Phoenix: A Substrate for Resilient Distributed Graph Analytics

Abstract

This paper presents Phoenix, a communication and synchronization substrate that implements a novel protocol for recovering from fail-stop faults when executing graph analytics applications on distributed-memory machines. The standard recovery technique in this space is checkpointing, which rolls back the state of the entire computation to a state that existed before the fault occurred. The insight behind Phoenix is that this is not necessary since it is sufficient to continue the computation from a state that will ultimately produce the correct result. We show that for graph analytics applications, the necessary state adjustment can be specified easily by the application programmer using a thin API supported by the Phoenix system.

Phoenix has no observable overhead during fault-free execution, and it is resilient to any number of faults while guaranteeing that the correct answer will be produced at the end of the computation. This is in contrast to other systems in this space which may either have overheads even during fault-free execution or produce only approximate answers when faults occur during execution.

We incorporated Phoenix into D-Galois, the state-of-the-art distributed graph analytics system, and evaluated it on two production clusters. Our evaluation shows that in the absence of faults, Phoenix is \(\sim 24\times\) faster than GraphX, which provides fault-tolerance using the Spark system. Phoenix also outperforms a checkpoint-restart technique implemented in D-Galois: in fault-free execution, Phoenix has no observable overhead, whereas the checkpoint-restart technique has 31\% overhead. In addition, Phoenix mostly outperforms checkpointing when faults occur, particularly in the common case when only a small number of hosts fail simultaneously.

1. Introduction

Graph analytics systems are now being used to analyze properties of very large graphs such as social network graphs with tens of billions of nodes and edges. Most of these systems use clusters: the graph is partitioned between the hosts of the cluster, and a programming model such as BSP [39] is used to organize the computation and communication [21, 12, 46, 42, 28].

Fault-tolerance is an important concern for long-running graph analytics applications on large clusters. Some state-of-the-art high-performance graph analytics systems such as D-Galois [8] and Gemini [46] do not address fault-tolerance; since the mean time between failures in medium-sized clusters is of the order of days [34], the approach taken by these systems is to minimize overheads for fault-free execution and restart the application if a fault occurs. Older distributed graph analytics systems rely on checkpointing [21, 12, 19, 30, 35], which adds overhead to execution even if there are no failures. Furthermore, all hosts have to be rolled back to the last checkpoint if even one host fails.

More recent systems such as GraphX [43], Imitator [42], Zorro [28], and CoRAL [41] have emphasized the need to avoid taking checkpoints and/or to perform confined recovery in which surviving hosts do not have to be rolled back when a fault occurs. The solution strategies used in these systems are remarkably diverse (described in more detail in Section 2). For example, GraphX is built on top of Spark [45], which supports a dataflow model of computation in which nodes represent computations with transactional semantics and only failed nodes need to be re-executed. Imitator requires an \textit{a priori} bound on the number of faulty hosts, and it uses replication to tolerate failure, reducing the size of graphs it can handle with a cluster of a fixed size. Zorro provides zero overhead in the absence of faults, but it may provide only approximate results if faults happen. CoRAL takes asynchronous checkpoints and performs confined recovery, but it is applicable only to certain kinds of graph applications.

In this paper, we present Phoenix, a communication and synchronization substrate that provides fail-stop fault-tolerance for distributed-memory graph analytics applications. The insight behind Phoenix is that to recover from faults, it is sufficient to restart the computation from a state that will ultimately produce the correct results. These states, called \textit{valid} states in this paper, are defined formally in Section 3. All states that are reached during fault-free execution are valid states, but in general, there are valid states that are not reachable during fault-free execution. On failure, Phoenix sets the state of revived hosts appropriately so that the global state is valid and continues execution, thus performing confined recovery in which non-crashed machines do not lose progress.

This paper makes the following contributions.

- We present Phoenix, a substrate that can be used to achieve fault tolerance to fail-stop faults for graph analytics applications without any observable overhead in the absence of faults. Phoenix does not use checkpoints, but it can be integrated seamlessly with global and local checkpointing.
- We describe properties of graph analytics algorithms that can be exploited to recover efficiently from faults and classify these algorithms into several categories based on these properties (Section 3). The key notions of valid states and globally consistent states are also introduced in this section.
- We describe techniques for recovering from fail-stop faults
without losing progress of non-crashed machines for the different classes of algorithms, and show how they can be implemented by programmers using the Phoenix API (Section 4).

- We show that D-Galois [8], the state-of-the-art distributed graph analytics system, can be made fault tolerant without performance degradation by using Phoenix, outperforming GraphX [43], a fault tolerant system, and Gemini [46], a non-fault tolerant system, by a geometric mean factor of \( \sim 24 \times \) and \( \sim 4 \times \), respectively. We also show that when faults occur, Phoenix generally outperforms the checkpoint-restart technique implemented in the D-Galois system (Section 5).

2. Background and motivation

Section 2.1 describes the programming model and execution model used in current distributed graph analytics systems. Prior work on fault-tolerance in these systems is summarized in Section 2.2.

2.1. Programming model and execution model

Like other distributed graph analytics systems [21, 12, 46], Phoenix uses a variant of the vertex programming model. Each node has one or more labels that are updated by an operator that reads the labels of the node and its neighbors, performs computation, and updates the labels of some of these nodes. Edges may also have labels that are read by the operator. Operators are generally categorized as push-style or pull-style operators. Pull-style operators update the label of the node to which the operator is applied while push-style operators update the labels of the neighbors. In topology-driven algorithms, execution is in rounds, and in each round, the operator is applied to all the nodes of the graph; these algorithms terminate when no node label is changed during a round. The Bellman-Ford algorithm for single-source shortest-path (sssp) computation is an example. Data-driven algorithms, on the other hand, maintain worklists of active nodes where the operator must be applied, and they terminate when the worklist is empty. The delta-stepping algorithm for sssp is an example [17].

To execute these programs on a distributed-memory cluster, the graph is partitioned among the hosts of the cluster, and each host applies operators on nodes in its own partition. Some systems partition the graph by partitioning nodes among hosts, and they permit edges to cross partitions to nodes on other hosts (to enable this, a global address space is implemented in software). Figure 1 shows a graph (used as a running example in this paper), and Figure 2 shows a partitioning of the nodes of this graph between two hosts; the edges between nodes B and C cross host boundaries.

Most distributed graph analytics systems, however, are implemented using disjoint address spaces on the hosts [21, 12, 46]. Edges are partitioned among hosts, and proxy nodes (or replicas) are created for the end-points of each edge on a given host. Therefore, if the edges connected to a given node in the original graph are assigned to multiple hosts in the machine, that node will have multiple proxies. Figure 3 shows an example in which there are proxies for nodes B and C on both hosts. Some systems execute bulk-synchronous parallel (BSP) rounds and reconcile labels of proxies at the end of each round, whereas other systems update proxy labels asynchronously. Phoenix can be used with both execution models, but for the sake of simplicity, we describe and evaluate Phoenix using a proxy-based system with BSP-style computation.

2.2. Fault-tolerant distributed graph analytics systems

Fail-stop faults in distributed systems can be handled with a rebirth-based approach, in which the failed host is replaced with an unused host in the cluster, or with a migration-based approach [36], which distributes failed host’s partition among the surviving hosts. We describe and evaluate Phoenix using rebirth-based recovery, but it can also be used with migration-based recovery.

Several fault-tolerant distributed graph analytics systems rely on checkpointing [21, 1, 19, 12, 25, 42, 35]: the state of the computation is saved periodically on stable storage by taking a globally consistent snapshot [6], and when a failure is detected, the computation is restarted from the last checkpoint [44]. This incurs high synchronization and I/O cost. Another disadvantage of checkpointing is that every host has to be rolled back to the last checkpoint even if it has not failed. To avoid this, many existing distributed graph analytics systems [43, 42, 28, 41] have devised ways to avoid taking global checkpoints or to avoid rolling back live hosts. Phoenix is in this design space, so we discuss these systems in more detail.

GraphX [43] is built on Spark [45], and achieves fault-tolerance by leveraging RDDs from Spark to store information on how to reconstruct graph states in the event of failure; if the reconstruction information becomes too large, it checkpoints the graph state. GraphX is very general and can recover from failure of any number of hosts, but unlike Phoenix, it does not exploit semantic properties of graph analytics applications to reduce overhead.

Table 1 compares the performance of GraphX and Phoenix.
on 32 hosts of the Wrangler cluster (experimental setup described in Section 5). Phoenix programs are ∼ 24× faster, on average, than GraphX programs in the absence of faults while providing the same level of fault-tolerance. D-Galois [8] and Gemini [46] are state-of-the-art distributed graph analytics systems that do not support fault-tolerance. The Phoenix system in Table 1 is D-Galois made resilient by using the Phoenix substrate. Phoenix performance is the same as D-Galois: it does not have any observable overhead in the fault-free case. Phoenix programs also run on average ∼ 4× faster than the same programs in Gemini.

Imitator [42] is designed to tolerate a given number of faults: if it is desired to tolerate n faults, the system ensures that at least n+1 proxies are created for every graph node, and it updates the labels of all these proxies at every round to ensure that at least one proxy survives simultaneous host failure. The memory requirements of Imitator grow proportionately with n, and keeping these additional proxies synchronized adds overhead even when there are no failures. In contrast, Phoenix does not require an a priori bound on the number of simultaneous faults, and it does not require creating any more proxies than are introduced by graph partitioning.

Like Imitator, Zorro [28] relies on node proxies for recovery, but it only uses the proxies created by graph partitioning. If no proxies of a given node survive failure, execution continues, but it may produce an incorrect answer. An advantage of Zorro is that it does not have any overhead in the absence of failures, unlike GraphX and Imitator in which fault tolerance comes at the cost of substantial overhead even when no failures occur. Phoenix shares this advantage, but unlike Zorro, it is guaranteed to produce the correct answer even when failures happen.

CoRAL [41] is based on asynchronous checkpointing [38]: hosts take local checkpoints independently, so no global coordination is required for checkpointing. Nonetheless, it incurs overhead during fault-free execution. Moreover, techniques like CoRAL that do not take a globally consistent checkpoint are not as general as Phoenix. Therefore, CoRAL is applicable to only a subset of algorithms that can be handled by Phoenix (specifically, CoRAL only works for algorithms that we term as self-stabilizing or locally-correcting in Section 3). In CoRAL, like other confined recovery techniques [21, 35], surviving hosts wait for the failed hosts to recover their lost state before continuing algorithm execution. Phoenix, in contrast, performs confined recovery without waiting for the failed hosts to recover their lost state.

### 3. Classes of graph analytics algorithms

This section describes the key properties of graph analytics algorithms that are exploited by Phoenix. Although Phoenix is handling fail-stop faults in distributed-memory machines, these algorithmic properties are easier to understand in the context of data corruption errors (or transient soft faults) in shared-memory programming, so we introduce them in that context. Section 3.1 introduces the key notions of valid and globally consistent states. Section 3.2 shows how these concepts can be used to classify graph analytics algorithms into a small number of categories. Section 3.3 shows how this classification is applicable to a distributed-memory setting.

#### 3.1. Overview

In shared-memory execution, the state of the computation can be described compactly by a vector of node labels in which there is one entry for each node. If s is such a vector and v is a node in the graph, let s(v) refer to the label of node v in state s. For example, if the initial state in a BFS computation is denoted by s1 and the source is node r, then s1(r) = 0 and s1(v) = ∞ for all other nodes v in the graph. During the computation, these labels are updated by applying the operator to nodes in the graph, changing the state. When the algorithm terminates, the final state s_f will contain the BFS labels of all nodes. Therefore, the evolution of the state can be viewed as a trajectory beginning at s1 and ending at s_f in the set S of all states. In general, there are many such trajectories, and since the scheduler is permitted to make non-deterministic choices in the order of processing nodes, different executions of a given program for a given input graph may follow different trajectories from s1 to s_f. Figure 4 shows a trajectory for an application.

We define these concepts formally below.

**Definition 1** For a given graph analytics program and input graph, let S be the set of states, and let s1 and s_f denote the initial and final states respectively.

**Definition 2** For s_m, s_n ∈ S, s_n is said to be a successor state of s_m if applying the operator to a node when the computation is in state s_m changes the state to s_n. A trajectory is a sequence of states s_0, s_1, ..., s_l such that for all 0 ≤ j < l, s_j+1 is a successor state of s_j.

Two subsets of the set of states, which we call valid states (S_V) and globally consistent states (S_GC), are of interest.

**Definition 3** A state s_g is globally consistent if there is a trajectory s_0, ..., s_g from the initial state s_1 to s_g. Denote S_GC the subset of globally consistent states in S.

### Table 1: Total time (s) on 32 hosts of Wrangler

<table>
<thead>
<tr>
<th>App</th>
<th>Input</th>
<th>Total Time (s)</th>
<th>Phoenix Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GraphX Gemini D-Galois Phoenix</td>
<td>GraphX Gemini</td>
</tr>
<tr>
<td>cc</td>
<td>twitter50</td>
<td>110.8 29.0 8.2 8.2</td>
<td>13.5 3.5</td>
</tr>
<tr>
<td></td>
<td>rmat28</td>
<td>166.5 80.0 20.6 20.6</td>
<td>8.1 3.9</td>
</tr>
<tr>
<td></td>
<td>kron30</td>
<td>1538.2 346.7 46.0 46.0</td>
<td>14.3 7.0</td>
</tr>
<tr>
<td>pr</td>
<td>twitter50</td>
<td>2111.6 75.1 31.6 31.6</td>
<td>66.7 2.4</td>
</tr>
<tr>
<td></td>
<td>rmat28</td>
<td>5355.5 180.9 47.7 47.7</td>
<td>112.3 3.8</td>
</tr>
<tr>
<td></td>
<td>kron30</td>
<td>797.6 426.4 78.9 78.9</td>
<td>10.1 5.4</td>
</tr>
<tr>
<td>sssp</td>
<td>twitter50</td>
<td>153.1 24.2 8.5 8.5</td>
<td>18.0 2.8</td>
</tr>
<tr>
<td></td>
<td>rmat28</td>
<td>158.1 77.7 11.1 11.1</td>
<td>14.3 7.0</td>
</tr>
<tr>
<td></td>
<td>kron30</td>
<td>1893.2 346.7 46.0 46.0</td>
<td>41.2 7.5</td>
</tr>
</tbody>
</table>

This table shows the total time (in seconds) of Wrangler systems that do not support fault-tolerance. The Phoenix system in Table 1 is D-Galois made resilient by using the Phoenix substrate. Phoenix performance is the same as D-Galois: it does not have any observable overhead in the fault-free case. Phoenix programs also run on average ∼ 4× faster than the same programs in Gemini.
A state \( s_i \) is valid if there is a trajectory \( s_i, \ldots, s_f \) from \( s_i \) to the final state \( s_f \). Denote \( S_V \) the subset of valid states in \( S \).

Intuitively, a state is globally consistent if it is “reachable” (along some trajectory) from the initial state, and a state is valid if the final state (the “answer”) is reachable from that state \([10, 14, 32, 24, 31]\). Every globally consistent state is valid; otherwise, there is a state \( s \), reachable from \( s_i \), from which \( s_f \) is not reachable, which is impossible. Therefore,

\[
S_{GC} \subseteq S_V \subseteq S
\] (1)

In general, both containments can be strict, so there can be valid states that are not globally consistent, and there can be states that are not valid. This can be illustrated with BFS. Consider a state \( s_i \) in which the label of the root is 0, the labels of its immediate neighbors are 2, and all other node labels are \( \infty \). This is not a state reachable from \( s_i \), so \( s_i \notin S_{GC} \). However, it is a valid state since applying the operator to the root node changes the labels of the neighbors of the root node to their correct (and final values). This shows that \( S_{GC} \subset S_V \) in general. To show that not all states are valid, consider the state \( s_z \) where \( s_z(v) = 0 \) for all nodes \( v \). No other state is reachable from this state; in particular, \( s_f \) is not reachable, showing that \( s_z \) is not a valid state. Therefore, \( S_V \subset S \) in general.

The recovery approach used in Phoenix can be explained using the notions of valid states and globally consistent states. Consider Figure 4, which shows a trajectory of states from the initial state \( s_i \) to the final state \( s_f \) in a fault-free execution. Every intermediate state in the trajectory between \( s_i \) and \( s_f \) is a globally consistent state. Checkpointing schemes save globally consistent states on stable storage and recover from a fault by restoring the last globally consistent state saved on stable storage. Figure 4 shows this pictorially: \( s_b \) is an invalid state generated by data corruption errors, and the checkpointing scheme, denoted by CR, restores the state to a globally consistent state. The key insight in Phoenix is that for recovery, we can restart the computation from a valid state that is not necessarily a globally consistent state.

3.2. Classification of graph algorithms

The way Phoenix generates a valid state for recovery depends on the structure of the algorithm, and there are four cases which we explain in increasing order of complexity below.

**Self-stabilizing graph algorithms:** All states are valid states, i.e.,

\[
S_{GC} \subseteq S_V = S
\] (2)

Examples are topology-driven algorithms for collaborative filtering using stochastic gradient descent (cf), belief-propagation, pull-style pagerank, and pull-style graph coloring. For example, in both cf and pagerank, node labels are initialized to random values at the start of the computation, and the algorithm converges regardless of what those values are. Therefore, every state is valid, and in principle, no correction of the state is required when a data corruption error is detected in the shared-memory scenario.

**Locally-correcting graph algorithms:** Valid states are a proper superset of the set of globally consistent states and a proper subset of the set of all states. In addition, each node \( v \) has a set of valid values \( L_v \) and the set of valid states \( S_V \) is the Cartesian product of these sets.

\[
S_{GC} \subset S_V \subset S
\]

\[
S_V = L_1 \times L_2 \times \cdots \times L_N
\] (3)

Some examples are breadth-first search (bfs), single-source shortest path (sssp), connected components using label propagation, data-driven pagerank, and topology-driven k-core. For example, in bfs, all values are valid for the root node since the operator sets its label to 0. Therefore, \( L_r = [0, \infty] \). For an immediate neighbor \( v \) of the root node, \( L_v = [1, \infty] \), and so on for the neighbors of those nodes. Data corruption in the shared-memory case can be handled by setting the value of each corrupted node to \( \infty \); the labels of all other nodes can remain unchanged. This may produce a state that is not globally consistent, but the properties of the algorithm guarantee that the final state is reachable from this valid state.

**Globally-correcting graph algorithms:** Valid states are distinct from globally consistent states and from the set of all states, but unlike in the previous case, validity of a state depends on some global condition on the labels of all nodes and cannot be reduced to the Cartesian product of valid values for individual node labels.

In these algorithms, typically, the label of a node is not only dependent on the labels of its neighbors but also on the history of these labels. These algorithms are usually more work-efficient than their equivalent locally-correcting counterparts. Some examples are residual-based pagerank, degree-decrementing k-core, and latent Dirichlet allocation.

**Globally consistent algorithms:** Only globally consistent states are valid states i.e.,

\[
S_{GC} = S_V \subset S
\] (4)
Traditional HPC algorithms fall in this category. Globally consistent snapshots [6] are required for recovery. In such cases, Phoenix can be used along with traditional checkpointing. We list this class only for the sake of completeness, and we do not discuss this class of algorithms further in this paper.

### 3.3. Distributed graph analytics

The concepts introduced in this section for the shared-memory case can be applied to distributed memory implementations of graph analytics algorithms as follows. The main difference from the shared-memory case is that a given node in the graph can have proxies on several hosts that can be updated independently during the computation. In BSP-style execution, all proxies of a given node are synchronized at the end of every round, and at that point, they all have the same labels. Therefore, in the distributed-memory case, we consider a round to constitute one step of computation, and the global state at the end of each round is simply the vector containing the labels of the nodes at that point in time.

**Definition 4** For $s_m, s_h \in S$, $s_h$ is said to be a successor state of $s_m$ if a single BSP round transforms state $s_m$ to state $s_h$.

A trajectory is a sequence of states $s_0, s_1, \ldots, s_j$ such that for all $0 \leq j < l$, $s_{j+1}$ is a successor state of $s_j$.

Globally consistent states and valid states can be defined as in Definition 3, and the classification of graph algorithms in Section 3.2 can now be used in the context of distributed-memory implementations.

In Phoenix, fail-stop faults are detected and handled during the synchronization phase of a BSP round. Graph partitions are reloaded from stable storage on the hosts that replace the failed hosts. Some of the nodes on these hosts may have proxies on healthy hosts, and if so, their state can be recovered from their proxies. In Phoenix, this is done using a minor variation of the synchronization call used to reconcile proxies during normal execution (this is called a $\text{Sync}$ call). There may also be nodes for which no proxy exists on a healthy node. If so, Phoenix restores the global state to a valid state. The Phoenix approach to restore a valid state is specific to the class of algorithm. This is described in detail in Section 4 using concrete examples for each class of algorithm.

### 4. Fault-tolerance in distributed memory

In this section, we describe how Phoenix performs confined recovery when a fault occurs without incurring observable overhead during fault-free execution. Sections 4.1, 4.2, and 4.3 illustrate Phoenix’s recovery mechanisms for the three different classes of algorithms introduced in Section 3. Algorithms are presented in pseudocode since they can be implemented in different programming models or runtimes. Each host $h$ executes the algorithm on its partition of the graph. The worklist, when used, is local to the host. All algorithms use a generic interface, $\text{Sync}$ (which directly maps to Gluon’s [8] $\text{Sync}$ interface used in D-Galois) or $\text{SyncW}$, to synchronize the state of all proxies; $\text{SyncW}$ is similar to $\text{Sync}$, but it also adds nodes whose state is updated locally during synchronization to the worklist. $\text{Sync}$ and $\text{SyncW}$ return a failed message when at least one host has crashed, and when that occurs, the algorithms calls a generic API for Phoenix that recovers from the failure. Section 4.4 summarizes the Phoenix API and discusses how the programmer can use it in their applications.

#### 4.1. Self-stabilizing graph algorithms

We use greedy graph coloring to illustrate how self-stabilizing algorithms are handled by Phoenix. Algorithm 1 shows the greedy graph coloring algorithm for an undirected or symmetric graph. Every node has a label to denote its color which is initialized randomly. Nodes and colors are ordered according to some ranking functions. The algorithm is executed in rounds. In each round, every node picks the smallest color that was not picked by any of its smaller neighbors in the previous round. The algorithm terminates when the color assignment does not change in a round.

To tolerate faults, the programmer calls the Phoenix API (line 14) when $\text{Sync}$ fails. Figure 5 shows possible state transitions after each round of algorithm execution for the graph in Figure 1. In this figure, colors are encoded as integers. Suppose that faults are detected in the fourth round when performing the $\text{Sync}$ operation. To recover, Phoenix will initialize the colors of the proxies of the nodes on each

---

**Algorithm 1**: Greedy graph coloring (self-stabilizing)

```
Input : Partition $G_h = (V_h, E_h)$ of graph $G = (V, E)$
Output: A set of colors $s(v) \forall v \in V$
1 Let $t(v) \forall v \in V$ be a set of temporary colors
2 Function $\text{Init}(v, s)$:
3 $s(v) = \text{random color}$
4 foreach $v \in V_h$ do
5    $\text{Init}(v, s)$
6 repeat
7    foreach $v \in V_h$ do
8       $t(v) = s(v)$
9    foreach $v \in V_h$ do
10       foreach $u \in \text{adj}(v)$ and $u < v$ do
11         $nc = nc \cup \{t(u)\}$
12       $s(v) = \text{smallest } c \text{ such that } c \notin nc$
13 while $\text{Runtime.}$ $\text{Sync}(s) == \text{Failed}$ do
14    $\text{Phoenix.}$ $\text{Recover}(\text{Init}, s)$
15 until $\forall v \in V, s(v) = t(v)$
```

---

**Figure 5**: States of the graph in Figure 1, treated as an undirected graph, during the execution of greedy graph coloring.
failed host to a random color using the programmer supplied function. Some of these nodes might have proxies on healthy hosts, which will be recovered by the subsequent Sync call (line 13) after Phoenix returns. The nodes which do not have any proxies on healthy hosts are shaded red in Figure 5. Algorithm execution then continues, converging in three more rounds. Phoenix supports such confined recovery by providing a thin API defined in Algorithm 2. Phoenix recovers a valid state for all self-stabilizing algorithms in this way because any state is a valid state (Equation 2).

4.2. Locally-correcting graph algorithms

We use level-by-level breadth-first search, shown in Algorithm 3, to explain how locally-correcting algorithms are handled by Phoenix. In this algorithm, every node has a label that denotes its distance or level from the source. The distance of the source node is initialized to 0, and the distance of every other node is initialized to infinity. The algorithm is executed in rounds, and in each round, the relaxation operator is applied to nodes on the bfs frontier, which is tracked by worklists. The relaxation operator on a node updates its outgoing-neighbors (successors) if their distance is more than the node’s distance plus 1. Initially, only the source node is in the worklist. If the distance of the neighbor changes when relaxed, then the node is added to the worklist for the next round. The algorithm terminates (global_termination) when there are no nodes in the worklist on any host.

The programmer inserts the Phoenix API (line 15) to support resilience when Sync fails. When a fault occurs,
Algorithm 5: Data-driven k-core (global-correcting)

Input: Partition $G_0 = (V_0, E_0)$ of graph $G = (V, E)$
Input: $k$
Output: A set of flags and degrees $s(v) \forall v \in V$

Function DecrementDegree $(v, s)$:
1. foreach $u \in outgoing_{adj}(v)$ do
2. $s(u).degree = s(u).degree - 1$

Function ReInitW $(v, s, W)$:
3. $s(v).degree = |outgoing_{adj}(v)|$
4. $W_0 = W_0 \cup \{v\}$

Function InitW $(v, s, W)$:
5. $s(v).flag = True$
6. $\text{DecrementDegree} (v, s)$
7. $W_0 = W_0 \cup \{v\}$

Function ComputeW $(v, s, W)$:
8. if $s(v).degree < k$ then
9. $s(v).flag = False$
10. $\text{DecrementDegree} (v, s)$
11. else
12. $W_0 = W_0 \cup \{v\}$

Function ReComputeW $(v, s, W)$:
13. if $s(v).flag = False$ then
14. $\text{DecrementDegree} (v, s)$
15. else
16. $W_0 = W_0 \cup \{v\}$

foreach $v \in V_0$ do
17. $\text{InitW} (v, s, W_0)$
18. repeat
19. $W_0 = W_0 \cup W_0$
20. foreach $v \in V_0$ do
21. $\text{ComputeW} (v, s, W_0)$
22. while Runtime.SyncW $(s, W_0) = Failed$ do
23. $\text{ReComputeW} (v, s, W_0)$
24. until global_termination;

Figure 7: State of the graph in Figure 1, treated as a symmetric graph, during the execution of degree-decrementing k-core (globally-correcting): $k = 4$.

Algorithm 6: Phoenix recovery for globally-correcting algorithms

Function Recover $(\text{InitW}, \text{ReInitW}, \text{ReComputeW}, s, W_0)$:
1. if $h \in \text{failed hosts} H_f$ then
2. foreach $v \in V_0$ do
3. $\text{InitW} (v, s, W_0)$
4. else
5. foreach $v \in V_0$ do
6. $\text{ReInitW} (v, s, W_0)$
7. $\text{ReComputeW} (v, s, W_0)$
8. if Runtime.SyncW $(s, W_0) = Failed$ then
9. return
10. $W_0 = W_0 \cup W_0$
11. foreach $v \in V_0$ do
12. $\text{ReComputeW} (v, s, W_0)$

4.4. Phoenix API

The Phoenix API is specific to the class of algorithm, as defined in Algorithms 2, 4, and 6. For self-stabilizing algorithms, the API takes the initialization function as input and updates the state. For locally-correcting algorithms, the API takes the initialization function as input and updates both the state and the work-list. For globally-correcting algorithms, the API updates the state and the work-list by taking functions for initialization, re-initialization, and re-computation as input. The Sync function is expected to be called soon after the Phoenix API so that proxies are synchronized with each other.
rithms to enable Phoenix is straight-forward because the ini-
tialization function required by the API would be used in fault-
free execution regardless. On the other hand, for globally-
correcting algorithms, the programmer must write new re-
initialization and re-computation functions to use the Phoenix 
API. This is akin to replacing the work-efficient algorithm
used during fault-free execution by writing a naive algorithm
that computes the new state using only the old state.

The Phoenix API replaces the checkpoint restore functions
in Checkpoint-Restart (CR) systems, and it can be incorpor-
ated into existing synchronous or asynchronous distributed
graph analytics programming models or runtimes (Phoenix
recovery itself involves a BSP-style computation and commu-
nication round). Phoenix can also be combined with existing
checkpoint-restart fault-tolerant techniques that take globally
consistent snapshots or locally consistent snapshots [21, 41].
In this case, the failed hosts initialize their values from the last
checkpoint instead of calling the initialization function.

5. Experimental evaluation

D-Galois [8] is the state-of-the-art distributed graph analytics
system, but it does not support fault tolerance. The Phoenix
fault-tolerance approach discussed in this paper was imple-
mented in D-Galois, and in the results below, we just call this
system Phoenix. We also implemented a checkpoint-restart
fault-tolerant technique in D-Galois, and this system is termed
CR. D-Galois is bulk-synchronous parallel (BSP), so CR takes
a globally-consistent snapshot soon after bulk-synchronization
and checkpoints it synchronously (Asynchronous checkpoint-
ing techniques like CoRAL [41] that do not take a globally-
consistent snapshot cannot be used because it is not applicable
for all applications we evaluate). The periodicity of check-
pointing can be specified at runtime. We evaluated checkpoint-
ing after every 5, 50, and 500 rounds of BSP-style execution,
and these are called CR-5, CR-50, and CR-500 respectively.

Experiments were conducted on the Stampede [37] and
Wrangler clusters at the Texas Advanced Computing Cen-
ter [2]. The configurations used are listed in Table 2. We
used Wrangler to compare Phoenix with GraphX [43] and
Gemini [46] (GraphX cannot be installed on Stampede). All
other experiments were conducted on 128 hosts of Stampede.

Our evaluation uses 5 benchmarks: connected components
(cc), k-core decomposition (kcore), pagerank (pr), single-
source shortest path (sssp), and collaborative filtering using
stochastic gradient descent (cf). In D-Galois (consequently,
CR and Phoenix), cf is a self-stabilizing algorithm, cc and
sssp are locally-correcting algorithms, and kcore and pr are
globally-correcting algorithms. We used GraphX and Gem-
ini implementations of the same benchmarks (they use self-
stabilizing algorithm for pr); they do not have kcore or cf. sssp
uses the maximum out-degree node as the source node. The
tolerance for pr is 10−7 for kron30, 10−4 for clueweb12, and
10−3 for wdc12. The k-core value used is 100. All benchmarks
are run until convergence. Results presented are for a mean of
3 runs. The number of BSP-style rounds for wdc12 is between
270 and 3700 depending on the application.

Table 3 specifies the input graphs: wdc12 [23, 22] and
clueweb12 [27, 4, 3] are the largest publicly available web-
crawls; kron30 [18] and rmat28 [5] are randomized synthetic
scale-free graphs; twitter50 [16] is a social network graph;
amazon [13] is the largest publicly available bipartite graph.
Smaller inputs are used for comparison with GraphX because
it exhausts memory quickly.

5.1. Fault-free performance

Table 1 compares the performance of GraphX, Gemini, D-
Galois, and Phoenix on Wrangler in the absence of faults.
All systems read the graph from a single file on disk and
partition it among hosts. The total time includes the time to
load and partition the graph. Phoenix is on average ∼ 24× and
∼ 4× faster than GraphX and Gemini, respectively. Using
the Phoenix substrate to make D-Galois resilient adds no
overhead during fault-free execution. GraphX is more than
an order of magnitude slower than Phoenix during fault-free
5.2. Overhead of fault-tolerance

To evaluate the behavior of Phoenix and CR when fail-stop faults occur, we simulate a fault by clearing all in-memory data structures on the failed hosts and running recovery techniques on the failed hosts (rebirth-based recovery). We simulate various fault scenarios by varying the number of failed hosts (1, 4, 16, 64 hosts) as well as by causing the hosts to fail at different points in the algorithm execution (failing after executing 25%, 50%, 75%, 99% of rounds). The number of failed hosts does not impact CR because all hosts rollback to the last checkpoint (the graph partition re-loading time does not vary much because we are limited by the host’s bandwidth, not the network’s bandwidth). Note that these scenarios are designed to test the best and worst fault scenarios for Phoenix, and we do not choose the point of failure to test the best or worst fault scenarios for checkpointing.

For each benchmark and input, Figure 8 shows the total time overhead of faults for Phoenix and CR-50 over fault-free execution of Phoenix using a box-plot\(^1\) to summarize all fault scenarios. We omit CR-5 and CR-500 because the overheads are more than 100% in several cases. Phoenix has lower overhead than CR-50 in most cases. For all benchmarks, the overhead reduces as the size of the graph increases. Smaller graphs have very little computation, so almost all of the recovery overhead is due to re-loading the partitions from disk on the failed hosts. Except for the smaller kron30, Phoenix has at most 50% overhead when faults occur in most cases.

We now analyze the overhead of faults with Phoenix and CR in more detail by comparing execution time, which excludes initial loading of graph partitions as well as re-loading of graph partitions on failed hosts. Figures 9 and 10 compare the execution time of Phoenix and CR in different fault scenarios, including no faults. Execution time is divided into algorithm, recovery, and checkpointing time. Recovery time in Phoenix and CR is the time to restore the state to a valid state and globally consistent state, respectively. We omit CR-5 in the figure because its overhead is too high. We observe that for realistic fault scenarios [34] such as up to 16 hosts failing, Phoenix outperforms CR-50 and CR-500, except for kcore on cluweb12. When 4 hosts fail, the mean execution time overhead over fault-free execution of Phoenix, CR-50, and CR-500 is ∼14%, ∼48.5%, and ∼59%, respectively. CR-500 is worse than CR-50 is many cases because it loses more progress when faults occur due to less frequent checkpointing. For Phoenix, as we increase the number of failed hosts, the execution overhead increases. The mean execution time overhead for 16 failed hosts is ∼20.8%, and it increases to ∼44% when 64 hosts fail. In contrast, the point of failure does not increase the overhead by much in most cases. The takeaway is that even in the worst case scenario for Phoenix (64 hosts fail after 99% of rounds), the performance of Phoenix is comparable or better than CR.

When some state is lost due to faults, Phoenix and CR-50 restore the state to a valid and globally consistent state, respectively. However, the algorithm may need to execute more computation and communication to recover the lost state. For Phoenix and CR-50, Figure 11 shows the % increase in rounds and communication volume due to failures over fault-free execution of Phoenix (box-plot summarizes all fault scenarios). The overhead is under 100% for Phoenix and CR-

\(^1\)The box represents the range of 50% of the values, the line dividing the box is the median of those values, and the circles are outliers.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Recovery</th>
<th>Checkpointing</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoFault</td>
<td>Faults at 25%</td>
<td>Faults at 50%</td>
</tr>
<tr>
<td>cc</td>
<td><img src="image" alt="cc" /></td>
<td><img src="image" alt="cc" /></td>
</tr>
<tr>
<td>kcore</td>
<td><img src="image" alt="kcore" /></td>
<td><img src="image" alt="kcore" /></td>
</tr>
<tr>
<td>pr</td>
<td><img src="image" alt="pr" /></td>
<td><img src="image" alt="pr" /></td>
</tr>
<tr>
<td>sssp</td>
<td><img src="image" alt="sssp" /></td>
<td><img src="image" alt="sssp" /></td>
</tr>
</tbody>
</table>

Figure 10: Execution time (s) of Phoenix (Ph) and CR with different fault scenarios.
50; therefore, both schemes of fault-tolerance are better than simply re-executing applications in case of failure. In most cases, Phoenix has lower overhead than CR-50. Furthermore, the overhead of Phoenix is less than 50% in almost all cases.

5.3. Combining Phoenix with checkpointing

As observed in our analysis of CR-50 and CR-500, reducing the frequency of checkpointing reduces overhead in fault-free execution, but it can lead to high overheads in case of failure as more progress can be lost. On the other hand, Phoenix has no overhead in fault-free execution, but in worst case scenarios (such as 64 hosts crashing after 99% of execution), it can have considerable overhead. To overcome this, we can combine Phoenix with checkpointing (CPh) so that we take checkpoints less frequently (every 500 rounds), but in case of failures, crashed hosts can rollback to the last saved checkpoint, and all hosts can use Phoenix to help crashed hosts recover faster. Figure 12 shows the evaluation of CPh in case of 64 hosts crashing at different points of execution for all benchmarks on wdc. CPh is faster than CR-500 in all cases, as expected.

6. Related work

System level fault-tolerance. Many systems [20, 11, 7, 40, 29] implement checkpointing or message logging to provide fault tolerance. Although these system-level mechanisms are transparent to the programmer, exploiting domain knowledge can reduce overheads substantially as shown in this paper.

Fault-tolerant graph-analytical systems. Many systems [21, 43, 1, 19, 12, 25, 42, 35, 36, 28, 41, 26] exist for distributed graph analytics that support fault-tolerance. Greft [26] is the only one that tolerates Byzantine faults; the rest tolerate fail-stop faults. Section 2.2 contrasts Phoenix with other fail-stop fault-tolerant systems in detail.

Self-stabilizing algorithms. Phoenix generalizes the classical concept of self-stabilizing algorithms [10, 14, 32, 24, 31], as explained in Section 3.

Data-parallel systems. Some data-parallel systems [9, 15, 45] save sufficient information to re-execute computation and restore lost data when faults occur. Schelter et al. [33] allow users to specify functions that can recover state when faults occur but their technique is applicable only to self-stabilizing and locally-correcting algorithms.

7. Conclusions

We presented Phoenix, a substrate used to provide fail-stop fault tolerance in distributed graph analytics applications. The key observation it uses is that recovery from failure can be accomplished by continuing the computation from a state that will ultimately produce the correct result. We presented three classes of graph algorithms and the methods used to adjust the state after failure for each class. Experiments showed that D-Galois augmented with Phoenix outperforms GraphX by an order of magnitude and almost always outperforms traditional checkpoint-based recovery in the presence of faults.
References

[43] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A Resilient Distributed Graph System on Spark. In First International Workshop on Graph Data Management Experiences and Systems, GRADES ’13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

