Greft: Arbitrary Fault-Tolerant Distributed Graph Processing

Daniel Presser Lau Cheuk Lung
Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)
Florianópolis, Brasil
daniel.presser@posgrad.ufsc.br, lau.lung@ufsc.br

Miguel Correia
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Lisboa, Portugal
miguel.p.correia@tecnico.ulisboa.pt

***Abstract*—Many large-scale computing problems can be modeled as graphs. Example areas include the web, social networks, and biological systems. The increasing sizes of datasets has led to the creation of various distributed large scale graph processing systems, e.g., Google Pregel. Although these systems tolerate crash faults, the literature suggests they are vulnerable to a wider range of accidental arbitrary faults (also called Byzantine faults). In this paper we present an algorithm and a prototype of a distributed large-scale graph processing system that can tolerate arbitrary faults. The prototype is 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 3-4 times as usual in Byzantine fault-tolerant systems. This cost may be acceptable for critical applications that require this level of fault tolerance.**

I. Introduction

Graphs are used to model a large number of real problems in areas such as the web, social networks, and biological systems. The increasing size of datasets and complexity of analysis has brought researchers' attention to this subject. Several *large-scale distributed graph processing engines* were recently proposed, e.g., Pregel [1], PowerGraph [2], Trinity [3], GraphX [4], Mizan [5], Ligra [6], and Giraph++ [7]. Among these, Google's Pregel is the most cited. Pregel is a distributed graph processing system that can run in clusters with thousands of machines and process graphs with billions of vertices. Pregel is a proprietary solution, but there are open source implementations such as Apache Giraph [8], Apache Hama [9] and GPS [10].

Pregel uses a *checkpointing* mechanism to tolerate faults. The state of each process is periodically saved to a reliable global storage, such as a distributed file system. When a machine fails, the remaining machines reload their states from the last checkpoint. The failed machine state is also loaded and distributed among the remaining ones, allowing the computation to restart.

Processes and machines crash so often in large-scale systems that tolerating these faults is essential in graph processing systems. However, there are more subtle faults that may affect process/machine correctness without actually stopping them [11]. These faults are part of the wider class of *accidental arbitrary faults*, also known as *accidental Byzantine faults* [12]. This class of faults is known for long to affect real systems [13]. More recently, a two-year study conducted at Google's data centers revealed that more than 8% of DIMMs are affected by errors per year [14]. Another study from Microsoft showed that errors in processors are relatively frequent as well [15]. The checkpointing mechanism used by Pregel and the other similar systems in the literature cannot cope with accidental arbitrary faults that can be caused by these errors in DIMMs and processors.

Replication is often used to deal with arbitrary faults, masking their effects [16]. An example of such technique is the practical Byzantine fault-tolerant replication algorithm presented in [17]. However, this algorithm is designed for client/server applications, so it is not suited for distributed graph processing systems. Besides, it needs $3f + 1$ replicas to tolerate f faults, thus a minimum of 4 replicas ($f = 1$), which is too expensive for large scale systems. A more efficient model was used to make MapReduce tolerate Byzantine faults [18]. This model replicates $f + 1$ times each map and reduce task to tolerate f faults (minimum 2 replicas). Task results are compared after they finish and if they do not match, new tasks are executed until $f + 1$ matching results are obtained. Despite requiring less replicas, this model is also not suited because graph processing in MapReduce is up to ten times slower than in Pregel-like systems [19].

In this paper we present *Greft*, a distributed graph processing system based on Pregel that can tolerate arbitrary accidental faults in an efficient way.[1] Greft replicates each graph vertex in different machines so that their states can be compared during the computation. Using several techniques, we were able to reduce the number of replicas to $f + 1$ to tolerate f faults. Network traffic is greatly reduced because only a cryptographic hash of each replica state is transmitted and compared to detect arbitrary faults. As computation is replicated, checkpoints can be stored more efficiently in local disks instead of on a distributed file system, which is used to store only the overall computation inputs and outputs.

The main contributions of the paper are: the first large-scale distributed graph processing system that tolerates accidental arbitrary (or Byzantine) faults; a prototype of the system based on GPS, an open source Pregel implementa-

[1] *Greft* was composed freely from the terms graph and fault tolerance.

tion; a detailed experimental evaluation of the prototype at Amazon Web Services using a real graph.

II. Pregel and GPS

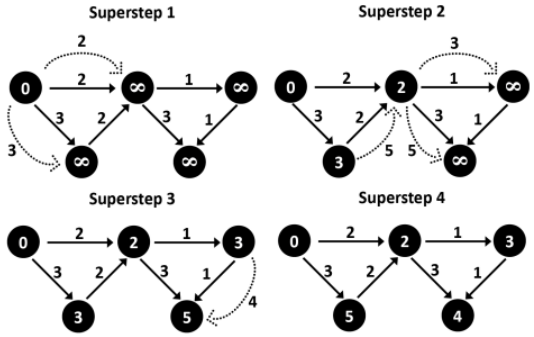
Pregel is a programming model and a framework that supports the model for large-scale distributed graph processing. A graph is composed by *vertices* (or nodes) connected by *edges*. In Pregel, each graph vertex has an unique ID, a value, a state (active or inactive), a message queue and an adjacency list. The adjacency list has the directed edges that connect this vertex to the others, with an user-defined value for each edge. The vertices are broken in a set of *partitions* (sets of vertices) and distributed to machines in a cluster.

A typical Pregel program is expressed as a sequence of iterations, called *supersteps*. In the beginning there is an *input graph*. In each superstep, a user-defined function is called in each vertex of the graph, conceptually all in parallel (only conceptually, as each machine has a partition of the vertices). This function defines the behaviour of vertex V in superstep S. In the function, V can receive messages sent by other vertices in superstep $S-1$, update its value and state and send messages to other vertices that will be received only in superstep $S+1$. This model is inspired in Valiant's Bulk Synchronous Parallel model [20].

A coordination system manages the execution of supersteps in each machine, making sure that each superstep only starts after all machines finished executing the previous. Execution finishes when all vertices are inactive and there are no pending messages in queues. This model is flexible enough to implement a wide range of graph algorithms and ensures that the system is free of dead-locks and data races, common in asynchronous systems [1]. Since Pregel is proprietary, Greft's prototype is based on GPS [10], which is similar but open.

Figure 1 illustrates an execution of the Single Source Shortest Path algorithm in the Pregel/GPS model [1]. The input graph has the origin vertex with zero and the other vertices with $+\infty$. In the first superstep, the origin sends to all its adjacent vertices messages with its value added to the weight of the edge that connects them. These messages are received by the destination vertices in the second superstep. Each vertex then updates its value with the received messages minimum value and sends new messages to their adjacent vertices. Processing continues until a superstep ends with no new messages being sent by any vertex and all vertices being in inactive state. At this point, the value of each vertex will be the shortest path to the origin.

GPS [10] is an open implementation of Pregel that also introduces improvements in graph partitioning and a global computation interface. Improvements in graph partitioning aim to keep vertices that communicate a lot in the same machine to reduce messages sent and improve performance. The global computation interface was introduced to simplify algorithms that would be too complicated to implement with




```
1: function SINGLE SOURCE SHORTEST PATH(vertex)
2:     minDist ← isOrigin(vertex) ? 0 : +∞
3:     for all msg in vertex.messages do
4:         minDist ← min(msg.dist, minDist)
5:     if minDist < vertex.value then
6:         vertex.value ← minDist
7:         for all edge in vertex.edges do
8:             send(edge.id, edge.weight + vertex.value)
9:     vertex.active ← FALSE
10: end function
```

Figure 1. Single Source Shortest Path execution and algorithm in Pregel

a Pregel's vertex-centric view, e.g., clustering algorithms. GPS is composed of a master process called *GPSMaster* and worker processes called *GPSWorkers*. Hadoop Distributed File System (HDFS) is used to store checkpoints and computation inputs and outputs. A checkpoint in GPS has all vertices's data, including their message queue. HDFS is a file system optimized to deal with large files and tolerant to crash faults [21].

III. Greft

This section presents Greft's architecture, system model and algorithms.

A. Architecture and System Model

Greft is composed of a set of distributed processes. Clients request the execution of an algorithm on a dataset and wait the computation to finish. A coordination process, GMaster, receives the request, manages the graph partitioning and distribution among the set of worker processes $\mathcal{W}$ (GWorkers). Each graph partition is replicated in two or more GWorkers in different machines. GWorkers run the user-defined function of each vertex in each superstep and save their states in local checkpoints when requested by the GMaster. HDFS is used only to store the input and output files, and is composed of a NameNode (master process that manages access to the file system), and several DataNodes (store file blocks). Figure 2 shows Gref's architecture.

We assume that the clients and GMaster are always *correct*, i.e., that they run as expected, which is the same assumption done for Pregel, GPS, MapReduce and similar systems. GWorkers can be correct or *faulty*: they can deviate arbitrarily from their algorithm, e.g., by crashing, omitting some messages, or jumping to arbitrary states. We assume

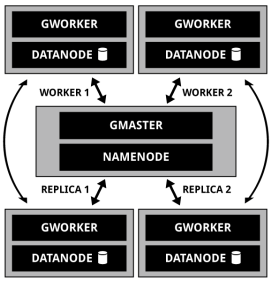


Figure 2. Greft's architecture

that HDFS tolerates arbitrary faults (we used the standard HDFS in the experiments, but a Byzantine fault-tolerant HDFS has been presented in the literature [22]).

We express our assumptions on GWorker faults in terms of parameter f. Usually this parameter means the number of faulty replicas tolerated [17], [22], [23]. In our case, this parameter has a different meaning: given a replica set $\{V_1, V_2, \cdots, V_n\}$ where V_i is a vertex, we assume that f is the maximum number of faulty replicas that finish a superstep in the same incorrect state. The assumption for GWorkers is equivalent as vertex replicas must run in different workers.

The system is asynchronous, i.e., we make no assumptions about bounds on communication delays and processing times, except those needed for a ping-based health-check mechanism between master and workers. We assume that the processes are connected through reliable channels, where no messages are lost, duplicated or corrupted, similarly to those provided by TCP connections. We also assume the existence of a collision-resistant hash function (e.g., SHA-3).

B. GMaster Algorithm

Besides f, the algorithm has two other parameters: f_{max}, the maximum number of faults that can happen in a given set of workers before that set is removed from the system; spc, the number of supersteps between checkpoints.

Figure 3 shows the algorithm executed by GMaster. Initially, the input graph is partitioned in $\mathcal{P}$ partitions. A partition is created for each set of $f+1$ available workers in $\mathcal{W}$ (line 1). Then, each partition is assigned to $f+1$ GWorkers (lines 3 to 6). Therefore, each partition is replicated in $f+1$ workers that should go over the same sequence of states during the computation. By comparing their states, the system can detect arbitrary values. These sets of replicas are stored in Variable $\mathcal{R}$. The array $\mathcal{F}$ counts the faults detected in each set of replicas and is also initiated at this point. Any of the partitioning mechanisms available in GPS can be used to partition the input graph.

The main part of the algorithm is the superstep loop (lines 10-35). In this loop, GMaster defines which operations the workers should perform next and sends messages with commands instructing the workers to do so. Each message sent by the GMaster takes one or more commands, which can be seen in Table I. After sending the message to all GWorkers (line 14), GMaster waits for them to respond or marks them as suspects of crashing (line 17). A process is marked as suspect if it fails to respond to the heartbeat messages sent periodically from GMaster to each GWorker. This mechanism is simple so it is not detailed in the algorithm.

Command	Instructs workers to:
PARTITION	Load a partition from the input graph
START_SUPERSTEP	Start a new superstep
CREATE_CHECKPOINT	Write current state to a checkpoint
RESTORE_CHECKPOINT	Load a checkpoint to replace current state
RESTORE_REPLICAS	Load part of checkpoint from another worker

Table I
COMMANDS SENT IN MESSAGES FROM GMASTER TO GWORKERS

To detect an arbitrary value in the vertices's states, GMaster needs a way to compare the states of each vertex and its replica(s). However, it would be burdensome to transmit every vertex state of each GWorker to the GMaster, as this state may be large. Instead, Greft uses cryptographic hashes of the replica states. In the response received from each GWorker after a superstep there is a hash of all its vertex states. Since the workers are grouped in sets of $f+1$ replicas and each set processes the same graph partition, every vertex of each GWorker of a set of replicas should be in the same state after each superstep. From this follows that a hash of every vertex state computed in a defined order in a GWorker will be equal to the hash computed by its replicas if they are correct. By comparing the hashes of each set of replicas, Greft can detect if any of the replicas has produced arbitrary values, as the collision resistance provided by cryptographic hash functions implies the hashes will be different in that case. If any of the processes is marked as suspect or there are differences in the hashes received from the replicas (lines 18-19), a system recovery is started. If not, processing continues to the next superstep.

To recover from a fault, GMaster first orders the load of the last checkpoint. The current superstep number is replaced with the number of the superstep in which the checkpoint was created, and the RESTORE_CHECKPOINT command is added to the message that will be sent to all GWorkers (lines 20-22). Whenever a checkpoint is created by a GWorker, a hash of that checkpoint is computed and returned to GMaster. These hashes are included in the message so that GWorkers can test if their checkpoints are consistent. If the recovery was motivated by a difference in replica hashes, the system assumes, optimistically, that a *transient fault* [12] occurred. In this case, simply reverting to the last checkpoint puts the system back in a consistent state, allowing the computation to continue and finish correctly.

The case of a *permanent fault* (e.g., some memory bits stuck-at-0 or 1 [24]) is more complicated as it can prevent the computation completion, since the replicas may never produce equal hashes. To deal with this kind of problem the

master counts the faults detected in each set of replicas in the $\mathcal{F}$ array (line 19, 23). If the number of faults detected in a replica set exceeds f_{max} or there is one crashed, the set of workers is removed from the system (lines 24-25). If there are additional machines available in $\mathcal{N}$, they are used to replace those removed. To do so, the PARTITION command is added to the message that will be sent to them, instructing them to load the removed machine's partition (lines 27-31). If not, the RESTORE_REPLICAS command is added to the message, so that the removed replicas partition can be redistributed among the remaining GWorkers.

The superstep loop continues while there are active vertices in the graph and/or pending messages to be delivered, i.e., while the graph is active.

C. GWorker Algorithm

Figure 4 presents the algorithm executed by the GWorkers. The algorithm is composed of a loop that is executed while GMaster is active. In this loop, GWorker waits for messages from GMaster and, when received, executes the commands in the message. Each message can have more than one command, and they are executed in the sequence presented in the algorithm.

When a PARTITION command is received, the worker loads the indicated graph partition from the input graph (lines 5-6). The way a partition is loaded is abstracted in the algorithm by the *loadPartition* function, as it depends on which partitioning mechanism is used and the way the graph is stored. If the CREATE_CHECKPOINT command is received, GWorker creates a checkpoint of its partition and stores it in the local disk (lines 7-9). Because there is at least one replica of each GWorker, there is no need to store checkpoints in a distributed file system to tolerate faults (unlike HDFS and GFS). If a GWorker fails, its state can be recovered from its replica(s). A hash of the checkpoint is computed and added to the response sent to the GMaster.

If RESTORE_CHECKPOINT is received, GWorker replaces its state with the state loaded from the last checkpoint (lines 10-13). The checkpoint can be loaded from local disk or from a replica, if the local checkpoint is absent or inconsistent (done by function *partition.restoreChkpt*). Checkpoint consistency is checked using hashes computed when the checkpoint was created, that are received along with the RESTORE_CHECKPOINT command from GMaster. This procedure restores the GWorker to the state it was when the checkpoint was created and is used to recover from faults. It can be used jointly with the PARTITION command when new nodes are added to the system: first the partition is loaded, then the checkpoint is restored, leaving the GWorker ready to continue. As for the RESTORE_REPLICAS command, it is used when a set of replicas was removed from the system and there are no available machines to replace them. In this case, the removed replicas graph partition and state has to be redistributed among the remaining workers.

```
Require: 𝒲: set of workers, 𝒢: input graph, 𝒩: available nodes
 1: function GMaster(f, f_max, spc)
 2:     𝒫 ← partition 𝒢 into ⌊|𝒲|/(f + 1)⌋ subgraphs
 3:     for all p ∈ 𝒫 do
 4:         ℛ[p] ← {w ∈ 𝒲 | w_1...w_{f+1}}
 5:         𝒲 ← 𝒲 − ℛ[p]
 6:         ℱ[p] ← 0
 7:     msg ← empty message
 8:     msg.addCmd(PARTITION, ℛ)
 9:     superstep ← 1
10:     while 𝒢.active() do
11:         msg.addCmd(START_SUPERSTEP, superstep)
12:         if superstep mod spc == 0 then
13:             msg.addCmd(CREATE_CHECKPOINT, superstep)
14:         sendToAll(ℛ, msg)
15:         msg ← empty message
16:         superstep ← superstep + 1
17:         ∀r ∈ ℛ wait for responses or suspicions
18:         for all p ∈ 𝒫 do
19:             if (∃w ∈ ℛ[p] | suspicions[w] = True) or (∃w, w' ∈
    ℛ[p] | resp[w].hash ≠ resp[w'].hash) then
20:                 checkpoint ← lastCheckpoint()
21:                 superstep ← checkpoint.superstep
22:                 msg.addCmd(RESTORE_CHECKPOINT, checkpoint)
23:                 ℱ[p] ← ℱ[p] + 1
24:                 if (ℱ[p] > f_max) or (∃w ∈ ℛ[p] | suspicions[w] =
    True) then
25:                     removed ← ℛ[p]
26:                     ℛ ← ℛ − {removed}
27:                     if |𝒩| > f + 1 then
28:                         ℛ[p] ← {w ∈ 𝒩 | w_1...w_{f+1}}
29:                         𝒩 ← 𝒩 − ℛ[p]
30:                         ℱ[p] ← 0
31:                         msg.addCmd(PARTITION, ℛ[p])
32:                     else
33:                         msg.addCmd(RESTORE_REPLICS, removed)
34: end function
```

Figure 3. GMaster (Greft)

When GWorker receives this command, instead of loading the entire checkpoint and replacing its current state with the one loaded from the checkpoint, only a part of the checkpoint is loaded and its contents are appended to the worker's current state (done by function *partition.restoreChkptPart*). The checkpoint is obtained by the worker from one of the removed replicas and the same partitioning mechanism used in the input graph is used in the checkpoint to determine which part of the checkpoint each set of workers should load.

When the START_SUPERSTEP command is received, the user-defined function is executed on each of the GWorker partition vertices (lines 14 to 16). After that, a hash of all vertices's states is computed and added to the message that will be sent to GMaster (line 18). The user function defines what is the *state* of each vertex: its value, weights of its edges or both. Vertices are ordered by their numeric id before the hash is computed. This way the system ensures that if the states are the same between the replicas, so will be the hashes. When the system finishes processing, only one of the replicas of each replica set writes the result to the output files (function *partition.writeResult*).

```
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.restoreChkpt(msg.superstep,
    msg.checkpoint.hashes)
12:            if msg.RESTORE_REPLICAS then
13:                partition.restoreChkptPart(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
```

Figure 4. GWorker (Greft)

IV. Greft's Implementation

Greft's prototype was implement using GPS' initial version, as made available by its authors[2]. GPS is written in Java, so modifications will be described in terms of classes. HDFS was not modified, as explained. Besides implementing the algorithm described, we had to create saving and restoring checkpoint routines, because they were not available in GPS.

Parameters f, f_{max} and spc were included in the GMaster class. This class' graph partitioning routine was modified to replicate $f + 1$ times each partition in different GWorkers. The message to start a new superstep was modified to include commands to create or restore checkpoints. In the superstep control loop, after receiving end of superstep messages from all GWorkers, a routine to compare the hashes of each replica set was created. If a divergence is detected, the command to restore the last checkpoint is included in the next start superstep message, along with the checkpoint file hashes. If a replica set reached f_{max} faults or a machine has crashed, the replicas are removed. If there are enough available machines to replace this replica set, they are used. If not, information about the removed replicas checkpoints are included in the start superstep message.

In the GWorker class, the superstep control loop was modified to include the checkpoint creation and restore routines, and the computation of vertex state hashes. Upon receiving a start superstep message, GWorker creates or restores a checkpoint according to the command received. After the user-defined function is executed for all vertices, GWorker computes a hash of all vertex states, ordering them by their numeric ID. This way, replicas with the same set of vertices with the same values will produce identical hashes. This hash is added to the end of superstep message sent to GMaster. If a checkpoint was created in this superstep, its hash is also added to the message.

[2]Obtained in 2014-09-23 at https://subversion.assembla.com/svn/phd-projects/gps/trunk/

V. Experimental Evaluation

In this section we discuss the experimental results obtained by executing Greft in a cluster. The experiments aimed to answer the following questions: (1) What is the additional cost of tolerating arbitrary faults? (2) What is the performance improvement obtained by saving checkpoints to local disk instead of a distributed file system? (3) What is the overhead introduced by recovering from a fault?

Although there are no specific benchmarks for distributed graph processing, there are several real-world graph datasets available to be used in experiments. We used a large graph dataset extracted from Twitter [25]. In this directed graph, each vertex represents an user and the edges his/hers followers. This graph has approximately 47.1 million vertices and 1.47 billion edges. The experiments were performed at Amazon Web Services (AWS), using 17 *r3.large* instances from Elastic Cloud Computing (EC2). This type of instance has 15GB RAM, 32 GB of SSD storage and 2 Intel Xeon E5-2670 v2 processor virtual CPUs. All instances were located in the same availability zone with Ubuntu Server 14.04 LTS operating system. As GPS and Greft need the whole graph to fit in memory, this type of instance was chosen because it has the lowest cost per GiB of RAM among Amazon EC2 instance types. Moreover, it is in cloud providers such as Amazon that most real-world large scale processing is executed, including graph processing.

Three algorithms were used in the experiments: *PageRank*, *Single Source Shortest Path* (SSSP) and *Weak Connected* Components (WCC). Pagerank is characterized by high traffic but low processing per vertex, with all vertices active during the computation. SSSP and WCC are both characterized by a decreasing number of active vertices during the computation, so the number of messages exchanged between vertices also decreases with time. SSSP and WCC are also characterized by a variable number of supersteps, depending on the graph topology. PageRank, on the other hand, has a fixed number of iterations (40 in our experiments).

Greft was parametrized with $f = 1$, since this is the value used in most arbitrary fault-tolerant algorithm evaluations [17], [22], [23], [18]. Besides, considering the meaning of f in our algorithm, the probability of this assumption being broken is negligible. Only *transient faults* were simulated, because replica removal routines were not fully implemented in the available version of GPS. The interval of supersteps between checkpoint parameter (spc) was defined according to the algorithm characteristics and expected run time. It was set at 8 for PageRank, 6 for SSSP and 10 for WCC. We only consider the superstep execution time in our results. Although the initial partitioning is affected by our

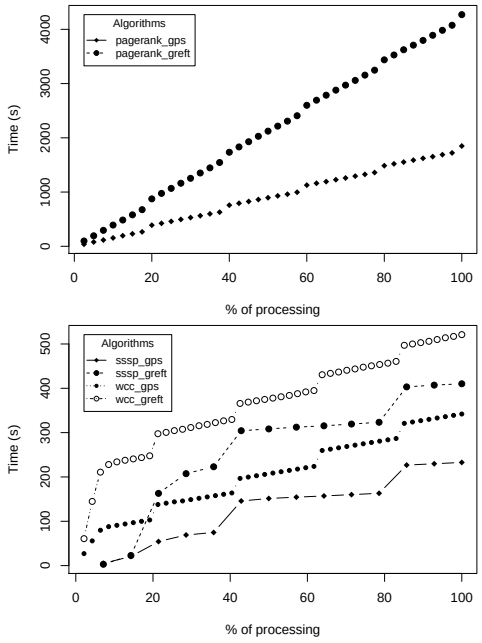


Figure 5. Execution time on Greft and GPS for SSSP/WCC

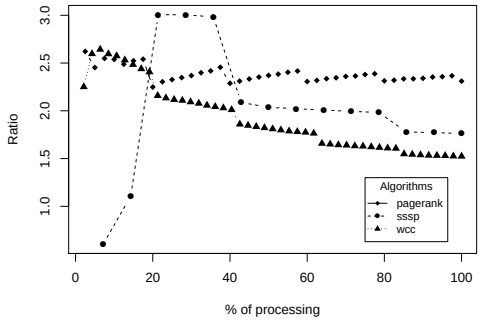


Figure 6. Execution time ratio between Greft and GPS for PageRank

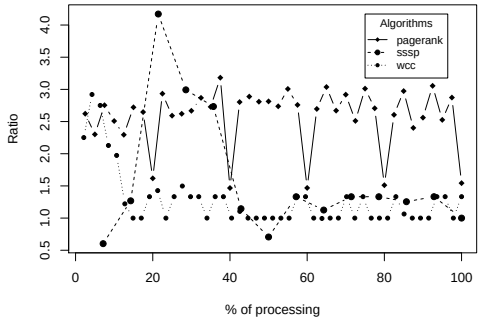


Figure 7. Superstep execution time ratio between Greft and GPS

algorithm, the overhead introduced is very small because what dominates this phase is the way input files are stored in HDFS and the partitioning mechanism chosen, whose optimization is beyond the scope of this paper. The storage of the results produces no significant overhead because only one replica writes the results in the file system. All values shown represent the mean time of 3 executions of each algorithm.

A. Cost of tolerating arbitrary faults

In the first experiment we compare Greft with the original GPS by running all three algorithms on Twitter's dataset. The average run times (and standard deviations) for PageRank on GPS and Greft were 1849 ($\pm$ 62) and 4272 ($\pm$ 119) seconds; for SSSP, 233 ($\pm$ 13) and 411 ($\pm$ 21) seconds; for WCC, 342 ($\pm$ 11) and 568 ($\pm$ 17) seconds. The total run times and the times for achieving 20/40/60/80% of the processing for each algorithm are shown in Figure 5. Because PageRank's execution time is much longer than SSSP and WCC, its time is shown separately in a second chart (same figure). Figure 6 shows the same results, but instead of absolute times, it shows Greft's execution time divided by GPS's execution time (i.e. the ratio between execution times). Both graphs show that Greft processing time is approximately two times longer than GPS's. This was expected as Greft does twice as much computation as GPS, so with the same resources the processing time should be the double.

B. Benefit of saving checkpoints to local disk

Greft optimizes checkpoint management by saving the checkpoints in local disk instead of a distributed file system. A performance improvement was expected for this reason, so an experiment was performed where the times to execute a superstep in Greft and GPS were compared. Figure 7 displays the ratio between execution times of each superstep for each algorithm. This figure shows that in supersteps where a checkpoint is created – the lowest points, at 20, 40, 60, 80, and 100% – Greft performs almost as well as GPS (i.e., the ratio approaches 1), even considering that, because Greft replicates its workers, its checkpoints are almost twice as big as GPS's.

Figure 8 shows the percentage of time spent processing and creating checkpoints. This figure shows that Greft manages checkpoints more efficiently than GPS, as it spends less time on checkpoints and more on processing when compared to GPS.

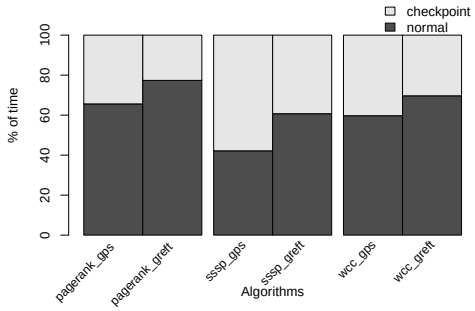


Figure 8. Percentage of time processing and creating checkpoints

C. Overhead of recovering from a fault

To test the overhead introduced by recovering from a fault, an experiment was performed where a transient fault was injected during the computation. Two scenarios were evaluated. Recovery time is dominated by running again the supersteps passed between the checkpoint creation and the fault detection. Therefore, the best case is when a fault is detected right after a checkpoint was created, because only one superstep would have to be executed again. The worst case, however, is when a fault is detected right before a checkpoint would be created, because in this case all supersteps in the interval defined by *spc* would have to be executed again.

Figure 9 display the results that the recovery time in the best case is small, limited to reloading the state and running one superstep. The worst case has a longer recovery time, but is also limited to the number of supersteps that had to be executed again. Table II details these values. The worst value is for the worst case of SSSP. The reason is that SSSP has only 17 supersteps in the graph tested, with $spc = 6$, so processing these supersteps again implies redoing a high percentage of the processing. Otherwise, even in the worst case the maximum overhead is 16.08%.

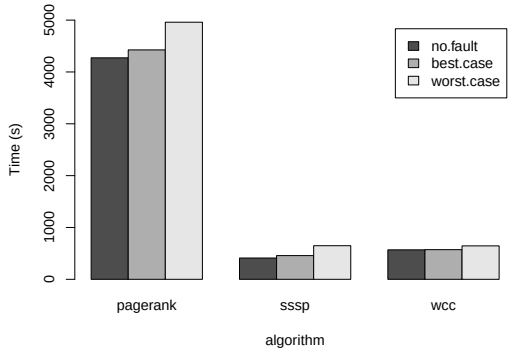


Figure 9. Execution times with and without faults

Time	*no fault*	*best case*	*worst case*
PageRank	4272 s	4425 s	4959 s
SSSP	411 s	457 s	648 s
WCC	568 s	572 s	645 s
Overhead	*no fault*	*best case*	*worst case*
PageRank	0	3.6%	16.08%
SSSP	0	11.2%	57.7%
WCC	0	0.7%	13.6%

Table II
EXECUTION TIMES (SECONDS) AND OVERHEADS OF A FAULT

VI. Related Work

There is a vast literature on the subject of arbitrary / Byzantine fault tolerance, with models dating from the 1980s [26]. State Machine Replication is a generic technique to mask faults [16] that has been shown to be able to tolerate Byzantine faults efficiently [17]. After that work, several other efficient algorithms were published, such as the UpRight library [22] and MinBFT [23]. Byzantine fault tolerance in large scale processing models such as MapReduce were also the subject of studies such as [18], [27]. However, as already discussed, these models are not suited for graph processing due to the costs involved.

	F.T. mechanism	**Fault type**	**Replicas**
Pregel/GPS	Backward recovery	crash	-
PowerGraph	Backward recovery	crash	-
Trinity	Backward recovery	crash	-
GraphX	RDD	crash	-
Imitator	Replication	crash	$f + 1$
Greft	Replication	arbitrary	$f + 1$

Table III
RELATED WORKS AND THEIR CHARACTERISTICS

The graph processing models more related to this work are briefly described in Table III. Pregel [1] is one of the first and the one that introduced many of the concepts used by the other works, e.g., supersteps and vertex-oriented programming. However, Pregel tolerates only crash faults using a checkpoint-based recovery mechanism. GPS adds a few features to Pregel, as explained in Section II. PowerGraph [2] and Trinity [3] use similar mechanisms and also tolerate only crash faults. GraphX [4] tolerates faults extending Spark's Resilient Distributed Datasets (RDD). This mechanism stores changes made to each vertex instead of the data itself. None of these systems uses replication to tolerate faults.

Imitator [28] is based on Pregel but uses in-memory vertex replication to tolerate crash faults without using checkpoints. Some distributed graph processing engines have a vertex replication mechanism that is used to optimize access to high degree vertices [2]. These vertices are replicated in different machines so that its values can be accessed directly, without exchanging messages between machines. Imitator extends this mechanism creating a replica of every vertex. When a machine crashes, their replicas are used to restore the system.

VII. Conclusion and Future Work

In this paper we presented Greft, an algorithm and prototype of an accidental arbitrary fault-tolerant distributed graph processing engine. Experimental results were also presented, confirming that Greft can recover from accidental arbitrary faults in an efficient manner, using only twice the original version resources with $f = 1$. We consider this a realistic value, because accidental arbitrary faults are rare and the probability of such a fault leaving two replicas in the same state is negligible. The optimizations introduced in Greft's checkpoint management also proved to be satisfactory, significantly reducing the time spent in this task when compared to the original version.

As future work we plan to study ways to tolerate *malicious* faults in Greft. We also plan to improve its fault detection algorithm so that it can detect precisely which process of a replica set is faulty. This will improve the fault recovery

time by using healthy replicas states to recover from faults instead of restoring checkpoints, whenever possible.

Acknowledgment Miguel Correia is bolsista CAPES/Brasil (project LEAD Clouds). This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 and CNPq 455303/2014-2.

REFERENCES

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

[2] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *Proceedings of the 10th Symposium on Operating System Design and Implementation*, 2012.

[3] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 International Conference on Management of Data*, 2013, pp. 505–516.

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

[5] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 169–182.

[6] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM SIGPLAN Notices*, vol. 48, 2013, pp. 135–146.

[7] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "Think like a vertex" to "Think like a graph"," in *Proceedings of the VLDB Endowment*, vol. 7, 2013, pp. 193–204.

[8] C. Avery, "Giraph: Large-scale graph processing infrastructure on Hadoop," in *Proceedings of the Hadoop Summit*, 2011.

[9] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the MapReduce framework," in *Proceedings of the IEEE 2nd International Conference on Cloud Computing Technology and Science*, 2010, pp. 721–726.

[10] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, 2013.

[11] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," in *Journal of Physics: Conference Series*, vol. 78, no. 1, 2007.

[12] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[13] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, "Byzantine fault tolerance, from theory to reality," in *Computer Safety, Reliability, and Security*. Springer, 2003, pp. 235–248.

[14] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, 2009, pp. 193–204.

[15] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs," in *Proceedings of the EuroSys 2011 Conference*, 2011, pp. 343–356.

[16] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[17] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[18] P. Costa, M. Pasin, A. N. Bessani, and M. Correia, "On the performance of Byzantine fault-tolerant MapReduce," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 5, pp. 301–313, 2013.

[19] T. Kajdanowicz, P. Kazienko, and W. Indyk, "Parallel processing of large graphs," *Future Generation Computer Systems*, vol. 32, pp. 324–337, 2014.

[20] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[21] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010, pp. 1–10.

[22] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009, pp. 277–290.

[23] G. S. Veronese, M. Correia, A. N. Bessani, L. C., and P. Verissimo, "Efficient Byzantine fault tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[24] X. Li, M. C. Huang, K. Shen, and L. Chu, "An empirical study of memory hardware errors in a server farm," in *Proceedings of the 3rd Workshop on on Hot Topics in System Dependability*, 2007.

[25] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, 2010, pp. 591–600.

[26] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programing Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[27] J. J. Stephen and P. Eugster, "Assured cloud-based data analysis with ClusterBFT," in *Middleware 2013*. Springer, 2013, pp. 82–102.

[28] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan, "Replication-based fault-tolerance for large-scale graph processing," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 562–573.