpointWriter* e *CheckpointReader*. Elas fazem isso de maneira eficiente, armazenando apenas os dados necessários usando uma codificação binária. A classe *CheckpointReader* obtém o arquivo localmente ou de uma réplica do processo, caso ele esteja corrompido ou ausente. No caso de restauração de *checkpoint* de processos removidos, os dados são carregados para um local temporário e a rotina de distribuição dos vértices usadas no particionamento inicial do grafo é invocada, distribuindo esses vértices entre os processos restantes.

4.7 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados a arquitetura, o modelo do sistema, os algoritmos e informações sobre a implementação do protótipo do Greft.

De acordo com os objetivos deste trabalho, foi apresentado um sistema de processamento de grafos de larga escala capaz de tolerar faltas arbitrárias acidentais de maneira eficiente. Os detalhes do sistema são apresentados de maneira teórica, incluindo uma análise de corretude de seus algoritmos. No próximo capítulo é apresentada uma avaliação experimental do protótipo implementado.

5 AVALIAÇÃO EXPERIMENTAL

Neste capítulo são apresentados os resultados de experimentos realizados com o Greft. O propósito destes experimentos é demonstrar que o Greft atende aos objetivos do trabalho, descritos na seção 1.3, e é uma alternativa viável a aplicações de processamento distribuído de grafos que necessitem de garantias mais amplas a tolerância a faltas.

Especificamente, os experimentos foram desenhados para avaliar as seguintes questões:

- Eficiência: qual o custo adicional do algoritmo tolerante a faltas arbitrárias acidentais?
- Gerenciamento de *checkpoints*: qual o ganho de desempenho obtido com o armazenamento dos *checkpoints* em disco local?
- Recuperação de faltas: qual o *overhead* introduzido pela recuperação de uma falta?

5.1 CENÁRIO

Embora não exista um *benchmark* específico para processamento de grafos, há diversos *datasets* com grafos reais disponíveis. Nos testes foi usado um *dataset* extraído da rede social Twitter, contendo um grafo onde vértices representam usuários e arestas os seus seguidores na rede (KWAK et al., 2010). Este grafo tem aproximadamente 41,7 milhões de vértices e 1,47 bilhão de arestas. Os experimentos foram executados na nuvem computacional Amazon Web Services (AWS), usando 17 instâncias (máquinas virtuais) tipo *r3.large* do serviço Elastic Compute Cloud (EC2). Este tipo de instância possui 15 GB de RAM, 32 GB de armazenamento em *Solid State Disk* (SSD) e 2 CPUs virtuais de processadores Intel Xeon E5-2670 v2. Todas as instâncias estavam na mesma zona de disponibilidade (*availability zone*) e usavam o Ubuntu Server 14.04 LTS. Como o GPS e o Greft precisam que todo o grafo e as mensagens trocadas fiquem em memória, esse tipo de instância foi escolhido por ter a melhor relação de custo/benefício entre poder de processamento e memória disponível. Além disso, é no próprio ambiente AWS ou similares que aplicações reais de processamento de larga escala são executadas, incluindo processamento de grafos.

Para realizar os experimentos, foi necessário selecionar algoritmos para a função definida pelo usuário que é executada pelo sistema

sobre o grafo de entrada. Foram selecionados três algoritmos, que foram executados sobre o grafo nos experimentos. De acordo com o modelo do Pregel, os algoritmos recebem como argumento um vértice v do grafo de entrada $\mathcal{G}$. O vértice contém todas as informações descritas no capítulo 4.1. Os algoritmos também podem acessar valores do conjunto de argumentos opcionais $\mathcal{A}$, que é disponibilizado como uma variável global. Além dos valores recebidos na requisição feita pelo usuário, o GWorker também inclui valores do ambiente de execução nesta variável, como o número do *superstep* atual ($current\_superstep$) e o número de vértices do grafo $\mathcal{G}$ ($num\_of\_vertices$). As características de cada algoritmo são descritas a seguir:

- **PageRank**: ranqueia de maneira recursiva os vértices do grafo com base no seu grau de entrada (PAGE et al., 1999). É caracterizado por um alto número de mensagens trocadas pelos vértices durante todo o processamento e por um número fixo de *supersteps*. Seu funcionamento é descrito no algoritmo 3. Este algoritmo necessita do argumento $max\_iterations$, que indica o número de iterações a realizar e deve ser informado pelo usuário na requisição da computação.

- ***Single Source Shortest Path (SSSP)***: assumindo os pesos das arestas como distâncias entre os vértices, o algoritmo calcula a menor distância de um vértice de origem a todos os outros vértices do grafo (MALEWICZ et al., 2010). É caracterizado por uma redução do número de vértices ativos com o andamento do processamento, o que faz que o volume de mensagens trocadas entre os vértices e o processamento propriamente dito diminuam com o tempo. O número de *supersteps* necessários para completar a computação varia de acordo com a estrutura do grafo. No grafo usado nos experimentos foram necessários 17 *supersteps* para completar. Seu funcionamento é descrito no algoritmo 4.

- ***Weakly Connected Components***: encontra as componentes fracamente conexas de um grafo direcionado - ou seja, encontra os maiores subgrafos possíveis em que cada par de vértices é mutualmente alcançável se removida a direcionalidade das arestas (SALIHOGLU; WIDOM, 2013). Como o SSSP, é caracterizado por um número variável de vértices ativos durante o processamento. O número de *supersteps* necessários para completar uma computação também depende do grafo usado. Nos experimentos, foram necessários 47 *supersteps*. Seu funcionamento e descrito no algoritmo 5.

Algoritmo 3 PageRank

```
1: function PageRank(v)
2:     if superstep == 1 then
3:         rank ← 1 / A.num_of_vertices
4:     else
5:         sum ← 0
6:         count ← 0
7:         for msg in v.messages do
8:             sum ← sum + msg.value
9:             count ← count + 1
10:        rank ← (0.85 × sum ÷ count) + (0.15 ÷ A.num_of_vertices)
11:    v.value ← rank
12:    for e in v.edges do
13:        sendMessage(e.vertexId, rank)
14:    if A.current_superstep == A.max_iterations then
15:        v.active ← false
16: end function
```

Algoritmo 4 Single Source Shortest Path

```
1: function SSSP(v)
2:     minDist ← origem(v.id) ? 0 : +∞
3:     for msg in v.messages do
4:         minDist ← min(msg.value, minDist)
5:     if minDist < v.value then
6:         v.value ← minDist
7:         for e in v.edges do
8:             valueMsg gets v.value + e.weight
9:             sendMessage(e.vertexId, valueMsg)
10:    v.active ← false
11: end function
```

Algoritmo 5 Weakly Connected Components

```
1: function WCC(v)
2:     if A.superstep == 1 then
3:         minValue ← v.id
4:         v.value ← +∞
5:     else
6:         minValue ← v.value
7:     for msg in v.messages do
8:         minValue = min(msg.value, minValue)
9:     if minValue < v.value then
10:        v.value ← minValue
11:        for e in v.edges do
12:            sendMessage(e.vertexId, minValue)
13:    v.active ← false
14: end function
```

Os algoritmos escolhidos para os experimentos não costumam ser usados diretamente para resolver problemas do mundo real. No entanto, são úteis como experimentos pois cada um deles captura características comuns em algoritmos em grafos. O PageRank é um exemplo de pior caso em termos de processamento e troca de mensagens, pois todos os vértices ficam ativos durante a computação e enviam mensagens a todos os vértices adjacentes em cada *superstep*. O SSSP é um exemplo de algoritmo cujo desempenho é afetado pelo diâmetro do grafo, ou seja, a distância entre os dois pares de vértices mais distantes do grafo. O WCC, ao agrupar os vértices fracamente conexos, simula o funcionamento de um algoritmo de clusterização.

Nos experimentos, os algoritmos foram parametrizados com $f = 1$, pois esse é o valor usualmente assumido em testes de sistemas tolerantes a faltas Bizantinas e porque é pouco provável ocorrer um número de faltas superior a esse num passo de processamento. Além disso, apenas faltas transientes foram simuladas, pois a implementação da remoção de uma réplica do sistema não estava completa na versão disponível do GPS original. O parâmetro *spc*, que define o intervalo de *supersteps* para geração de um *checkpoint*, foi definido conforme as características de cada algoritmo e ficou em 8 para o PageRank, 6 para o SSP e 10 para o WCC. Levou-se em consideração apenas o tempo de execução dos *supersteps* de cada algoritmo, sem considerar o tempo para particionar o grafo entre as réplicas no início e de armazenamento do resultado no final do algoritmo. Embora o particionamento inicial do grafo seja afetado pela implementação do novo algoritmo, o fator dominante é a forma como os arquivos de entrada estão distribuídos no HDFS, cuja otimização foge do escopo desse trabalho. Já a escrita final do resultado não foi afetada pelas implementações, pois apenas uma réplica armazena o resultado. Os valores exibidos são a média de 3 execuções de cada algoritmo. Foram realizadas apenas 3 execuções, pois observou-se uma dispersão muito pequena dos resultados.

5.2 EXPERIMENTOS

5.2.1 Eficiência

Um dos objetivos do Greft é ser capaz de tolerar faltas arbitrárias acidentais de maneira eficiente. Esta eficiência é expressa principalmente no número de réplicas necessárias para alcançar este objetivo. No capítulo 4 descreve-se que o Greft necessita de $f + 1$ réplicas para

tolerar f faltas. Este é um número menor que o comumente necessário para tolerância de faltas Bizantinas, que é de $2f + 1$ ou $3f + 1$ réplicas. No entanto, é necessário avaliar se os mecanismos de verificação de faltas e comparação de *hashes* descritos nos algoritmos mantêm essa eficiência.

Para avaliar esta característica, os três algoritmos foram executados sobre o grafo do Twitter, tanto no modo original do GPS quanto no Greft, para comparação do desempenho. Os tempos médios de execução (e desvios-padrão) para o PageRank no GPS e Greft foram, respectivamente, 1849 (± 62) e 4272 (± 119) segundos; para o SSSP, 233 (± 13) e 411 (± 21) segundos; para o WCC, 342 (± 11) e 568 (± 17) segundos. Os resultados para o PageRank são exibidos na figura 15. Já os resultados para o SSSP e WCC na figura 16. Os resultados estão em figuras separadas pois o PageRank leva muito mais tempo que os outros dois algoritmos.

Embora os números absolutos sejam interessantes, pode ser difícil perceber neles a diferença real de desempenho entre o Greft e o GPS. Para facilitar, a figura 17 apresenta a razão do tempo de processamento do Greft e da versão original do GPS (ou seja, tempo do Greft dividido pelo tempo do GPS). Nota-se que a versão experimental leva, em média, duas vezes o tempo da versão oficial, variando conforme o algoritmo. Isso é esperado, já que o Greft tem à disposição metade dos recursos da versão original devido à replicação dos vértices.

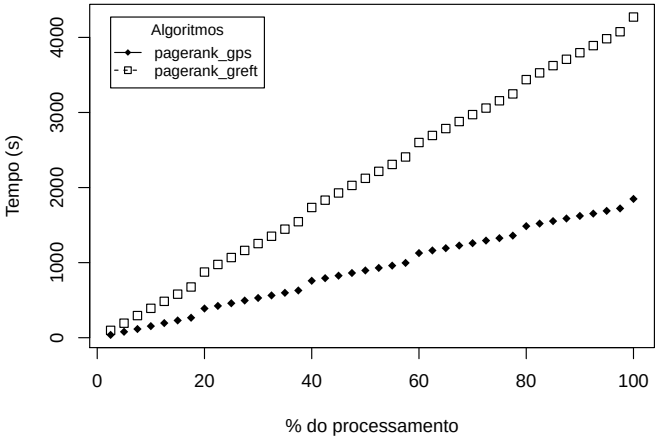


Figura 15 – Tempo de execução do PageRank no GPS original e no Greft

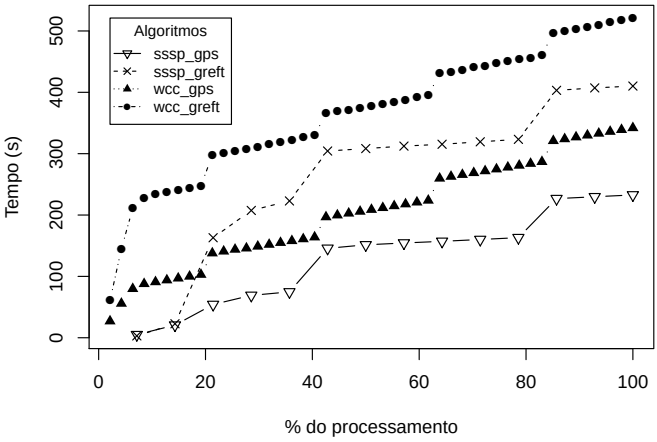


Figura 16 – Tempos de execução do SSSP e WCC no GPS original e no Greft

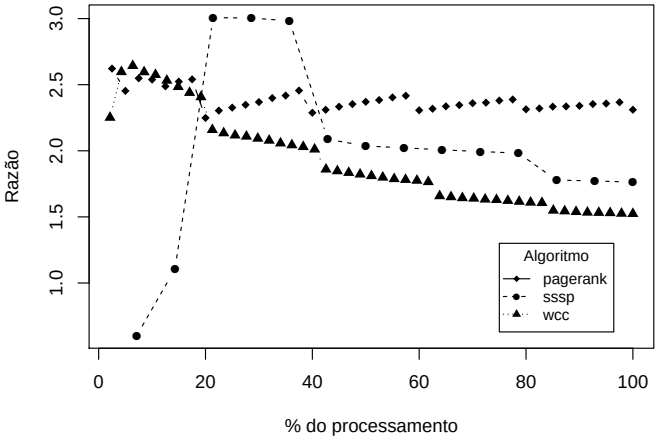


Figura 17 – Razão dos tempos de execução dos algoritmos no GPS original e no Greft

5.2.2 Gerenciamento de *Checkpoints*

Uma das otimizações do novo algoritmo é o armazenamento dos *checkpoints* em disco local. Esperava-se um ganho no desempenho por

se dispensar o uso do HDFS, mesmo considerando que os *checkpoints* no Greft teriam o dobro do tamanho dos do GPS original. Esta suposição foi testada num experimento onde se compara os tempos levados pelo GPS e Greft na tarefa de criação de *checkpoints*. A rotina de criação de *checkpoints* é idêntica nos dois casos, apenas o destino do arquivo é diferente. Esse ganho foi alcançado, como pode ser visto na figura 18. Neste gráfico é exibido o percentual de tempo de processamento gasto em *supersteps* que geram *checkpoints* e em *supersteps* normais. Observa-se que em todos os casos, Greft usa menos tempo na tarefa de criar *checkpoints* que o GPS, destinando mais tempo ao processamento propriamente dito.

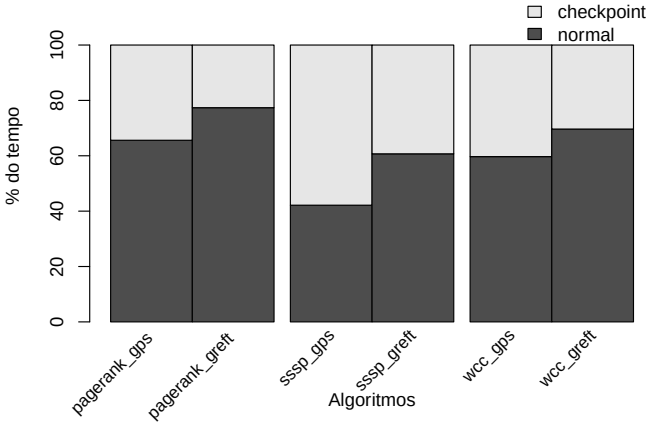


Figura 18 – Percentual de tempo gasto em *supersteps* com e sem geração de *checkpoints*

Esse ganho também pode ser observado na figura 19, que contém a razão do tempo de execução do Greft e GPS em cada *superstep*. Os vales visíveis neste gráfico ocorrem nos *supersteps* onde há geração de um *checkpoint* e demonstram que o desempenho do Greft fica próximo do GPS original nestes pontos (ou seja, a razão se aproxima de 1).

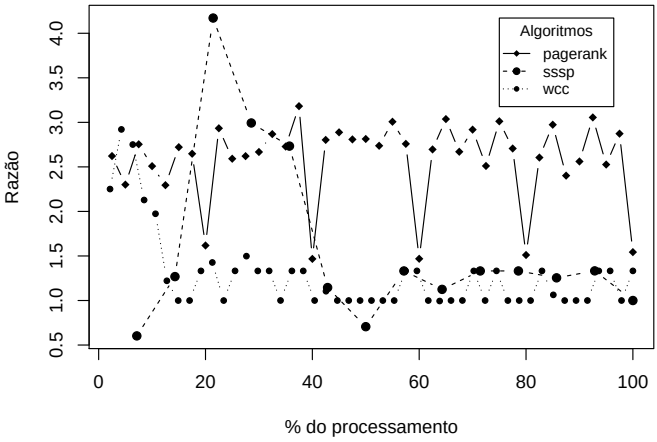


Figura 19 – Razão dos tempos de execução dos *superstep* no Greft e GPS

5.2.3 Recuperação de uma Falta

Outro experimento realizado foi a execução dos algoritmos com a injeção de uma falta transiente, para avaliar o *overhead* introduzido pela rotina de recuperação de faltas. Foram simulados os cenários de melhor e pior caso. Assumindo que o fator dominante na restauração de uma falta é o número de *supersteps* que serão reexecutados após a restauração do *checkpoint*, o melhor caso é uma falta que ocorre logo após um *superstep* onde um *checkpoint* foi gerado, pois apenas um *superstep* precisará ser reexecutado. Já o pior caso é o de uma falta que ocorra imediatamente antes da geração de um *checkpoint*. Na figura 20, é possível perceber que nos melhores casos o *overhead* é realmente muito pequeno - limitando-se ao tempo de restauração do estado e de execução de mais um *superstep*. Já no pior caso, há um *overhead* maior, mas também limitado ao número de *supersteps* que precisaram ser reexecutados após a restauração do *checkpoint*.

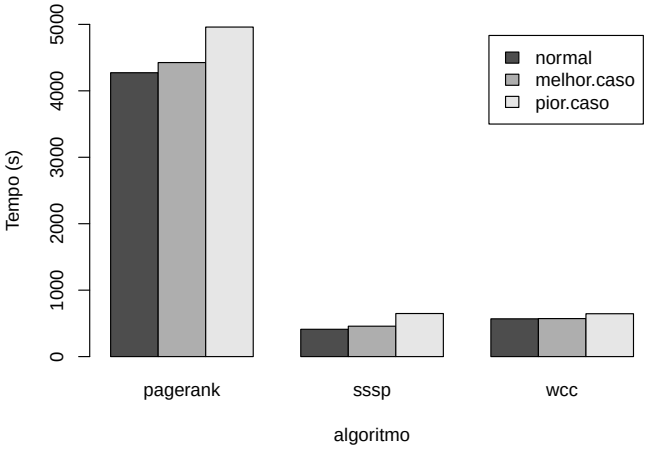


Figura 20 – Tempos de execução dos algoritmos no Greft no sem faltas (normal), melhor caso e pior caso

5.3 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados resultados experimentais que demonstram que Greft atinge os objetivos descritos no capítulo 1. Quando comparado ao GPS original, Greft leva em média o dobro do tempo para realizar uma computação. Isso demonstra que o sistema é eficiente, pois como o processamento é replicado, Greft tem à disposição metade dos recursos do GPS original. Dessa forma, é razoável que leve o dobro do tempo.

O mecanismo de gerenciamento de *checkpoints* introduzido pelo Greft é mais eficiente que o original do GPS, já que Greft gasta proporcionalmente menos tempo criando *checkpoints* que o GPS durante uma computação.

Com relação à recuperação de faltas, ficou demonstrado que há um *overhead*, mas ele é dominado pelo número de *supersteps* que precisam ser reexecutados após a restauração de um *checkpoint*. Essa é uma característica comum aos sistemas que utilizam *checkpoints* para recuperação de faltas.

6 CONCLUSÃO

6.1 REVISÃO DAS MOTIVAÇÕES E OBJETIVOS

Nos últimos anos observou-se uma tendência de crescimento das bases de dados em diversas áreas, incluindo aquelas cujos dados são modelados como grafos. Isso tem atraído a atenção dos pesquisadores e diversos modelos estão sendo propostos para lidar com grafos muito grandes de maneira paralela e distribuída.

O levantamento do estado da arte realizado neste trabalho demonstrou que há uma preocupação com tolerância a faltas nos modelos recentemente propostos de sistemas de processamento distribuído de grafos. Isso é fundamental, já que estes modelos podem ser usados em *clusters* com muitas máquinas para processar grafos muito grandes. Neste tipo de ambiente, é provável que algum componente falhe em algum momento.

No entanto, também observou-se que há lacunas nos modelos propostos no que se refere a tolerância a faltas. Todos os modelos pesquisados suportam apenas faltas simples de parada. Porém, a literatura sugere que estes sistemas também estão expostos a faltas arbitrárias, que são mais sutis e mais difíceis de tratar. A preocupação com esse tipo de falta é justificada pelo uso de sistemas de processamento de grafos em tarefas críticas, como detecção de fraudes em transações comerciais e análise de redes de terroristas por parte de agências de inteligência.

O objetivo geral do trabalho foi apresentar um modelo de processamento distribuído de grafos capaz de lidar com faltas arbitrárias acidentais de maneira eficiente.

Visando atender este objetivo, os seguintes objetivos específicos foram perseguidos:

1. Avaliação do estado da arte em modelos de processamento distribuído de grafos, incluindo os mecanismos de tolerância a faltas destes modelos;

2. Proposição de uma arquitetura, um modelo de sistema e um algoritmo de processamento distribuído de grafos capaz de tolerar faltas arbitrárias acidentais de maneira eficiente através da replicação dos vértices do grafo;

3. Otimização do armazenamento de *checkpoints* para recuperação de faltas, dispensando o uso de sistema de arquivos distribuído;

4. Implementação de um protótipo do modelo e realização de uma avaliação experimental com o uso de grafos reais;

6.2 VISÃO GERAL DO TRABALHO

Nesta seção é feita uma revisão do trabalho realizado e de como foram atendidos os objetivos listados na seção anterior.

O capítulo 1 desta dissertação descreveu o contexto em que o trabalho está inserido, o objetivo geral e os objetivos específicos. Os capítulos 2 e 3 apresentaram a revisão da literatura. O capítulo 2 tratou da fundamentação teórica e trouxe os conceitos utilizados no restante do trabalho. Já o capítulo 3 trouxe os principais modelos de processamento distribuído de grafos existentes na literatura e suas características. O capítulo 4 trouxe a descrição do Greft, um sistema de processamento distribuído de grafos capaz de tolerar faltas arbitrárias acidentais de maneira eficiente. Finalmente, o capítulo 5 trata de uma avaliação experimental do Greft, onde demonstrou-se que o modelo é eficiente e que a otimização no armazenamento de *checkpoints* reduziu significativamente o tempo gasto nesta tarefa.

6.3 CONTRIBUIÇÕES

De acordo com os objetivos definidos para este trabalho, pode-se listar as seguintes contribuições:

1. A criação de um algoritmo tolerante a faltas arbitrárias acidentais para processamento distribuído de grafos;

2. A implementação de um protótipo deste algoritmo com base no GPS, uma implementação do Pregel, incluindo otimizações no gerenciamento dos *checkpoints*;

3. Uma avaliação experimental detalhada do funcionamento e desempenho do protótipo no AWS, utilizando um grafo real para os testes.

Com o objetivo de divulgar os resultados obtidos com o trabalho e, principalmente, submeter seus resultados para uma avaliação da comunidade científica que trata do tema desta dissertação, foram produzidos artigos que foram submetidos a eventos na área de computação distribuída e sistemas de larga escala. As revisões e discussões

destas publicações contribuíram para o amadurecimento do trabalho, incluindo melhorias nos algoritmos e recuperação de faltas. Os artigos publicados são listados a seguir:

1. PRESSER, Daniel; LUNG, Lau Cheuk; CORREIA, Miguel. Tolerância a Faltas Arbitrária em Processamento Distribuído de Grafos. In: XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC 2015, Anais Trilha Principal do SBRC 2015. Vitória, Brasil, 2015. V. Único, p. 305-318. Qualis **EB2**.

2. PRESSER, Daniel; LUNG, Lau Cheuk; CORREIA, Miguel. Greft: Arbitrary fault-tolerant distributed graph processing. In: 2015 IEEE International Congress on Big Data (BigData Congress). New York, USA, 2015. V. Único, p. 452-459. Qualis em análise

A classificação Qualis-CC do evento BigData Congress ainda está em análise pois trata-se de um evento recente, que está apenas na 4ª edição. Entretanto, é um evento em ascensão e concorrido na área de sistemas distribuídos de larga escala, que justifica sua escolha para submissão de artigo.

6.4 LIMITAÇÕES

Embora o trabalho tenha atingido os objetivos desejados, algumas decisões tomadas no seu desenvolvimento trazem limitações à sua utilização. Estas limitações são discutidas a seguir.

O trabalho trata de tolerância a faltas, no entanto, é assumido que o processo GMaster é sempre correto. Isso significa que qualquer falha nesse processo compromete o andamento da computação. Isso se justifica pelo fato que o processamento realizado pelo GMaster é pequeno em relação aos GWorkers, que são intensivos em uso de processador e memória. Além disso, é uma alteração trivial no algoritmo do GMaster permitir que uma computação que tenha parado por falha no GMaster seja manualmente reiniciada a partir de um *checkpoint* existente. Isso não o torna tolerante a faltas, mas facilita uma recuperação manual do sistema.

Uma característica comum aos mecanismos de recuperação de faltas, como o usado pelo Greft, é o fato do processamento não avançar enquanto uma falta estiver presente no sistema. Greft controla essa situação com o uso do parâmetro f_max, que acaba por eliminar os

processos com faltas permanentes ou intermitentes em algum momento. No entanto, isso pode introduzir um *overhead* significativo caso muitas faltas ocorram durante uma computação. No pior caso, para n GWorkers disponíveis, um total de $\frac{n}{f+1} \times f\_max\ checkpoints$ terão que ser restaurados. Modelos de mascaramento de faltas são mais adequados para essas situações, pois avançam mesmo na presença de faltas. Entretanto, estas técnicas demandam um número maior de réplicas, como $2f + 1$ para tolerar f faltas.

O objetivo de ser eficiente também limita a capacidade de Greft detectar as faltas de maneira específica. O que o sistema detecta são faltas de parada ou diferenças entre estados de réplicas. Entretanto, nesse caso não é possível determinar exatamente qual é a réplica faltosa, o que leva ao descarte de conjuntos inteiros de réplicas no caso de muitas faltas. Como Greft é planejado para ambientes de nuvens computacionais, é assumido que descartar instâncias e obter novas não seja custoso. Mas esse pode não ser o caso em outros ambientes.

6.5 TRABALHOS FUTUROS

Podem ser exemplos de trabalhos futuros a exploração de modelos mais amplos de faltas, incluindo faltas Bizantinas considerando comportamento malicioso e tolerância a faltas no processo GMaster. Uma forma de tolerar faltas no GMaster é replicá-lo. Como não há necessidade de ordenação de requisições no Greft, é possível utilizar um total de $2f + 1$ réplicas para tolerar f réplicas faltosas. Os GWorkers, por sua vez, esperariam ao menos $f + 1$ mensagens iguais de GMasters diferentes antes de realizar cada tarefa. Embora o número de réplicas do GMaster seja maior que o dos GWorkers, o impacto no desempenho do sistema não deverá ser significativo, uma vez que o processamento realizado pelo GMaster é pequeno se comparado aos GWorkers. Além disso, um processo GMaster e um GWorker podem rodar na mesma máquina sem prejuízo ao sistema. Junto com a replicação do GMaster também seria necessário utilizar algum mecanismo de chaves para autenticar cada processo do sistema e evitar que mensagens sejam falsificadas. Já o tratamento de comportamento malicioso traz desafios maiores, principalmente no que diz respeito à diversidade e às garantias de sincronia do sistema. Para se alcançar diversidade, duas ou mais implementações do Greft precisariam ser criadas. Já no caso das garantias de sincronia, seria necessário estabelecer mecanismos para evitar que uma réplica maliciosa atrase indefinidamente a execução de

um *superstep*, mas ao mesmo tempo evitar a imposição de um limite para a conclusão de cada *superstep*, já que a ausência desse tipo de limite é observada em todas os modelos de processamento de grafos.

Outra otimização que pode ser feita no algoritmo é permitir a recuperação de uma falta de parada sem a necessidade de restauração de *checkpoint*. Neste caso, é possível usar as réplicas saudáveis de cada vértice para restaurar o estado da réplica faltosa. Isso permitiria aumentar o intervalo de *supersteps* para criação de *checkpoints*, que representaria um ganho de desempenho quando não há faltas ou apenas faltas de parada, em troca de um tempo maior de recuperação no caso de uma falta arbitrária. Essa é uma troca aceitável, já que é razoável supor que faltas arbitrárias sejam mais raras que faltas de parada.

A parte experimental do trabalho também pode ser ampliada, principalmente envolvendo experimentos que tratem de faltas permanentes. Comparações de desempenho variando os valores de f e spc também auxiliariam na definição de valores adequados para esses parâmetros. Novos experimentos podem ser feitos variando o tamanho do grafo de entrada, para demonstrar a escalabilidade do sistema e como ele mantém a eficiência com outros tipos de grafos.

REFERÊNCIAS

Amazon S3 Team. **Amazon S3 availability event: July 20, 2008**. 2008. Disponível em: <See: http://status.aws.amazon.com/s3-20080720html>. Acesso em: 14 dez. 2015.

Amazon Web Services. **Amazon Elastic MaReduce**. 2015. Disponível em: <https://aws.amazon.com/pt/elasticmapreduce/>. Acesso em: 14 dez. 2015.

ARORA, A.; GOUDA, M. Closure and convergence: A foundation of fault-tolerant computing. **Software Engineering, IEEE Transactions on**, IEEE, v. 19, n. 11, p. 1015–1027, 1993.

AVERY, C. Giraph: Large-scale graph processing infrastructure on Hadoop. In: **Proceedings of Hadoop Summit**. Santa Clara: [s.n.], 2011.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, IEEE, v. 1, n. 1, p. 11–33, 2004.

BONDY, J. A.; MURTY, U. S. R. **Graph theory with applications**. London: Macmillan, 1976.

CACHIN, C.; GUERRAOUI, R.; RODRIGUES, L. **Introduction to reliable distributed programming**. Heidelberg: Springer, 2011.

CASTRO, M.; LISKOV, B. Practical Byzantine fault tolerance and proactive recovery. **ACM Transactions on Computer Systems**, ACM, v. 20, n. 4, p. 398–461, 2002.

CHEN, R. et al. Large graph processing in the cloud. In: **Proceedings of the 2010 ACM SIGMOD International Conference on Management of data**. Indiana: ACM, 2010. p. 1123–1126.

CLEMENT, A. et al. UpRight cluster services. In: **Proceedings of the 22nd ACM Symposium on Operating Systems Principles**. New York: ACM, 2009. p. 277–290.

COSTA, P. et al. Byzantine fault-tolerant MapReduce: Faults are not just crashes. In: **Proceedings of the IEEE 3rd International Conference on Cloud Computing Technology and Science**. Athens: IEEE, 2011. p. 32–39.

COULOURIS, G. F.; DOLLIMORE, J.; KINDBERG, T. **Distributed systems: concepts and design**. New Jersey: Pearson Education, 2005.

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

DRISCOLL, K. et al. Byzantine fault tolerance, from theory to reality. In: **Computer Safety, Reliability, and Security**. Edinburgh: Springer, 2003. p. 235–248.

DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. **Journal of the ACM (JACM)**, ACM, v. 35, n. 2, p. 288–323, 1988.

EBERLE, W.; GRAVES, J.; HOLDER, L. Insider threat detection using a graph-based approach. **Journal of Applied Security Research**, Taylor & Francis, v. 6, n. 1, p. 32–81, 2010.

GONZALEZ, J. E. et al. PowerGraph: Distributed graph-parallel computation on natural graphs. In: **Proceedings of the 10th Symposium on Operating System Design and Implementation**. Hollywood: USENIX Association, 2012.

HADZILACOS, V.; TOUEG, S. **A Modular Approach to Fault-Tolerant Broadcasts and Related Problems**. [S.l.], 1994.

HILL, M.; MCCOLL, W.; SKILLICORN, D. Questions and answers about bsp. **Scientific Programming**, v. 6, n. 3, p. 249–274, 1997.

HUNT, P. et al. ZooKeeper: Wait-free coordination for internet-scale systems. In: **USENIX Annual Technical Conference**. Boston: USENIX Association, 2010. v. 8, p. 9.

KAJDANOWICZ, T.; KAZIENKO, P.; INDYK, W. Parallel processing of large graphs. **Future Generation Computer Systems**, Elsevier, v. 32, p. 324–337, 2014.

KWAK, H. et al. What is Twitter, a social network or a news media? In: **Proceedings of the 19th International Conference on World Wide Web**. Raleigh: ACM, 2010. p. 591–600.

LAMPORT, L. Proving the correctness of multiprocess programs. **Software Engineering, IEEE Transactions on**, IEEE, n. 2, p. 125–143, 1977.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The Byzantine generals problem. **ACM Transactions on Programing Languages and Systems**, ACM, v. 4, n. 3, p. 382–401, 1982.

LATORA, V.; MARCHIORI, M. How the science of complex networks can help developing strategies against terrorism. **Chaos, solitons & fractals**, Elsevier, v. 20, n. 1, p. 69–75, 2004.

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

MIRZA, B. J.; KELLER, B. J.; RAMAKRISHNAN, N. Studying recommendation algorithms by graph analysis. **Journal of Intelligent Information Systems**, Springer, v. 20, n. 2, p. 131–160, 2003.

NIGHTINGALE, E. B.; DOUCEUR, J. R.; ORGOVAN, V. Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer pcs. In: **Proceedings of the EuroSys 2011 Conference**. Salzburg: ACM, 2011. p. 343–356.

PAGE, L. et al. The pagerank citation ranking: bringing order to the web. Stanford InfoLab, Stanford, 1999.

SALIHOGLU, S.; WIDOM, J. GPS: A graph processing system. In: **Proceedings of the 25th International Conference on Scientific and Statistical Database Management**. Baltimore: ACM, 2013.

SCHROEDER, B.; GIBSON, G. A. Understanding failures in petascale computers. **Journal of Physics: Conference Series**, IOP Publishing, Boston, v. 78, n. 1, 2007.

SCHROEDER, B.; PINHEIRO, E.; WEBER, W.-D. Dram errors in the wild: A large-scale field study. **SIGMETRICS Perform. Eval. Rev.**, ACM, New York, v. 37, n. 1, p. 193–204, 2009.

SENGUPTA, S. Facebook unveils a new search tool. **NY Times**, 2013.

SEO, S. et al. Hama: An efficient matrix computation with the mapreduce framework. In: **Proceedings of the IEEE 2nd International Conference on Cloud Computing Technology and Science**. Indianapolis: IEEE, 2010. p. 721–726.

SHAO, B.; WANG, H.; LI, Y. Trinity: A distributed graph engine on a memory cloud. In: **Proceedings of the 2013 International Conference on Management of Data**. New York: ACM, 2013. p. 505–516.

SHVACHKO, K. et al. The Hadoop Distributed File System. In: **IEEE 26th Symposium on Mass Storage Systems and Technologies**. Hyatt Regency: IEEE, 2010. p. 1–10.

STEPHEN, J. J.; EUGSTER, P. Assured cloud-based data analysis with ClusterBFT. In: **Middleware 2013**. Beijing: Springer, 2013. p. 82–102.

TANENBAUM, A.; STEEN, M. **Distributed systems**. New Jersey: Prentice-Hall, 2007.

VALIANT, L. G. A bridging model for parallel computation. **Communications of the ACM**, ACM, v. 33, n. 8, p. 103–111, 1990.

VERONESE, G. S. et al. EBAWA: Efficient Byzantine agreement for wide-area networks. In: **IEEE 12th International Symposium on High-Assurance Systems Engineering**. San Jose, CA: IEEE, 2010. p. 10–19.

WANG, P. et al. Replication-based fault-tolerance for large-scale graph processing. In: **Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks**. Atlanta: IEEE, 2014. p. 562–573.

WEST, D. B. et al. **Introduction to graph theory**. New Jersey: Prentice-Hall, 2001.

WHITE, T. **Hadoop: The definitive guide**. Sebastopol: O'Reilly Media, Inc., 2012.

XIN, R. S. et al. GraphX: A resilient distributed graph system on spark. In: **First International Workshop on Graph Data Management Experiences and Systems**. New York: ACM, 2013.

ZAHARIA, M. et al. Spark: cluster computing with working sets. In: **Proceedings of the 2nd USENIX conference on Hot topics in cloud computing**. Boston: USENIX Association, 2010. v. 10, p. 10.