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 ∼ 24× 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:
Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2019. Phoenix: A Substrate for Resilient Distributed Graph Analytics. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3297858.3304056

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].

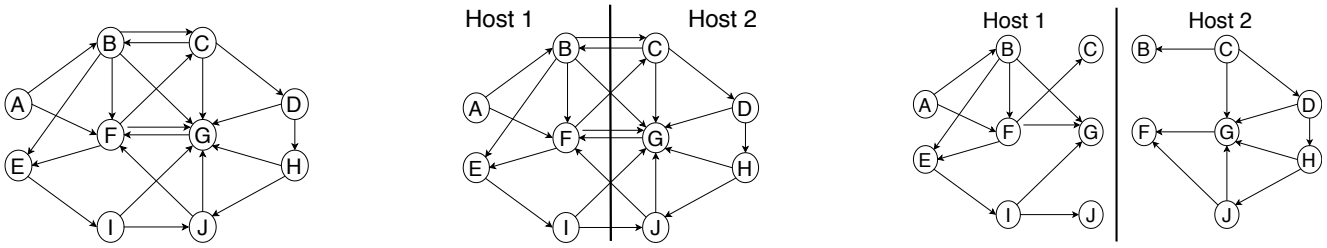
Fault tolerance is an important concern for long-running graph analytics applications on large clusters. Some state-of-the-art high-performance graph analytics systems such as D-Galois [18] and Gemini [73] do not address fault tolerance; since the mean time between failures in medium-sized clusters is 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

*Both authors contributed equally.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304056



(a) Original graph. **(b)** Node partitioning. **(c)** Edge partitioning.

Figure 1. An example of partitioning a graph for two hosts in two different ways.

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 continues execution, thus performing confined recovery in which surviving hosts 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 distributed 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 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 work-lists 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 ∼ 4× faster than the same programs in Gemini. *In addition, Phoenix programs are* ∼ 24× *faster, on average, than GraphX programs in the absence of faults while providing the same level of fault tolerance.*

Table 1. Fault-free execution on 32 hosts of Wrangler.

App	Input	Total Time (s)				Phoenix Speedup	
		GraphX	Gemini	D-Galois	Phoenix	GraphX	Gemini
cc	twitter50	110.8	29.0	8.2	8.2	13.5	3.5
	rmat28	166.5	80.0	20.6	20.6	8.1	3.9
	kron30	1538.2	347.1	56.2	56.2	27.4	6.2
pr	twitter50	2111.6	75.1	31.6	31.6	66.7	2.4
	rmat28	5355.5	180.9	47.7	47.7	112.3	3.8
	kron30	797.6	426.4	78.9	78.9	10.1	5.4
sssp	twitter50	153.1	24.2	8.5	8.5	18.0	2.8
	rmat28	158.1	77.7	11.1	11.1	14.3	7.0
	kron30	1893.2	346.7	46.0	46.0	41.2	7.5

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[1]. However, for globally-correcting algorithms, Zorro will yield an incorrect result

[1]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 is 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[2] [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 $s_0, s_1, ..., s_l$ 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 $s_i, ..., s_g$ from the initial state s_i to s_g. Denote S_{GC} the subset of globally consistent states in S.*

A state s_v is valid *if there is a trajectory $s_v, ..., s_f$ from s_v 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,

$$S_{GC} \subseteq S_V \subseteq S \tag{1}$$

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_x in which the label of the root is 0, the labels of its immediate neighbors are 2, and all other node labels are ∞. This is not a state reachable from s_i, so $s_x \notin S_{GC}$. 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_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, so 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 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_b is an invalid state and a checkpoint-restart scheme, denoted by CR, recovers by restoring the state to a globally consistent state. In

[2]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.

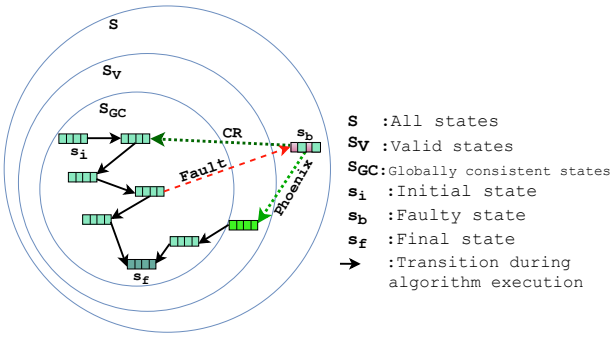


Figure 2. States during algorithm execution and recovery.

contrast, as illustrated in Figure 2, *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 in which Phoenix generates a valid state for recovery depends on the structure of the algorithm. There are four cases, discussed below in increasing order of complexity.

Self-stabilizing graph algorithms: All states are valid.

$$S_{GC} \subset S_V = S \tag{2}$$

Examples are topology-driven algorithms for collaborative filtering using stochastic gradient descent (cf), belief-propagation, pull-style pagerank, and pull-style graph coloring. In these algorithms, 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 no correction of the state is required when a data corruption error is detected in the shared-memory case.

Locally-correcting graph algorithms: The set of valid states is 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.

$$\begin{aligned} S_{GC} \subset S_V \subset S \\ S_V = L_1 \times L_2 \times \ldots \times L_N \end{aligned} \tag{3}$$

Some examples are breadth-first search (bfs), single-source shortest path (sssp), connected components using label propagation (cc), 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 label of each corrupted node to ∞; 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.

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

$$S_{GC} = S_V \subset S \tag{4}$$

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_h = (V_h, E_h) of graph G = (V, E)
Output : A set of colors s(v) ∀v ∈ V
Let t(v) ∀v ∈ V be a set of temporary colors
Function Init(v, s):
    s(v) = random color
foreach v ∈ V_h do
    Init(v, s)
repeat
    foreach v ∈ V_h do
        t(v) = s(v)
    foreach v ∈ V_h do
        foreach u ∈ adj(v) and u < v do
            nc = nc ∪ {t(u)}
        s(v) = smallest c such that c ∉ nc
    while Runtime.Sync(s) == Failed do
        Phoenix.Recover(Init, s)
until ∀v ∈ V, s(v) = t(v);
```

Algorithm 2: Phoenix API for self-stabilizing algorithms.

```
Function Recover(Init, s):
    if h ∈ failed hosts H_f then
        foreach v ∈ V_h do
            Init(v, s)
```

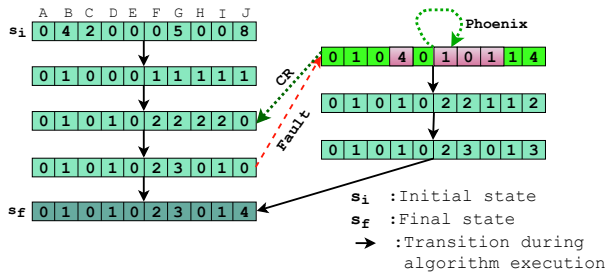


Figure 3. States of the graph in Figure 1a, treated as an undirected graph, during the execution of greedy graph coloring.

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 Sync to synchronize the state of all proxies of all nodes. Synchronization of different proxies of a node involves an *all-reduce* operation, and different systems support this in different ways. For example, the interface maps directly to Gluon's [18] Sync interface used in D-Galois. In Gather-Apply-Scatter (GAS) models like PowerGraph [25], the interface can be implemented using gather and scatter. During normal execution, Sync only synchronizes proxies that have been updated. In contrast, proxies of all nodes are synchronized during Phoenix recovery. The *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 *all-reduce* during normal execution, but we do not use this during Phoenix recovery. We omit these variations of 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-Galois with atomic updates to the local state. All algorithms use the generic interface, Sync or SyncW, to synchronize the state of all proxies; SyncW is similar to 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. Sync and 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 Sync at the end of its recovery. In the algorithms presented, we push this 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 Sync called 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.

Algorithm 3: Breadth first search (locally-correcting).

```
   Input  : Partition G_h = (V_h, E_h) of graph G = (V, E)
   Input  : Source v_s
   Output : A set of distances s(v) ∀v ∈ V
1  Function InitW(v, s, W_n):
2      if v == v_s then
3          s(v) = 0 ; W_n = W_n ∪ {v}
4      else
5          s(v) = ∞
6  foreach v ∈ V_h do
7      InitW(v, s, W_n)
8  repeat
9      W_o = W_n ; W_n = ∅
10     foreach v ∈ W_o do
11         foreach u ∈ outgoing_adj(v) do
12             if s(u) > s(v) + 1 then
13                 s(u) = s(v) + 1 ; W_n = W_n ∪ {u}
14     while Runtime.SyncW(s, W_n) == Failed do
15         Phoenix.Recover(InitW, s, W_n)
16 until global_termination;
```

Algorithm 4: Phoenix API for locally-correcting algorithms.

```
1  Function Recover(InitW, s, W_n):
2      if h ∈ failed hosts H_f then
3          foreach v ∈ V_h do
4              InitW(v, s, W_n)
```

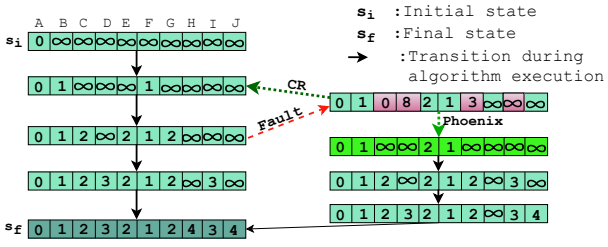


Figure 4. State of the graph in Figure 1a during the execution of data-driven breadth first search (locally-correcting).

4.2 Self-Stabilizing Graph Algorithms

To illustrate how self-stabilizing algorithms are handled by Phoenix, Algorithm 1 shows greedy graph coloring 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 function. 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 color assignments do not change in a round.

The programmer calls the Phoenix API (line 14) if Sync fails. Figure 3 shows possible state transitions after each round of algorithm execution for the graph in Figure 1a. In this figure, colors are encoded as integers. Suppose that faults are detected in the fourth round when performing the Sync operation. To recover, Phoenix will initialize the colors of the proxies of the nodes on each 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). The nodes which do not have any proxies on healthy hosts are shaded red in Figure 3. 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).

4.3 Locally-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 ∞ 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 ∞ 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

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

```
Input  : Partition G_h = (V_h, E_h) of graph G = (V, E)
Input  : k
Output : A set of flags and degrees s(v) ∀v ∈ V
1  Function DecrementDegree(v, s):
2      foreach u ∈ outgoing_adj(v) do
3          s(u).degree = s(u).degree − 1
4  Function ReInitW(v, s, W_n):
5      s(v).degree = |outgoing_adj(v)|
6      W_n = W_n ∪ {v}
7  Function InitW(v, s, W_n):
8      s(v).flag = True
9      ReInitW(v, s, W_n)
10 Function ComputeW(v, s, W_n):
11     if s(v).degree < k then
12         s(v).flag = False
13         DecrementDegree(v, s)
14     else
15         W_n = W_n ∪ {v}
16 Function ReComputeW(v, s, W_n):
17     if s(v).flag = False then
18         DecrementDegree(v, s)
19     else
20         W_n = W_n ∪ {v}
21 foreach v ∈ V_h do
22     InitW(v, s, W_n)
23 repeat
24     W_o = W_n ; W_n = ∅
25     foreach v ∈ W_o do
26         ComputeW(v, s, W_n)
27     while Runtime.SyncW(s, W_n) == Failed do
28         Phoenix.Recover(InitW, ReInitW, ReComputeW, s, W_n)
29 until global_termination;
```

Algorithm 6: Phoenix API for globally-correcting algorithms.

```
1  Function Recover(InitW, ReInitW, ReComputeW, s, W_n):
2      if h ∈ failed hosts H_f then
3          foreach v ∈ V_h do
4              InitW(v, s, W_n)
5      else
6          foreach v ∈ V_h do
7              ReInitW(v, s, W_n)
8      if Runtime.SyncW(s, W_n) == Failed then
9          return
10     W_o = W_n ; W_n = ∅
11     foreach v ∈ W_o do
12         ReComputeW(v, s, W_n)
```

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

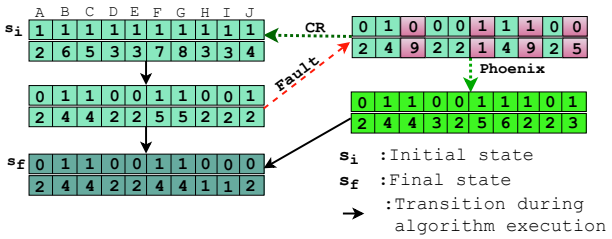


Figure 5. State of the graph in Figure 1a, treated as undirected, during the execution of data-driven k-core ($k = 4$).

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

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 straight-forward 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 programming 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.

	Stampede	Wrangler
NIC	Omni-path	Infiniband
Machine	Intel Xeon Phi KNL	Intel Xeon Haswell
No. of hosts	128	32
Threads per host	272	48
Memory	96GB DDR4	128GB DDR4
Compiler	g++ 7.1	g++ 4.9.3

Table 3. Inputs and their key properties.

	amazon	twitter50	rmat28	kron30	clueweb12	wdc12
$\|V\|$	31M	51M	268M	1,073M	978M	3,563M
$\|E\|$	82.5M	1,963M	4,295M	10,791M	42,574M	128,736M
$\|E\|/\|V\|$	2.7	38	16	16	44	36
max D_{out}	44,557	779,958	4M	3.2M	7,447	55,931
max D_{in}	25,366	3.5M	0.3M	3.2M	75M	95M
Size on Disk (GB)	1.2	16	35	136	325	986

Table 4. Fault-free execution of Phoenix and CR (R stands for the number of BSP-style rounds).

App	Input	R	Total Time (s)			Execution Time (s)		
			Phoenix	CR-50	CR-500	Phoenix	CR-50	CR-500
cc	kron30	7	52.1	52.8	62.6	2.5	3.9	5.5
	clueweb12	24	73.1	72.8	78.5	12.6	14.8	17.4
	wdc12	401	303.8	356.5	306.1	90.5	146.0	95.5
kcore	kron30	6	89.0	95.7	125.3	2.7	5.7	16.9
	clueweb12	680	254.5	309.3	266.9	166.4	221.7	177.6
	wdc12	270	563.6	625.7	573.0	269.5	334.2	279.5
pr	kron30	120	245.3	256.1	269.6	210.0	219.8	220.6
	clueweb12	570	290.8	326.8	303.0	243.7	280.6	249.8
	wdc12	747	871.4	1012.8	893.2	732.2	875.1	748.4
sssp	kron30	11	40.2	42.3	47.0	3.1	4.3	5.8
	clueweb12	200	85.1	87.6	85.3	26.0	32.8	30.5
	wdc12	3779	777.8	1183.7	822.2	620.1	1020.2	659.2
cf	amazon	908	266.1	342.2	273.3	254.1	330.5	262.3

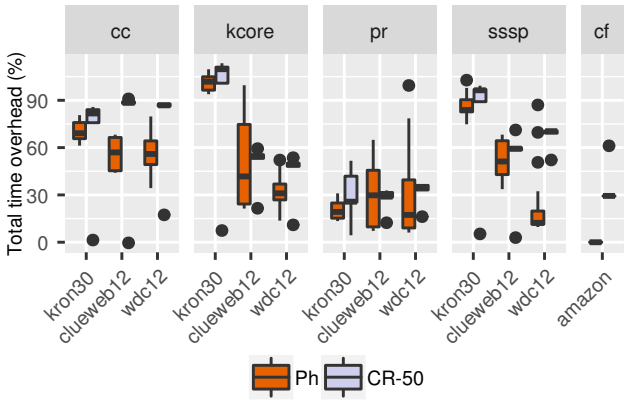


Figure 6. Overheads in total time (%) of different fault scenarios with Phoenix and CR over fault-free execution.

snapshot soon after bulk-synchronization and checkpoints it to the Lustre network filesystem [4] using stripe count 22 and stripe size 2 MB (Asynchronous checkpointing techniques like CoRAL [64] that do not take a globally consistent snapshot cannot be used because it is not applicable for all benchmarks we evaluate). The periodicity of checkpointing can be specified at runtime. We evaluated checkpointing after every 5, 50, and 500 rounds of BSP-style execution, and these are called CR-5, CR-50, and CR-500 respectively.

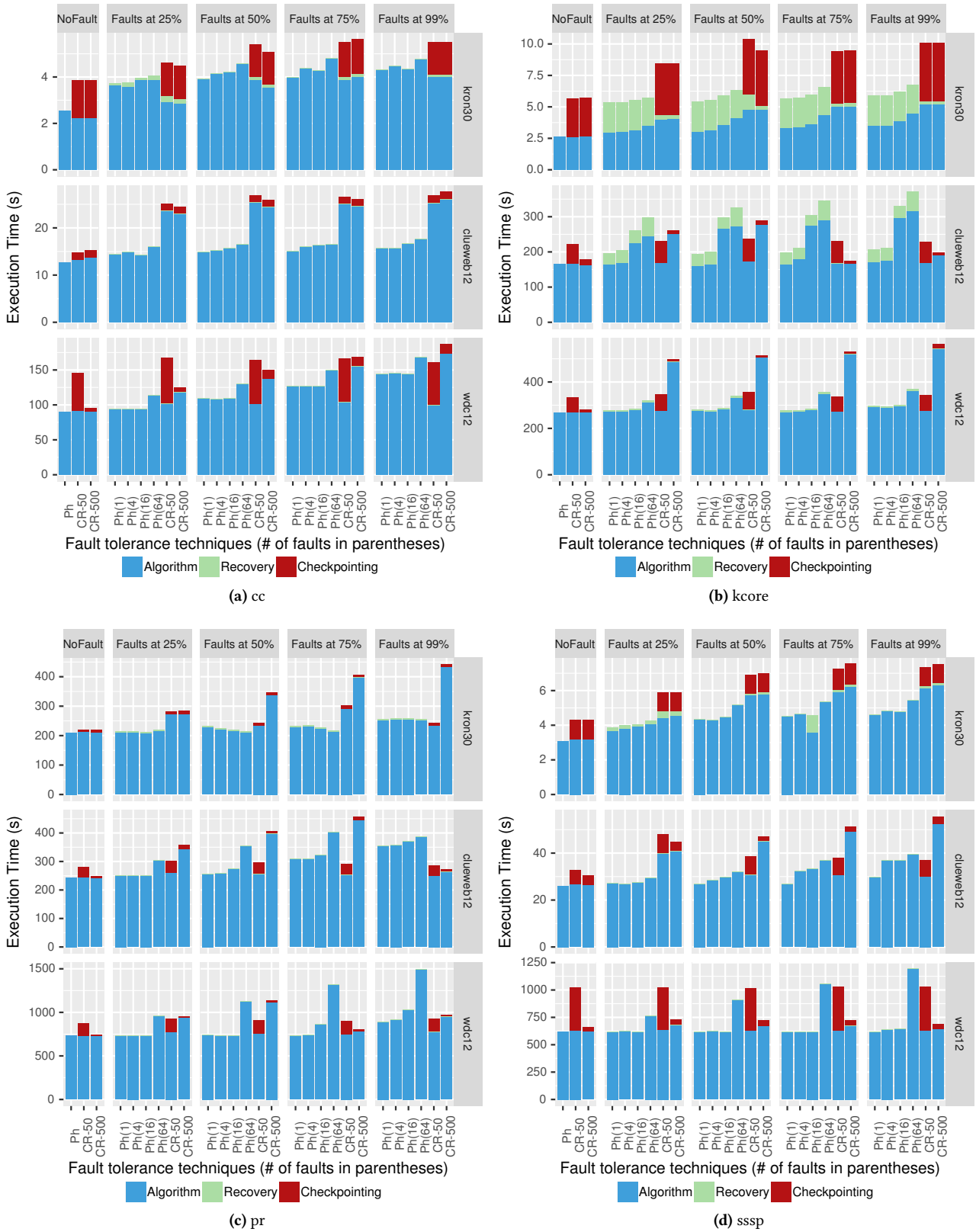
5.2 Fault-free performance

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* ∼ 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 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.



(a) cc **(b)** kcore **(c)** pr **(d)** sssp

Figure 7. Execution time (s) of Phoenix (Ph) and CR with different fault scenarios.

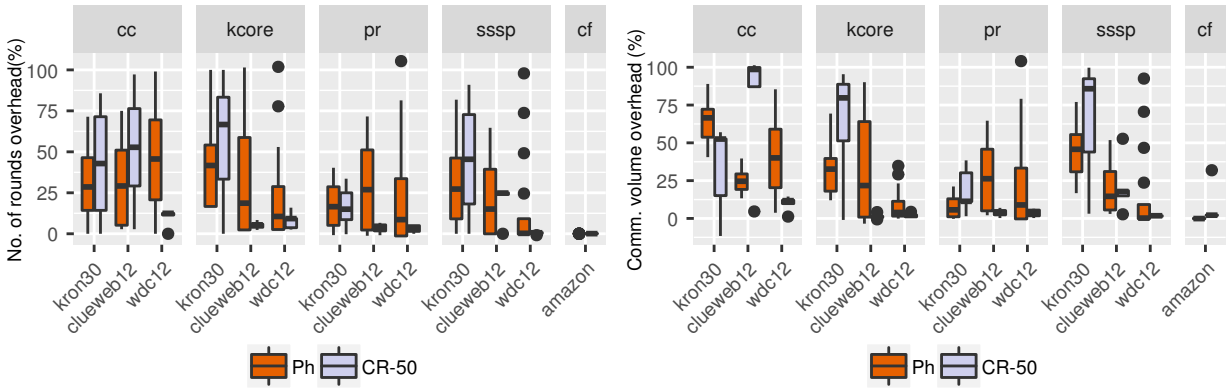


Figure 8. Overheads of different fault scenarios with Phoenix and CR over fault-free execution.

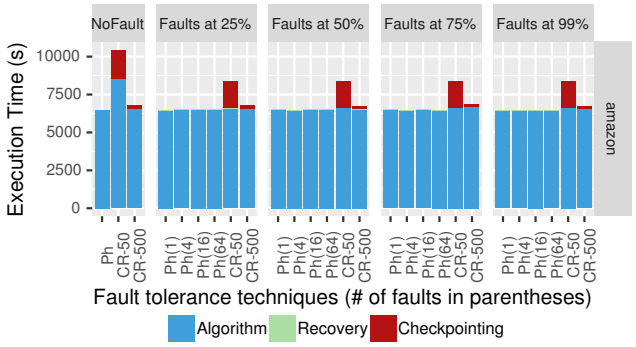


Figure 9. Execution time (s) of Phoenix (Ph) and CR with different fault scenarios for cf.

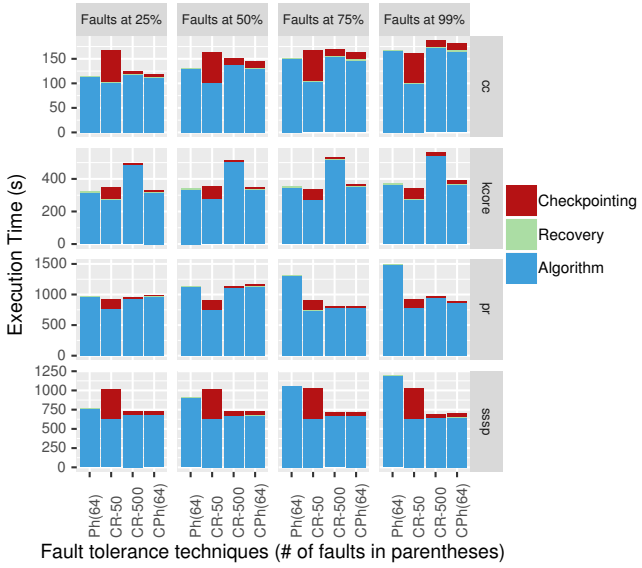


Figure 10. Execution time (s) of Phoenix (Ph), CR, and Checkpointing with Phoenix (CPh): 64 faults for wdc12.

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 occurs 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 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 [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 ~14%, ~48.5%, and ~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 ~20.8%, and it increases to ~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-500 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.

Fault tolerant algorithms: Algorithm-Based Fault Tolerance (ABFT) approaches [9, 15, 16, 19, 29, 66, 67, 69] have modified several computational science algorithms to tolerate faults without checkpointing. In a similar vein, many iterative solvers [28, 44, 54] have used self-stabilization [21] to tolerate failures. Sao et al. [53] build on this to design and implement a self-correcting topology-driven connected components algorithm. Some of these algorithms detect and recover from data corruptions (transient soft faults) in shared-memory systems. Phoenix generalizes the concept of self-stabilizing and self-correcting algorithms as explained in Section 3. In addition, Phoenix provides a simple API to implementing such algorithms on distributed-memory as explained in Section 4.

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 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

[1] 2010. Graph 500 Benchmarks. http://www.graph500.org

[2] 2013. Apache Giraph. http://giraph.apache.org/

[3] 2018. The Galois System. http://iss.ices.utexas.edu/?p=projects/galois

[4] 2018. Lustre File System. http://lustre.org/

[5] 2018. Texas Advanced Computing Center (TACC), The University of Texas at Austin. https://www.tacc.utexas.edu/

[6] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. 2004. Adaptive Incremental Checkpointing for Massively Parallel Systems. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS '04)*. ACM, New York, NY, USA, 277–286. https://doi.org/10.1145/1006209.1006248

[7] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[8] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[9] George Bosilca, Remi Delmas, Jack Dongarra, and Julien Langou. 2009. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel and Distrib. Comput.* 69, 4 (2009), 410 – 416. https://doi.org/10.1016/j.jpdc.2008.12.002

[10] Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. 2006. Recent Advances in Checkpoint/Recovery Systems. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*. IEEE Computer Society, Washington, DC, USA, 282–282. http://dl.acm.org/citation.cfm?id=1898699.1898802

[11] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. 2003. Automated Application-level Checkpointing of MPI Programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*. ACM, New York, NY, USA, 84–94. https://doi.org/10.1145/781498.781513

[12] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. 2004. C3: A System for Automating Application-Level Checkpointing of MPI Programs. 357–373. https://doi.org/10.1007/978-3-540-24644-2_23

[13] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. *R-MAT: A Recursive Model for Graph Mining*. 442–446. https://doi.org/10.1137/1.9781611972740.43

[14] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75. https://doi.org/10.1145/214451.214456

[15] Zizhong Chen. 2013. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 167–176. https://doi.org/10.1145/2442516.2442533

[16] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. 2005. Fault Tolerant High Performance Computing by a Coding Approach. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065944.1065973

[17] Ge-Ming Chiu and Cheng-Ru Young. 1996. Efficient Rollback-Recovery Technique in Distributed Computing Systems. *IEEE Trans. Parallel Distrib. Syst.* 7, 6 (June 1996), 565–577. https://doi.org/10.1109/71.506695

[18] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 752–768. https://doi.org/10.1145/3192366.3192404

[19] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. 2011. High Performance Linpack Benchmark: A Fault Tolerant Implementation Without Checkpointing. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 162–171. https://doi.org/10.1145/1995896.1995923

[20] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: simplified data processing on large clusters. In *OSDI'04*.

[21] Edsger W. Dijkstra. 1974. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM* 17, 11 (Nov. 1974), 643–644. https://doi.org/10.1145/361179.361202

[22] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. 1992. Manetho: Transparent Roll Back-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Trans. Comput.* 41, 5 (May 1992), 526–531. https://doi.org/10.1109/12.142678

[23] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.* 34, 3 (Sept. 2002), 375–408. https://doi.org/10.1145/568522.568525

[24] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. 2010. Distributed Diskless Checkpoint for Large Scale Systems. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. 63–72. https://doi.org/10.1109/CCGRID.2010.40

[25] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 17–30. http://dl.acm.org/citation.cfm?id=2387880.2387883

[26] Ruining He and Julian McAuley. 2016. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. In *Proceedings of the 25th International Conference on World Wide Web (WWW '16)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 507–517. https://doi.org/10.1145/2872427.2883037

[27] Loc Hoang, Matteo Pontecorvi, Roshan Dathathri, Gurbinder Gill, Bozhi You, Keshav Pingali, and Vijaya Ramachandran. 2019. A Round-Efficient Distributed Betweenness Centrality Algorithm. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'19) (PPoPP)*. 15. http://doi.acm.org/10.1145/3293883.3295729

[28] Mark Frederick Hoemmen and Michael Allen Heroux. 2011. *Fault-tolerant iterative methods.* Technical Report. Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States).

[29] Kuang-Hua Huang and Abraham. 1984. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Trans. Comput.* C-33, 6 (June 1984), 518–528. https://doi.org/10.1109/TC.1984.1676475

[30] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, New York, NY, USA, 59–72. https://doi.org/10.1145/1272996.1273005

[31] R. Koo and S. Toueg. 1987. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering* SE-13, 1 (Jan 1987), 23–31. https://doi.org/10.1109/TSE.1987.232562

[32] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[33] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2016. Parallel Graph Analytics. *Commun. ACM* 59, 5 (April 2016), 78–87. https:

//doi.org/10.1145/2901919

[34] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker Graphs: An Approach to Modeling Networks. *J. Mach. Learn. Res.* 11 (March 2010), 985–1042. http://dl.acm.org/citation.cfm?id=1756006.1756039

[35] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (April 2012), 716–727. https://doi.org/10.14778/2212351.2212354

[36] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. 2000. Exploring Failure Transparency and the Limits of Generic Recovery. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4 (OSDI'00)*. USENIX Association, Berkeley, CA, USA, Article 20. http://dl.acm.org/citation.cfm?id=1251229.1251249

[37] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD Intl Conf. on Management of Data (SIGMOD '10)*. 135–146. https://doi.org/10.1145/1807167.1807184

[38] D. Manivannan and M. Singhal. 1999. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (July 1999), 703–713. https://doi.org/10.1109/71.780865

[39] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. 2015. Image-Based Recommendations on Styles and Substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '15)*. ACM, New York, NY, USA, 43–52. https://doi.org/10.1145/2766462.2767755

[40] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2012. Web Data Commons - Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph/

[41] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2014. Graph Structure in the Web — Revisited: A Trick of the Heavy Tail. In *Proceedings of the 23rd International Conference on World Wide Web (WWW '14 Companion)*. ACM, New York, NY, USA, 427–432. https://doi.org/10.1145/2567948.2576928

[42] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. 2010. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2010.18

[43] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 29–41. https://doi.org/10.1145/2043556.2043560

[44] Carlos Pachajoa and Wilfried N. Gansterer. 2018. On the Resilience of Conjugate Gradient andÂăMultigrid Methods to Node Failures. In *Euro-Par 2017: Parallel Processing Workshops*. Springer International Publishing, Cham, 569–580.

[45] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The TAO of parallelism in algorithms. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '11)*. 12–25. https://doi.org/10.1145/1993498.1993501

[46] Russell Power and Jinyang Li. 2010. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 293–306. http://dl.acm.org/citation.cfm?id=1924943.1924964

[47] D. Presser, L. C. Lung, and M. Correia. 2015. Greft: Arbitrary Fault-Tolerant Distributed Graph Processing. In *2015 IEEE International Congress on Big Data*. 452–459. https://doi.org/10.1109/BigDataCongress.2015.73

[48] The Lemur Project. 2013. The ClueWeb12 Dataset. http://lemurproject.org/clueweb12/

[49] Mayank Pundir, Luke M. Leslie, Indranil Gupta, and Roy H. Campbell. 2015. Zorro: Zero-cost Reactive Failure Recovery in Distributed Graph Processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 195–208. https://doi.org/10.1145/2806777.2806934

[50] Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. 2007. Rx: Treating Bugs As Allergies - a Safe Method to Survive Software Failures. *ACM Trans. Comput. Syst.* 25, 3, Article 7 (Aug. 2007). https://doi.org/10.1145/1275517.1275519

[51] Semih Salihoglu and Jennifer Widom. 2013. GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management (SSDBM)*. ACM, New York, NY, USA, Article 22, 12 pages. https://doi.org/10.1145/2484838.2484843

[52] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Vishal Sahay, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. 2005. The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing. *The International Journal of High Performance Computing Applications* 19, 4 (2005), 479–493. https://doi.org/10.1177/1094342005056139

[53] Piyush Sao, Oded Green, Chirag Jain, and Richard Vuduc. 2016. A Self-Correcting Connected Components Algorithm. In *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS '16)*. ACM, New York, NY, USA, 9–16. https://doi.org/10.1145/2909428.2909435

[54] Piyush Sao and Richard Vuduc. 2013. Self-stabilizing Iterative Solvers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA '13)*. ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/2530268.2530272

[55] Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. 2013. "All Roads Lead to Rome": Optimistic Recovery for Distributed Iterative Data Processing. In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management (CIKM '13)*. ACM, New York, NY, USA, 1919–1928. https://doi.org/10.1145/2505515.2505753

[56] B. Schroeder and G. Gibson. 2010. A Large-Scale Study of Failures in High-Performance Computing Systems. *IEEE Transactions on Dependable and Secure Computing* 7, 4 (Oct 2010), 337–350. https://doi.org/10.1109/TDSC.2009.4

[57] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 505–516. https://doi.org/10.1145/2463676.2467799

[58] Yanyan Shen, Gang Chen, H. V. Jagadish, Wei Lu, Beng Chin Ooi, and Bogdan Marius Tudor. 2014. Fast Failure Recovery in Distributed Graph Processing Systems. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 437–448. https://doi.org/10.14778/2735496.2735506

[59] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 297–310. https://doi.org/10.1145/2694344.2694348

[60] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, S. Mehringer, Eric Wernert, H. Tufo, D. Panda, and P. Teller. 2017. Stampede 2: The Evolution of an XSEDE Supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact (PEARC17)*. ACM, New York, NY, USA, Article 15, 8 pages. https://doi.org/10.1145/

3093338.3093385

[61] Rob Strom and Shaula Yemini. 1985. Optimistic Recovery in Distributed Systems. *ACM Trans. Comput. Syst.* 3, 3 (Aug. 1985), 204–226. https://doi.org/10.1145/3959.3962

[62] Xiongchao Tang, Jidong Zhai, Bowen Yu, Wenguang Chen, and Weimin Zheng. 2017. Self-Checkpoint: An In-Memory Checkpoint Method Using Less Space and Its Practice on Fault-Tolerant HPL. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 401–413. https://doi.org/10.1145/3018743.3018745

[63] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111. https://doi.org/10.1145/79173.79181

[64] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. 2017. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 223–236. https://doi.org/10.1145/3037697.3037747

[65] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan. 2014. Replication-Based Fault-Tolerance for Large-Scale Graph Processing. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 562–573. https://doi.org/10.1109/DSN.2014.58

[66] R. Wang, E. Yao, M. Chen, G. Tan, P. Balaji, and D. Buntinas. 2011. Building algorithmically nonstop fault tolerant MPI programs. In *2011 18th International Conference on High Performance Computing*. 1–9. https://doi.org/10.1109/HiPC.2011.6152716

[67] Panruo Wu and Zizhong Chen. 2014. FT-ScaLAPACK: Correcting Soft Errors On-line for ScaLAPACK Cholesky, QR, and LU Factorization Routines. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/2600212.2600232

[68] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2013. GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*. ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/2484425.2484427

[69] E. Yao, R. Wang, M. Chen, G. Tan, and N. Sun. 2012. A Case Study of Designing Efficient Algorithm-based Fault Tolerant Application for Exascale Parallelism. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 438–448. https://doi.org/10.1109/IPDPS.2012.48

[70] John W. Young. 1974. A First Order Approximation to the Optimum Checkpoint Interval. *Commun. ACM* 17, 9 (Sept. 1974), 530–531. https://doi.org/10.1145/361147.361115

[71] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[72] G. Zheng, Xiang Ni, and L. V. KalÃĺ. 2012. A scalable double in-memory checkpoint and restart scheme towards exascale. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. 1–6. https://doi.org/10.1109/DSNW.2012.6264677

[73] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 301–316. http://dl.acm.org/citation.cfm?id=3026877.3026901