Zorro: Zero-Cost Reactive Failure Recovery in Distributed Graph Processing

Mayank Pundir and Luke M. Leslie

*Apache Hama implementation and results contributed by Tanmay Gangwani and Yi Zhou

CS 525, UIUC, Spring 2015

Abstract

Today, distributed graph processing systems largely rely on proactive techniques for failure recovery. Unfortunately, these approaches (such as checkpointing) entail a significant overhead. In this paper, we argue that distributed graph processing systems should instead use a reactive approach to failure recovery. The reactive approach trades off completeness of the result (generating a slightly inaccurate result) while reducing the overhead during failure-free execution to zero. We build a system called Zorro that imbues this reactive approach, and integrate Zorro into three graph processing systems – PowerGraph, LFGraph, and Hama. When a failure occurs, Zorro opportunistically exploits inherent vertex replication to quickly rebuild the state of failed servers. Experiments using real-world graphs demonstrate that Zorro is able to recover over 99% of the graph state when a few servers fail, and between 87-92% when half the cluster fails. Furthermore, using various graph algorithms, Zorro incurs little to no accuracy loss in all experimental failure scenarios, and achieves a worst-case accuracy of 97%.

1. Introduction

Distributed graph processing systems are widely employed to process large graphs, including online social networks [20], web graphs [5, 6], and biological networks [7]. Such graphs may comprise billions of vertices and trillions of edges [10]. Consequently, distributed graph processing systems such as Pregel [23], Giraph [1], PowerGraph [15], LFGraph [18] and GPS [28] are often run on large clusters [23].

However, despite their widespread and increasing popularity, systems for distributed graph processing offer recovery after server failures only through expensive proactive mechanisms such as checkpointing [15, 21, 23, 26, 28]. Such failure recovery mechanisms periodically and synchronously save the global graph state, consisting of application-specific values associated with vertices and/or edges. For instance, PowerGraph, Pregel and Giraph offer the option for each server to periodically save a synchronous snapshot of its graph partition to reliable storage such as HDFS. After failure, the most recent snapshot is used to rebuild the last persisted graph state.

Despite the success of checkpoint-based failure recovery mechanisms in storage [14, 27] and virtualization systems [11, 25], we find that proactive recovery mechanisms incur unnecessary and expensive overhead during common-case failure-free processing. Figure 1 illustrates that in PowerGraph [15] (running PageRank), turning on checkpointing incurred an $8 - 31 \times$ increase in per-iteration time – the larger the graph, the higher was the overhead. This overhead is largely because checkpointing incurs periodic and excessive I/O, and is particularly prohibitive given the large mean time between failures (MTBF) of a machine in modern clusters (e.g., 360 days [21]). As a result, many users in fact prefer to disable failure recovery mechanisms and simply restart [16].
In this paper, we argue that distributed graph processing systems should instead adopt a reactive approach to failure recovery. Achieving this in practice requires several challenges to be met. First, a reactive approach does not prepare for failures, and hence can only use information available after failure has occurred. Second, failures should be allowed to occur at any time during computation without inconsistencies during recovery. Third, failures should be allowed to occur during recovery itself (cascading failures) without interfering with ongoing recovery. Finally, the recovery mechanism must be fast and use few resources.

The reactive approach trades off completeness of the result (generating slight inaccuracy) while reducing the overhead during failure-free execution to zero. We build a failure recovery mechanism called Zorro that realizes this reactive philosophy, and we integrate Zorro into three graph processing systems – PowerGraph [15], LFGraph [18] and Hama [2]. Zorro does not prepare for server or network failures and incurs essentially zero additional cost during failure-free execution (at most 0.8% added execution time). When failure occurs, Zorro opportunistically exploits vertex replication inherent in graph processing systems to quickly and consistently rebuild the state of failed servers.

Experiments using real-world graphs containing billions of edges demonstrate that Zorro is able to quickly recover over 99% of the graph state when a few servers fail, and between 87-92% when half the cluster fails. Furthermore, using various graph processing algorithms, Zorro incurs little to no accuracy loss in all experimental failure scenarios, and achieves 97% worst-case accuracy.

2. Background and Motivation

In this section, we first give an overview of distributed graph processing systems, and then discuss challenges and limitations of existing failure recovery mechanisms.

2.1 Distributed Graph Processing Systems

A distributed graph processing system performs computation on a graph partitioned among servers.

Partitioning: Distributed graph partitioning can take the form of either vertex or edge partitions, where vertices or edges are uniquely assigned to servers, forming a local subgraph at each. Although existing frameworks offer mechanisms to partition graphs with the aim of reducing communication overhead, recent studies [18] have demonstrated that such mechanisms can occupy up to 80% of processing time, and are therefore inefficient compared to cheap hash-based partitioning.

Computation: We assume graph processing follows the synchronous, vertex-centric Gather-Apply-Scatter (GAS) decomposition, as supported by most popular systems (e.g., Giraph [1], Hama [2], PowerGraph [15] and LFGraph [18]). In this model, computations occur in iterations (also called supersteps in some systems), wherein vertices Gather values from neighbors, aggregate and Apply the values, and then Scatter the results to neighbors. Depending on the system and algorithm, vertices may be inactive within an iteration, restricting computation to only active vertices. We define the vertex state to be a vertex’s most recent applied value.

Communication: Partitioning the graph across servers requires vertex states to be propagated over the network to neighbors at remote servers. Different systems implement separate ways of performing this communication but, as we will discuss in Section 2.4, all approaches introduce a level of vertex state replication.
**Failure Recovery:** We define failure recovery in distributed graph processing systems as the recovery of all vertex states to the iteration from just before failure occurrence. We define state loss as all vertex states that must be recomputed.

The most common mechanisms for failure recovery are checkpoint-based [15, 21, 23, 26]. These approaches offer the ability to recover the entire distributed graph state after failure by periodically saving each node’s local subgraph state to reliable storage such as HDFS [31]. Failed nodes are replaced and all nodes load their subgraph state from the most recent checkpoint. Lost progress from iterations after the point of checkpoint creation must be recomputed before failure recovery is complete.

### 2.2 Failure Recovery

#### Failure Model:
We consider only fail-stop failures. One or more servers may fail simultaneously or in a cascading manner. Network failures such as rack outages are special cases of this failure model. We do not consider high message loss rates or Byzantine failures.

#### Limitations of Checkpointing
Despite the success of checkpoint-based failure recovery mechanisms in storage [14, 27] and virtualization systems [11, 25], two serious issues arise with checkpoint-based recovery in distributed graph processing. First, the process of synchronous snapshot determination and checkpointing can incur high runtime overhead, resulting in significant execution delays for relatively short-lived graph computations. As demonstrated in Figure 1, per-iteration checkpointing slowdown in our experiments ranges from $8 \times 31 \times$. Second, recovery from a checkpoint requires all progress from the interval between checkpointing and failure to be recomputed (potentially comprising many repeated iterations).

Recent work is consistent with our concerns. For example, the authors of GraphX [16] mention that most users of distributed graph processing systems leave checkpointing disabled due to performance overheads. Even the authors of Distributed GraphLab [21] state users must explicitly balance failure recovery costs against restarting computation.

### 2.4 Replication in Existing Systems

In order to realize zero overhead during the common case of failure-free executions, we must adopt a reactive approach for when failures do occur. However, since we do not a priori replicate vertex state, we are forced to rely opportunistically on replication that the underlying graph processing system already provides.

In this section, we argue that existing graph processing systems provide sufficient replication of vertices for real-world graphs, thus making our reactive approach feasible. Below, we classify popular systems into two classes based on their replication methodology. We refer to a vertex state replica as a copy of the vertex state created on a remote server by the communication model of the system.

1. **Out-Neighbor Replication:** In this class of systems, vertex states are replicated at out-neighbors on remote servers (see Figure 3(b)). Concretely, replication of a vertex $v$’s state

![Figure 2. Fraction of recovered state as a function of the number of failed servers (out of 16) for the graphs in Table 1.](image-url)

---

**Table 1.** Graph datasets. (E) represents an exponential graph, and (P) a powerlaw graph.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Edges</th>
<th>Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P) Stanford Web Graph</td>
<td>2,312,497</td>
<td>281,903</td>
</tr>
<tr>
<td>(E) CA Road Network</td>
<td>2,766,607</td>
<td>1,965,206</td>
</tr>
<tr>
<td>(P) Twitter</td>
<td>1,468,365,182</td>
<td>41,652,230</td>
</tr>
<tr>
<td>(P) UK Web Graph</td>
<td>3,738,733,648</td>
<td>105,896,555</td>
</tr>
</tbody>
</table>
exists at servers containing v’s out-neighbors. These systems can be further divided into two subclasses as a function of how these replicas are maintained:

(a) Message-Based: E.g., Pregel [23], Giraph [1] and Hama [2]. Each vertex is assigned to one server and maintains its out-edges. Replication of a vertex v’s state exists as buffered messages received from v during the previous iteration, at v’s out-neighbors.

(b) Value-Based: E.g., LFGraph [18]. Each vertex is hashed to one server and maintains its in-neighbors and their states. Each server maintains updated neighbor values of local vertices. Updated vertex states are sent to servers containing out-neighbors of the vertex.

2. All-Neighbor Replication: E.g., PowerGraph [15] and its predecessor, Distributed GraphLab [21]. In PowerGraph, each edge is assigned to one server (see Figure 3(c)) – thus, each vertex is present on all servers which store adjacent edges. For vertices having edges on multiple servers, one replica is labeled as the master while others are labeled as as mirrors. Replication may thus occur at all remote neighbors. In each iteration, mirrors send the local results of Gather to the master, which combines them and synchronizes the result with mirrors.

Figures 2(a) and 2(b) illustrate the fraction of recoverable vertex states in out- and all-neighbor replication frameworks, respectively. The replication models of both classes allow recovery of a large fraction of the vertex states – half the servers failing still results 87-92% of vertices recovered. As we will demonstrate in this paper, Zorro is able to achieve little to no inaccuracy in popular graph algorithms using this recovered state, even in the face of high numbers of failures.

3. Zorro Design

In this section, we present Zorro, a general protocol for zero overhead reactive failure recovery in distributed graph processing systems. After a failure occurs, Zorro reactively kicks in and executes the following stages:

R1 (Replace): After failure, each failed server is substituted by a new server – we call these replacement servers. Replacement servers start with zero state.

R2 (Rebuild): Each replacement server collects relevant state information from all surviving servers and rebuilds its local state.

R3 (Resume): After all replacement servers have finished rebuilding their local states, computation restarts from the beginning of the last iteration before failure.

3.1 Replacing Failed Servers

After failure is detected, survivors suspend computation, retain the local subgraph state in memory, and wait for replacements to rejoin. We assume the presence of a membership service that detects failures and informs the surviving servers. Such mechanisms are already running inside to-
day’s graph processing systems, e.g., ZooKeeper is supported by PowerGraph [15] and LFGraph [18], heartbeating mechanisms are used in Pregel [23], Giraph [1] and Hama [2], etc. In particular, the synchronous nature of the execution (per iteration) requires the use of barriers, which rely on a membership service by design. Thus, Zorro’s use of a membership service does not add extra overhead.

3.2 Rebuilding Local Subgraph States

In the most general case, Zorro has survivors send all local vertex states which are required to rebuild the replacement’s local subgraph state. Replacement servers receive vertex state data in parallel with initialization (e.g., loading graph partitions from persistent storage), and apply the received values afterward. As a result, the Rebuild stage of each replacement is independent and concurrent across survivors; this facilitates recovery in the case of scenarios such as cascading failures (discussed in Section 3.4). Furthermore, the GAS decomposition helps ensure consistency of state information due to the enforcement of synchronicity via barriers.

3.3 Resuming Computation

Recall that the Scatter stage in GAS involves notifying neighbors about updates. After failure recovery, the messages available from the previous Scatter will be unavailable at replacement servers. Therefore, for execution correctness, Zorro performs a partial Scatter stage after recovering from failures. In systems such as PowerGraph, this stage of Zorro is entirely local and merely involves transferring the value of a vertex to local neighbors. In other systems such as LFGraph and Giraph, the additional communication overhead is incurred only among replacement servers, and is less than that of a normal Scatter stage during computation.

3.4 Cascading Failures

Zorro treats cascading failures during either the Replace or Resume stages in the same fashion as failures during execution. When failures occur during Rebuild, Zorro performs nested Replace stages alongside vertex state transfer to existing replacements. To prevent a scenario where all servers fail during recovery and no progress can be made, existing replacements also assist in Rebuild by sending back any previously-received vertex states to new replacements.

3.5 Recovery Flow

An example run of a graph processing system during failure recovery using Zorro is illustrated in Figure 4. Events are numbered in order of our discussion below.

For this example, we assume Zorro manages membership lists through a membership service (MS in the figure) such as ZooKeeper. Upon detecting failures (1), the MS issues a callback (leave_cb) to surviving servers (2), after which the surviving servers suspend computation and wait for the replacement servers to reload their graph partitions. The replacement server joins the cluster by notifying the MS (3), which then issues a callback (join_cb) to survivors (4). After receiving a join callback, surviving servers send the most recent state of vertices (send_state) hosted on replacement servers if replicas are locally available (5). Now, as the cluster is recovering from the failure of server $s_1$, another server ($s_2$) fails. This failure of server $s_2$ is detected by MS (6) and the surviving servers are informed about the failure using a callback (leave_cb) (7). We note that the failure of server $s_2$ does not interfere with the recovery of server $s_1$. After the transfer of replicated state from server $s_3$ to server $s_1$ completes, the replacement server sends an acknowledgment (8). Since the failure of server $s_2$ is a cascading failure, replacement server of $s_1$ also participates in its recovery (11) by transferring state that it might have received from $s_2$ in (5). Finally, $s_2$’s replacement server acknowledges completion of state transfer from server $s_1$ and $s_3$ (12).

4. Recovery Analysis

In this section, we analyze the number of vertex states recovered by Zorro, as well as the overhead
For a graph $G = (V, E)$, we define the set of recovery neighbors $\Gamma_r(v)$, $\forall v \in V$, as the set of vertices that enable remote replication of $v$ (e.g., in Figures 3(b) and 3(c), $\Gamma_r(v_1) = \{v_2, v_3\}$ and $\{v_2, v_4\}$). Let $\Gamma_{in}(v)$ and $\Gamma_{out}(v)$ be the set of in- and out-neighbors, respectively. For the two classes described in Section 2.4, $\Gamma_r(v)$ exhibits the following property:

- **Out-Neighbor:** $|\Gamma_r(v)| = |\Gamma_{out}(v)|$
- **All-Neighbor:** $|\Gamma_r(v)| = |\Gamma_{out}(v) \cup \Gamma_{in}(v)|$

We note that PowerGraph has $|\Gamma_r(v)| = |\Gamma_{out}(v) \cup \Gamma_{in}(v)| - 1$ due to neighbor collocation from edge partitioning.

Suppose graph processing is performed on a set $S$ of $m$ servers and some set of $f \leq m$ servers fail. Let $V_S$ be the set of vertices that were primarily hosted at surviving servers, $V_F = V \setminus V_S$ be the set of vertices whose state must be recovered from failed servers, and $V' \subseteq V$ ($|V'| = n'$) be the true set of vertex states recovered after failure.

### 4.1 State Recovery

We wish to quantify $n_r = n' - |V_S|$, the number of vertex states recovered by Zorro from $V_F$. Under the assumption that vertices/edges are randomly assigned to servers using a consistent hashing function, we have the following results.

**Theorem 1.** The expected number of vertex states recoverable from $V_F$ is given by:

$$\mathbb{E}[n_r] = \sum_{v \in V_F} \left(1 - \left(\frac{f}{m}\right)^{|\Gamma_r(v)|}\right)$$  \hspace{1cm} (1)

**Proof.** For $v \in V_F$, the probability that $v$ is recoverable $(v \in V')$ is equal to the probability that $\Gamma_r(v) \cap V_S \neq \emptyset$, i.e., that $v$ has at least one surviving recovery neighbor. Let $r(v)$ be the binary recovery event for vertex $v$. The expected probability of recovery is:

$$\mathbb{E}[r(v)] = 1 - \prod_{v' \in \Gamma_r(v)} \frac{f}{m} = 1 - \left(\frac{f}{m}\right)^{|\Gamma_r(v)|}$$ \hspace{1cm} (2)

By linearity of expectation, $\mathbb{E}[n_r] = \sum_{v \in V_F} \mathbb{E}[r(v)]$. \hfill $\square$

Theorem 1 says that the expected number of recovered vertices and the probability of recovery are both dependent on the fraction of servers that fail, rather than the actual number of server failures. Moreover, the probability of vertex recovery exhibits rapid convergence to 1 as the number of recovery neighbors increases, or as the fraction of servers that fail decreases.

**Theorem 2.** If the number of recovery neighbors of vertices in $G$ follows a power-law distribution, $\mathbb{E}[n_r]$ can be expressed directly in terms of the power-law constant $\gamma$:

$$\mathbb{E}[n_r] = |V_F| \left(1 - \frac{1}{\sum_{d=1}^{|V_F|-1} d^{-\gamma}} \sum_{d=1}^{|V_F|-1} \left(\frac{f}{m}\right)^d d^{-\gamma}\right)$$ \hspace{1cm} (3)
Proof. Let $|\Gamma_r(v)|$ be a Zipf random variable with constant $\gamma$. The expectation in Equation 1 is equivalent to:

$$E \left( \left( \frac{f}{m} \right)^{|\Gamma_r(v)|} \right) = \frac{1}{\sum_{d=1}^{\frac{|V_F| - 1}{d - \gamma}}} \sum_{d=1}^{\frac{|V_F| - 1}{d - \gamma}} \left( \frac{f}{m} \right)^d d^{-\gamma}$$

(4)

Substituting back in gives:

$$E[n_r] = \sum_{v \in V_F} \left( 1 - \frac{1}{\sum_{d=1}^{\frac{|V_F| - 1}{d - \gamma}}} \sum_{d=1}^{\frac{|V_F| - 1}{d - \gamma}} \left( \frac{f}{m} \right)^d d^{-\gamma} \right)$$

(5)

which reduces to our result.

As demonstrated in [13] and [15], the power-law constant of most natural graphs is typically around $\gamma \approx 2$; for example, the Twitter graph has in-degree $\gamma = 1.7$, out-degree $\gamma = 2$, and recoverability illustrated in Figure 2.

Expected Recovery: The expected number of vertex states lost by Zorro is therefore equal to $|V_F| - E[n_r]$, and the expected number recovered is given by:

$$E[n'] = |V_S| + E[n_r]$$

(6)

This value represents a general metric on the approximation given by Zorro, specified as the total number of recovered vertex states after failure. However, application specific metrics may present more useful results to the user and will be discussed in Section 6.

5. Implementation

To demonstrate the generality and effectiveness of Zorro across the two classes of systems discussed in Section 2.4, we discuss implementation on two out-neighbor replication system, LFGraph and Hama, and one all-neighbor replication system, PowerGraph. In both systems, Replace is handled using ZooKeeper to identify failures.

5.1 LFGraph

Our implementation of Zorro in LFGraph modifies the computation_worker and communication_worker classes within the JobServer, which implements the GAS decomposition.

Rebuild: LFGraph maintains, for each vertex, the vertex state, a copy (for lock-free read/write) in the local value store, and vertex state replicas of remote in-neighbors in the remote value store. After failure occurs, survivors send replacements all vertex states previously hosted at that server in both the remote and local value store.

Resume: As discussed in Section 3.3, Zorro performs a partial Scatter before computation resumes. In LFGraph, Zorro performs this operation only among replacement servers. Each replacement performs a Scatter (over the network) to other replacements, rebuilding the vertex states of incoming neighbors on these servers.

Maintaining State Consistency: To enforce consistency, Zorro ensures that servers receive all updated values from incoming neighbors before updating the remote value store in the background during the Apply phase. Zorro also merges vertex state copies after Scatter, rather than between Apply and Scatter as in vanilla LFGraph. The resulting average per-iteration overhead incurred by ensuring vertex state consistency is just 0.8%. This is the only instance of overhead we ever found in Zorro.

5.2 PowerGraph

Our implementation of Zorro in PowerGraph modifies both the synchronous_engine class, which implements the GAS decomposition, and the local_graph class, which encapsulates the local subgraph at each server.

Rebuild: As discussed in Section 2.4, PowerGraph maintains, for each vertex, a master and set of mirrors. After failure occurs, survivors retain the local subgraph state in memory and send replacements all vertex states (either masters or mirrors) previously hosted at that server. Replacements receive vertex state data concurrently with initialization (i.e., graph loading and ingress) and then update local states.

Resume: As discussed in Section 3.3, Zorro performs a partial Scatter before computation re-
In PowerGraph, this operation is entirely local, performed only at replacements, and is used to rebuild local message buffers.

**Maintaining State Consistency:** In the Apply stage, partial accumulator (the results of performing local Gather at mirrors) are aggregated at the master and the results synchronized back to the mirrors. Failures during Apply may result in vertex state inconsistency at survivors. Zorro ensures that servers receive updates for all mirrors before applying the updated values, aborting if failures occur during transfer.

### 5.3 Hama

Our implementation of Zorro in Hama mainly modifies the GraphJobRunner class, which implements the GAS decomposition, and GroomServer class which handles task execution and failures.

**Rebuild:** Hama maintains, for each graph task, a local queue ($q_r$) which contains messages for the APPL Y phase of the running superstep, a queue ($q_n$) to hold messages from peers to be used in the next superstep, and a queue ($q_t$) to hold messages being sent out to peers in the running superstep. If a failure occurs in superstep $i$, survivors respond to replacement nodes with messages deposited in the $q_n$ and $q_t$ queues in superstep $i-1$.

**Resume:** As in LFGraph, Zorro performs a partial scatter between replacement servers before computation resumes. The replacement servers redo the failed superstep and sync-up with the survivors waiting on a barrier.

**Maintaining State Consistency:** Tasks in Hama use two barrier points ($b_1, b_2$). $b_1$ is executed following the APPL Y and SCATTER phases. After $b_1$, contents from the queue $q_n$ are transferred to $q_r$, followed by execution of $b_2$. To provide the recovering task with consistent data, the survivors query Zookeeper to gauge the point of failure i.e. before $b_1$ or between $b_1$ and $b_2$.

### 6. Evaluation

In this section, we describe the experimental setup and evaluation of Zorro using our exemplar systems and the graphs from Table 1.

**Cluster Setup:** All experiments were conducted on a 16-machine cluster. Each machine has $2 \times 4$-core Intel Xeon E5620 processors with hyper-threading enabled (16 virtual cores), 64 GB of RAM, a 500 GB SSD and 2 TB HDD.

#### 6.1 Algorithm Accuracy

We evaluate Zorro’s resulting accuracy using four algorithms: (1) PageRank, (2) Single-Source Shortest Paths, (3) Connected Components, and (4) K-Core Decomposition.

##### 6.1.1 PageRank

We first evaluate Zorro’s accuracy loss after failures while running PageRank with 10 iterations on the Twitter and UK Web graphs. Let $P_t$ be the set of top-$k$ PageRank vertices after failure recovery, and $P_n$ be the true top-$k$ PageRank vertices from execution without failure. We use the following metrics from [24] to evaluate Zorro’s accuracy:

- **Top-$k$ Lost (TL):** The fraction of lost top-$k$ ranked vertices: $|P_t \setminus P_n| / |P_t|$.
- **Mass Lost (ML):** The fraction of total top-$k$ PageRank mass lost: $\sum_{v \in P_t \setminus P_n} p(v) / \sum_{v \in P_t} p(v)$, where $p(v)$ is the PageRank score of $v$.

As demonstrated in Figure 5, Zorro on both frameworks achieves no accuracy loss in a majority of failure scenarios. In fact, even when half the servers fail (8 out of 16), Zorro results in an inaccuracy of only 2% top-$k$ lost (i.e., two of the top-100 PageRank vertices are not present in the new result), and even lower mass lost (i.e., the two lost vertices were low in the ranking). Even with a lower replication model (out-neighbor only), Zorro on LFGraph still manages to achieve a maximum inaccuracy of only 3% for failures at the last iteration, or with half the servers failing.

##### 6.1.2 Single-Source Shortest Paths (SSSP)

We evaluate Zorro’s accuracy after failures while running SSSP on all three graphs. The SSSP algo-
Zorro achieves zero (or near-zero) inaccuracy for evaluating Zorro’s accuracy: We use the following metrics graphs with large directed diameter (UK Web and increased to 15 to reach a larger set of vertices in Graph’s default setting was used for source selection. Power-algorithm computes the distance from a given source vertex to all other vertices in the graph. PowerGraph’s default setting was used for source selection in both systems. The number of iterations is increased to 15 to reach a larger set of vertices in graphs with large directed diameter (UK Web and CA Road Network). We use the weak connected components algorithm popular in distributed graph processing systems [23]. We evaluate Zorro’s inaccuracy using the following metric:

- **Paths Lost (PL):** The fraction of reachable vertices with lost paths after failure.
- **Average Difference (AD):** The average normalized difference of shortest paths [17]: 
  \[
  \frac{1}{|V'|} \sum_{v \in V'} (l_v(v) - l_n(v))/l_v(v), \text{ where } V' \text{ is the set of reachable vertices and } l_v(v) \text{ and } l_n(v) \text{ are the original and resulting shortest path length.}
  \]

As in PageRank, Figure 6 demonstrates that Zorro achieves zero (or near-zero) inaccuracy for most failure scenarios. Even with 8 failures, the maximum inaccuracy on LFGraph resulted in only 0.06% of total paths lost (i.e., only 0.06% of the original reachable vertices were unreachable after failure), and no AD. Accuracy on PowerGraph was consistently higher than LFGraph due to PowerGraph’s replication model, and achieved zero loss in all but a few scenarios.

6.1.3 **Connected Components (CC)**

We evaluate Zorro’s inaccuracy after failures while running CC with 10 iterations on all three graphs. We use the weak connected components algorithm popular in distributed graph processing systems [23]. We evaluate Zorro’s inaccuracy using the following metric:

- **Incorrect Labels (IL):** The fraction of vertices with a different label than the original.

Figure 7 illustrates the result. The CA Road network resulted in the highest inaccuracy, with a
maximum of 2.2% of vertices incorrectly labeled (i.e., assigned to the wrong component) in LFGraph, even with half of the servers failing. Zorro on PowerGraph resulted in 1.6% incorrectly labeled in the same scenario. There was no inaccuracy under all scenarios using the Twitter graph. Inaccuracy again increased with the iteration in which failure occurred, due to a lower likelihood of re-covergence in later iterations.

### 6.1.4 K-Core Decomposition

K-core decomposition [29] of a graph identifies induced sub-graphs such that included vertices have at least \( k \) neighbors. We evaluate Zorro’s accuracy after failures while running K-core decomposition with 10 iterations on all three graphs. We use the same metric as in Connected Components. In this context, the label is a binary value corresponding to a vertex’s inclusion in the K-core subgraph.

The results are illustrated in Figure 8. Zorro’s inaccuracy is again low, achieving a maximum of 1.4% of vertices incorrectly labeled with half of the servers failing using LFGraph. Inaccuracy is lower for smaller numbers of failures, and again increases in later iterations.

### 6.2 Rebuild Network Overhead

The network overhead incurred during recovery for PowerGraph and LFGraph, relative to the total overhead of 10 PageRank iterations, is presented in Figure 9. For PowerGraph (Figure 9(a)), we illustrate the overhead both with and without our optimization from Section 5.2 – overhead with the optimization is approximately 10% of that without (attaining a maximum of around 2%). For LFGraph (Figure 9(b)), recovery overhead is less than the average overhead of a single iteration (∼ 8% vs 10%). We also note that relative overhead scales with graph size for unoptimized Rebuild in PowerGraph, but remains static in LFGraph – this result can be explained by the difference in replication models between the two frameworks.

### 6.3 Different Partitioning Methods

To examine the effect of different partitioning techniques (applied to the graph during initialization), we perform experiments comparing our previous results with two further approaches available in PowerGraph [15] (LFGraph utilizes only cheap hash-based partitioning):
Figure 8. K-Core inaccuracy vs. (1) the number of failures (at the middle iteration), and (2) the iteration at which a quarter of the servers fail. There are a total of 16 servers and 15 iterations.

(a) PowerGraph: number of failures.
(b) LFGraph: number of failures.
(c) PowerGraph: failed iteration.
(d) LFGraph: failed iteration.

Figure 10. PageRank and SSSP inaccuracy with various PowerGraph partitioning methods, for half (8 out of 16) servers failing in the middle iteration, and a quarter (4 out of 16) servers failing in the last iteration.

(a) PageRank: half failures.
(b) PageRank: last iteration failure.
(c) SSSP: half failures.
(d) SSSP: last iteration failure.

7. Related Work

Failure recovery has been widely researched [8, 12], including optimistic failure recovery in distributed systems [4, 19, 32, 33] and large networks [22]. To the best of our knowledge, we are the first to explore reactive failure recovery in distributed graph processing systems.

Proactive Checkpoint-Based Recovery: Failure recovery using checkpoints is most common in distributed graph processing systems (e.g., Pregel [23], Piccolo [26], GPS [28], Giraph [1] and PowerGraph [15]).

In Pregel [23], workers checkpoint the state of vertices, edge values and received messages. The Pregel master checkpoints the state of global aggregators and detects worker failures via heartbeating. [23] also proposes a confined recovery mechanism in which workers checkpoint outgoing messages, restricting recomputation to failed.
Piccolo [26], an open-source implementation of Pregel, and Distributed GraphLab [21], both use the Chandy-Lamport algorithm [9] to calculate either a synchronous or asynchronous global snapshot of the system. The asynchronous variant of Chandy-Lamport allows computation to proceed along side the snapshot algorithm to mask checkpointing cost. However, after failures, some iterations may need to be repeated, significantly increasing recovery cost. The authors in [30] propose a partition-based recovery (PBR) mechanism that relies on checkpointing. PBR achieves faster recovery than traditional checkpointing by parallelizing re-computation of failed partitions among survivors. PBR handles cascading failures by initiating a new recovery plan considering the most recent cluster state. However, PBR additionally logs all outgoing messages to disk, increasing overhead during failure-free execution.

Proactive Replication-Based Recovery: Imitator [34] proactively ensures that vertices have at least $(K+1)$ replicas to tolerate the failure of at most $K$ servers. However, the choice of $K$ needs to be implicitly linked to the cluster size (given MTBF), and thus the induced network overhead to update replicas would become infeasible with large clusters. Furthermore, such an approach: (1) requires estimating the number of failures an application must tolerate, (2) enforces an artificial lower bound on replication that prevents the use of graph partitioning heuristics, and (3) is unable to handle cascading failures without re-replicating after failure.

8. Conclusions

In this paper, we proposed Zorro, a reactive recovery mechanism for distributed graph processing systems. We have implemented Zorro in both PowerGraph and LFGraph. Using real world graphs and popular algorithms, we demonstrated that Zorro recovers a highly accurate approximation of the graph state, even after a significant number of failures. Moreover, Zorro is able to achieve at least 97% accuracy when compared to the original output of graph algorithms, achieving perfect accuracy in many scenarios. We have argued that failure recovery in distributed graph processing systems is best done via reactive approaches like Zorro, rather than expensive proactive approaches that are the norm today.

9. APPENDIX

We present the evaluation of Zorro on Hama using a 7-machine cluster on Google Cloud. Each machine has 1 vCPU and 4GB of RAM. The algorithms used are (1) PageRank, (2) SSSP, and (3) CC. The metrics used are same as those outlined in section 6. For each benchmark, in the first set of experiments, different number of servers fail at the middle iteration; and in the second set, 4 servers fail at different iteration counts.

(a) PageRank(10 iterations). Maximum inaccuracy of 5% top-$k$ lost.

(b) SSSP(730 iterations). No inaccuracy.

(c) CC(30 iterations). Maximum inaccuracy of 7.4% wrongly labeled vertices.
References


