

© 2016 Luke M. Leslie

APPROXIMATE FAILURE RECOVERY IN DISTRIBUTED GRAPH
PROCESSING SYSTEMS

BY

LUKE M. LESLIE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Indranil Gupta

ABSTRACT

Distributed graph processing systems are an emerging area of big data systems. As graphs continue to grow in size and prevalence, these systems must become faster and more scalable. However, after failures, distributed graph processing systems either largely rely on proactive fault tolerance techniques such as checkpointing, or use no fault tolerance mechanisms at all and simply restart computation. The former approach entails significant proactive overheads that increase with the size of the graph, while the latter wastes time and resources in potentially lengthy recomputation. In this thesis, we argue that distributed graph processing systems should instead use an approximate approach to failure recovery that trades off minimal amounts of application accuracy while reducing the overhead during failure-free execution to zero, and allowing fast and scalable recovery.

We build a system called Zorro that imbues the approximate reactive approach, and integrate Zorro into two distributed graph processing systems – PowerGraph and LFGGraph. When a failure occurs, Zorro opportunistically exploits vertex replication (inherent in today’s graph processing systems) to quickly and scalably rebuild the state of failed servers. In addition, we describe three other novel failure recovery mechanisms that aim to address several of Zorro’s shortcomings. The first utilizes optimistic accuracy results from graph sampling and hence continues after failure without taking any action. The second repartitions the graph after failure to avoid waiting for replacement servers, and then continues computation with the recovered state. The last allows a small amount of proactive overhead to significantly increase the fraction of recovered state.

Experiments using five real-world graphs and eight benchmark applications 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, with recovery taking only a fraction of the cost of a single iteration. Furthermore, using eight common graph processing algorithms, Zorro incurs little to no accuracy loss in all experimental failure scenarios. Furthermore, preliminary analysis and experi-

ments using our three alternative approaches suggest that they are able to address many of the potential issues Zorro faces with minimal overhead and accuracy loss.

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Indranil Gupta, and my colleague, Mayank Pundir, for their support and contributions to this project and others. I would also like to thank my family for their continual support.

CONTENTS

Chapter 1	INTRODUCTION	1
1.1	Contributions of this Thesis	2
1.2	Outline of this Thesis	4
Chapter 2	BACKGROUND AND MOTIVATION	6
2.1	Distributed Graph Processing Systems	6
2.2	Desirable Properties for Failure Recovery	7
2.3	Limitations of Checkpointing	9
2.4	Replication in Existing Systems	10
Chapter 3	ZORRO DESIGN	13
3.1	Replacing Failed Servers	13
3.2	Rebuilding Local Subgraph States	14
3.3	Resuming Computation	15
3.4	Cascading Failures	16
3.5	Recovery Flow	16
3.6	Handling Lost Vertex States	17
Chapter 4	RECOVERY ANALYSIS	19
4.1	State Recovery	19
4.2	Rebuild Overhead	22
Chapter 5	IMPLEMENTATION	24
5.1	LFGGraph	24
5.2	PowerGraph	25
Chapter 6	EVALUATION	27
6.1	Algorithm Accuracy	27
6.2	PowerGraph-Specific Algorithms	32
6.3	Overhead during Recovery	34
6.4	Different Partitioning Methods	36
6.5	The Trade-off Space	37

Chapter 7	RELATED WORK	39
7.1	Proactive Checkpoint-Based Recovery	39
7.2	Proactive Replication-Based Recovery	40
7.3	Failure Recovery in Iterative Distributed Computation	40
Chapter 8	IMPROVING ZORRO	42
8.1	Don't Let Failures Stop You	42
8.2	Shrink to Fit	43
8.3	Some Proactive Overhead Might be OK	43
Chapter 9	CONCLUSION	47
	BIBLIOGRAPHY	48

Chapter 1

INTRODUCTION

Distributed graph processing systems are an emerging area of big data systems used to process graphs that can range up to petabytes in size and comprise billions of vertices and trillions of edges [2, 11]. These graphs include online social networks such as Facebook [11] and Twitter [27], regional and global web graphs [5, 6, 36], and biological networks [7]. In order to process these graphs, distributed graph processing systems are often run on large clusters comprising hundreds or thousands of servers [11, 36]. Examples of distributed graph processing systems include Pregel [36], Giraph [11], PowerGraph [17], GraphX [18], LFGGraph [21] and GPS [45].

As graphs continue to grow in size and prevalence, systems for distributed graph processing must become faster and more scalable. However, after failures, these systems either largely rely on proactive fault tolerance techniques such as checkpointing, or use no fault tolerance mechanisms at all and simply restart computation [11, 17, 33, 36, 41, 45]. Checkpointing-based mechanisms periodically and synchronously save the global *graph state*, consisting of application-specific values associated with *all* 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 and lost work since the checkpoint must then be recomputed.

Although checkpoint-based failure recovery mechanisms have been employed successfully in storage [15, 43] and virtualization systems [12, 38], we find that these proactive recovery mechanisms incur unnecessary and expensive overhead during common-case failure-free processing. In the world of big data, where modern graphs can occupy gigabytes to petabytes of storage space, these mechanisms also exhibit another major flaw – poor scalability; the relative slowdown of checkpointing increases in proportion to the size of the input graph.

Figure 1.1 illustrates some of these limitations of checkpointing in distributed

graph processing. In particular, turning on checkpointing in PowerGraph [17] (running PageRank) incurs an $8 - 31\times$ increase in the checkpointed iteration time – the bigger the graph, the higher the overhead. This overhead is largely because checkpointing incurs periodic and excessive I/O that scales with the size of the graph. This I/O cost is particularly prohibitive given the large mean time between failures of a machine in modern clusters (e.g., 360 days [33]). As a result, many users and administrators of these systems prefer to disable failure recovery mechanisms and simply restart computation [11, 18]. Furthermore, with computation on large graphs potentially taking hours, and with the possibility of failure at any time during computation, recomputing iterations from the time of checkpoint (or simply restarting computation) can lead to a large amount of redundant work occupying many resources. Instead, to efficiently tolerate failures when processing large graphs, a scalable approach to failure recovery is required that avoids both proactive overhead and lengthy recomputation.

1.1 Contributions of this Thesis

In this thesis, we argue that distributed graph processing systems should instead adopt a scalable, approximate, and 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 resulting in inconsistencies during recovery. Third, failures should be allowed to also occur during recovery itself (*cascading failures*) without interfering with ongoing recovery. Finally, the recovery mechanism must be fast and use few resources – in particular, to accommodate the recent orders-of-magnitude increases in graph sizes, relative recovery time must be insensitive to graph size.

An approximate, reactive approach trades off completeness of the result (generating slight inaccuracy) while eliminating overhead during failure-free execution. We first build a failure recovery mechanism called Zorro that realizes this reactive philosophy, and we integrate Zorro into two distributed graph processing systems – PowerGraph [17] and LFGGraph [21]. 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), thereby eliminating the upfront costs

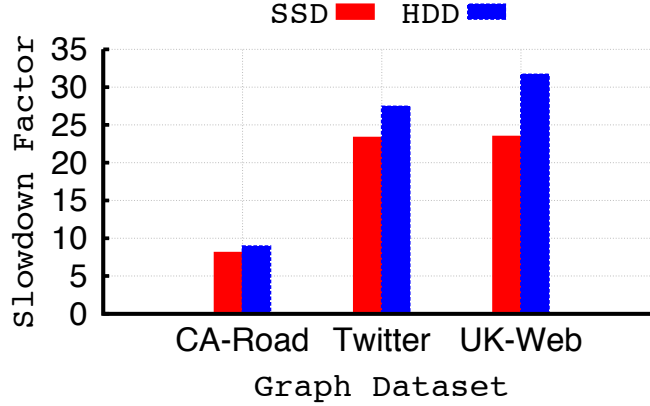


Figure 1.1: Per-iteration checkpointing slowdown with 16 servers (using SSDs and HDDs), for the graphs in Table 2.2.

of checkpointing and improving scalability of systems for processing big graphs. On the other hand, when failures do occur, Zorro opportunistically exploits vertex replication inherent in today’s graph processing systems to quickly, scalably, and consistently rebuild the state of failed servers. Our primary finding is that this existing level of replication (a function of the system and graph structure) is sufficient to achieve high accuracy after failure, and facilitates scalable recovery among servers.

Experiments using real-world big data 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. Zorro is able to achieve this in a scalable manner, with recovery taking only a fraction of the time of a normal iteration. Furthermore, using eight common graph processing algorithms, Zorro incurs little to no accuracy loss in all experimental failure scenarios.

During the design of Zorro, we explored a few alternative (and simpler) options for reactive failure recovery. We describe these alternatives below. Section 6.5 expands on the trade-offs between various failure recovery mechanisms (including restarting computation).

In addition, we describe the implementation details and preliminary results of three potential successors to Zorro that promise to address many of the issues Zorro faces, specifically with respect to waiting for replacement servers, decreasing recovery time, and increasing Zorro’s accuracy.

1. The first successor proceeds as follows: after failure occurs, we take no action and simply let computation continue on the surviving servers. this

approach results in the final part of computation being executed on what amounts to a *sample* of the input graph - vertices/edges that exist only on failed servers are permanently lost.

2. The second successor repartitions the graph after failure occurs, in order to fit on the smaller number of surviving servers. This approach entails significantly increased recovery time, but removes the reliance on waiting for new servers.
3. The third successor introduces a small amount of additional proactive replication in order to increase the accuracy of Zorro after failure. Specifically, theoretical analysis of several popular exemplar graphs has shown that proactively ensuring each vertex is replicated at least three times can significantly raise the number of recovered vertex states after failure.

We discuss these successors in more detail in Chapter 8 and present preliminary experimental and theoretical evaluations.

1.2 Outline of this Thesis

The overall contributions of this thesis are as follows.

1. In Chapter 2, we provide background on distributed graph processing systems and motivate the problems with proactive checkpointing in these systems. We then demonstrate how existing systems provide enough replication for an alternative strategy.
2. In Chapter 3, we present the design of a reactive approach to failure recovery in distributed graph processing systems called Zorro.
3. In Chapter 4, we mathematically analyze Zorro’s performance for different systems.
4. In Chapter 5, we describe the implementation of Zorro on two distributed graph processing systems: LFGraph and PowerGraph.
5. In Chapter 6, we evaluate Zorro on multiple real-world graphs, applications, and both distributed graph processing systems.

6. In Chapter 7, we present and contrast related work on this area.
7. In Chapter 8, we present and compare three additional novel approaches to failure recovery in distributed graph processing systems that aim to address many of the shortcomings of Zorro and failure recovery mechanisms in general. In addition, we provide preliminary theoretical and experimental evaluations.
8. In Chapter 9, we present our conclusions.

Chapter 2

BACKGROUND AND MOTIVATION

In this chapter, 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 a set of servers. As noted in the previous chapter, these graphs can range up to trillions of edges and billions of vertices in size [11], and computation can occur on hundreds or thousands of 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 intelligent mechanisms to partition graphs (with the aim of reducing communication overhead), and there exists a great deal of research on creating more efficient partitioning strategies (e.g., [10, 29, 40]) recent studies [21] have demonstrated that such mechanisms can occupy up to 80% of processing time, and therefore should be reconsidered in favor of cheap hash-based partitioning.

Computation: The synchronous, vertex-centric Gather-Apply-Scatter (GAS) decomposition is the most common graph computation model, and is supported by most popular systems (e.g., [1, 11, 17, 21]). As illustrated in Figure 2.1, computation in GAS occurs in iterations (also called supersteps), wherein vertices *Gather* values from neighbors, aggregate and *Apply* the values, and then *Scatter* the results to neighbors. Depending on the system and algorithm, computation within an iteration may be restricted 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

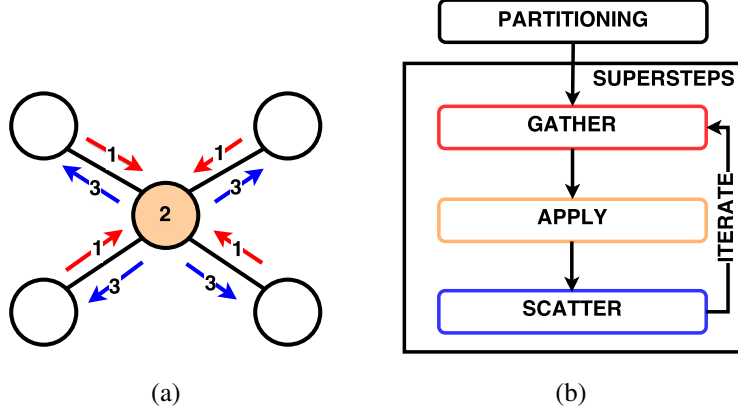


Figure 2.1: Gather-Apply-Scatter (GAS) decomposition, illustrating the Gather (1), Apply (2) and Scatter (3) stages.

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 [11, 17, 33, 36, 41]. 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 [49]. Failed nodes are replaced and all nodes load their subgraph state from the most recent checkpoint. Lost progress from iterations after checkpoint creation must be recomputed.

2.2 Desirable Properties for Failure Recovery

Scope: We assume graph processing follows the synchronous GAS decomposition. We consider only fail-stop failures, where 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.

Desirable Properties: Under failure-prone executions, there are three desirable properties of failure recovery mechanisms for distributed graph processing

systems:

ZO (Zero Overhead): No overhead is incurred during failure-free execution.

CR (Complete Recovery): Results in the face of failures are fully accurate.

FR (Fast Recovery): Recovery after failure is quick, scalable, and does not require additional iterations.

We note that we consider only the results of a graph processing application, rather than full recovery of all vertex states.

It is difficult to fully satisfy all three properties in a distributed graph processing failure recovery mechanism. To see why, consider a system that does not checkpoint dynamic graph state. When machine failures occur, the in-memory state of the graph application will be incomplete, potentially violating CR (in our experiments, this option gave 25-51% inaccuracy). On the other hand, without *a priori* knowledge of failure occurrence, the only option is to proactively checkpoint the in-memory graph state – however, this incurs high overhead (Figure 1.1) and violates ZO. Existing mechanisms for failure recovery strive to achieve completeness (CR) over zero-overhead (ZO). In this paper, we demonstrate that it is possible to achieve FR and ZO without sacrificing application accuracy by too much (i.e., achieving *almost*-CR).

	Recovery Mechanism			
Property	Checkpoint	Restart	Continue	Zorro
ZO	No	Yes	Yes	Yes
CR	Yes	No	Low	High
FR	Low	No	Yes	Yes

Table 2.1: Characterization of four recovery mechanisms

Scalability: We characterize four failure recovery mechanisms in Table 2.1. Here, *Restart* refers to restarting computation after failure and *Continue* refers to simply continuing computation after replacement servers join (as discussed in Chapter 1). Both checkpointing and restarting pose a serious scalability challenge when processing big data graphs in the form of *repeated iterations*. Given a static set of resources, the time to complete an iteration will generally increase with the size of the input graph. Although checkpointing might avoid having to repeat *all* iterations as with restarting, the high proactive costs (that themselves do not scale)

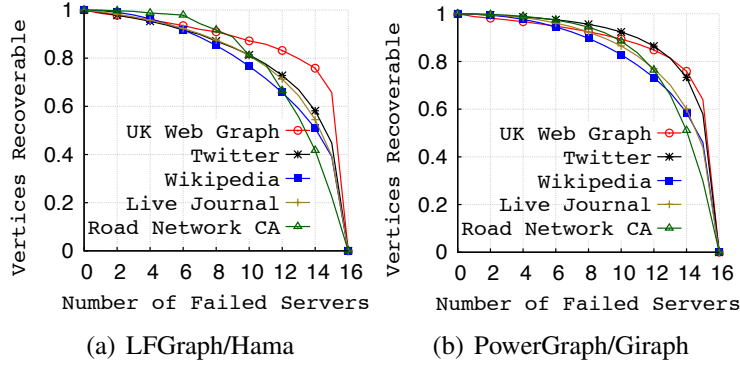


Figure 2.2: Fraction of recovered state as a function of the number of failed servers (out of 16) for the graphs in Table 2.2.

given relatively rare failures, coupled with the need to repeat *some* iterations after failure, have made it an unattractive alternative. On the other hand, Zorro and the Continue mechanism have no such scalability limitations, with Zorro able to provide high accuracy.

2.3 Limitations of Checkpointing

Despite the success of checkpoint-based failure recovery mechanisms in storage [15, 43] and virtualization systems [12, 38], 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 1.1, per-iteration checkpointing slow-down in our experiments ranges from $8 - 31\times$ and increases with the size of the graph, posing serious limitations for processing big data graphs. Second, recovery from a checkpoint requires all progress from the interval between checkpointing and failure to be recomputed, potentially comprising many repeated iterations on a large number of servers.

The two issues above jointly imply another drawback: significant user involvement – frequent checkpoints produce waste, whereas infrequent checkpoints produce risk. That is, if the checkpointing interval for an application is low, most checkpoints will be unused due to comparably high mean time between failures (MTBF). On the other hand, high checkpointing intervals make it likely that either no checkpoint will exist after failure, or that it will be stale and result in lengthy re-

Dataset	Edges	Vertices
(E) CA Road Network [3]	2,766,607	1,965,206
(P) Live Journal [3]	68,993,773	4,847,571
(P) Wikipedia [25]	340,309,824	11,196,007
(P) Twitter [27]	1,468,365,182	41,652,230
(P) UK Web Graph [6, 5]	3,738,733,648	105,896,555

Table 2.2: Graph datasets. (E) represents an exponential graph, and (P) a powerlaw graph.

computation on large pools of resources. For example, with a single-server MTBF of 360 days [33], the MTBF of a cluster of 16 servers is 22 days.

Young’s model [52] provides an approximation for the optimal checkpointing interval as $t^* = \sqrt{2t_c \cdot t_{MTBF}}$, where t_c is the checkpointing time and t_{MTBF} is the per-server MTBF [33, 36]. In our experiments, PowerGraph takes on average $t_c = 493.81$ seconds to create a *single* checkpoint of PageRank running on the UK Web graph [6, 5] partitioned across 16 servers with SSDs. Using Young’s model, the optimal checkpointing interval in the scenario above turns out to be 12 hours, which can far exceed typical application execution times (~ 22 seconds per iteration in this scenario).

Recent work is consistent with our concerns. For example, the authors of GraphX [18] mention that most users of distributed graph processing systems leave checkpointing disabled due to performance overheads. The authors of Distributed GraphLab [33] state users must explicitly balance failure recovery costs against restarting computation. Further, the authors of Giraph[11] state that they prefer to disable checkpointing in practice, and instead restart computation in the event of failure.

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 system already provides.

In this section, we argue that existing graph processing systems provide sufficient replication of vertices for real-world big data graphs, thus making our reac-

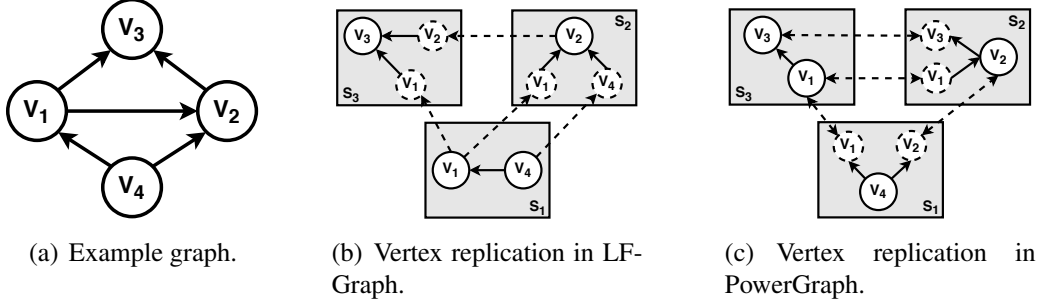


Figure 2.3: Vertex replication in the two system classes, for a graph partitioned across three servers using consistent hashing.

tive 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 2.3(b)). Concretely, replication of a vertex v 's state 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 [36] and Hama [1]. 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., LFGGraph [21]. 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 [17] and its predecessor, Distributed GraphLab [33], and Giraph [11]. In PowerGraph, each edge is assigned to one server (see Figure 2.3(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 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.

Given our focus on distributed processing of big graphs, we exclude centralized graph processing systems (e.g., [28, 35, 44]). GraphX [18] provides failure recovery using lineages from the RDD abstraction, with optional support for checkpointing in the case of long lineage chains [53].

Figures 2.2(a) and 2.2(b) illustrate the fraction of recoverable vertex states in out- and all-neighbor replication frameworks, respectively, for existing real-world graphs. The replication models of both classes allow recovery of a large fraction of the vertex states – half the servers failing still results 87-95% 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.

Chapter 3

ZORRO DESIGN

In this chapter, we present Zorro, a general protocol for zero overhead reactive failure recovery in distributed graph processing systems. Zorro gives preference to the zero overhead (ZO) characteristic of an ideal failure recovery mechanism and, as such, does not add overhead during failure-free execution. Rather, after a failure occurs, Zorro reactively kicks in and executes the following stages:

- R1 (**Replace**): After failure, each failed server is replaced 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, all servers resume computation from the start of the iteration in which failure occurred.

As we will discuss in Section 3.4, stage R1 may be nested inside R2 in order to handle failures during recovery. Figure 3.1 contrasts proactive and reactive recovery mechanisms. Proactive failure recovery mechanisms (Figure 3.1(a)) periodically save the graph state to persistent storage during computation. After failure, servers initialize from the checkpoint. In contrast, reactive failure recovery mechanisms (Figure 3.1(b)) do not persist state during computation. Rather, after server failures, replacements initialize their local subgraph from persistent storage and receive states from survivors.

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 today’s graph processing systems, e.g., ZooKeeper is supported by PowerGraph [17] and LFGGraph [21], heartbeating mechanisms are used in Pregel [36], Giraph [11] and Hama [1], 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.

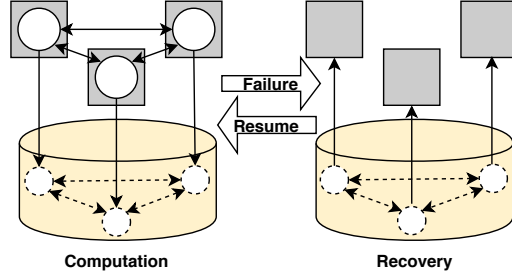
3.2 Rebuilding Local Subgraph States

We refer to the *local subgraph state* of distributed graph processing systems as the vertex values of locally-hosted vertices, as well as the replicated values of remote neighbors.

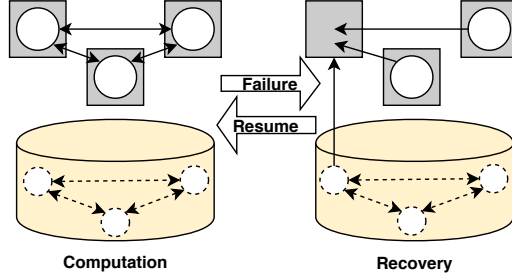
In the most general case, Zorro has survivors send all local vertex states which are required to rebuild the replacement’s local subgraph state. Replacement servers receive vertex state data in parallel with initialization (e.g., loading graph partitions from persistent storage), and apply the received 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, GAS helps ensure consistency of state information due to the enforcement of synchronicity via barriers. We later prove in Theorem 1 (Section 4.1) that, under the assumption of hash-based partitioning, the number of vertex states recovered during Rebuild is independent of the cluster size, and depends instead on the fraction of servers that fail.

As an example of general Rebuild, consider the failure of server s_3 in Figure 2.3(c). The values of vertex v_1 (replicated at servers s_1 and s_2) and v_3 (replicated at server s_2) are sent to the replacement server, and are thus both recovered.

We quantify the overhead during Rebuild in Section 6.3.2. In our experiments, state transfer during Rebuild is masked by concurrent graph initialization, (the resulting overhead is a fraction of the cost of a single iteration). We note that in systems such as PowerGraph, edges must be distributed among servers during graph loading, and recovery may thus introduce extra overhead. To help mitigate this, we discuss system-specific optimizations to eliminate redundant value transfer and balance network overhead in Chapter 5.



(a) Proactive checkpointing-based recovery.



(b) Reactive replication-based recovery.

Figure 3.1: Proactive vs. reactive failure recovery.

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 LFGGraph and Gigraph, the additional communication overhead is incurred only among replacement servers, and is less than that of a normal Scatter stage during computation.

After the partial Scatter, Zorro resumes computation from the start of the iteration during which failure occurred. Zorro either sends the iteration number through networking channels, or stores it on the membership service after failure detection. Synchronicity ensures that only a single value will be available for each vertex at the end of Resume.

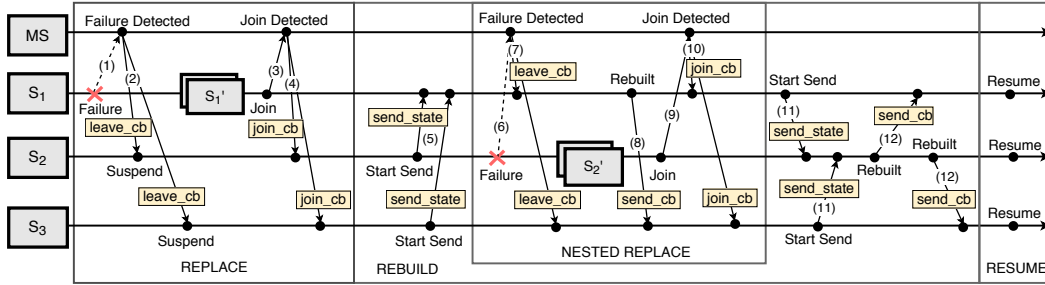


Figure 3.2: Zorro reactive recovery protocol under a cascading failure.

3.4 Cascading Failures

We define *cascading failures* as failures that occur while the system is recovering from a previous failure. Handling cascading failures can be a difficult task due to the interleaving of recovery stages and failures. However, Zorro utilizes the independence guarantees of its generalized Rebuild to significantly alleviate issues typically associated with recovery subject to 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.

For example, in Figure 2.3(c) consider the case where the system is recovering from the failure of server s_3 and server s_1 crashes. The recovery of server s_3 involves sending the state of vertex v_1 from servers s_2 and s_3 and vertex v_3 from server s_2 . If server s_1 fails during recovery, the Rebuild of s_3 from server s_2 is unaffected and s_3 assists in the rebuild of s_1 by sending back v_1 's state.

3.5 Recovery Flow

An example run of a graph processing system during failure recovery using Zorro is illustrated in Figure 3.2. 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. Graph reloading after failure rehashes the graph as during initialization, but rebuilds only the partitions on replacement servers.

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).

3.6 Handling Lost Vertex States

After failure, Zorro recovers only an approximation of the global state from before failure. So, after Rebuild, the resulting graph may contain vertices without recovered state information. For example, the value of vertex v_3 in Figure 2.3(b) cannot be recovered from a survivor if s_3 fails. When a vertex state is unrecoverable, Zorro reinitializes the vertex with the default value used for initialization by the application (e.g., for single-source shortest path, default initialization is zero for the source vertex and infinity for others).

Zorro minimizes the impact that lost values have on the global graph state through an implicit prioritization of high-degree vertices; for most partitioning functions, the probability of vertex recovery monotonically increases with the degree. As a result, vertices with lost states are generally confined to those whose neighbors do not span partitions. In our experiments, we find that those vertices with lost states are still able to quickly reconverge.

Edge States: Some applications in PowerGraph maintain edge states in addition to vertex states. However, these values can be obtained from their source and/or target vertex values and do not need to be maintained separately. Alternatively, edge states assume static values (e.g., edge weights) for some applications,

and can therefore be restored from graph partitions during Rebuild.

Chapter 4

RECOVERY ANALYSIS

In this chapter, we analyze the number of vertex states recovered by Zorro, as well as the overhead of Rebuild. Proofs below can be skipped without loss of continuity.

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 2.3(b) and 2.3(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 assigned to servers using a consistent hashing function, we have the following results.

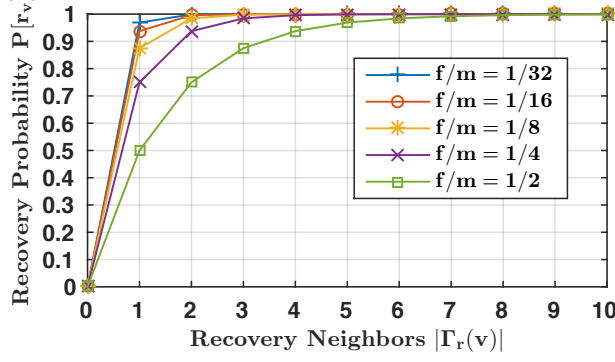


Figure 4.1: Probability of vertex recovery given the fraction of failures (f/m) and the number of recovery neighbors ($|\Gamma_r(v)|$), assuming random hash-based partitioning.

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

$$\mathbb{E}[n_r] = \sum_{v \in V_F} \left(1 - \left(\frac{f}{m} \right)^{|\Gamma_r(v)|} \right) \quad (4.1)$$

Proof. $\forall v \in V_F$, 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 v . We therefore have:

$$\mathbb{E}[r(v)] = 1 - \prod_{v' \in \Gamma_r(v)} \frac{f}{m} = 1 - \left(\frac{f}{m} \right)^{|\Gamma_r(v)|} \quad (4.2)$$

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

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

In Figure 4.1, we plot the probability of recovering the state of a vertex v discussed in Theorem 1 as a function of the number of recovery neighbors $|\Gamma_r(v)|$, and the fraction f/m of servers that have failed. We assume random hash-based partitioning. As shown in the figure, the probability of vertex state recovery for almost all failure scenarios exhibits rapid convergence to 1, and is high for even a small number of recovery neighbors.

Theorem 2. Letting V_F equal the total set of vertices primarily hosted on servers that fail before and during recovery, $\mathbb{E}[n_r]$ presents a lower bound on the expected number of vertices recovered after cascading failures.

Proof. Failures during state transfer from survivors to replacements will occur after some subset $V'_F \subseteq V_F$ of vertex states have been received at a replacement. Therefore, the expected number of vertex states recovered is equal to:

$$|V'_F| + \sum_{v \in V_F \setminus V'_F} \left(1 - \left(\frac{f}{m} \right)^{|\Gamma_r(v)|} \right) \geq \mathbb{E}[n_r] \quad (4.3)$$

□

Theorem 3. 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 γ :

$$\mathbb{E}[n_r] = |V_F| \left(1 - \frac{1}{\sum_{d=1}^{|V_F|-1} d^{-\gamma}} \sum_{d=1}^{|V_F|-1} \left(\frac{f}{m} \right)^d d^{-\gamma} \right) \quad (4.4)$$

Proof. Let $|\Gamma_r(v)|$ be a Zipf random variable with constant γ . The expectation in Equation 4.1 is equivalent to:

$$\mathbb{E} \left[\left(\frac{f}{m} \right)^{|\Gamma_r(v)|} \right] = \frac{1}{\sum_{d=1}^{|V_F|-1} d^{-\gamma}} \sum_{d=1}^{|V_F|-1} \left(\frac{f}{m} \right)^d d^{-\gamma} \quad (4.5)$$

Substituting back into Equation 4.1 yields:

$$\mathbb{E}[n_r] = \sum_{v \in V_F} \left(1 - \frac{1}{\sum_{d=1}^{|V_F|-1} d^{-\gamma}} \sum_{d=1}^{|V_F|-1} \left(\frac{f}{m} \right)^d d^{-\gamma} \right) \quad (4.6)$$

which reduces to our result. ¹

□

As demonstrated in [17] and [14], 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 2.2.

Expected Recovery: The expected number of vertex states lost by Zorro is

¹By modifying the expectation in Equation 4.5, we may derive a similar result for exponential (and other) graphs.

therefore equal to $|V_F| - \mathbb{E}[n_r]$, and the expected number recovered is given by:

$$\mathbb{E}[n'] = |V_S| + \mathbb{E}[n_r] \quad (4.7)$$

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 Chapter 6.

4.2 Rebuild Overhead

We define the *rebuild time* of Zorro after failures as the total time to rebuild the local graph states at replacement servers. Let $S_R, S_S \subseteq S$ be the set of replacement and surviving servers, respectively, and let $V(s)$ be the set of vertices primarily hosted on some server in S . Using the approach in Section 3.2, the expected upper bound on rebuild network overhead between two servers $s \in S_R$ and $s' \in S_S$ can be calculated as:

$$c_r(s, s') = \sum_{v \in V_F(s)} |\Gamma_r(v) \cap V_S(s')| \cdot \eta(v) + \sum_{v \in V_S(s')} |\Gamma_r(v) \cap V_F(s)| \cdot \eta(v) \quad (4.8)$$

where $\eta(v)$ is the size of v 's vertex state message. Hence, the expected communication cost is given by:

$$\begin{aligned} \mathbb{E}[c_r(s, s')] &= \sum_{v \in V_F(s)} |\Gamma_r(v)| \cdot \left(1 - \frac{1}{m-f}\right) \cdot \eta(v) \\ &\quad + \sum_{v \in V_S(s')} |\Gamma_r(v)| \cdot \left(1 - \frac{1}{f}\right) \cdot \eta(v) \end{aligned} \quad (4.9)$$

Expected Rebuild Time: Let $\varphi(s, s')$ be the symmetric bandwidth between any two servers $s, s' \in S, s \neq s'$. The expected upper bound on total rebuild time can be approximated as the maximum per-server rebuild time using [20]:

$$\mathbb{E}[t_r] = \max_{s \in S_R} \sum_{s' \in S_S} \frac{\mathbb{E}[c_r(s, s')]}{\varphi(s, s')} \quad (4.10)$$

An optimization for rebuild when neighbors are known is provided in Section 5.2.

Furthermore, the upper bound on network overhead is less than that of a single operation during a normal GAS iteration (e.g., master-mirror synchronization), as vertex state values are transferred to only a subset of servers, rather than to all (Section 6.3 further demonstrates this result).

Chapter 5

IMPLEMENTATION

In this chapter, to demonstrate the generality and effectiveness of Zorro across the two classes of systems discussed in Section 2.4, we discuss implementation on one out-neighbor replication system, LFGraph, and one all-neighbor replication system, PowerGraph (v2.2). In both systems, *Replace* is handled using ZooKeeper to identify failures. Hence, we focus on the *Rebuild* and *Resume* stages in our descriptions. We then discuss how Zorro maintains state consistency after failure.

5.1 LFGraph

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

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. Replacements receive vertex state data concurrently with initialization (i.e., graph loading) and apply the received values afterward.

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: In the Scatter stage, servers send the states of updated vertices to servers hosting outgoing neighbors. As a consequence, failures during Scatter may result in states being received at only a subset of survivors, leading to possible inconsistency during subsequent computation. To enforce consistency, Zorro ensures that servers receive all updated values from

incoming neighbors *before* updating the remote value store by creating a copy of 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 GAS, 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 a 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.

PowerGraph stores the IDs of servers containing each vertex, and thus allows an optimization to reduce network overhead during the rebuild stage. Per replacement server, $s \in S_F$, and relevant vertex, $v \in V_F(s)$, only a single survivor sends the associated state information. Relevant survivors evaluate candidacy using the following function:

$$s_r(s, v) = \operatorname{argmin}_{s' \in S_S(v)} |s'.id - ((v.id - s.id) \% m)| \quad (5.1)$$

where $S_S(v)$ is the set of surviving servers that contain recovery neighbor(s) of v . This approach ensures balanced state transfer and low network overhead by (1) minimizing the transfer of redundant state information and (2) removing skew associated with high-degree vertices. After cascading failures, servers reiterate over vertices and send any values for which they *newly* satisfy Equation 5.1. We present the performance improvement of this approach in Section 6.3.2.

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

Maintaining State Consistency: In the Apply stage, the master aggregates partial accumulators (the results of performing local Gather at mirrors) and synchronizes the results back to the mirrors. Failures during Apply may result in vertex state inconsistency at survivors. For example, a failed server may synchronize a mirror at one survivor but not at another. Zorro ensures that servers receive updates for all mirrors before Applying the updated values, aborting if failures occur during transfer. This modification required changing only two lines of code and, interestingly, *reduced* the average per-iteration time of PowerGraph by $\sim 26\%$ in all experiments. We attribute this reduction to the overhead incurred by synchronously interleaving send and apply in the original PowerGraph.

Chapter 6

EVALUATION

In this chapter, we describe the experimental setup and evaluation of Zorro using our exemplar systems and the graphs from Table 2.2. The goals of our evaluation are to measure: (1) the accuracy of graph algorithms after various numbers of server failures and after failures in different iterations, (2) the recovery overhead with Zorro and scalability implications, and (3) the effects of using different partitioning strategies. Failures encompass random servers and results are averaged across three trials.¹

Cluster Setup: All experiments were conducted on a 16-machine cluster. Each machine has 2×4 -core Intel Xeon E5620 processors with hyperthreading enabled (16 logical cores), 64 GB of RAM, a 500 GB SSD and 2 TB HDD. The connectivity between any two machines is 1 Gbps.

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. The above encompass all algorithms common to both PowerGraph and LFGGraph; results for four more PowerGraph-specific algorithms are presented in the next subsection.

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.² Let P_n be the set of top- k PageR-

¹We evaluate the effect of up to half the cluster servers failing. Related work has evaluated only a small fraction (e.g., 5 out of 72 servers [48]).

²CA-Road was excluded from the PageRank results due to many values remaining 1 after 10 iterations, thus yielding no inaccuracy but also offering no information about Zorro’s performance.

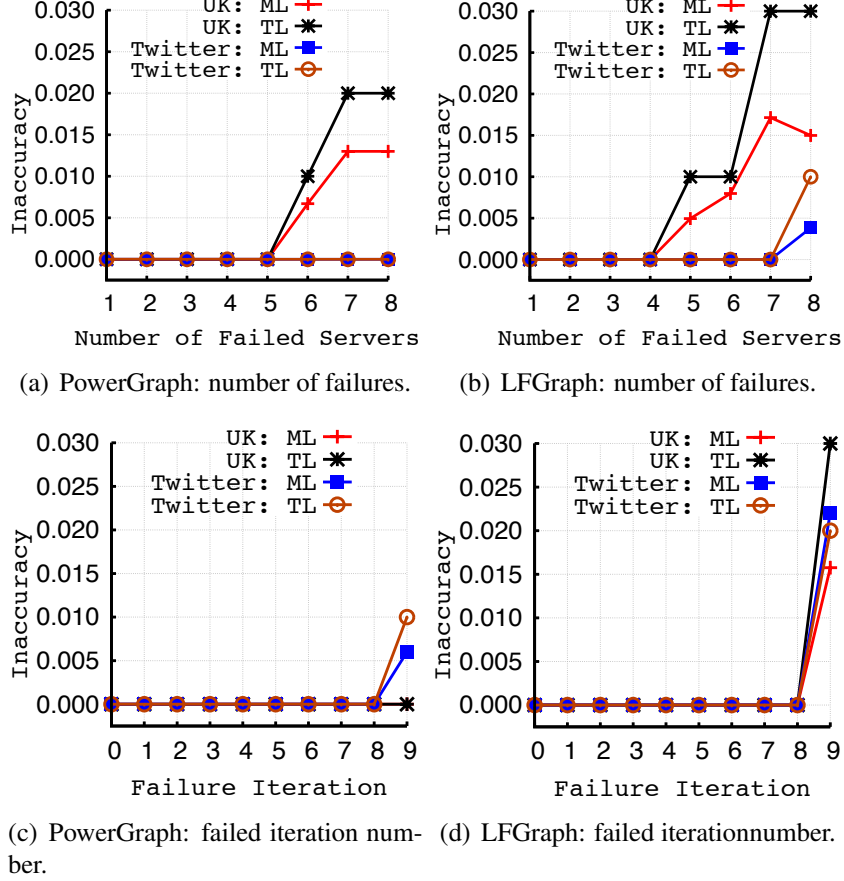


Figure 6.1: PageRank inaccuracy vs. (1) the number of failures (at the middle iteration), and (2) the iteration at which a quarter of the servers fail. There are a total of 16 servers and 10 iterations.

ank vertices after failure recovery, and P_t be the true top- k PageRank vertices from execution without failure. We use the following metrics from [37] 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 .

These metrics evaluate both how many of the top PageRank vertices are lost (TL), as well as their relative importance (ML). For our experiments, we set $k = 100$.

As demonstrated in Figure 6.1, 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

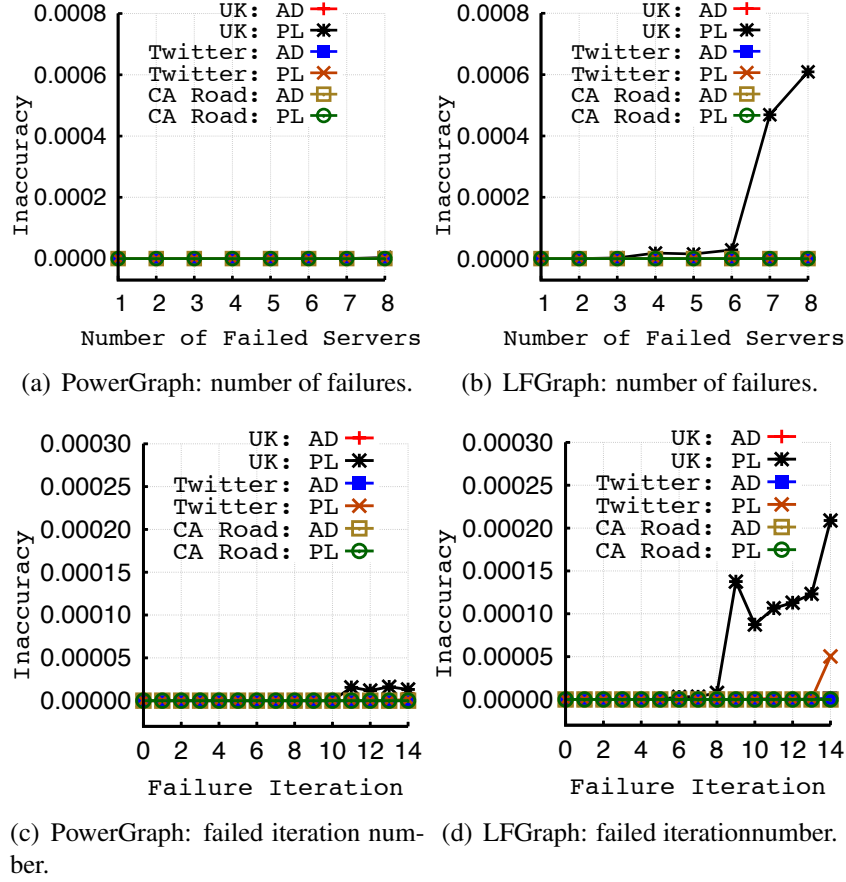


Figure 6.2: SSSP inaccuracy vs. (1) the number of failures (at the middle iteration), and (2) the iteration at which a quarter of the servers fail. There are a total of 16 servers and 15 iterations.

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. In both systems, lost mass is always less than top- k lost, implying that lost vertices rank low in the original top- k result.

From Figures 6.1(c) and 6.1(d), we note that, even with 4 servers failing, Zorro incurred inaccuracy only for failures in the last iteration. This is due to a high likelihood of subsequent reconvergence to the correct value in later iterations.

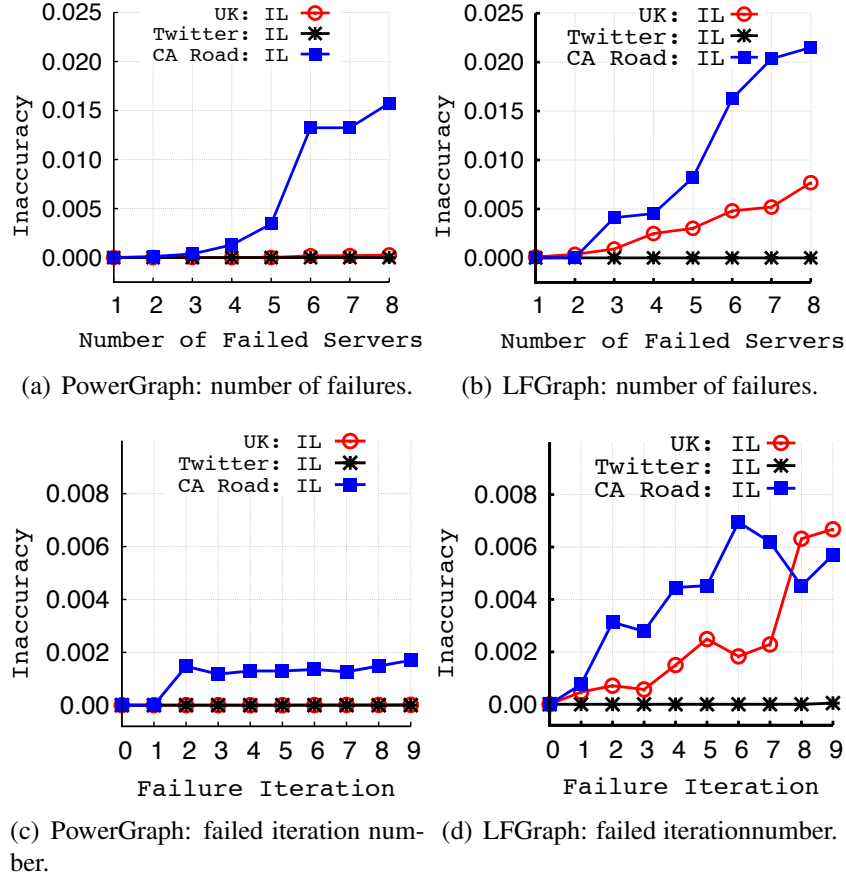


Figure 6.3: CC inaccuracy vs. (1) the number of failures (at the middle iteration), and (2) the iteration at which a quarter of the servers fail. There are a total of 16 servers and 15 iterations.

6.1.2 Single-Source Shortest Paths (SSSP)

We evaluate Zorro’s accuracy after failures while running SSSP on all three graphs. The SSSP 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 following metrics to evaluate Zorro’s accuracy:

- **Paths Lost (PL):** The fraction of reachable vertices with lost paths after failure.
- **Average Difference (AD):** The average normalized difference of shortest paths [19]: $\frac{1}{|V'|} \sum_{v \in V'} (l_t(v) - l_n(v)) / l_t(v)$, where V' is the set of reachable

vertices and $l_t(v)$ and $l_n(v)$ are the original and resulting shortest path length, respectively, from the source to vertex v .

As in PageRank, Figure 6.2 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 [36]. We evaluate Zorro’s inaccuracy using the following metric:

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

Figure 6.3 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-convergence in later iterations.

6.1.4 K-Core Decomposition

K-core decomposition [47] 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 induced K-core subgraph.

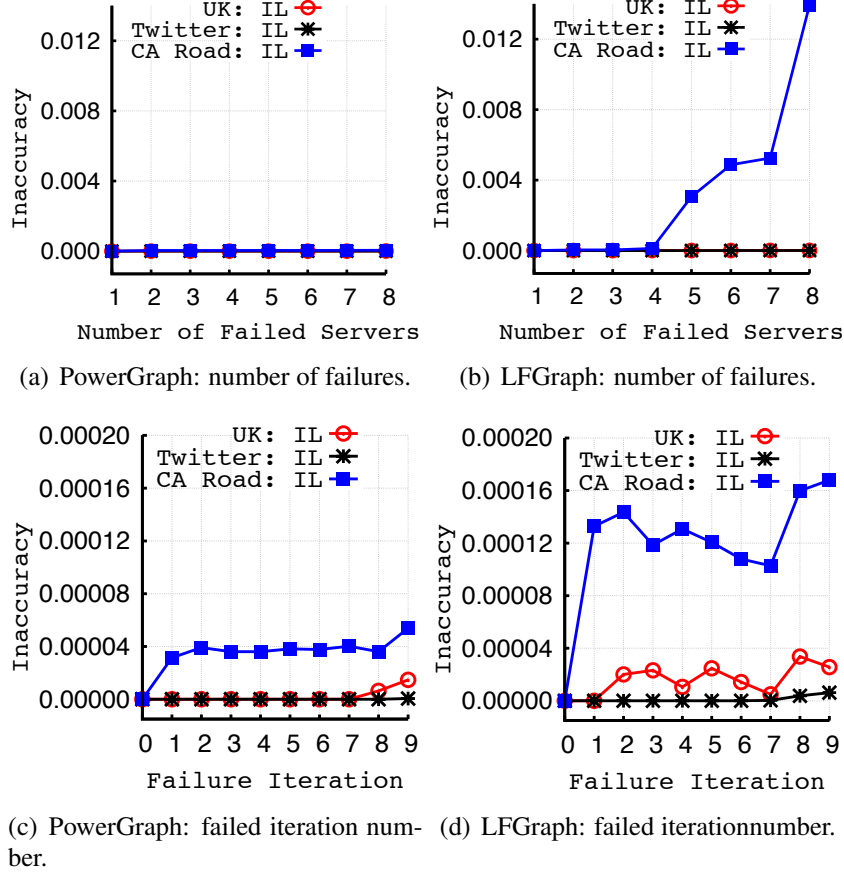


Figure 6.4: K-Core inaccuracy vs. (1) the number of failures (at the middle iteration), and (2) the iteration at which a quarter of the servers fail. There are a total of 16 servers and 15 iterations.

The results are illustrated in Figure 6.4. 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 PowerGraph-Specific Algorithms

We next present the results using PowerGraph-specific algorithms not available in LFGraph. The algorithms evaluated include (1) Graph Coloring, (2) Group-Source Shortest Paths, (3) Undirected Triangle Count, and (4) Approximate Diameter. Results are shown for the worst-case failure scenarios only. For Triangle Count and Approximate Diameter, we instead run experiments using the CA-

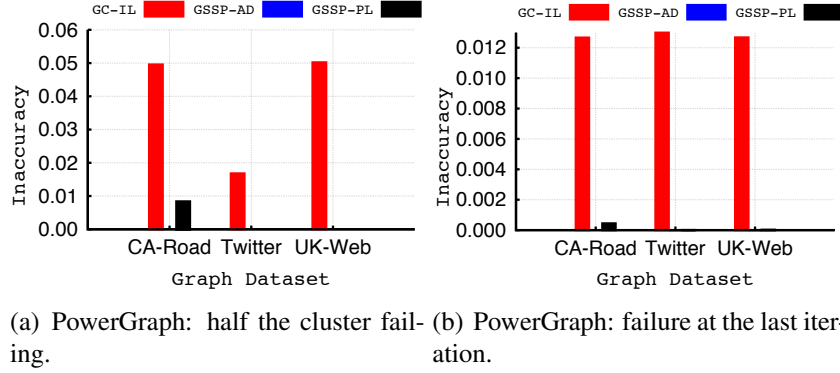


Figure 6.5: GC and GSSP inaccuracy using the algorithm-specific metrics in Sections 6.2.1 and 6.2.2 vs. (1) the number of failures (at the middle iteration), and (2) the iteration at which a quarter of the servers fail. There are a total of 16 servers and 15 iterations.

Road, Wikipedia, and LiveJournal graphs. The reason for this is that PowerGraph ran out of memory with the Twitter and UK-Web graphs using the Triangle Count and Approximate Diameter algorithms.

6.2.1 Graph Coloring (GC)

In this algorithm, each vertex assigns itself the smallest color not already used by its neighbors. We use the same metric, *Incorrect Labels* (IL), as Connected Components and K-Core Decomposition, with labels representing colors. We run the algorithm for 10 iterations. As illustrated in Figure 6.5, Zorro achieves a maximum worst case inaccuracy of 5% for UK Web and CA road with 8 (out of 16) server failures. This inaccuracy is the highest observed for Zorro.

6.2.2 Group-Source Shortest Paths (GSSP)

Group-source shortest paths is a variant of SSSP that instead measures the minimum distance from every vertex to those in a group of source vertices. We used the same metrics as SSSP, *Paths Lost* (PL) and *Average Difference* (AD). We selected the source set to be the top-5 degree vertices and, as in SSSP, ran the algorithm for 15 iterations. As demonstrated in Figure 6.5, the maximum measured inaccuracy was 0.8% with CA-Road for the Paths Lost metric with 8 (out of 16) server failures; most other experiments showed no inaccuracy.

6.2.3 Undirected Triangle Count

This algorithm counts the number of incident triangles on each vertex in the graph, as well as the total number of triangles in the graph. The metric used is incorrect labels, where the label is the number of incident triangles. Due to memory restrictions with PowerGraph, we ran this algorithm on the CA-Road, LiveJournal and Wikipedia graphs. In all graphs and both worst-case failure scenarios, Zorro achieves *zero* inaccuracy and thus we omit the plots.

6.2.4 Approximate Diameter

PowerGraph’s implementation of Approximate Diameter is based on the work from [23]. As with Undirected Triangle Count, we ran this algorithm on the CA-Road, LiveJournal and Wikipedia graphs, due to memory restrictions. As with Undirected Triangle Count, Zorro achieves *zero* inaccuracy in both worst-case failure scenarios and thus we omit the plots.

6.3 Overhead during Recovery

In this section, we evaluate the overhead Zorro incurs during failure recovery in terms of (1) added recovery time beyond initialization, and (2) network communication cost.

6.3.1 Recovery Time

Figure 6.6 shows the total recovery time excluding initialization (i.e., subgraph loading) for simultaneous failures in PowerGraph and LFGraph with the UK Web and Twitter graphs (CA Road is excluded due to negligible recovery time). Zorro allows replacement servers to rebuild their graph state while loading their respective graph partitions, resulting in quick recovery. Using PowerGraph, the recovery time for both graphs does not vary significantly with increasing numbers of failed servers, and increases for larger graph sizes. Using LFGraph, the recovery time increases linearly with the number of failed server due to the non-local partial scatter discussed in Section 5.1. PowerGraph has a slightly higher average recovery

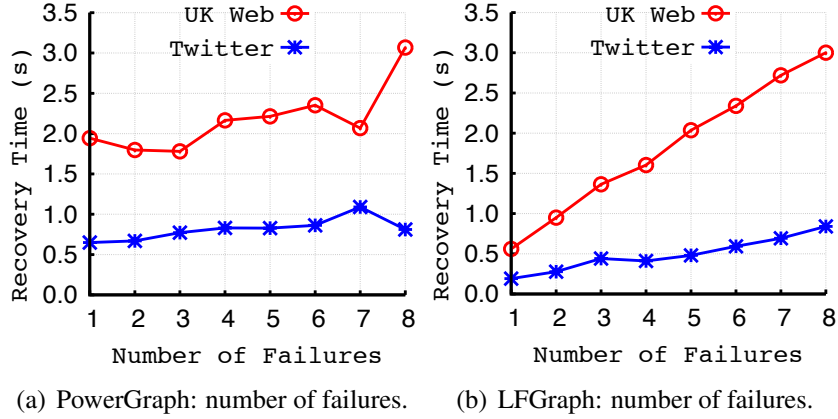


Figure 6.6: Additional recovery time beyond initialization as a function the number of failures (in the middle iteration).

time than LFGGraph due to all-neighbor replication. Most importantly, the recovery time is a small fraction of the average iteration time in both PowerGraph (11.7 seconds and 22 seconds for PageRank with Twitter graph and UK Web graph, respectively) and LFGGraph (2 seconds and 5.6, respectively).

6.3.2 Rebuild Network Overhead

The network overhead incurred during recovery for PowerGraph and LFGGraph, relative to the total overhead of 10 PageRank iterations, is presented in Figure 6.7. For PowerGraph (Figure 6.7(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 LFGGraph (Figure 6.7(b)), recovery overhead is less than the average overhead of a single iteration ($\sim 8\%$ vs 10%). We also note that relative overhead scales with graph size for unoptimized Rebuild in PowerGraph, but remains static in LFGGraph – this result can be explained by the difference in replication models between the two frameworks. Zorro on both systems also exhibits plateauing behavior as the number of failures increase. In addition, Zorro on PowerGraph (with optimizations) and LFGGraph appears insensitive to the size of the input graph (relative to normal iteration time), thus providing the required scalability for processing big data graphs.

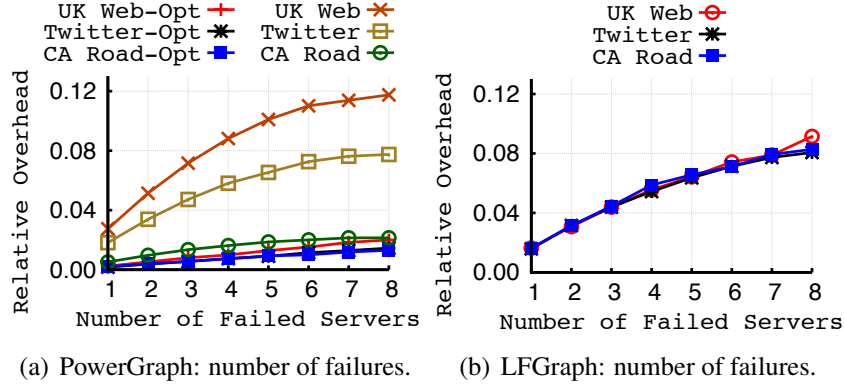


Figure 6.7: Network communication overhead of recovery with Zorro relative to total failure-free PageRank overhead.

6.4 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 [17] (LFGraph utilizes only cheap hash-based partitioning):

- **Oblivious:** Servers in the cluster greedily and independently partition the graph segment locally read during initialization.
- **Grid:** Randomly places edges using a grid constraint. We note that this approach only works if the cluster comprises a perfect-square number of machines.

For brevity, we only present results under the worst-case scenarios (i.e., half the cluster fails or a quarter fails on the last iteration) with PageRank, SSSP, Connected Components (CC) and K-Core on the Twitter graph.

As illustrated in Figure 6.8(a)-(h), using intelligent partitioning methods results in *at most* a 1% and 1.2% increase in inaccuracy with PageRank and SSSP, respectively. In some cases, changing the partitioning function resulted in no (or negligible) increase in inaccuracy (e.g., Figures 6.8(b) and 6.8(c)). In others, the inaccuracy increase is very small, e.g, Incorrect Labels (IL) increased to 1.8×10^{-5} for Connected Components and 5×10^{-6} for K-Core.

	Recovery Mechanism			
Overhead	Checkpoint	Restart	Continue	Zorro
Normal (s)	261 per chkpt	0	0	0
Re-init. (s)	60	117	117	117
Recover (s)	$11.7 \times (i_f - i_c)$	$11.7 \times i_f$	0	≤ 1
Inaccuracy	0%	0%	up to 25%	$\leq 1\%$

Table 6.1: A comparison of various failure recovery techniques for PowerGraph with the Twitter graph (16 servers, SSDs). i_f corresponds to the iteration at which failure occurs and i_c corresponds to the last checkpointed iteration.

6.5 The Trade-off Space

As an example of the trade-offs between various failure recovery mechanisms, we compare various failure recovery mechanisms with PowerGraph running PageRank on the Twitter Graph in Table 6.1. Our comparison contrasts the overhead incurred, both during normal execution and with failures, and the resulting inaccuracy.

We can observe the following results about the four recovery mechanisms given in Table 6.1.

- **Checkpointing** incurs high overhead both during failure-free execution (261 seconds per checkpoint with SSDs, 321 with HDDs) and after failures. Furthermore, checkpointing requires recomputation of iterations between the time of checkpoint and failure.
- **Restarting** computation incurs initialization overhead (117 seconds) and must recompute all previous iterations. Such a mechanism also fails to make progress during cascading failures.
- **Continuing** processing after failures suffers from very high inaccuracy - up to 25% for PageRank on the Twitter graph, and often higher in other scenarios.
- **Zorro** performs better than all alternatives, exhibiting no overhead during failure-free execution, requiring no recomputation, and achieving $\leq 1\%$ inaccuracy.

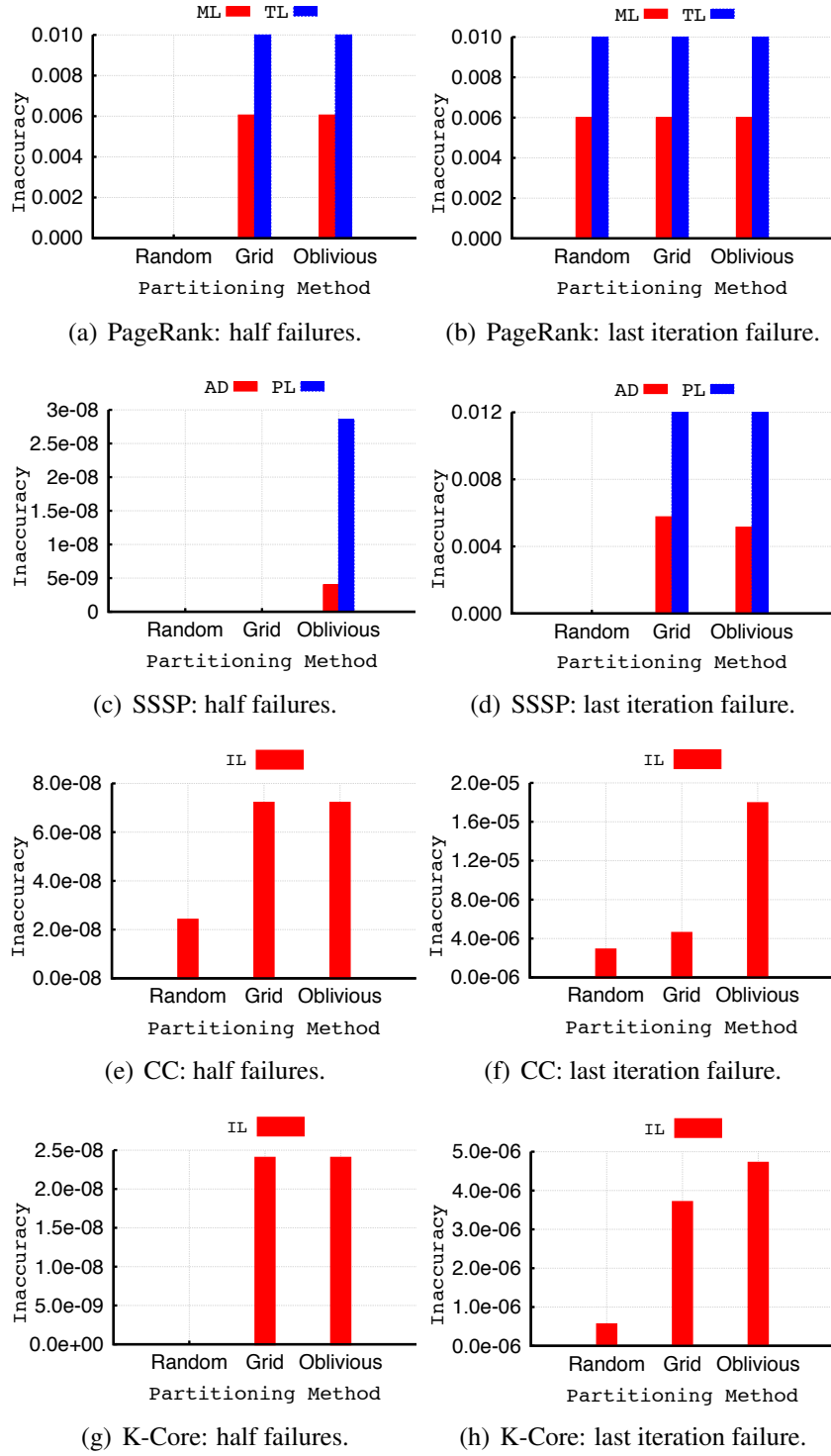


Figure 6.8: Inaccuracy with various PowerGraph partitioning methods, for half (8 out of 16) servers failing in the middle iteration, and a quarter (4 out of 16) servers failing in the last iteration.

Chapter 7

RELATED WORK

Failure recovery has been widely researched [8, 13], including optimistic recovery in systems and networks [4, 22, 34, 50]. To the best of our knowledge, we are the first to explore reactive failure recovery in distributed graph processing. In this chapter, we present and contrast existing research on failure recovery in systems and networks with Zorro.

7.1 Proactive Checkpoint-Based Recovery

Failure recovery using checkpoints is most common in distributed graph processing systems (e.g., Pregel [36], Piccolo [41], GPS [45], Giraph [11] and PowerGraph [17]).

In Pregel [36], 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. [36] also proposes a *confined* recovery mechanism in which workers checkpoint outgoing messages, restricting recomputation to failed workers. However, the authors do not evaluate the overhead involved in checkpointing or during failure recovery.

Piccolo [41], an open-source implementation of Pregel, and Distributed GraphLab [33], 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 [48] propose a partition-based recovery (PBR) mechanism that relies on checkpointing. PBR achieves faster recovery than traditional checkpointing by parallellizing recomputation 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, therefore further increasing the overhead during failure-free execution.

7.2 Proactive Replication-Based Recovery

Imitator [51] 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-replication after failure.

Zorro, on the other hand, recovers a near-perfect approximation of the graph state without any upper bounds on the number of failures. Furthermore, the number of vertex states recovered is independent of the cluster size (depending instead on the fraction that fails), and Zorro can recover from arbitrary numbers of independent and cascading failures.

7.3 Failure Recovery in Iterative Distributed Computation

To eliminate checkpointing, GraphX [18] uses the Resilient Distributed Datasets (RDD) abstraction provided by Spark [53]. Spark allows fast reconstruction of RDDs using their lineage graph. However, even with the fast reconstruction of RDDs, the execution time with one server failure incurs an overhead of 36% [18], and checkpointing is required in the case of long lineage chains.

The authors in [46] propose a reactive mechanism to recover from failures in iterative data-flow systems. In the proposed mechanism, the processing state can reach consistency even after failures using correct “algorithmic compensations”. The mechanism allows users to specify the “compensate” function and discuss such functions for algorithms involving link/path exploration and matrix factorization.

In distributed storage systems, RAMCloud [39] distributes data replicas across

the cluster servers. In case of failures, the surviving servers jointly reconstruct the state of failed servers in parallel for fast failure recovery. [24] performs opportunistic recovery for MapReduce with forced proactive replication.

Chapter 8

IMPROVING ZORRO

In this chapter, we provide high level details of three potential successors to Zorro. These successors aim to address subsets of the following limitations of Zorro:

1. *Reliance on replacements:* Zorro requires failed servers to be replaced by new servers, and thus recovery time is influenced by the time required to provide replacements. Although waiting for replacement servers is also a requirement of checkpointing, we wish to eliminate this dependence if possible.
2. *Recovery time:* Zorro still has a small amount of recovery time after failure. Although Zorro’s recovery time is negligible relative to techniques such as checkpointing and restarting, which often must repartition the graph and recompute lost iterations, we wish to see how much further we can reduce this time.
3. *Inaccuracy:* Zorro incurs a small amount of inaccuracy in exchange for eliminating proactive overhead. We instead wish to see if we can provide a tuneable tradeoff between proactive overhead and inaccuracy.

To address the limitations above, we will discuss three novel failure recovery mechanisms that build on Zorro’s results and seek to address the above limitations.

8.1 Don’t Let Failures Stop You

The first successor build on results from graph sampling literature that demonstrate high accuracy even with relatively small samples [16, 26, 30, 31, 32]. After failures occur, a natural question arises: what if we do nothing and simply continue computation on the surviving set of servers? We note that this approach is different from **Continue** in previous chapters, which still waits for replacement

servers and performs the remaining iterations on the whole graph. In contrast, this recovery mechanism performs the final part of computation (i.e., after failure) on what amounts to a *sample* of the input graph. Hence, vertices/edges that existed only on failed servers (without replicas) are permanently lost.

8.2 Shrink to Fit

One of the primary reasons that Zorro required replacement servers to be made available is that the input graph was already partitioned for a certain number of servers. However, it is possible to *shrink* the graph to fit on the smaller number of surviving servers by repartitioning. The second approach thus stores the most recent graph state, repartitions the graph to fit on the surviving servers, and propagates the stored vertex states to the servers that are now responsible for it. This approach can entail significantly increased recovery time due to repartitioning, but removes the reliance on waiting for new servers while still increasing accuracy compared to the first successor above. Existing mechanisms for on-demand elasticity (i.e., repartitioning a graph on-demand) can be seen in recent literature such as [42].

8.3 Some Proactive Overhead Might be OK

The third successor introduces the notion that proactive overhead can instead be tuneable against inaccuracy, and thus introduces a small amount of additional proactive replication of vertex states in order to increase the accuracy of Zorro after failure. Therefore, this successor ensures that each vertex is proactively replicated *at least* r times, where r is a user-provided constant based on theoretical analysis or some heuristics.

One conclusion immediately apparent from Figure 4.1 is that low degree vertices are especially susceptible to loss under Zorro. In fact, in LFGraph, lost vertices with no neighbors have no chance of recovery; in PowerGraph, the same holds true for vertices with ≤ 1 neighbor. Hence, the successor we propose to combat this property is *Low-Degree Vertex Replication* (LDVR). LDVR $_k$ replicates each vertex enough times to ensure at least k recovery neighbors per vertex exist in the cluster. Low-degree vertices are replicated in-memory on neighboring

servers in the logical ring constructed through the hashing function. Thus, 0-neighbor vertices are replicated at two neighboring servers in the ring, 1-neighbor vertices are replicated at a single neighboring server, etc.

Therefore, contrary to existing replication techniques such as those in PowerGraph, we seek only to replicate a small percentage of the total number of vertices. In power-law graphs, such as LiveJournal and Yelp, such vertices comprise up to approximately 30% of the total number of vertices (see Figure 8.1 for example). However, the overhead in our proposed replication strategy will be significantly lower than this value as in LFGGraph, for example, 1+-degree vertices might already have an existing replica through their neighbor, and 2+-degree vertices in PowerGraph might also have the same.

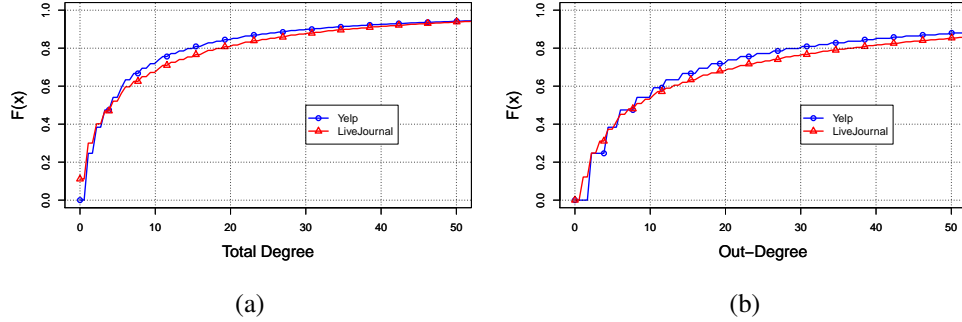


Figure 8.1: CDF of (a) total degrees and (b) out-degrees for Yelp and LiveJournal graphs.

To further motivate this approach, we provide experimental results and theoretical analysis using LDVR₂ (i.e., with the minimum number of in-memory replicas $k = 2$). Experiments on real-world graphs demonstrate that the storage and network transfer overhead of LDVR₂ is only around an additional 6-8% in GraphLab and 14-15% in LFGGraph, while vertex state retention after failure is significantly increased.

More formally, the probability of vertex recovery under LDVR₂ is equivalent to the following:

$$P_r(v) = \begin{cases} 1 - \frac{f}{m}^{|\Gamma_r(v)|} & \text{if } |\Gamma(v)| \geq 2 \\ 1 - \frac{f(f-1)}{m(m-1)} & \text{if } |\Gamma(v)| = 1 \\ 1 - \frac{(f-1)(f-2)}{(m-1)(m-2)} & \text{if } |\Gamma(v)| = 0 \end{cases} \quad (8.1)$$

Note that the case when $|\Gamma(v)| = 0$ will only occur when the set of recovery neighbors are colocated with the vertex on the same server.

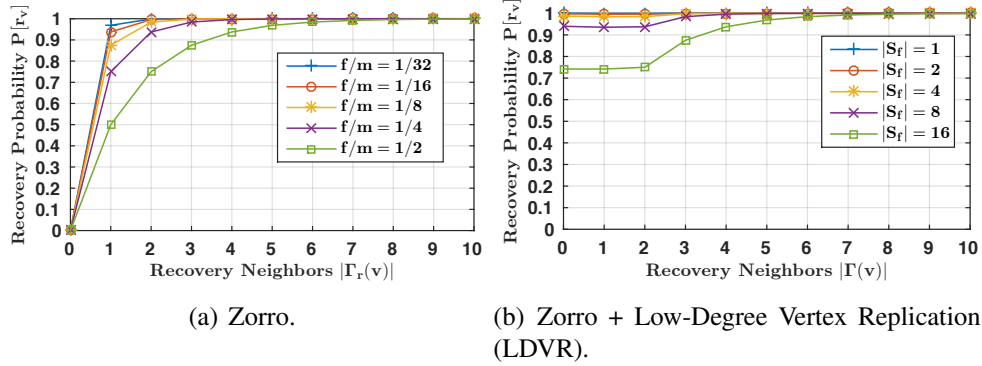
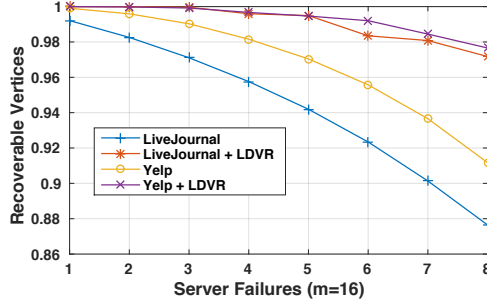


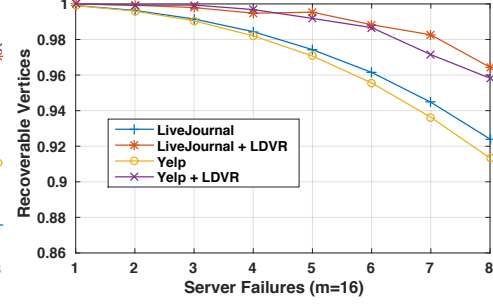
Figure 8.2: Probability of recovery with $m = 32$ servers using (a) no replication and (b) low-degree vertex replication with $k = 2$.

Figure 8.2 illustrates the probability of vertex recovery using Zorro and Zorro + LDVR. In contrast to using only Zorro, for ≤ 8 simultaneous failures the probability of vertex recovery is above 0.93 for all values of $|\Gamma(v)|$, and around 1 for ≤ 4 simultaneous failures.

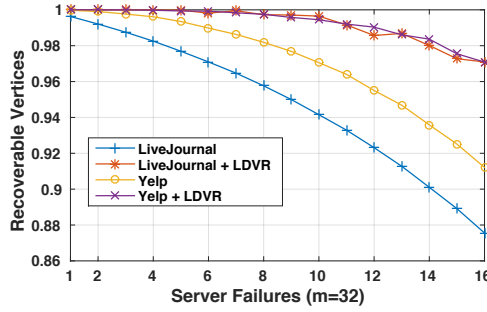
Figure 8.3 illustrates the number of recoverable vertices using Zorro and Zorro + LDVR for the Yelp and LiveJournal graphs. As evidenced by those figures, using LDVR can significantly increase the fraction of recoverable vertices on both LFGGraph and PowerGraph, for all failure scenarios, with the results being comparable for larger clusters. For example, with half the cluster failing, the fraction of recoverable vertices using only Zorro falls around 88-92% for both systems. However, with LDVR, the fraction of recoverable vertices increases to $>96\%$. Therefore, due to the low proactive overhead compared to checkpointing and the higher recoverability compared to Zorro, Low-Degree Vertex Replication may prove an attractive alternative for some applications.



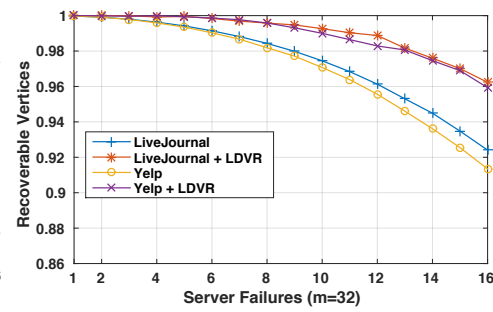
(a) LFGGraph



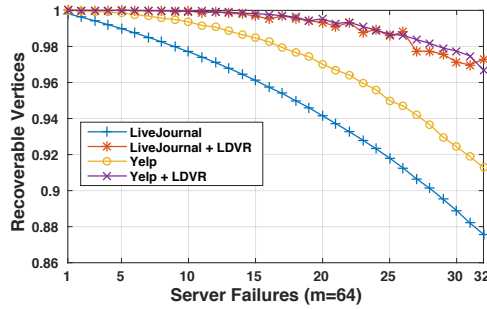
(b) PowerGraph



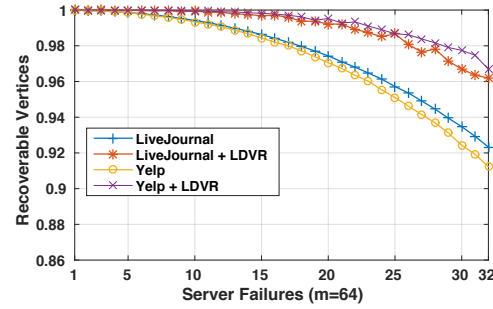
(c) LFGGraph



(d) PowerGraph



(e) LFGGraph



(f) PowerGraph

Figure 8.3: Fraction of vertices recovered given group failures of servers, for $m = 16, 32, 64$. LFGGraph is in the left column - (a), (c), and (e), and PowerGraph is in the right column - (b), (d), and (f)

Chapter 9

CONCLUSION

In this thesis, we have motivated the case for approximate failure recovery in distributed graph processing. To this end, we have presented, analyzed, and evaluated Zorro, a, approximate, reactive recovery mechanism for distributed graph processing systems. We have implemented Zorro in both PowerGraph and LFG-Graph. Using real world graphs and popular algorithms, we demonstrated that Zorro is:

- **Accurate:** 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 95% accuracy when compared to the original output of graph algorithms, achieving perfect accuracy in many scenarios.
- **Fast:** Recovery costs less than a single iteration, and system-specific optimizations are available to further reduce this overhead.
- **Scalable:** Both the expected number of recovered vertices and the probability of recovery (assuming hash partitioning) depend only on the fraction of failed servers. Recovery time is insensitive to the size of the input graph.
- **Resilient:** Rebuild of each failed server is independent and concurrent across survivors, easily handling cascading failures.

We have argued that failure recovery in distributed graph processing systems is best done via approximate, reactive approaches like Zorro, rather than expensive fully-complete, proactive approaches that are the norm today. We believe that this approach opens up the avenue to explore approximate reactive recovery in other computation systems. In addition, we have detailed three potential successors to Zorro that each aim to address a subset of Zorro's limitations. Preliminary experiments show that these approaches can further expand on the trade-offs made available to Zorro to increase accuracy and reduce recovery time.

BIBLIOGRAPHY

- [1] Apache Hama. <https://hama.apache.org/>. Last accessed 2016-04-18.
- [2] Scaling Apache Giraph to a Trillion Edges. <https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920/>. Last accessed 2016-04-18.
- [3] Stanford Network Analysis Project. <http://snap.stanford.edu/>. Last accessed 2016-04-18.
- [4] BHARGAVA, B., AND LIAN, S. R. Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. In *Proceedings of the Symposium on Reliable Distributed Systems* (1988), IEEE.
- [5] BOLDI, P., ROSA, M., SANTINI, M., AND VIGNA, S. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the International Conference on World Wide Web (WWW)* (2011), ACM.
- [6] BOLDI, P., AND VIGNA, S. The WebGraph framework I: Compression techniques. In *Proceedings of the International World Wide Web Conference (WWW)* (2004), ACM.
- [7] BOTA, M., DONG, H.-W., AND SWANSON, L. W. From gene networks to brain networks. *Nature Neuroscience* 6, 8 (2003), 795–799.
- [8] CAMPBELL, R. H., AND RANDELL, B. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, 8 (1986), 811–826.
- [9] CHANDY, K. M., AND LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. 63–75.
- [10] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2015), ACM.

- [11] CHING, A., EDUNOV, S., KABILJO, M., LOGOTHETIS, D., AND MUTHUKRISHNAN, S. One trillion edges: graph processing at Facebook-scale. *Proceedings of the VLDB Endowment* (2015).
- [12] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)* (2005), USENIX.
- [13] ELNOZAHY, E. N., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)* 34, 3 (2002), 375–408.
- [14] FALOUTSOS, M., FALOUTSOS, P., AND FALOUTSOS, C. On power-law relationships of the internet topology. *ACM SIGCOMM Computer Communication Review* 29, 4 (1999), 251–262.
- [15] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *SIGOPS Operating Systems Review* (2003), ACM.
- [16] GJOKA, M., KURANT, M., BUTTS, C. T., AND MARKOPOULOU, A. Walking in Facebook: A case study of unbiased sampling of OSNs. In *Proceedings of the International Conference on Computer Communications (INFOCOM)* (2010), IEEE.
- [17] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)* (2012), USENIX.
- [18] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)* (2014), USENIX.
- [19] GUBICHEV, A., BEDATHUR, S., SEUFERT, S., AND WEIKUM, G. Fast and accurate estimation of shortest paths in large graphs. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)* (2010), ACM.
- [20] HOCKNEY, R. W. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Elsevier Parallel computing* 20, 3 (1994), 389–398.
- [21] HOQUE, I., AND GUPTA, I. LFGraph: Simple and fast distributed graph analytics. In *Proceedings of Conference on Timely Results In Operating Systems (TRIOS)* (2013), ACM.

- [22] JOHNSON, D. B., AND ZWAENEPOEL, W. Recovery in distributed systems using asynchronous message logging and checkpointing. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)* (1988), ACM.
- [23] KANG, U., TSOURAKAKIS, C. E., APPEL, A. P., FALOUTSOS, C., AND LESKOVEC, J. Hadi: Mining radii of large graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 5, 2 (2011), 8.
- [24] KO, S. Y., HOQUE, I., CHO, B., AND GUPTA, I. Making cloud intermediate data fault-tolerant. In *Proceedings of the Symposium on Cloud Computing (SoCC)* (2010), ACM.
- [25] KUNEGIS, J. Konect: The Koblenz network collection. In *Proceedings of the International Conference on World Wide Web companion* (2013).
- [26] KURANT, M., GJOKA, M., BUTTS, C. T., AND MARKOPOULOU, A. Walking on a graph with a magnifying glass: stratified sampling via weighted random walks. In *Proceedings of the SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (2011), ACM.
- [27] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a social network or a news media? In *Proceedings of International Conference on World Wide Web (WWW)* (2010), ACM.
- [28] KYROLA, A., BLELLOCH, G. E., AND GUESTRIN, C. Graphchi. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*.
- [29] LEBEANE, M., SONG, S., PANDA, R., RYOO, J. H., AND JOHN, L. K. Data partitioning strategies for graph workloads on heterogeneous clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2015), ACM.
- [30] LEE, C.-H., XU, X., AND EUN, D. Y. Beyond random walk and Metropolis-Hastings samplers: why you should not backtrack for unbiased graph sampling. *ACM SIGMETRICS Performance Evaluation Review* 40, 1 (2012), 319–330.
- [31] LESKOVEC, J., AND FALOUTSOS, C. Sampling from large graphs. In *Proceedings of the SIGKDD International Conference on Knowledge Discovery and Data Mining* (2006), ACM.
- [32] LESKOVEC, J., KLEINBERG, J., AND FALOUTSOS, C. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 1, 1 (2007), 2–14.

- [33] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. In *Proceedings of VLDB Endowment* (2012), VLDB Endowment.
- [34] LOWRY, A., RUSSELL, J. R., AND GOLDBERG, A. P. Optimistic failure recovery for very large networks. In *Proceedings of the Symposium on Reliable Distributed Systems* (1991), IEEE.
- [35] MACKO, P., MARATHE, V. J., MARGO, D. W., AND SELTZER, M. I. LLAMA: Efficient graph analytics using large multiversioned arrays. In *Proceedings of the International Conference on Data Engineering (ICDE)* (2015), IEEE.
- [36] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of International Conference on Management of Data (SIGMOD)* (2010), ACM.
- [37] MITLIAGKAS, I., BOROKHOVICH, M., DIMAKIS, A. G., AND CARAMANIS, C. FrogWild!—fast PageRank approximations on graph engines. In *Proceedings of VLDB Endowment* (2015), VLDB Endowment.
- [38] NAGARAJAN, A. B., MUELLER, F., ENGELMANN, C., AND SCOTT, S. L. Proactive fault tolerance for HPC with Xen virtualization. In *Proceedings of the International Conference on Supercomputing (SC)* (2007), ACM.
- [39] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., AND OUSTERHOUT, J. Fast crash recovery in RAMCloud. In *Proceedings of Symposium on Operating Systems Principles (SOSP)* (2011), ACM.
- [40] PETRONI, F., QUERZONI, L., DAUDJEE, K., KAMALI, S., AND IACOBONI, G. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)* (2015), ACM, pp. 243–252.
- [41] POWER, R., AND LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)* (2010), USENIX.
- [42] PUNDIR, M., KUMAR, M., LESLIE, L. M., GUPTA, I., AND CAMPBELL, R. H. Supporting on-demand elasticity in distributed graph processing. In *Proceedings of the International Conference on Cloud Engineering (IC2E)* (2016), IEEE. Best Paper Award.
- [43] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.

- [44] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of Symposium on Operating Systems Principles (SOSP)* (2013), ACM.
- [45] SALIHOGLU, S., AND WIDOM, J. GPS: A graph processing system. In *Proceedings of International Conference on Scientific and Statistical Database Management* (2013), ACM.
- [46] SCHELTER, S., EWEN, S., TZOUMAS, K., AND MARKL, V. All roads lead to Rome: Optimistic recovery for distributed iterative data processing. In *Proceedings of International Conference on Information and Knowledge Management (CIKM)* (2013), ACM.
- [47] SEIDMAN, S. B. Network structure and minimum degree. *Elsevier Social Networks* 5, 3 (1983), 269–287.
- [48] SHEN, Y., CHEN, G., JAGADISH, H. V., LU, W., OOI, B. C., AND TUDOR, B. M. Fast failure recovery in distributed graph processing systems. In *Proceedings of the VLDB Endowment* (2015), VLDB Endowment.
- [49] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In *Proceedings of the Symposium on Mass Storage Systems and Technologies (MSST)* (2010), IEEE.
- [50] STROM, R., AND YEMINI, S. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 3 (1985), 204–226.
- [51] WANG, P., ZHANG, K., CHEN, R., CHEN, H., AND GUAN, H. Replication-based fault-tolerance for large-scale graph processing. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)* (2014), IEEE.
- [52] YOUNG, J. W. A first order approximation to the optimum checkpoint interval. In *Communications of the ACM* (1974), ACM.
- [53] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of Conference on Networked Systems Design and Implementation (NSDI)* (2012), USENIX.