Tolerância a Faltas Arbitrárias em Processamento Distribuído de Grafos

Daniel Presser[1], **Lau Cheuk Lung**[1], **Miguel Correia**[2]

[1]Departamento de Informatica e Estatística – Universidade Federal de Santa Catarina (UFSC)
88040-900 – Trindade – Florianopolis – Santa Catarina – Brasil

[2]Instituto Superior Tecnico, Technical University of Lisbon, INESC-ID – Lisboa, Portugal

Abstract. *Graphs are used to model a large number of real problems in areas such as machine learning and data mining. The increasing datasets 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 paper we present an algorithm and a prototype of such system that can tolerate this kind of fault, based on GPS, an open source implementation of Pregel. Experimental results of the prototype in Amazon AWS are presented, showing that it uses only 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.*

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 artigo é apresentado um algoritmo e protótipo de sistema de processamento de grafos distribuído capaz de lidar com essas faltas, baseado no GPS, uma implementação de código aberto do Pregel. São apresentados também resultados experimentais do protótipo obtidos na Amazon AWS, onde demonstra-se que este algoritmo usa apenas 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.*

1. Introdução

Grafos são extensivamente usados na modelagem e solução de problemas reais em áreas como aprendizado de máquina e mineração de dados. Por isso, algoritmos e modelos de processamento em grafos também enfrentam os desafios impostos pelo crescente tamanho das bases de dados e complexidade das análises realizadas por eles. 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 et al. 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. A implementação do Pregel é proprietária da Google, mas existem implementações de código aberto, como o Apache Giraph [Avery 2011] e o GPS [Salihoglu and Widom 2013].

Por ser tão escalável, o Pregel tolera faltas através 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 os processos restantes. Após isso, o sistema recomeça o processamento do ponto onde parou.

Tolerar problemas como falha de uma máquina ou de disco é fundamental. Entretanto, há problemas conhecidos e mais sutis, que podem afetar a corretude do resultado de um processamento sem causar sua interrupção [Schroeder and Gibson 2007]. Há vários anos um estudo já mostrava a existência de faltas arbitrárias ou Bizantinas em vários sistemas reais [Driscoll et al. 2003]. Mais recentemente, um estudo conduzido por 2 anos e meio nos servidores da Google evidenciou que mais de 8% das DIMMS apresentaram falhas em algum momento [Schroeder et al. 2009]. Outro estudo da Microsoft mostrou que erros em processadores também são frequentes [Nightingale et al. 2011]. Os mecanismos de tolerância a faltas do Pregel e de suas implementações atuais não são capazes de lidar com falhas arbitrárias acidentais, que podem ser causadas por esse tipo de erro em memórias ou processadores.

Para lidar com esse tipo de falta, 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 and 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, que torna seu custo proibitivo no contexto de larga escala. Outra técnica mais eficiente é a que foi utilizada para tornar o *MapReduce* [Dean and 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, um estudo mostrou que o *MapReduce* é até 10 vezes mais lento que modelos como o do Pregel para processamento de grafos [Kajdanowicz et al. 2014], o que inviabiliza seu uso.

Neste artigo é apresentado o *Greft*, um sistema de processamento distribuído de grafos baseado no modelo do Pregel capaz de lidar com faltas arbitrárias acidentais de maneira eficiente. Neste modelo, 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. A verificação da evolução do estado dos vértices é otimizada usando-se resumos criptográficos dos dados das réplicas, reduzindo enormemente o volume de dados trafegados entre os processos. Além disso, foi 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 características do sistema. O sistema de arquivos distribuído é usado apenas para armazenar os arquivos de entrada e os resultados da computação.

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 AWS, utilizando um grafo real para os testes.

2. Pregel e GPS

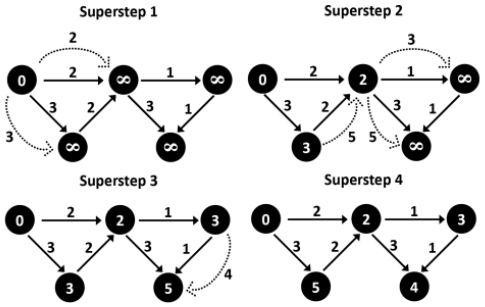
Pregel é um modelo de desenvolvimento e um *framework* que suporta este modelo para processamento distribuído de grafos em larga escala. Neste modelo, cada vértice do grafo possui uma identificação única, um valor, um estado (ativo ou inativo), uma lista de adjacências com ou sem pesos e uma fila de mensagens recebidas. Os vértices são distribuídos entre as máquinas de um *cluster* para realizar o processamento. Cada máquina geralmente processa muitos vértices do grafo.

Os programas são expressos como uma sequência de iterações, chamadas de *supersteps*, nas quais uma função definida pelo usuário é executada em cada vértice do grafo, conceitualmente em paralelo. A função define o comportamento de um único vértice V dentro do *superstep* S. Nela, 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* $S+1$. Com isso, garante-se o paralelismo da execução da função nos vértices dentro de um *superstep* mesmo que, na prática, ela ocorra sequencialmente nas máquinas. Este modelo de processamento em iterações é baseado no modelo *Bulk Synchronous Parallel* [Valiant 1990].

Um sistema de coordenação gerencia a execução dos *supersteps* nas máquinas, garantindo que cada *superstep* só inicie depois que todas as máquinas finalizaram o *superstep* anterior. A execução continua até que todos os vértices estejam inativos e nenhuma mensagem esteja na fila de entrega. Este modelo é suficientemente flexível para permitir a implementação de um grande número de algoritmos em grafos. Além disso, o esquema de sincronia facilita a implementação dos algoritmos e garante que o sistema estará livre de *dead-locks* e condições de corrida - comuns em sistemas assíncronos.

A figura 1 ilustra esse modelo com a execução em um grafo do algoritmo *Single Source Shortest Path* [Malewicz et al. 2010], que encontra o menor caminho de um vértice de origem a todos os outros. No primeiro *superstep*, cada vértice inicia seu valor com $+\infty$, exceto pela origem, que se inicia com zero. Ainda nesse *superstep*, o vértice de origem envia mensagens para seus adjacentes com seu valor atual somado ao peso da aresta que os conecta. Estas mensagens serão recebidas pelos destinatários depois que o próximo *superstep* for iniciado, garantindo o paralelismo da execução do algoritmo em

cada vértice durante o *superstep*. No segundo *superstep*, os vértices adjacentes à origem recebem as mensagens e atualizam seu valor com a menor distância entre elas. Os que atualizaram seu valor então enviam mensagens para os seus adjacentes e o processamento continua até que nenhum vértice tenha mensagens pendentes de recebimento.



Single Source Shortest Path

```
1: minDist ← origem(this.id)  0 : +∞
2: for msg em 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 em this.arestas do
8:         valorMsg ← this.valor + aresta.peso
9:         enviar(aresta.id, valorMsg)
10:    end for
11: end if
12: this.ativo ← FALSE
```

Figure 1. Exemplo de execução do algoritmo num grafo pelo modelo do Pregel

O *Graph Processing System* (GPS) [Salihoglu and 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 tem 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. Sua arquitetura conta com um processo *coordenador*, chamado *GPSMaster* que gerencia a execução dos *supersteps* nos processos, chamados de *GPSWorker*. O *Hadoop Distributed File System* (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. O HDFS é otimizado para o armazenamento de arquivos de grandes dimensões tolerando faltas de parada [Shvachko et al. 2010].

O processo coordenador é o responsável por detectar eventuais falhas nos outros processos através de troca de mensagens periódicas. Caso um processo falhe, o coordenador solicita aos restantes que reiniciem o processamento a partir do último *checkpoint* armazenado. Os vértices que estavam sob a responsabilidade do processo faltoso são redistribuídos entre os restantes e o processamento é reiniciado.

3. Greft - Processamento de Grafos Tolerante a Faltas Arbitrárias

Nesta seção é apresentado a proposta do Greft, um sistema de processamento de grafos tolerante a faltas arbitrárias acidentais baseado no GPS. Inicialmente são apresentadas a arquitetura e o modelo do sistema, depois os algoritmos e as técnicas usadas.

3.1. Arquitetura

O sistema é composto de um conjunto de processos distribuídos. Os clientes solicitam a execução de processamentos em um grafo e aguardam sua conclusão. O coordenador,

chamado *GPSMaster*, recebe a solicitação do cliente e gerencia a distribuição dos vértices do grafo entre os processos. Nessa distribuição, o *GPSMaster* cria duas ou mais réplicas de cada processo atribuindo a eles os mesmos conjuntos de vértices. Os processos sob supervisão do *GPSMaster*, chamados de *GPSWorkers*, recebem os vértices, realizam a computação de cada *superstep*, e armazenam seu estado nos *checkpoints* quando solicitado pelo *GPSMaster*. Além disso, o sistema de arquivos distribuído utilizado (HDFS) também é composto de dois tipos de processos. O *NameNode* é um processo mestre que gerencia o armazenamento dos arquivos do sistema. Já os *DataNodes* são os processos que armazenam dos dados dos arquivos nas máquinas.

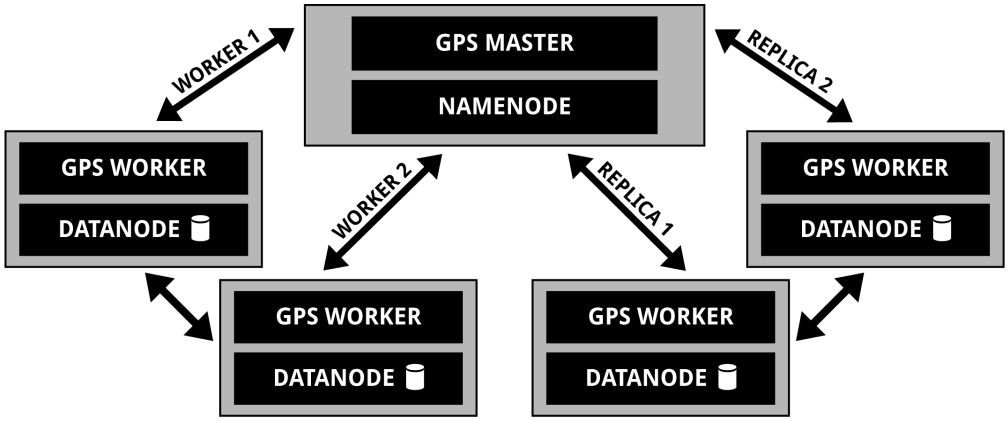


Figure 2. Arquitetura do Greft, baseada no GPS

3.2. Modelo do Sistema

É definido que o processo é correto se ele segue os algoritmos estabelecidos. Caso contrário, trata-se de um processo faltoso. Os processos são executados em servidores em um *datacenter*. O modelo de faltas é o de faltas arbitrárias acidentais, permanentes ou transientes [Avizienis et al. 2004]. Assume-se que um *GPSWorker* 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 processo coordenador 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 tolerando faltas arbitrárias, pois já existe uma versão do HDFS 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 *GPSMaster* nos *GPSWorkers* para detectar eventuais problemas. 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*), por exemplo o SHA-1, resistentes à colisão, isto é, que seja inviável encontrar duas entradas que resultem na mesma saída.

O algoritmo é parametrizado com f. Normalmente este parâmetro indica a quantidade de processos que podem falhar arbitrariamente sem comprometer o funcionamento

do sistema. Neste caso, 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*. O algoritmo também tem os parâmetros f_max, que indica o número máximo de vezes que um conjunto de réplicas pode apresentar divergências antes de ser removido do sistema, e spc, que define o intervalo de *supersteps* entre cada *checkpoint*.

3.3. Algoritmo

Os algoritmos 1 e 2 detalham o funcionamento do *GPSMaster* e GPSWorker. O algoritmo do *GPSMaster* consiste em particionar o grafo de entrada e distribuir cada partição para $f + 1$ *GPSWorkers* diferentes. Cada partição é um subgrafo do grafo original, gerado usando qualquer uma das técnicas de particionamento disponibilizadas pelo GPS. Depois disso, os *supersteps* são iniciados com o envio do comando *START_SUPERSTEP* para os processos. Ao final de cada *superstep*, o coordenador compara o estado de cada vértice com suas réplicas. Caso sejam diferentes, é sinal que o estado de uma das réplicas do vértice foi corrompido durante o processamento e precisa ser recuperado. Para isso, o *GPSMaster* envia o comando *RESTORE_CHECKPOINT* para que os *GPSWorkers* restaurem o último *checkpoint*. Dado que o último *checkpoint* foi feito num momento em que nenhum vértice estava corrompido, ao restaurá-lo todos os vértices voltarão a um estado consistente a partir do qual o processamento pode recomeçar e continuar.

Comparar o estado de cada vértice de um grafo com centenas de milhões de vértices distribuídos em várias máquinas é dispendioso, tanto pelo tempo de comparação quanto pelo volume de dados trafegados até o coordenador. Para evitar isso, cada *GPSWorker* computa um resumo criptográfico dos vértices sob sua responsabilidade ao final de cada *superstep*, e apenas esses resumos são comparados. Isso pode ser visto no algoritmo 2. O que é o estado de cada vértice varia conforme a função definida pelo usuário: pode ser apenas o valor do vértice, ou os pesos das suas arestas, ou ambos.

No caso 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* pode fazer com que o conjunto de *GPSWorkers* nunca dê resultados iguais. Para este caso, é definido o parâmetro f_max como o limite a partir do qual o conjunto de réplicas 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 *GPSMaster*. Neste caso o conjunto de réplicas também é removido do sistema. Quando o GPSMaster remove um conjunto de processos, primeiramente ele verifica se existem máquinas adicionais que possam ser usadas para dar continuidade ao processamento. 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 *GPSWorkers*. Cada conjunto restante, por sua vez, restaura uma partição do arquivo de *checkpoint* do conjunto removido (função *restoreCheckpointPartition* do algoritmo 2), de forma que os vértices presentes no arquivo sejam igualmente distribuídos entre eles e sejam replicados. Em ambos casos o *GPSMaster* também envia o comando *RESTORE_CHECKPOINT* para

que os processos restaurem o último *checkpoint* e recomecem o processamento.

Algorithm 1 GPSMaster (Greft)

```
Require: 𝒲: set of workers, 𝒢: input graph, 𝒩: available nodes
function GPSMaster(f, f_max, spc)
    𝒫 ← partition 𝒢 into ⌊|𝒲|/(f + 1)⌋ subgraphs
    for all p ∈ 𝒫 do
        ℛ[p] ← {w ∈ 𝒲 | w_1...w_{f+1}} //create a set of f + 1 replicas to each partition
        𝒲 ← 𝒲 − ℛ[p] //remove used workers from set
        ℱ[p] ← 0 //initialize faults per partition
    msg ← empty message
    msg.addCommand(PARTITION, ℛ) //send partitions to workers
    superstep ← 1
    while 𝒢.active() do
        msg.addCommand(START_SUPERSTEP, superstep)
        if superstep mod spc == 0 then
            msg.addComand(CREATE_CHECKPOINT, superstep) //create checkpoint if spc interval has passed
        sendToAll(ℛ, msg)
        msg ← empty message
        superstep ← superstep + 1
        ∀r ∈ ℛ wait for responses or suspicions
        for all p ∈ 𝒫 do
            //If there are suspicions of crashes or differences between hashes...
            if (∃w ∈ ℛ[p] | suspicions[w] = True) or (∃w, w' ∈ ℛ[p] | resp[w].hash ≠ resp[w'].hash) then
                checkpoint ← lastCheckpoint() //load last checkpoint info
                superstep ← checkpoint.superstep //replace current superstep number
                msg.addCommand(RESTORE_CHECKPOINT, checkpoint) //add command to restore last checkpoint
                ℱ[p] ← ℱ[p] + 1
                if (ℱ[p] > f_max) or (∃w ∈ ℛ[p] | suspicions[w] = True) then
                    removed ← ℛ[p]
                    ℛ ← ℛ − {removed} //remove replicas from set
                    if |𝒩| > f + 1 then
                        ℛ[p] ← {w ∈ 𝒩 | w_1...w_{f+1}} //if there are available nodes, use them
                        𝒩 ← 𝒩 − ℛ[p] // remove used nodes from set
                        ℱ[p] ← 0
                        msg.addCommand(PARTITION, ℛ[p]) // Send partition only to new replicas
                    else
                        msg.addCommand(RESTORE_REPLICAS, removed) // restore vertices in remaining workers

end function
```

Algorithm 2 GPSWorker (Greft)

```
function GPSWorker(master)
    while master.active() do
        msg ← waitMessage(master)
        resp ← empty response
        if msg.PARTITION then
            partition ← msg.partition()
        if msg.CREATE_CHECKPOINT then
            checkpoint ← partition.createCheckpoint()
            resp.addCheckpointHash(checkpoint.computeHash())
        if msg.RESTORE_CHECKPOINT then
            partition.restoreFromLocalOrReplica(msg.superstep, msg.checkpoint.hashes)
            if msg.RESTORE_REPLICAS then
                partition.restoreCheckpointPartition(msg.replicas, msg.checkpoint.hashes)
        if msg.START_SUPERSTEP then
            for vertex ∈ partition do
                vertex.execute(msg.superstep)
            resp.addSuperstepHash(partition.computeHash())
        send(resp, master)
    if not partition.isReplica then
        partition.writeResult()
```

Com a introdução das réplicas de cada processo não há mais necessidade de ar-

mazenar os *checkpoints* num sistema de arquivos distribuídos. Os *checkpoints* passam a ser armazenados na máquina local e, no caso de falha de uma máquina, seu estado pode ser restaurado a partir do último *checkpoint* armazenado em sua réplica (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 *GPSWorker*, um resumo criptográfico do arquivo é enviado ao *GPSMaster*. Com isso, quando for necessário restaurar *checkpoints* o *GPSMaster* devolve estes resumos para os *GPSWorkers*, que podem verificar a integridade de seus arquivos e solicitar o arquivo de sua réplica no caso de uma divergência. Ao final da execução dos *supersteps*, caso o estado dos vértices esteja consistente, apenas uma das réplicas escreve o resultado final no sistema de arquivos distribuídos.

4. Protótipo

O protótipo do Greft foi implementado sobre a versão inicial do GPS disponibilizada pelos autores.[1] 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 `GPSMaster` foi alterada com a inclusão dos 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 *GPSWorkers* 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.

Na classe `GPSWorker`, 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 `GPSWorker` 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

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

resumo e o de um eventual *checkpoint* são adicionados à mensagem de finalização de *superstep* enviada ao `GPSMaster`.

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.

5. Avaliação Experimental

Nesta seção são discutidos os resultados experimentais da execução do protótipo do Greft num *cluster*. Os experimentos foram planejados para avaliar os seguintes aspectos: (1) Qual o custo adicional do algoritmo tolerante a faltas arbitrárias acidentais? (2) Qual o ganho de desempenho obtido com o armazenamento dos *checkpoints* em disco local? (3) Qual o *overhead* introduzido pela recuperação de uma falta?

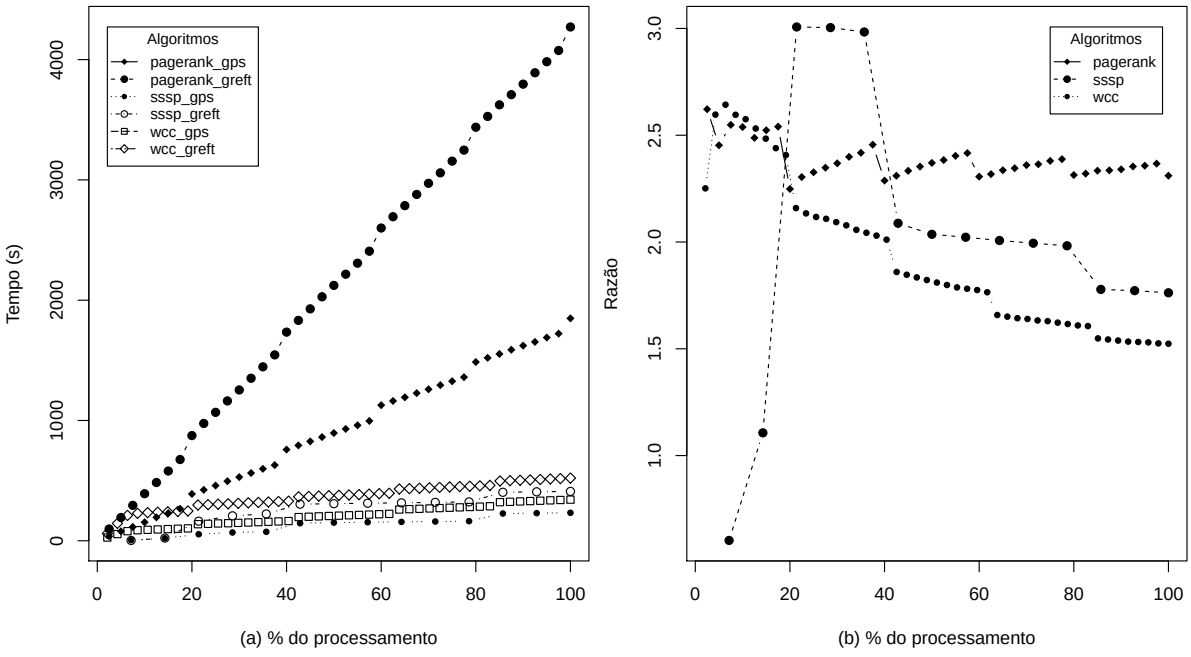


Figure 3. Tempo de execução e razão dos tempos de execução dos algoritmos no GPS original e no Greft

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 numa nuvem computacional Amazon Web Services (AWS), usando 17 instâncias (máquinas virtuais) tipo *r3.large* do serviço Elastic Cloud Computing (EC2). Este tipo de instância

possui 15 GB de RAM, 32 GB de armazenamento em 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.

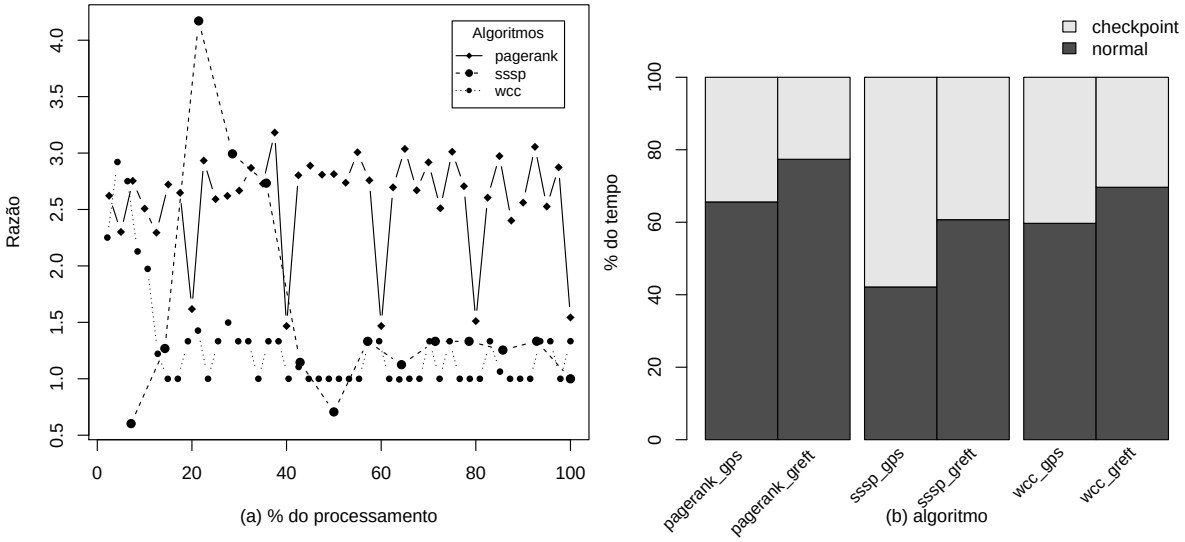


Figure 4. Razão dos tempos de execução dos supersteps e percentuais em supersteps com geração de checkpoints e normais

Três algoritmos foram executados sobre o grafo nos experimentos: *PageRank* - que ranqueia os vértices com base no seu grau de entrada, *Single Source Shortest Path* (SSP) - que calcula a menor distância de um vértice a todos os outros - e *Weak Connected Components* (WCC), que determina as componentes fracamente conexas do grafo. O *PageRank* é caracterizado por um alto número de mensagens trocadas pelos vértices durante todo o processamento e um número fixo de *supersteps* - neste caso, 40. Já o SSP e o WCC têm como característica 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 o algoritmo depende da estrutura do grafo. No grafo usado nos experimentos, o SSP finalizou com de 17 *supersteps*, e o WCC com 47.

Nos experimentos o algoritmo foi parametrizado com $f = 1$, pois esse é o valor usualmente assumido em testes de sistemas tolerantes a faltas Bizantinas e porque é pouco provável ter 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* 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 artigo. 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.

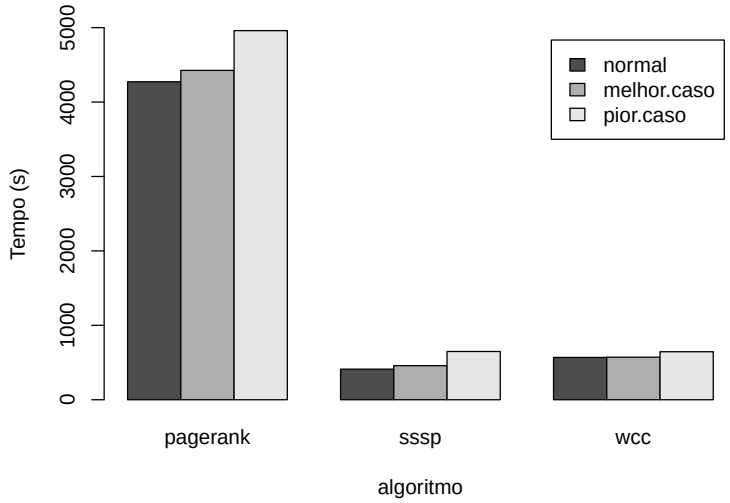


Figure 5. Tempo de execução dos algoritmos no GPS original e no Greft

O primeiro experimento foi a execução dos três algoritmos sobre o grafo do Twitter, tanto no modo original do GPS quanto no modo tolerante a faltas arbitrárias acidentais do Greft, para comparação do desempenho. Os tempos médios de execução (e desvios-padrão) para o PageRank no GPS e Greft foram 1849 ($\pm$ 62) e 4272 ($\pm$ 119) segundos; para o SSSP, 233 ($\pm$ 13) e 411 ($\pm$ 21) segundos; para o WCC, 342 ($\pm$ 11) e 568 ($\pm$ 17) segundos. A figura 3 mostra (a) estes tempos de execução e (b) 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.

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. Esse ganho foi alcançado, como pode ser visto na figura 4. Neste gráfico é exibida (a) a razão do tempo de execução entre o Greft e GPS original de cada *superstep* dos algoritmos e (b) o percentual de tempo gasto em *supersteps* com geração de *checkpoints* e *supersteps* normais. Os vales visíveis no gráfico (a) ocorrem nos *supersteps* onde há geração de um *checkpoint*, e demonstram que o desempenho do Greft fica próximo do GPS original nestes pontos. No gráfico (b) percebe-se que o percentual do tempo gasto em *supersteps* que geram *checkpoints* é menor no Greft.

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 (figura 5). 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, 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*

6. Trabalhos Correlatos

Existe uma vasta literatura sobre tolerância a faltas Bizantinas ou arbitrárias, com modelos propostos desde a década de 80 [Lamport et al. 1982]. Replicação de máquinas de estado é uma técnica genérica para lidar com faltas, que já se demonstrou poder ser implementada de forma eficiente mesmo tolerando faltas Bizantinas [Castro and 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 and Eugster 2013]. Entretanto, como já discutido, estes modelos não são adequados ao processamento distribuído de grafos, principalmente pelo custo envolvido nestas técnicas.

As características dos sistemas mais próximos deste trabalho são brevemente descritas na tabela 1. Pregel [Malewicz et al. 2010], em que é baseado o GPS, é um dos primeiros modelos e introduziu vários dos conceitos usados em outros trabalhos, como a definição do *superstep*. Entretanto, ele tolera apenas faltas de parada (*crash*) através de um mecanismo de recuperação em retrocesso baseado em *checkpoints*. PowerGraph [Gonzalez et al. 2012] e Trinity [Shao et al. 2013] usam mecanismos semelhantes e também toleram apenas faltas de parada. Graphx [Xin et al. 2013] implementa tolerância a faltas de parada estendendo o mecanismo *Resilient Distributed Datasets* (RDD), presente no *framework Spark*, que foi usado para construí-lo. Este mecanismo armazena as alterações realizadas nos *datasets* em vez de armazenar os dados propriamente ditos. Nenhum desses modelos usa replicação para implementar tolerância a faltas.

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

Table 1. Trabalhos relacionados e suas características

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, o *Imitator* se utiliza de uma técnica existente em alguns sistemas de processamento de grafos, 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, o *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.

7. Conclusão

Neste artigo foi apresentado o Greft, um algoritmo e protótipo de um sistema de processamento distribuído de grafos tolerante a faltas arbitrárias acidentais. Também foi apresentada uma avaliação experimental do protótipo, onde se confirmou que ele é capaz de lidar com este tipo de faltas usando, no máximo, duas vezes os recursos do modelo original, usando o parâmetro $f = 1$. Este valor é realista, já que este tipo de falta não acontece com frequência e, quando acontece, tem uma probabilidade muito baixa de deixar duas réplicas com o mesmo estado. As otimizações realizadas no armazenamento dos *checkpoints* também se mostraram satisfatórias, reduzindo significativamente o tempo gasto nesta tarefa quando comparada ao modelo original.

O modelo também se mostrou eficiente ao se recuperar de faltas transientes, adicionando um *overhead* referente apenas à restauração do estado e reexecução dos *supersteps* que já haviam sido processados desde o último *checkpoint*. Isso é um avanço se comparado ao modelo original, onde seria necessário executar o processamento três vezes para chegar ao mesmo resultado no caso de uma falta transiente.

References

Avery, C. (2011). Giraph: Large-scale graph processing infrastructure on Hadoop. In *Proceedings of Hadoop Summit*.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

Castro, M. and Liskov, B. (2002). Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461.

Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riche, T. (2009). UpRight cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 277–290.

Costa, P., Pasin, M., Bessani, A. N., and Correia, M. (2011). Byzantine fault-tolerant MapReduce: Faults are not just crashes. In *Proceedings of the IEEE 3rd International Conference on Cloud Computing Technology and Science*, pages 32–39.

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

Driscoll, K., Hall, B., Sivencrona, H., and Zumsteg, P. (2003). Byzantine fault tolerance, from theory to reality. In *Computer Safety, Reliability, and Security*, pages 235–248. Springer.

Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th Symposium on Operating System Design and Implementation*.

Kajdanowicz, T., Kazienko, P., and Indyk, W. (2014). Parallel processing of large graphs. *Future Generation Computer Systems*, 32:324–337.

Kwak, H., Lee, C., Park, H., and Moon, S. (2010). What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, pages 591–600.

Lamport, L., Shostak, R., and Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146.

Nightingale, E. B., Douceur, J. R., and Orgovan, V. (2011). Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer pcs. In *Proceedings of the EuroSys 2011 Conference*, pages 343–356.

Salihoglu, S. and Widom, J. (2013). GPS: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*.

Schroeder, B. and Gibson, G. A. (2007). Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78.

Schroeder, B., Pinheiro, E., and Weber, W.-D. (2009). DRAM errors in the wild: a large-scale field study. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 193–204.

Shao, B., Wang, H., and Li, Y. (2013). Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 International Conference on Management of Data*, pages 505–516.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The Hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies*, pages 1–10.

Stephen, J. J. and Eugster, P. (2013). Assured cloud-based data analysis with ClusterBFT. In *Middleware 2013*, pages 82–102. Springer.

Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.

Veronese, G. S., Correia, M., Bessani, A. N., and Lung, L. C. (2010). EBAWA: Efficient Byzantine agreement for wide-area networks. In *IEEE 12th International Symposium on High-Assurance Systems Engineering*, pages 10–19.

Wang, P., Zhang, K., Chen, R., Chen, H., and Guan, H. (2014). Replication-based fault-tolerance for large-scale graph processing. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 562–573.

Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2013). GraphX: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*.