Phoenix: A Substrate for Resilient Distributed Graph Analytics

Roshan Dathathri
University of Texas at Austin, USA
roshan@cs.utexas.edu

Gurbinder Gill
University of Texas at Austin, USA
gill@cs.utexas.edu

Loc Hoang
University of Texas at Austin, USA
loc@cs.utexas.edu

Keshav Pingali
University of Texas at Austin, USA
pingali@cs.utexas.edu

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 programmer using a thin API supported by Phoenix.

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 the traditional checkpoint-restart technique implemented in D-Galois: in fault-free execution, Phoenix has no observable overhead, while the checkpointing technique has 31% overhead. Furthermore, Phoenix mostly outperforms checkpointing when faults occur, particularly in the common case when only a small number of hosts fail simultaneously.

CCS Concepts • Software and its engineering → Software fault tolerance; • Computing methodologies → Distributed programming languages.

Keywords fault tolerance, distributed-memory graph analytics, big data, self-stabilizing algorithms

ACM Reference Format:

1 Introduction

Graph analytics systems are now being used to analyze properties of very large graphs such as social networks 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 [63] is used to organize the computation and communication [25, 37, 49, 65, 73].

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 [18] and Gemini [73] do not address fault tolerance; since the mean time between failures in medium-sized clusters is of the order of days [56], 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 [2, 25, 35, 37, 51, 57, 65], which adds overhead to execution even if there are no failures. Furthermore, all hosts have to be rolled back to the last saved checkpoint if even one host fails.

More recent systems such as GraphX [68], Imitator [65], Zorro [49], and CoRAL [64] 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 quite diverse and are described in Section 2. For example, GraphX is built on top of Spark [71], which supports a dataflow model of computation in which computations have transactional semantics and only failed computations need to be re-executed. Imitator requires an a priori bound on the number of faulty hosts, and it uses replication to tolerate failure, which constrains the size of graphs it can handle on a cluster of a given 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 to provide 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 result. These states, called 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 with the aid of the application programmer so that the global state is valid and classified 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 surviving hosts for the different classes of algorithms and show how they can be implemented by application developers using the Phoenix API (Section 4).

- We show that D-Galois [18], the state-of-the-art distributed graph analytics system, can be made fault tolerant without performance degradation by using Phoenix while outperforming GraphX [68], a fault tolerant system, and Gemini [73], a fault intolerant system, by a geometric mean factor of $\sim 24 \times$ and $\sim 4 \times$, respectively. In addition, when faults occur, Phoenix generally outperforms the traditional checkpoint-restart technique implemented in D-Galois (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 [25, 37, 73], 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 work-list is empty. The delta-stepping algorithm for sssp is an example [33, 45].

To execute these programs on a distributed-memory cluster, the graph is partitioned among the hosts of the cluster.
and each host applies the operator to nodes in its partition. Some systems implement a global address space in software. Nodes are partitioned between hosts and edges may connect nodes in different hosts. Figure 1a shows a graph used as the running example in this paper, and Figure 1b shows a partitioning of this graph between two hosts; the edges between nodes B and C connect nodes in different hosts.

Most distributed graph analytics systems, however, are implemented using disjoint address spaces [25, 37, 73]. Edges are partitioned among hosts, and proxy nodes (or replicas) are created for the end-points of each edge on a given host. Thus, if the edges connected to a given node in the original graph are assigned to multiple hosts in the cluster, that node will have multiple proxies. Figure 1c 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 simplicity’s sake, we describe and evaluate Phoenix using a proxy-based system with BSP-style computation.

2.2 Fault Tolerant Distributed Graph Analytics

A fail-stop fault in a distributed system occurs when a host in the cluster fails without corrupting data on other hosts. Such faults can be handled with a rebirth-based approach, in which the failed host is replaced with an unused host in the cluster, or a migration-based approach [58], which distributes the 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 [2, 25, 35, 37, 51, 57, 65]: the state of the computation is saved periodically on stable storage by taking a globally consistent snapshot [14], and when a fail-stop fault is detected, the computation is restarted from the last checkpoint [70]. One disadvantage of checkpointing is that every host has to be rolled back to the last checkpoint even if only one host fails. Some existing distributed graph analytics systems have devised ways to avoid taking global checkpoints or rolling back live hosts, as described below.

GraphX [68] is built on Spark [71], and achieves fault tolerance by leveraging Resilient Distributed Datasets (RDD) 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 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). The table also shows the performance of D-Galois [18] and Gemini [73], which 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. In fault-free execution, Phoenix performance is the same as D-Galois, and the system does not have any observable overhead. Phoenix programs also run on average ∼ 4X faster than the same programs in Gemini. In addition, Phoenix programs are ∼ 24X faster, on average, than GraphX programs in the absence of faults while providing the same level of fault tolerance.

Imitator [65] is designed to tolerate a given number of faults: if n-way fault tolerance is desired, 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. As shown in [65], the ability to tolerate a single fault results in an overhead of ∼ 4% even if no faults occur; similarly, tolerating 3 faults increases the overhead to ∼ 8%. 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 [49] 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 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 overhead even when no failures occur. Phoenix shares this advantage with Zorro. While both Phoenix and Zorro incur overhead when failures happen, Phoenix is guaranteed to produce the correct answer unlike Zorro. Specifically, for self-stabilizing and locally-correcting algorithms (defined in Section 3), Zorro will yield the correct result if run till convergence. However, for globally-correcting algorithms, Zorro will yield an incorrect result.

The algorithms evaluated in Zorro [49] are self-stabilizing or locally-correcting. They were only run for a fixed number of total iterations when faults occur and a loss of precision was reported. However, if they were run until convergence, they would have been precise.
even if run till convergence. With respect to Zorro, our contributions are (1) defining the different classes of algorithms and (2) designing recovery mechanisms, specific to that class, that yield the correct result. For self-stabilizing algorithms, Phoenix uses a simpler recovery mechanism than Zorro. For locally-correcting algorithms, the recovery mechanisms of Phoenix are similar to that of Zorro. For globally-correcting algorithms, Phoenix uses a novel recovery approach with the help of the programmer.

CoRAL [64] is based on asynchronous checkpointing [38, 61]: hosts take local checkpoints independently, so no global coordination is required for checkpointing. Nonetheless, it incurs overhead during fault-free execution. CoRAL is applicable only to a subset of the algorithms that can be handled by Phoenix. Specifically, CoRAL can handle only self-stabilizing or locally-correcting algorithms (defined in Section 3). In CoRAL, like in other systems based on confined recovery [37, 57], the surviving hosts wait for the revived hosts to recover their lost state before continuing execution. Phoenix, in contrast, performs confined recovery without waiting for the revived 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 handles fail-stop faults in distributed-memory clusters, these algorithmic properties are easier to understand in the context of data corruption errors\(^5\) [59] (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 breadth-first search (bfs) computation is denoted by \(s_i\) and the source is node \(r\), then \(s_i(r) = 0\) and \(s_i(v) = \infty\) for all other nodes \(v\) in the graph. During the computation, these labels are updated by applying the relaxation operator to nodes in the graph to change 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 \(s_i\) 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 \(s_i\) to \(s_f\). Figure 2 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 \(s_i\) and \(s_f\) denote the initial and final states, respectively.

Definition 2. For \(s_m, s_n \in S\), \(s_n\) 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 \(s_0, s_1, \ldots, s_t\) such that for all \(0 \leq j < l\), \(s_j+1\) 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 \(s_i, \ldots, s_f\) from the initial state \(s_i\) to \(s_f\). Denote \(S_{GC}\) the set of globally consistent states in \(S\).

A state \(s_v\) is valid if there is a trajectory \(s_{v_1}, \ldots, s_f\) from \(s_{v_1}\) 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 [21, 28, 44, 53, 54]. 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 \(s_r\) 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 \(s_r \notin S_{GC}\). However, it is a valid state since applying the operator to the root node changes the labels of the neighbors of the root 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 \(s_{2z}\) where \(s_{2z}(v) = 0\) for all nodes \(v\). No other state is reachable from this state; in particular, \(s_f\) is not reachable, so \(s_{2z}\) 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 2, which shows a trajectory of states from the initial state \(s_i\) to the final state \(s_f\) in fault-free execution.

Every intermediate state in the trajectory between \(s_i\) and \(s_f\) is a globally consistent state. Checkpointing schemes save (some) globally consistent states on stable storage and recover from a fault by restoring the last globally consistent state saved. Figure 2 shows this pictorially: \(s_0\) is an invalid state and a checkpoint-restart scheme, denoted by CR, recovers by restoring the state to a globally consistent state.

\(5\)Data corruption means some bits in the data are flipped. In this paper, we consider fail-stop faults in distributed-memory and data corruption errors in shared-memory. We do not consider data corruption in distributed-memory.
The way in which Phoenix generates a valid state for recovery regardless of what those values are. Therefore, every values at the start of the computation, and the algorithm converges. This may produce a state that is not globally consistent, but the properties of the algorithm change of these labels. Hence, to recover from data corruption errors in shared-memory, re-initializing the label of each corrupted node is insufficient. After re-initializing the corrupted nodes, all nodes can re-compute their labels using only the current labels of their neighbors. This restores a valid state from which the work-efficient algorithm would reach the final state.

**Globally-consistent graph algorithms:** Only a globally consistent state is a valid state.

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

Betweenness centrality [27] is an example. Globally consistent snapshots [14] are required for recovery. In such cases, Phoenix may 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 shared-memory 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 \( s_m, s_n \in S \), \( s_n \) is said to be a successor state of \( s_m \) if a single BSP round transforms state \( s_m \) to state \( s_n \).

A trajectory is a sequence of states \( s_0, s_1, \ldots, s_l \) 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.
Algorithm 1: Greedy graph coloring (self-stabilizing).

Input: Partition \( G_s = (V_s, E_s) \) 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 \( v \in V_s \) do
5. \( \text{Init}(v, s) \)
6. repeat
7. foreach \( v \in V_s \) do
8. \( t(v) = s(v) \)
9. foreach \( v \in V_s \) do
10. foreach \( u \in \text{adj}(v) \text{ and } u < v \) do
11. \( nc = nc \cup \{t(u)\} \)
12. \( s(v) = \text{smallest } c \text{ such that } c \not\in nc \)
13. while Runtime.\text{Sync}(s) = \text{Failed} do
14. \( \text{Phoenix.\text{Recover}(Init, s)} \)
15. until \( \forall v \in V, s(v) = t(v) \);

Algorithm 2: Phoenix API for self-stabilizing algorithms.

1. Function \( \text{Recover}(\text{Init, s}) \):
2. if \( h \in \text{failed hosts } H_f \) then
3. foreach \( v \in V_h \) do
4. \( \text{Init}(v, s) \)

Figure 3. States of the graph in Figure 1a, treated as an undirected graph, during the execution of greedy graph coloring.

4 Fault Tolerance in Distributed-Memory

In this section, we describe how Phoenix performs confined recovery when a fault occurs without incurring overhead during fault-free execution. Section 4.1 presents Phoenix in the context of distributed execution of graph analytics applications. Sections 4.2, 4.3, and 4.4 illustrate Phoenix’s recovery mechanisms for the three different classes of algorithms introduced in Section 3. Section 4.5 summarizes the Phoenix API and discusses how programmers can use it when writing applications.

4.1 Overview

Fail-stop faults are detected and handled during the synchronization phase of a BSP round. Once a fail-stop fault is detected, the program passes the control to Phoenix using the Phoenix API. Phoenix reloads graph partitions from stable storage on the revived hosts that replace the failed hosts. Some of the nodes on these hosts may have proxies on surviving or 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 fault-free or normal execution. If the entire state can be recovered in this way, execution continues. However, if there are nodes for which no proxy exists on a healthy host, Phoenix restores the global state to a valid state using the approach described in detail in this section.

All algorithms presented use a generic interface called \( \text{Sync} \) to synchronize the state of all proxies of all nodes. Synchronization of different proxies of a node involves an \text{all-reduce} operation, and different systems support this in different ways. For example, the interface maps directly to Gluon’s [18] \text{Sync} interface used in D-Galos. In Gather-Apply-Scatter (GAS) models like PowerGraph [25], the interface can be implemented using gather and scatter. During normal execution, \text{Sync} only synchronizes proxies that have been updated. In contrast, proxies of all nodes are synchronized during Phoenix recovery. The \text{reduction} operation for the same field could be different during normal execution and Phoenix recovery (e.g., addition in normal execution corresponds to maximum during recovery). In addition, Gluon exploits the structural invariants of partitioning policies to avoid performing an \text{all-reduce} during normal execution, but we do not use this during Phoenix recovery. We omit these variations of \text{Sync} in the algorithms presented for the sake of simplicity.

Algorithms for each class 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 work-list, when used, is local to the host. The foreach loop denotes a parallel loop that can be executed on multiple threads, using fine-grained synchronization to update the local state. For example, the loop can be implemented using the do\_all construct in D-Galos with atomic updates to the local state. All algorithms use the generic interface, \text{Sync} 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 work-list. Without the explicit synchronization or Phoenix calls, these algorithms can be executed on shared-memory systems. \text{Sync} and \text{SyncW} fail when at least one host has crashed, and when that occurs, the algorithms call a generic API for Phoenix that recovers from the failure.

The Phoenix API calls \text{Sync} at the end of its recovery. In the algorithms presented, we push this \text{Sync} call to the user code and omit the loading of partitions on the hosts replacing the crashed hosts. If the failures cascade, i.e., if failures occur during recovery, then the \text{Sync} call at the end of Phoenix’s recovery will fail and Phoenix’s recovery is re-initiated after partitions are loaded on the hosts replacing the failed hosts.
It is clear that varying values of the k-core problem, shown in Algorithm 5. The k-core problem is to find the subgraphs of an undirected or symmetric graph in which every node has degree at least \( k \). Most k-core algorithms might have proxies on healthy hosts, which are recovered in the subsequent Sync call (line 14). This call also adds the proxies whose labels were recovered to the work-list. For all locally-correcting algorithms, the state is now valid because (i) the initial label of a node is a valid value (Equation 3) and (ii) the work-list ensures that all values lost will be recovered eventually. 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).

4.4 Globally-Correcting Graph Algorithms

We use data-driven breadth-first search (bfs), shown in Algorithm 3, to explain how locally-correcting algorithms are handled by Phoenix. Every node has a label that denotes its distance from the source, which is initialized to 0 for the source and \( \infty \) for every other node. 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 work-lists. Initially, only the source is in the work-list. If the neighbor’s distance changes when relaxed, then it is added to the work-list for the next round. The algorithm terminates when there are no nodes in the work-list on any host.

The program calls the Phoenix API (line 15) when SyncW fails. When a fault occurs, Phoenix uses the given function to initialize the labels of the proxies on the failed hosts and update the work-list. Some of the nodes on the failed host may have proxies on healthy hosts, which are recovered in the subsequent SyncW call (line 14). This call also adds the proxies whose labels were recovered to the work-list. For all locally-correcting algorithms, the state is now valid because (i) the initial label of a node is a valid value (Equation 3) and (ii) the work-list ensures that all values lost will be recovered eventually. Phoenix supports such confined recovery by providing a thin API defined in Algorithm 4.

Figure 4 illustrates this for the graph in Figure 1a. Hosts fail during the third round. CR would restore the state to that at the end of round 2, and this will need three more rounds to converge. Consider the nodes on the failed hosts that do not have proxies on the healthy hosts (shaded in red). Phoenix will initialize the distances of their proxies to \( \infty \) since none of them is the source. The incoming neighbors of these nodes that have proxies on the healthy hosts will be recovered by the subsequent SyncW call and added to the work-list. Phoenix will resume execution and converge in three more rounds.

4.4 Globally-Correcting Graph Algorithms

To illustrate how Phoenix handles globally-correcting graph algorithms, we use degree-decrementing, data-driven k-core (kcore), shown in Algorithm 5. The k-core problem is to find the sub-graphs of an undirected or symmetric graph in which each node has degree at least \( k \). Most k-core algorithms
execute by removing nodes with fewer than \( k \) neighbors in the graph since these nodes cannot be part of a \( k \)-core. The removal of these nodes lowers the degrees of other nodes, which may enable more nodes to be removed from the graph.

Since explicitly deleting nodes and edges from the graph is expensive, implementations usually just mark a node as dead and decrement the degree of its neighbors. Therefore, each node has two labels, a flag and a degree, that denote whether the node is dead or alive and the number of edges it has to other alive nodes in the graph. A work-list maintains the currently alive nodes. The algorithm is executed in rounds. In each round, every node in the work-list marks itself as dead if its degree is less than \( k \) or adds itself to the work-list for the next round otherwise. When a node is marked as dead, it decrements the degrees of its immediate neighbors. The algorithm terminates (global_termination) when no node is marked as dead in a round on any host.

To present the fault recovery strategy for this algorithm, we first note that the state is valid if and only if (i) the degree of each node is equal to the number of the edges it has to currently alive nodes and (ii) all the currently alive nodes are in the work-list. During algorithm execution, a node’s degree is updated only when its neighbor changes (its flag) from alive to dead. Hence, when a node is lost, re-initializing its flag and degree does not restore the state to a valid state: the degree of all nodes needs to be recalculated based on currently alive nodes. To do so, new functions are required to reinitialize the degree of a healthy node and recompute the degree of a node using only the current flags (alive or dead) of its neighbors (in Algorithm 5, ReInitW and ReComputeW).

Figure 5 shows state transitions after each round of execution for the graph in Figure 1a. Consider detection of a fault after two rounds. CR restores the state to that at the end of the first round. In contrast, Phoenix resets both the flag and the degree of proxies on failed hosts to the initial value and resets only the degree of the proxies on healthy hosts to the initial value. All proxies are added to the work-list in both. SyncW is called to recover the flags of nodes on failed hosts that have proxies on healthy hosts. After that, all the dead nodes decrement the degree of their outgoing-neighbors and all the alive nodes are added to the work-list. The subsequent SyncW will synchronize the degrees of the proxies and restore the state to a valid state. Algorithm execution then converges in a round.

For all globally-correcting algorithms, a node’s state is updated using the change of its neighbors’ state. In kcore, degree is updated using the change in flag, while in residual-based pagerank (pr), residual is updated using the change in rank. Due to this, re-initializing the labels of failed proxies is insufficient. Some labels of healthy proxies must also be

---

**Algorithm 5: Data-driven k-core (global-correcting).**

```plaintext
Input : Partition \( G_h = (V_h, E_h) \) of graph \( G = (V, E) \)
Output: A set of flags and degrees \( s(v) \) \( \forall v \in V \)
1. foreach \( u \in \text{outgoing}_\text{adj}(v) \) do
2. \( s(u).\text{degree} = s(u).\text{degree} - 1 \)
3. \( W_n = W_n \cup \{v\} \)
4. if \( s(v).\text{flag} == True \) then
5. \( \text{DecrementDegree}(v, s) \)
6. \( W_n = W_n \cup \{v\} \)
7. if \( \text{DecrementDegree}(v, s) \) then
8. \( W_n = W_n \cup \{v\} \)
9. \( \text{DecrementDegree}(v, s) \)
10. if \( W_n = W_n \cup \{v\} \) then
11. \( \text{DecrementDegree}(v, s) \)
12. \( s(v).\text{flag} = False \)
13. \( W_n = W_n \cup \{v\} \)
14. \( \text{DecrementDegree}(v, s) \)
15. \( W_n = W_n \cup \{v\} \)
16. if \( \text{DecrementDegree}(v, s) \) then
17. \( W_n = W_n \cup \{v\} \)
18. \( \text{DecrementDegree}(v, s) \)
19. \( W_n = W_n \cup \{v\} \)
20. foreach \( v \in V_h \) do
21. \( \text{InitW}(v, s, W_n) \)
22. repeat
23. \( W_o = W_n ; W_n = 0 \)
24. foreach \( v \in V_h \) do
25. \( \text{ComputeW}(v, s, W_n) \)
26. \( \text{ReComputeW}(v, s, W_n) \)
27. while \( \text{Runtime.SyncW}(s, W_n) == \text{Failed} \) do
28. \( \text{Phoenix.Recover}(\text{InitW, ReInitW, ReComputeW}, s, W_n) \)
29. until \text{global_termination};
```

**Algorithm 6: Phoenix API for globally-correcting algorithms.**

```plaintext
1. Function \( \text{ReInitW}(v, s, W_n) \):
2. if \( h \in \text{failed hosts} \) then
3. foreach \( v \in V_h \) do
4. \( \text{InitW}(v, s, W_n) \)
5. else
6. foreach \( v \in V_h \) do
7. \( \text{ReInitW}(v, s, W_n) \)
8. if \( \text{Runtime.SyncW}(s, W_n) == \text{Failed} \) then
9. \( \text{return} \)
10. \( W_o = W_n ; W_n = 0 \)
11. foreach \( v \in W_o \) do
12. \( \text{ReComputeW}(v, s, W_n) \)
```
reinitialized. In kcore and pr, only degree and residual are reinitialized, respectively, which ensures that the progress of flag and rank, respectively, on healthy proxies is not lost. The new state must be recomputed using only the current state. In kcore and pr, degree and residual must be recomputed using only the current flag and rank, respectively. Given re-initialization and re-computation functions, the Phoenix API, defined in Algorithm 6, supports such confined recovery.

4.5 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.

Instrumenting self-stabilizing and locally-correcting algorithms to enable Phoenix is straightforward because the initialization 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. These functions can be written by considering algorithm execution in shared-memory where node labels are corrupted, which is much simpler than considering algorithm execution in distributed-memory with synchronization of proxies. This typically involves writing a naive topology-driven algorithm instead of the data-driven, work-efficient algorithm used by default. This can be learned from the pattern in Algorithm 5.

While we described Phoenix using BSP-style rounds, the recovery mechanisms of Phoenix do not assume that the execution preceding or following the recovery is BSP-style. Phoenix recovery itself is BSP-style and involves at least one computation and communication round. Phoenix restores the state to a valid state, regardless of whether the prior updates to the state were bulk-synchronous or asynchronous.

The Phoenix API replaces the checkpoint and restore functions in Checkpoint-Restart (CR) systems, and it can be incorporated into existing synchronous or asynchronous distributed graph analytics programming models or runtimes. Phoenix can also be combined with existing checkpointing techniques that take globally consistent or locally consistent snapshots [37, 64]. In this case, the failed hosts initialize their labels from the saved checkpoint instead of calling the initialization function, thereby leading to faster recovery of nodes for which no proxy exists on a healthy host.

5 Experimental Evaluation

D-Galois [3, 18] is the state-of-the-art distributed graph analytics system, but it does not support fault tolerance. The Phoenix fault tolerance approach presented in this paper was implemented in D-Galois (adding ~ 500 lines of code); for brevity, the resulting system is called Phoenix too. We also implemented a checkpoint-restart technique in D-Galois, and this is termed CR. We compare these with GraphX [68] and Gemini [73], which are fault tolerant and fault intolerant distributed graph analytics systems respectively. Section 5.1 describes the experimental setup, including implementation details of Phoenix and CR. Sections 5.2 and 5.3 present the results in the presence and absence of faults respectively.

5.1 Experimental Setup

Experiments were conducted on the Stampede [60] and Wrangler clusters at the Texas Advanced Computing Center [5]. The configurations used are listed in Table 2. We used Wrangler to compare Phoenix with D-Galois, Gemini, and GraphX (GraphX cannot be installed on Stampede). All other experiments were conducted on 128 KNL hosts of Stampede.

Table 3 specifies the input graphs: wdc12 [40, 41] and clueweb12 [7, 8, 48] are the largest publicly available web-crawls; kron30 [34] and rmat28 [13] are randomized synthetic scale-free graphs (we used weights of 0.57, 0.19, 0.19, and 0.05, as suggested by graph500 [1]); twitter50 [32] is a social network graph; amazon [26, 39] is the largest publicly available bipartite graph. Smaller inputs are used for comparison with GraphX because it exhausts memory quickly.

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, Phoenix and CR), 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 Gemini implementations of the same benchmarks (they use self-stabilizing algorithm for pr); they do not have kcore or cf. GraphX does not use edge-weights in its sssp, so Gemini and Phoenix also do not use it when compared with it. We present cf results only with the amazon graph because it requires a bipartite graph. The tolerance used for cf is $10^{-9}$. The source node for sssp is the maximum out-degree node. The tolerance for pr is $10^{-7}$ for kron30, $10^{-4}$ for clueweb12, and $10^{-3}$ for wdc12. The $k$ in kcore is 100. All benchmarks are run until convergence. We present the mean of 3 runs.

We implemented Phoenix recovery for each benchmark depending on the class of its algorithm. Most of the effort was spent for kcore and pr, which are globally-correcting algorithms. They took an estimated day’s worth of programmer time with ~ 150 lines changed or added (original code was ~ 300 lines). The rest of the benchmarks were self-stabilizing or locally-correcting algorithms, so they took little time and needed only ~ 30 lines of changes or additions.

CR only checkpoints the graph node labels and not the graph topology (which is read-only). Since D-Galois is bulk-synchronous parallel (BSP), CR takes a globally consistent
Table 2. Configuration of clusters.

<table>
<thead>
<tr>
<th>App</th>
<th>Input</th>
<th>R</th>
<th>Stampede</th>
<th>Wrangler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Phoenix CR-50</td>
<td>Phoenix CR-50</td>
</tr>
<tr>
<td>cc</td>
<td>kron30</td>
<td>Ph</td>
<td>20.1</td>
<td>52.5</td>
</tr>
<tr>
<td></td>
<td>clueweb12</td>
<td>Ph</td>
<td>27.1</td>
<td>72.8</td>
</tr>
<tr>
<td></td>
<td>wdc12</td>
<td>Ph</td>
<td>24.1</td>
<td>56.5</td>
</tr>
<tr>
<td>kcore</td>
<td>kron30</td>
<td>Ph</td>
<td>25.3</td>
<td>95.7</td>
</tr>
<tr>
<td></td>
<td>clueweb12</td>
<td>Ph</td>
<td>24.3</td>
<td>95.7</td>
</tr>
<tr>
<td></td>
<td>wdc12</td>
<td>Ph</td>
<td>27.6</td>
<td>56.8</td>
</tr>
<tr>
<td>pr</td>
<td>kron30</td>
<td>Ph</td>
<td>12.0</td>
<td>256.1</td>
</tr>
<tr>
<td></td>
<td>clueweb12</td>
<td>Ph</td>
<td>29.0</td>
<td>326.8</td>
</tr>
<tr>
<td></td>
<td>wdc12</td>
<td>Ph</td>
<td>74.7</td>
<td>81.4</td>
</tr>
<tr>
<td>sssp</td>
<td>kron30</td>
<td>Ph</td>
<td>11.0</td>
<td>42.3</td>
</tr>
<tr>
<td></td>
<td>clueweb12</td>
<td>Ph</td>
<td>85.1</td>
<td>87.6</td>
</tr>
<tr>
<td></td>
<td>wdc12</td>
<td>Ph</td>
<td>77.7</td>
<td>118.3</td>
</tr>
<tr>
<td>cf</td>
<td>amazon</td>
<td>Ph</td>
<td>908.2</td>
<td>342.2</td>
</tr>
</tbody>
</table>

Table 3. Inputs and their key properties.

<table>
<thead>
<tr>
<th>Input</th>
<th></th>
<th>v</th>
<th></th>
<th>E</th>
<th></th>
<th>E/V</th>
</tr>
</thead>
<tbody>
<tr>
<td>amazon</td>
<td>tw50</td>
<td>rmat28</td>
<td>kron30</td>
<td>clueweb12</td>
<td>wdc12</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>31M</td>
<td>51M</td>
<td>268M</td>
<td>1,073M</td>
<td>978M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>82.5M</td>
<td>1,963M</td>
<td>4,295M</td>
<td>10,791M</td>
<td>42,574M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>40</td>
<td>100</td>
<td>300</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2500</td>
<td>7799</td>
<td>352</td>
<td>320</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>40</td>
<td>100</td>
<td>300</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>44</td>
<td>55</td>
<td>77</td>
<td>99</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25,366</td>
<td>3.5M</td>
<td>0.3M</td>
<td>3.2M</td>
<td>75M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.7 38 16 16 44 36</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>986</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>325</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.2 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 ~ 24x and ~ 4x 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 execution, and Gemini and D-Galois do not tolerate faults, so we do not evaluate them with different fault scenarios.

For the rest of our experiments, we assume that the graph is already partitioned, and we load graph partitions directly from disk. Table 4 compares the fault-free performance of Phoenix and CR on Stampede (D-Galois is omitted as it is identical to Phoenix). We present the total time and execution time separately. Execution time includes the algorithm time and checkpointing time, while total time includes graph loading time and execution time. Phoenix has 0% overhead in the absence of faults. We omit CR-5 from the table since its overhead is too high. CR-5, CR-50, CR-500 have an execution time overhead of 542%, 31%, and 8%, respectively, on the average, over Phoenix in the absence of faults.

5.3 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). These fault scenarios are used for all results presented in this section, including Figures 6, 7, 8, 9, and 10. 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.
Figure 7. Execution time (s) of Phoenix (Ph) and CR with different fault scenarios.
Figure 8. Overheads of different fault scenarios with Phoenix and CR over fault-free execution.

Figure 9. Execution time (s) of Phoenix (Ph) and CR with different fault scenarios for cf.

Figure 10. Execution time (s) of Phoenix (Ph), CR, and Checkpointing with Phoenix (CPh): 64 faults for wdc12.

For each benchmark and input, Figure 6 shows the total time overhead of faults for Phoenix and CR-50 over fault-free execution of Phoenix using a box-plot\(^3\) 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 from 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 7 and 9 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 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 [56] such as up to 16 hosts failing, Phoenix outperforms CR-50 and CR-500, except for kcore on clueweb12. When 4 hosts fail, the mean execution time overhead over fault-free execution of Phoenix, CR-50, and CR-500 is \(\sim 14\%\), \(\sim 48.5\%\), and \(\sim 59\%\), respectively. CR-500 is worse than CR-50 in 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 \(\sim 20.8\%\), and it increases to \(\sim 44\%\) when 64 hosts fail. In contrast, the point of failure does not increase the overhead of CR 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 8 shows the % increase

\(^3\) 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.
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-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.

**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 10 shows the evaluation of CPh in case of 64 hosts crashing at different points of execution for all benchmarks on wdc12. CPh is similar to or faster than CR-50 in all cases, as expected.

6 Related Work

**System-level checkpointing**: Many systems [6, 17, 22, 24, 31, 36, 38, 43, 50, 52] implement checkpointing or message logging to rollback and recover from faults [23]. Although these system-level mechanisms are transparent to the programmer, exploiting application properties like Phoenix does can reduce the execution time with or without faults.

**Application-level checkpointing**: Instead of checkpointing the entire state or memory footprint of the application like in system-level checkpoint-restart approaches, many checkpointing approaches [42, 62, 72] allow the programmer to instrument their code to checkpoint only the live application state; Bronevetsky et al. [10–12] automate this using a compiler. Some of these approaches checkpoint to memory [62, 72] or use a combination of memory, local disk, and network filesystem [42]. In Section 5, we showed that in the presence of faults, Phoenix generally outperforms even a checkpoint-restart (CR) approach that checkpoints only node labels (and not the graph topology) to the network filesystem. Although CR can be further improved by using in-memory or multi-level checkpointing, there will always be overhead even in fault-free execution, unlike in Phoenix.

**Fault tolerant data-parallel systems**: Some data-parallel systems [20, 30, 46, 71] save sufficient information transparently to re-execute computation and restore lost data when faults occur. Schelter et al. [55] 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. Phoenix, in contrast, does not require the user to define new functions for these classes of algorithms, and it supports globally-correcting algorithms with user-defined functions. Moreover, Phoenix does not store any additional state information in memory or stable storage.

**Fault tolerant graph-analytical systems**: Many systems for distributed graph analytics [2, 25, 35, 37, 46, 47, 49, 57, 58, 64, 65, 68] support fault tolerance transparently. Greft [47] is the only one that tolerates Byzantine (data corruption) faults; the rest tolerate only fail-stop faults. Section 2.2 contrasts Phoenix with other fail-stop fault tolerant systems in detail.

7 Conclusions

We presented Phoenix, a communication and synchronization 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 mechanisms used to adjust the 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 mechanisms used to adjust the 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 mechanisms used to adjust the state after failure for each class. Experiments showed that D-Galois augmented with Phoenix has no overhead during fault-free execution and outperforms GraphX by an order of magnitude. Furthermore, Phoenix almost always outperforms traditional checkpoint-restart recovery in the presence of faults because it performs confined recovery such that surviving hosts do not lose their progress. We believe Phoenix-style recovery can be used in other application domains to tolerate faults without fault-free execution overheads.

Acknowledgments

This research was supported by NSF grants 1337217, 1337281, 1406355, 1618425, 1725322 and by DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563. This work used the XSEDE grant ACI-1548562 through allocation TG-CIE170005. We used the Stampede system at Texas Advanced Computing Center, University of Texas at Austin.
References


