CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing

Keval Vora[1] Chen Tian[2] Rajiv Gupta[1] Ziang Hu[2]

[1] *University of California, Riverside*
{kvora001,gupta}@cs.ucr.edu

[2] *Huawei US R&D Center*
{Chen.Tian,Ziang.Hu}@huawei.com

Abstract

Existing distributed asynchronous graph processing systems employ checkpointing to capture *globally consistent* snapshots and rollback *all machines* to most recent checkpoint to recover from machine failures. In this paper we argue that recovery in distributed asynchronous graph processing does not require the entire execution state to be rolled back to a globally consistent state due to the relaxed asynchronous execution semantics. We define the properties required in the recovered state for it to be usable for correct asynchronous processing and develop CoRAL, a lightweight checkpointing and recovery algorithm. First, this algorithm carries out *confined recovery* that only rolls back graph execution states of the failed machines to affect recovery. Second, it relies upon lightweight checkpoints that capture *locally consistent snapshots* with a reduced peak network bandwidth requirement. Our experiments using real-world graphs show that our technique recovers from failures and finishes processing 1.5× to 3.2× faster compared to the traditional asynchronous checkpointing and recovery mechanism when failures impact 1 to 6 machines of a 16 machine cluster. Moreover, capturing locally consistent snapshots significantly reduces intermittent high peak bandwidth usage required to save the snapshots – the average reduction in 99th percentile bandwidth ranges from 22% to 51% while 1 to 6 snapshot replicas are being maintained.

1. Introduction

As graphs have become a popular means for representing large data sets, many systems for performing iterative graph analytics have been developed. Due to the large sizes of real-world graphs, distributed systems are a natural choice for analyzing them as they provide scalability in both processing and memory resources. Some examples of distributed graph processing systems include Google's Pregel [13], GraphLab [12], GraphX [7], GPS [19], ASPIRE [25], Trinity [20] etc. In these frameworks, the user defined algorithm is expressed in a *vertex centric* manner, i.e., computations are written from the perspective of a single vertex. Graph is iteratively processed until computation converges to a fixed point, i.e., all computed vertex/edge values in the graph stabilize. The workload of performing vertex centric processing is distributed across machines by dividing the vertices among the machines.

There are two execution models to iteratively process a graph using vertex centric functions. The *synchronous* (or the bulk synchronous parallel [22]) model guarantees that processing in current iteration is based on values computed in the preceding iteration. The *asynchronous* model [8, 26, 31] on the other hand permits processing to be based upon values that become available during the current iteration. Moreover, it permits read-write dependences to be further relaxed to accelerate processing. For example, ASPIRE [25] relaxes remote reads in computations by satisfying them using local *stale* values, i.e. values computed in older iterations. It shows that resulting asynchronous algorithms significantly outperform synchronous ones. Pregel, GraphX, and GPS rely on the synchronous model and GraphLab supports both synchronous and asynchronous models.

Fault tolerance in distributed graph processing systems is provided by periodically snapshotting the vertex/edge values of the data-graph during processing, and restarting the execution from the latest saved snapshot during recovery [13, 18, 27]. The cost of fault tolerance includes: overhead of periodic checkpoints that capture *globally consistent snapshots* [3] of a graph computation; and *repeating computation* whose results are discarded by the roll back performed during the recovery process. For synchronous graph processing systems, solutions that lower these overheads have been proposed. Pregel [13] performs *confined recovery* that begins with the most recently checkpointed graph state of the failed machine and, by replaying the saved inputs used by boundary vertices, recovers the lost graph state. Zorro [18] avoids checkpointing and directly recovers the lost graph state at the cost of sacrificing accuracy of computed solutions.

Due to its higher performance [9, 25, 26], asynchronous graph processing is preferable to synchronous processing.

ASPLOS '17, April 08-12, 2017, Xi'an, China
Copyright © 2017 ACM 978-1-4503-4465-4/17/04. . . $15.00
DOI: http://dx.doi.org/10.1145/3037697.3037747

However, development of efficient fault tolerance techniques for asynchronous graph processing has not achieved much attention. Asynchronous graph processing systems necessitate rolling back the entire graph state to the most recently checkpointed globally consistent snapshot, i.e. graph state at machines which are unaffected by failures is also rolled back. Such a recovery is expensive since the overall progress suffers a set back and previously performed computations on non-failing machines are also repeated.

In this paper we present CoRAL, a highly optimized recovery technique for asynchronous graph processing that is the first *confined recovery* technique for asynchronous processing. We observe that the correctness of graph computations using asynchronous processing rests on enforcing the *Progressive Reads Semantics* (PR-Semantics) which results in the graph computation being in PR-Consistent State. Therefore recovery need not roll back graph state to a globally consistent state; it is sufficient to restore state to a PR-Consistent state. We leverage this observation in two significant ways. First, non-failing machines do not roll back their graph state, instead they carry out *confined recovery* by reconstructing the graph states of failed machines such that the computation is brought into a PR-Consistent state. Second, globally consistent snapshots are no longer required, instead *locally consistent snapshots* are used to enable recovery.

We have implemented our technique and evaluated it on a cluster on Amazon EC2. Our experiments using real-world graphs show that our technique recovers from failures and finishes processing 1.5× to 3.2× faster compared to the traditional asynchronous checkpointing and recovery mechanism when failures impact 1 to 6 machines of a 16 machine cluster. Moreover, capturing locally consistent snapshots significantly reduces intermittent high bandwidth usage required to save the snapshots – the average reduction in 99th percentile peak bandwidth ranges from 22% to 51% while 1 to 6 snapshot replicas are being maintained.

2. Background and Motivation

Distributed graph processing systems provide fault tolerance to handle machine failures that can occur in the midst of a graph computation. The failure model assumes *fail-stop* failures, i.e. when a machine fails, it does not lead to malicious/unexpected behavior at other machines. Once a machine in a cluster fails, its workload can be distributed across remaining machines in the cluster or the failed machine can be replaced by another (cold) server.

Fault tolerance is provided via *checkpointing* and rollback based *recovery* mechanism [5]. The checkpoints, that are performed periodically, save a *globally consistent snapshot* [3] of the state of a graph computation. A captured snapshot represents the state of the entire distributed graph computation such that it includes a valid set of values from which the processing can be resumed. Therefore *recovery*, that is performed when a machine fails, rolls back the computation to the latest checkpoint using the saved snapshot and resumes execution. For capturing a globally consistent snapshot, both synchronous and asynchronous checkpointing methods exist [14]. Synchronous checkpointing suspends all computations and flushes all the communication channels before constructing the snapshot by capturing the graph computation state at each machine whereas asynchronous checkpointing incrementally constructs the snapshot as the computation proceeds. The frequency of capturing snapshots balances checkpointing and recovery costs [28].

A *globally consistent snapshot* of a distributed graph computation is defined below.

Definition 2.1. Given a cluster $C = \{c_0, c_1, ..., c_{k-1}\}$, a *Globally Consistent Snapshot* of C is a set of local snapshots, denoted as $S = \{s_0, s_1, ..., s_{k-1}\}$, such that it satisfies [P-LCO] and [P-GCO], as specified below.

[P-LCO]: Each $s_i \in S$ represents a consistent state of the subgraph processed by $c_i \in C$ (i.e., vertex/edge values iteratively computed by c_i) that is computable by the processing model from the initial state of the graph.

[P-GCO]: S represents a globally consistent state of the entire graph that is computable by the processing model from the initial state of the graph.

Note that a local snapshot of a machine simply consists of vertex/edge values computed by that machine; the structure of the graph, along with any static vertex and edge values, are not captured in the snapshot because they remain the same throughout the computation.

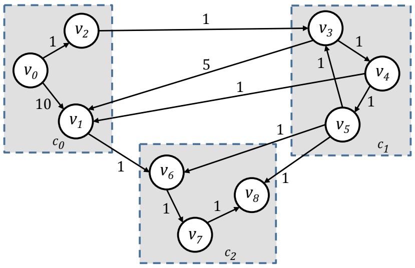


Figure 1: Example graph.

	c_0			c_1			c_2		
Step	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
t_0	0	∞	∞	∞	∞	∞	∞	∞	∞
t_1	0	10	1	∞	∞	∞	∞	∞	∞
t_2	0	10	1	2	∞	∞	11	∞	∞
t_3	0	7	1	2	3	∞	11	12	∞
t_4	0	4	1	2	3	4	8	12	13
t_5	0	4	1	2	3	4	5	9	5
t_6	0	4	1	2	3	4	5	6	5

Table 1: SSSP example.

Consider the graph shown in Figure 1. Vertices v_0 through v_8 are partitioned across machines c_0, c_1, and c_2. After partitioning, the edges that cross machines translate into *remote* reads. A vertex having at least one neighbor residing on a different machine is called a *boundary* vertex – in

	Execution Model		Checkpointing Model		Checkpointing Consistency Model		Recovery Model		Accuracy
	Sync	Async	Sync	Async	Globally Consistent	Locally Consistent	All Rollback	Minimal Rollback	
Pregel	✓	✗	✓	✗	✓	✗	✗	✓	✓
GraphLab	✓	✓	✓	✓	✓	✗	✓	✗	✓
GPS	✓	✗	✓	✗	✓	✗	✓	✗	✓
GraphX	✓	✗	✓	✗	✓	✗	✓	✗	✓
Imitator	✓	✗	Replication		Consistent Replication		None		✓
Zorro	✓	✗	None		None		✗	✓	✗
CoRAL	✗	✓	✗	✓	✗	✓	✗	✓	✓

Table 2: Key characteristics of existing graph processing systems and our CoRAL system.

this example, vertices v_1 and v_2 are boundary vertices of machine c_0. The boundary vertices are usually replicated on remote machines so that they are readily available to remote neighbors for their computation. Table 1 shows how computation of shortest paths proceeds with v_0 as source. The table shows steps t_0 through t_6 of the computation. Let us assume that a globally consistent checkpoint captures (highlighted) values at step t_3. Now, if c_2 fails at t_5, instead of starting from the state at t_0 (first row), all the execution states are rolled back to the snapshot taken at t_3 and the processing is resumed from this rolled back state.

Next we summarize the pros and cons of existing checkpointing and recovery methods to motivate our approach.

(1) Synchronous processing systems like Pregel, GPS, GraphLab's synchronous model, and Trinity's synchronous model use synchronous checkpointing that captures globally consistent snapshot by initiating the checkpointing process at the start of a global iteration (super step). Therefore the values captured are values that exist at the beginning of a global super step. Pregel performs ***confined recovery*** that requires rolling back the state of only the failed machine as follows. After capturing a snapshot, at each machine, the inputs read from other machines for boundary vertices are saved so that they can be replayed during recovery to construct the execution state of the failed machine at the point of failure. Trinity models confined recovery using buffered logging while [21] performs confined recovery in a distributed manner.

Two additional approaches for *fast recovery* have been proposed. Zorro [18] is motivated by the observation in other works (GraphX [7], Giraph [4], and Distributed GraphLab [12]) that users often disable the fault tolerance to accelerate processing. Thus it chooses to discard the checkpointing process altogether to eliminate its overheads. Upon failures, the recovery process constructs an approximate execution state using the replicated boundary vertices residing on remaining machines. Hence, it achieves fast recovery at the cost of sacrificing accuracy [18]. Imitator [27] maintains in-memory replicated globally consistent execution state throughout the execution so that recovery from failure is immediate. The cost of this approach is the overhead of maintaining consistent replicas in memory all the time. Finally, GraphX [7] relies on Spark [29] for tracking the lineage of data in memory, i.e., saving the intermediate results of high-level operations over data; however, when the lineage tree becomes very large, it resorts to checkpointing.

Although for synchronous processing systems recovery has been optimized, their overall performance can be significantly lower than that of asynchronous processing systems [23, 25]. Next, we discuss asynchronous systems.

(2) Asynchronous processing systems like GraphLab's asynchronous model uses asynchronous checkpointing technique that captures the vertex/edge values by developing a mechanism based on the Chandy-Lamport snapshot algorithm [3] that is also used in other domains like Piccolo [17]. While the snapshots captured by such asynchronous checkpointing reflect values coming from global states at different centralized clock times, the order in which the values are captured with respect to communication and computation performed guarantee that the snapshot is globally consistent. Trinity's asynchronous model, on the other hand, interrupts execution to capture the global snapshot.

While Pregel's confined recovery is useful as it only rolls back the state of the failed machine, it is applicable only for synchronous processing environments since the order in which iterations progress is deterministic with respect to the inputs. For asynchronous execution, the lost execution state cannot be reconstructed using the saved inputs because the order in which vertex computations observe and process the input values varies over the execution, thus making this technique inapplicable. As a result, to perform recovery, asynchronous processing systems roll back the states of *all the machines* to the last available snapshot and resume the computation from that point. This approach has the following two drawbacks:

– *Redundant computation*: Since recovery rolls back the states of *all* machines to the latest snapshot, when processing is resumed, the computation from snapshot to the current state is repeated for machines that did not fail.

– *Increased network bandwidth usage*: The local snapshots are saved on remote machines, either on the distributed file system or in memory. All machines bulk transfer snapshots over the network simultaneously. This stresses the network and increases peak bandwidth usage. The problem gets worse in case of a multi-tenant cluster.

Summary. Table 2 summarizes the key characteristics of above frameworks. The existing systems rely upon globally consistent snapshots for recovery. Apart from the synchronous solutions of Pregel and Zorro, none of the works perform minimal rollback. GraphLab's asynchronous engine captures globally consistent snapshots and rolls back the state of all the machines. Solutions that do not rely on recovery via rollback to a checkpoint, either incorporate consistent replication (Imitator) or relax the correctness guarantees that leads to imprecise results in case of failures (Zorro).

3. Confined Recovery for Asynchronous model via Lightweight checkpointing

The goal of our work is to develop a technique that: (1) uses *asynchronous* processing model as it provides high performance; (2) performs *minimal* rollback and avoids network bandwidth problem due to checkpointing; and (3) achieves complete recovery so that the final solutions are guaranteed to be *accurate*. Next we present **CoRAL**, a **Co**nfined **R**ecovery technique for iterative distributed graph algorithms being executed under the **A**synchronous processing model that uses **L**ightweight checkpoints.

Characteristics of Asynchronous Graph Computation. Under the asynchronous model [24, 25], graph computations are inherently non-deterministic because the model relaxes read-write dependences to allow machines to use stale values of remote vertices such that all machines can continue processing independently by skipping intermediate updates from remote vertices. As a consequence, under the asynchronous model, there are multiple legal executions, all of which upon convergence produce the same final results.

Asynchronous execution typically orders the values read for each vertex x via the *Progressive Reads Semantics* (PR-Semantics) such that over time, x is assigned different values $v(x, 0), v(x, 1), \cdots, v(x, n)$ by the machine on which it resides and these values are used (read) during processing on other machines.

Definition 3.1. *PR-Semantics* ensures that if a read of x performed by a thread observes the value $v(x, i)$, the subsequent read of x by that same thread must observe value $v(x, j)$ such that it either satisfies [V-SAM] or [V-FUT] as given below:

[V-SAM]: $j = i$, that is, the same value is observed; or

[V-FUT]: $j > i$, that is, a fresher value is observed on the second read.

This means that, once a value for any data item is read by a thread, no earlier values of that data item can be read by the same thread. The PR-Semantics ensures that each thread observes the values for a given data item in the same order as they were produced, and hence, convergence and correctness of asynchronous algorithms can be reasoned about.

Definition 3.2. An execution state $E = \{e_1, e_2, ..., e_{k-1}\}$ of a graph computation is *PR-Consistent* if it is reached by performing the graph computation using an asynchronous processing model that follows the PR-Semantics.

Thus, following a failure, the recovery process must construct the state of the subgraph(s) lost due to machine failure(s) such that the resulting computation is in a PR-Consistent state. Resuming execution from a PR-Consistent state guarantees that all future remote reads adhere to PR-Semantics.

The reliance of graph processing algorithms on *PR-Semantics* and *PR-Consistent* state can be found in literature. In [6] the self-stabilizing nature of PageRank algorithm is proven assuming that the underlying system guarantees progressive reads. Below we derive an equivalence between a PR-Consistent execution and a legal asynchronous bounded staleness based execution [24, 25].

Theorem 3.1. *Every PR-Consistent execution state of graph computation starting from an initial state I is equivalent to an execution state under some legal staleness based asynchronous execution* [24, 25] *starting from I.*

Proof. The full PR-Consistent execution can be viewed as a sequence of intermediate PR-Consistent execution states $E_0 \rightarrow E_1 \rightarrow E_2 \rightarrow ... \rightarrow E_n$ starting at the initial state $I = E_0$. Hence, we prove this theorem using induction on $E_i (0 \leq i \leq n)$.

BASE CASE ($E_i = E_0 = I$): E_0 is PR-Consistent since no reads are performed. It is the same starting execution state for staleness based asynchronous execution.

INDUCTION HYPOTHESIS ($E_i = E_k, k > 0$): E_k is PR-Consistent and is equivalent to an execution state under some legal staleness based asynchronous execution.

INDUCTION STEP ($E_i = E_{k+1}$): E_{k+1} is a PR-Consistent execution state constructed after E_k based on values read from vertices in V_{k+1}. Without loss of generality, let us consider a vertex $x \in V_{k+1}$ whose latest value read prior to computation of E_{k+1} is $v(x, p)$. When performing a computation requiring the value of x, the *current value* of x is first read and then used in the computation. From Definition 3.1, we know that the value read for computation of E_{k+1} is $v(x, q)$ such that $q \geq p$. Between the reading of this value and its use, the value of x can be changed to $v(x, r)$ by another computation, i.e., $r \geq q$. This leads to the following cases:

– Case 1 ($q = r$): The second read returned $v(x, q) = v(x, r)$. In the equivalent staleness based execution, this read is considered to have returned the *current* or the *freshest* value available for x, and hence is usable, which results in the same computation being performed.

– Case 2 ($q < r$): The second read returned $v(x, q) \neq v(x, r)$. In the equivalent staleness based execution, such a value is considered to be *stale* by $r - q$ versions, and is still usable in the asynchronous model where staleness bound b^x_{k+1} is at least $r - q$. Note that the staleness bound does not impact correctness, it merely impacts the performance of asynchronous execution [25].

The above reason can be applied to all the vertices in V_{k+1} whose values are used to compute E_{k+1}. Let $s = \max_{0>j\leq k+1}(\max_{x\in V_j}(\ b^x_j\ ))$ be the maximum staleness of reads across all the vertex values read in Case 2. Across all the possibilities, the computations in PR-Consistent execution and an asynchronous staleness based execution with staleness bound of at least s (i.e., $\geq s$) are equivalent since they are based on same values, and hence, they result in the equivalent or same execution state E_{k+1}. □

Corollary 3.1. *The final execution state reached by a PR-Consistent execution is equivalent to the final execution state under some legal staleness based asynchronous execution* [24, 25].

To further illustrate the efficacy of *PR-Semantics*, we consider SSSP, a popular example of monotonic graph algorithms (other examples include Connected Components, K-Core, etc.) where vertex values exhibit *monotonicity* which cannot be preserved without *PR-Semantics*. Table 3 shows the effect of violating the PR-Semantics at t_5 in our SSSP example from Figure 1. If at t_5, c_0 observes the old value of $v_4 = \infty$ after having observed $v_4 = 3$ at t_4, the value of v_1 is computed as 7 via v_3 as shown below.

$$\begin{aligned} path(v_1) = min(&path(v_3) + weight(v_3, v_1),\\ &path(v_4) + weight(v_4, v_2))\\ = min(&2 + 5, \infty + 1) = 7 \end{aligned}$$

	c_0			c_1			c_2		
Step	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
t_0	0	∞	∞	∞	∞	∞	∞	∞	∞
t_1	0	10	1	∞	∞	∞	∞	∞	∞
t_2	0	10	1	2	∞	∞	11	∞	∞
t_3	0	7	1	2	3	∞	11	12	∞
t_4	0	4	1	2	3	4	8	12	13
t_5	0	7	1	2	3	4	5	9	5

Table 3: Violation of PR-Semantics disrupting monotonicity in SSSP.

This violates the monotonicity property of SSSP because the shortest path value for v_2, instead of decreasing, increases from 4 to 7.

Overview of PR-Consistent Recovery. The above characteristics of asynchronous graph processing lead to new more *relaxed notion of recovery*, called *PR-Consistent* recovery, that allows use of confined recovery using lightweight checkpoints. Its key features follow.

(1) *Confined Recovery*: Given s_f and c_f such that s_f is the state of the subgraph on machine c_f just before it fails. The state of the subgraph on c_f following recovery, say s_r, need not be the same as s_f. However, both s_f and s_r must correspond to legal executions during which the PR-Semantics is preserved. We exploit this flexibility to achieve *Confined Recovery*, i.e. the subgraph states at non-failing machines is not rolled back.

(2) *Lightweight Checkpoints*: Deriving the recovered state s_r does not require globally consistent snapshots. It simply requires periodically taken local snapshots of all machines which we refer to as *Locally Consistent Snapshots*. The global ordering across the local snapshots, called *PR-Ordering*, must be captured to enforce PR-Semantics during confined recovery for *multiple machine failures*. The sufficiency of locally consistent snapshots solves the problem of *increased network bandwidth usage* due to bulk network transfer for saving snapshots during checkpointing. The decision to capture a local snapshot at a given point in time can be made either by a central coordinator to minimize the number of snapshots being simultaneously saved, or locally by the machine in which snapshot is to be captured.

(3) *Fast Recovery*: Once a machine in a cluster fails, its workload is distributed across remaining machines in the cluster which then collectively reconstruct the state s_r in parallel. To further reduce checkpointing overhead and speedup recovery, the replicated snapshots are stored in-memory on remote machines. Both of these design decisions are based on RAMCloud's approach [15] for fast replication and recovery; however, our technique is applicable if a failed machine is replaced by a cold server and snapshots are stored on a distributed file system.

In summary, CoRAL captures light-weight Locally Consistent Snapshots and PR-Ordering information that allow the recovery of the state(s) corresponding to failed machine(s) such that reconstructed state is PR-Consistent from which execution can be correctly resumed.

3.1 PR-Consistent Recovery: *Single Failure Case*

For ease of understanding, in this section we show how PR-Consistent state is restored in case of a single machine failure and in the next section we present the additions required to handle multiple simultaneous machine failures.

We introduce the concept of *Locally Consistent Snapshots* and then present a recovery algorithm that uses them to construct the *PR-Consistent* state following a failure. Since a globally consistent checkpoint captures a valid graph state, following a failure, rolling back entire graph state to such a captured state is sufficient to restore execution to a PR-Consistent state. However, restoring state via a globally consistent snapshot is too strong of a requirement, i.e. it is not necessary for satisfying PR-Semantics after recovery. In fact, allowing global inconsistencies in the captured graph state is acceptable due to the relaxed nature of asynchronous execution model semantics.

A *locally consistent checkpoint* represents this relaxed notion of a distributed snapshot. Next we define a locally consistent snapshot of the system.

Definition 3.3. Given a cluster $C = \{c_0, c_1, ..., c_{k-1}\}$, a *Locally Consistent Snapshot* $S = \{s_0, s_1, ..., s_{k-1}\}$, is defined as a set of local snapshots such that it satisfies [P-LCO] as specified below.

[P-LCO]: Each $s_i \in S$ represents a consistent state of the subgraph processed by $c_i \in C$ that is computable by the processing model from the initial state of the graph.

Note that locally consistent checkpoints do not enforce consistency requirement across different local snapshots and hence, eliminate the need to save snapshots at the same time; by staggering their collection over time, the stress on network bandwidth is lowered. Also, since failures can occur while a snapshot is being captured and transferred to remote machines, the snapshot should not be committed until the entire snapshot has been received.

The recovery process has two primary goals: first, the execution state should be restored to a PR-Consistent state; and second, the execution state of the machines which are not affected by failures must not be rolled back, i.e., the recovery process should be confined to workload of the failed machine.

Formally, let $E^c = \{e^c_0, e^c_1, ..., e^c_i, ..., e^c_{k-1}\}$ represent the latest execution state of machines in C right before failure of a single machine $c_i \in C$. Due to failure, the local execution state e^c_i is lost and the remaining available execution state is $E^{cf} = E^c \setminus \{e^c_i\}$. The recovery process must reconstruct the local execution state e^r_i of c_i such that, $E^r = E^{cf} \cup \{e^r_i\}$ represents a PR-Consistent state while, e^r_i may be different from e^c_i. Figure 2 shows this recovery process – when e^c_i is lost, the subgraph is processed using values from s_i and available inputs from E^c (i.e., E^{cf}) to generate e^r_i.

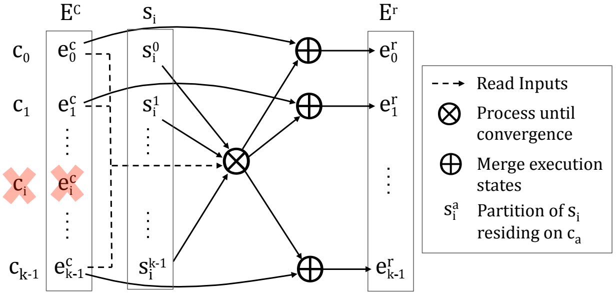


Figure 2: Recovery from single failure.

Next we consider the recovery algorithm. Let s_i be the last snapshot captured for e_i during checkpointing. Naïvely constructing e^r_i by directly using values from s_i does not represent a PR-Consistent state because $\forall e^c_j \in E^{cf}$, the values in e^c_j can be based on fresher values from c_i which became available after capturing s_i and hence, further reads from e^r_i will violate the PR-Semantics.

We use $=_{PR}$ to denote the PR-Consistent relationship between two local execution states, i.e., if e_a and e_b are PR-Consistent, then $e_a =_{PR} e_b$. Hence, we want to construct e^r_i such that $\forall e^c_j \in E^{cf}$, $e^r_i =_{PR} e^c_j$.

Algorithm 1 constructs a PR-Consistent state E^r. The algorithm first loads the subgraph which was handled by the failed machine and initializes it with: values from s_i (line 4); and current values of boundary vertices coming from E^{cf} (line 5). Note that this initialization of boundary vertex replicas does not violate PR-Semantics because values in s_i are based on older values of boundary vertices which were available when s_i was captured. Then the created subgraph e_i is iteratively processed in isolation until convergence (line 6) – this is the crucial step in the algorithm. Fully processing e_i ensures that the effects of fresher boundary vertex' values are fully propagated throughout the subgraph. Hence, the values in e^c_i before failure were either older than or at most same as the values in e^r_i. This means, any further reads from e^r_i performed by any $e^c_j \in E^{cf}$ return fresher values and hence, do not violate PR-Semantics, i.e., $\forall e^c_j \in E^{cf}$, $e^r_i =_{PR} e^c_j$. Hence, e^r_i is included in E^{cf} (line 7) to represent the PR-Consistent state E^r which is used to resume processing.

Algorithm 1: Recovery from single failure.

```
1: $s_i$: Snapshot of failed machine $c_i$
2: $E^{cf}$: Current execution state of remaining machines
3: function RECOVER ( )
4:     $e_i \leftarrow$ LOADSUBGRAPH($s_i$)
5:     READBOUNDARYVERTICES($e_i$, $E^{cf}$)
6:     $e^r_i \leftarrow$ PROCESSUNTILCONVERGENCE($e_i$)
7:     $E^r \leftarrow E^{cf} \cup \{e^r_i\}$
8:     return $E^r$
9: end function
```

3.2 PR-Consistent Recovery: *Multiple Failures*

Recovering from a failure impacting multiple machines introduces an additional challenge. To recover a PR-Consistent state, we must ensure that the recovery process operates on the snapshots of failed machines such that it does not violate the PR-Semantics. This means, the PR-Consistent state must be constructed by carefully orchestrating the *order* in which snapshots are included and processed for recovery. Hence, we introduce the concept of *PR-Ordering of Local Snapshots*, which is required to carry out PR-Consistent confined recovery following failure of multiple machines.

PR-Ordering of Local Snapshots. To recover a state after which any further reads will adhere to progressive reads semantics, we must capture the read-write dependences between data elements across different local snapshots that were truly imposed due to *PR-Semantics*. Capturing this information at the level of each data item is expensive due to two reasons: 1) the space of the snapshots blows up with number of inter-machine dependencies, which in graph processing is based on the edge-cut; and 2) capturing such information requires synchronization between the local machine and all other machines.

PR-Semantics naturally enforces dependency ordering across data values. We lift this dependency ordering to a higher level of abstraction – the local snapshots. If we only

	c_0			c_1			c_2		
Step	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
t_0	0	∞	∞	∞	∞	∞	∞	∞	∞
t_1	0	10	1	∞	∞	∞	∞	∞	∞
t_2	0	10	1	2	∞	∞	11	∞	∞
t_3	0	7	1	2	3	∞	11	12	∞
t_4	0	4	1	2	3	4	8	12	13
t_5	0	4	1	2	3	4	5	9	5

Table 4: State of execution till t_5; highlighted rows indicate latest locally consistent snapshots.

Step	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
t_6	0	7	1	∞	∞	∞	5	9	5
t_6	0	7	1	2	∞	6	5	9	5
t_7	0	7	1	2	3	6	5	9	5
t_8	0	7	1	2	3	4	5	9	5

Table 5: Recovering vertices v_3, v_4 and v_5.

Step	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
t_8	0	7	1	2	3	4	5	9	5
t_9	0	4	1	2	3	4	5	9	5

Table 6: Recovering vertices v_0 to v_5.

track the ordering incurred due to progressive reads across the local snapshots, the amount of information maintained per snapshot reduces drastically. This brings us to the definition of ordering of local snapshots.

A *PR-Ordering* of snapshots, denoted as $\leq_{PR}$, defines an ordering across local snapshots based on the order of reads performed by data-elements within the snapshot.

Definition 3.4. The *PR-Ordering* of a pair of local snapshots, denoted as $s_i \leq_{PR} s_j$, indicates that the values in s_i for machine c_i were computed based on values from machine c_j which were available no later than when s_j for c_j was captured.

In other words, $s_i \leq_{PR} s_j$ ensures that values captured in s_i were based on reads of data from c_j prior to capturing s_j. This naturally leads us to the following observation. While PR-Ordering is a pairwise ordering across local snapshots, we prove that a total PR-Ordering is required to perform recovery of PR-Consistent state.

Formally, let $E^c = \{e^c_0, e^c_1, ..., e^c_{k-1}\}$ represent the latest execution state of machines in C right before failure of machines in $F \subset C$. Let $E^l = \{e^c_i \mid c_i \in F\}$ be set of local execution states lost due to failure, leaving the remaining available execution state to be $E^{cf} = E^c \setminus E^l$. The goal of the recovery is to reconstruct the set of local execution states $E^{rf} = \{e^r_i \mid c_i \in F\}$ of failed machines such that, $E^r = E^{cf} \cup E^{rf}$ represents a PR-Consistent state while, $\forall c_i \in F$, e^r_i may be different from e^c_i.

Next we illustrate the necessity of PR-Ordering during recovery using our SSSP example. Let us assume that c_0 and c_1 fail after t_5. The locally consistent snapshots captured at c_0 and c_1 are highlighted in Table 4. Thus, during recovery, the local state of c_2 is that at t_5 while the latest available snapshots s_0 and s_1 from c_0 and c_1 represent their execution states at t_3 and t_1 respectively. By examining the dependences in the computation, we easily determine that $s_1 \leq_{PR} s_0 \leq_{PR} c_2$. Therefore, s_1 is processed first using values from s_0 and c_2 resulting in values shown in Table 5. After t_8, recovery for s_0 can read from the computed results for v_3, v_4 and v_5 because s_0 is PR-Consistent with these values. Finally, as s_0 is PR-Consistent with c_2, processing occurs for values v_0 to v_5 alone, as shown in Table 6. After t_9, all the values are PR-Consistent with each other, i.e., recovery is complete and processing resumes from this state. Note that if we had ignored PR-Ordering monotonicity would be violated. For example, if we had first performed computation over s_0, then by reading $v_3 = \infty$ from s_1, v_4 would have been computed as 10, violating monotonicity as $10 > 7$.

Theorem 3.2. *A total PR-Ordering of S is a necessary condition to recover from failures impacting machines in F, for all possible $F \subset C$, to a PR-Consistent state using a locally consistent snapshot S.*

Proof. We must show that if the execution state recovers to a PR-Consistent state using S, then a total PR-Ordering of S must be available. We prove this by contraposition. Let us assume that a total PR-Ordering of S is not available and hence, snapshots of failed machines $c_i, c_j \in F$, i.e., $s_i, s_j \in S$, are not ordered under $\leq_{PR}$. Without loss of generality, we focus on how the local execution state for c_i can be restored to e^r_i using s_i so that $E = E^{cf} \cup \{e^r_i\}$ is PR-Consistent. When e^r_i is initialized with s_i, E cannot be guaranteed to be PR-Consistent because $\forall e^c_k \in E^{cf}$, $e^c_k \leq_{PR} s_i$ cannot be guaranteed. Hence, e^r_i must be processed further after it is initialized using values from s_i. While processing e^r_i, values of boundary vertices from c_j can be either (Case 1) read from s_j or (Case 2) not read.

Case 1: Processing of e^r_i reads from s_j. In this case PR-Semantics cannot be guaranteed since $s_i \not\leq_{PR} s_j$ may be true. Hence, e^r_i represents an inconsistent state.

Case 2: Processing of e^r_i does not read from s_j. In this case after e^r_i is computed, $\forall e^c_k \in E^{cf}$, $e^c_k \leq_{PR} e^r_i$ cannot be guaranteed because e^c_k could have observed a fresher value from e^c_i prior to failure which was in turn calculated from e_j after s_j was captured. Moreover, recovery of local execution state for c_j cannot be initiated at this point due to the same choice of whether it could or could not read from e^r_i.

This means, e^r_i cannot be constructed such that PR-Consistency for $E = E^{cf} \cup \{e^r_i\}$ is guaranteed. Since failures can impact machines in any non-empty $F \in PowerSet(C)$, recovery of execution state of any failed machine such that it is PR-Consistent with e^c_k, $\forall e^c_k \in E^{cf}$ is not possible if total PR-Ordering of S is not available. □

Now that we know that the total PR-Ordering of S is required for recovery, we aim to construct the PR-Consistent state assuming that such a total PR-Ordering of S is available. The basic idea is to recover the execution states of individual

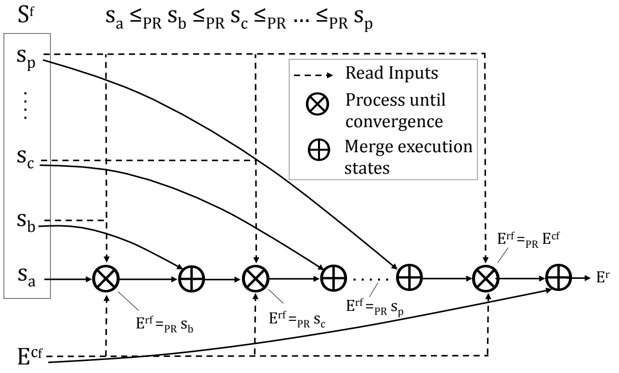


Figure 3: Recovery from multiple failures.

Algorithm 2: Recovery from multiple failures.

```
1: $S^f$: Local snapshots captured on failed machines
2: $E^{cf}$: Execution state of remaining machines at the time of failure
3: function RECOVER ( )
4:     $E^{rf} \leftarrow \varnothing$
5:     $P^f \leftarrow$ SORTASCENDING($S^f$)
6:     while $P^f \neq \varnothing$ do
7:       ▷ Adding Phase
8:         $s_i \leftarrow$ GETFIRST($P^f$)
9:         REMOVE($P^f, s_i$)
10:        $e^r_i \leftarrow$ LOADSUBGRAPH($s_i$)
11:        $E^{rf} \leftarrow E^{rf} \cup \{e^r_i\}$
12:      ▷ Forwarding Phase
13:        READBOUNDARYVERTICES($E^{rf}$, $P^f$, $E^{cf}$)
14:        PROCESSUNTILCONVERGENCE($E^{rf}$)
15:    end while
16:    $E^r \leftarrow E^{rf} \cup E^{cf}$
17:    return $E^r$
18: end function
```

machines one by one (see Figure 3), in an order such that PR-Semantics is never violated.

Algorithm 2 shows recovery from multiple failures. The PR-Consistent execution state (E^{rf}) is constructed by repetitively performing the *adding* and *forwarding* of the saved states from failed machines. The addition of execution states using saved snapshots is done in PR-Order (line 5) to guarantee that PR-Consistency is always retained in E^{rf}. During forwarding, the processing reads inputs, i.e., boundary vertices, from two sources (line 13): the snapshots from failed machines that are not yet incorporated in E^{rf} (i.e., $\forall s_i \in P^f$), and from current execution states of remaining machines (i.e., $\forall e^c_i \in E^{cf}$). At the end of the while loop (lines 6-15), E^{rf} represents the workload of failed machines that is PR-Consistent with the current execution state of the remaining machines (E^{cf}) and hence, the two execution states are merged (line 16) to form the required E^r.

Theorem 3.3. *Algorithm 2 recovers a PR-Consistent execution state.*

Proof. We prove this by first showing that at any point in the algorithm, E^{rf} is PR-Consistent, and then proving that at the end, E^{rf} becomes PR-Consistent with E^{cf}, resulting in E^r to be PR-Consistent.

The algorithm uses $S^f = \{s_i \mid c_i \in F\}$; without loss of generality, let $s_a \leq_{PR} s_b \leq_{PR} s_c \leq_{PR} ... \leq_{PR} s_p$ be the total PR-Ordering of S^f. At each step, the algorithm adds a snapshot into the recovery process in this PR-Order (see Figure 3).

Initially, $E^{rf} = \varnothing$ and e^r_a is initialized with values from s_a. This e^r_a is added to E^{rf}. Due to the available PR-Ordering, e^r_a can read boundary vertices available from s_i, $\forall s_i \in S^f \setminus \{s_a\}$. Also, e^r_a can read from e^c_i, $\forall e^c_i \in E^{cf}$. Hence, processing E^{rf} by allowing it to read boundary vertices from these sources does not violate PR-Semantics.

Since the forwarding phase fully processes E^{rf} until convergence, e^r_a is now based on values from s_b and from other sources which were available even after s_b was captured. On the other hand, when s_b was captured, its values were based on reads from c_a which were not based on fresher values from other sources. Hence, $s_b \leq_{PR} e^r_a$ which further leads to $e^r_a =_{PR} s_b$. This means, when e^r_b is added to E^{rf}, E^{rf} is still PR-Consistent.

Again, e^r_b can read from s_i, $\forall s_i \in S^f \setminus \{s_a, s_b\}$, and also from e^c_i, $\forall e^c_i \in E^{cf}$ which allows processing of E^{rf} to read boundary vertices from these sources without violating PR-Semantics. After the forwarding phase, using the same argument as above, we can show that $e^r_a =_{PR} e^r_b =_{PR} s_c$ which allows e^r_c to be added to E^{rf} while ensuring E^{rf} remains PR-Consistent.

At every step of this construction process, $|E^{rf}|$ increases by 1. When $|E^{rf}| = |F|$, we achieve E^{rf} such that it is only based on values from E^{cf} and hence, $\forall e^r_i \in E^{rf}$ and $\forall e^c_j \in E^{cf}$, $e^c_j \leq_{PR} e^r_i$ which further leads to $E^{rf} =_{PR} E^{cf}$. Hence, the constructed $E^r = E^{rf} \cup E^{cf}$ is a PR-Consistent execution state. □

Maintaining PR-Ordering after Recovery. After the recovery process, $\forall s_i \in S, s_i \leq_{PR} E^{rf}$. Hence, the future snapshots captured after the recovery process are also PR-Ordered with snapshots in S. In case of any further failures, the available snapshots in S before the previous failure can be used along with the newly captured snapshots following recovery. Thus, the snapshots in S and newly captured snapshots collectively guarantee that local states of all machines are available.

Cascading Failures. Failures can occur at any point in time during execution and hence, the recovery process can be affected by new failures at remaining machines. Such cascading failures need to be handled carefully so that the PR-Consistent state constructed by the recovery process includes the workload from newly failed machines.

Since Algorithm 2 incrementally constructs E^{rf} while maintaining the invariant that it is always PR-Consistent, the snapshots of newly failed nodes cannot be directly incorporated in the recovery process. This is because E^{rf} is processed based on values from E^{cf} and allowing a new snapshot to join the recovery process will cause older values to be read by E^{rf} thus violating PR-Semantics. Moreover, the new snapshots cannot be made PR-Consistent with E^{rf} since that in turn requires these snapshots to be PR-Consistent with E^{cf}. Hence, upon cascading failures, the recovery process discards the partially constructed E^{rf} and resumes the process by recreating the linear plan (P^f) consisting of all the failed nodes and then incrementally constructing the PR-Consistent execution state E^{rf}.

Machines Participating in Recovery. In a fail-stop failure model, the snapshots must be replicated on different machines so that they are available for recovery. There are two main ways to replicate a snapshot: either replicate it in entirety on a remote machine, or partition the snapshot into smaller chunks and distribute them across different machines. Both strategies have pros and cons. Placing the snapshot entirely on a single machine allows *confined* recovery for single machine failure with minimal communication. However, this comes at the cost of workload imbalance during post-recovery processing. Partitioning the snapshot and scattering the chunks across the remaining machines provides better load balancing.

3.3 Capturing PR-Ordering

Since the PR-ordering captures the causality relationship across different machines, we use logical timestamps to enable ordering of snapshots. We rely on a light-weight centralized timestamp service to ensure that correct global ordering of logical timestamps is possible. The role of the timestamp service is to atomically provide monotonically increasing timestamps; this does not require synchronization between the machines, allowing asynchronous processing and checkpointing to continue concurrently.

The ordering is captured using a lightweight 3-phase protocol by ensuring that the local execution state to be checkpointed does not change with respect to any new remote input coming during the checkpointing process. The first phase is the *Prepare* phase that blocks the input stream representing remote reads, and then gets a logical timestamp for the snapshot from the distributed coordinator. The second phase is the *Snapshot* phase during which the execution state of the snapshot is actually captured. This phase overlaps computation over vertices while capturing the local snapshot by enforcing that vertex values are saved before they are updated (as in GraphLab [12]) which leads to a locally consistent snapshot (i.e., ensures [P-LCO]). Finally, the third phase is the *Resume* phase which marks the end of snapshot with the acquired logical timestamp and unblocks the input stream to allow future reads. Algorithm 3 summarizes the above protocol for performing local checkpointing which generates correct PR-Ordering across the captured snapshots. The GETNEXTLOGICALTIMESTAMP() function atomically provides a monotonically increasing logical timestamp.

Algorithm 3: Local checkpointing algorithm.

```
function CHECKPOINT ( )
  ▷ Prepare Phase
    BLOCKINCOMINGMESSAGES( )
    $t_s \leftarrow$ GETNEXTLOGICALTIMESTAMP( )
  ▷ Snapshot Phase
    SNAPSHOTUPDATE( )
  ▷ Resume Phase
    SAVE(END-CHECKPOINT, $t_s$)
    UNBLOCKINCOMINGMESSAGES( )
end function

function SNAPSHOTUPDATE ( )
    for $v \in V$ do
      ▷ Snapshot Vertex
        if $v$ is to be checkpointed then
            SAVE($v$)
        end if
      ▷ Process Vertex
        if $v$ is to be processed then
            PROCESS($v$)
        end if
    end for
end function
```

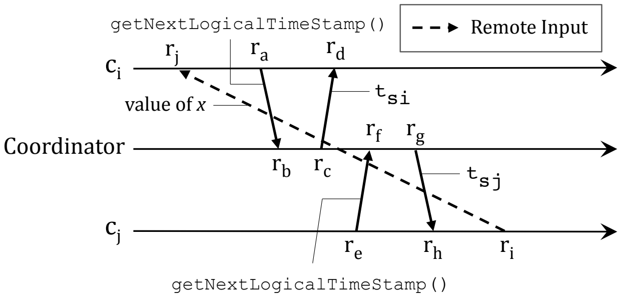


Figure 4: Event sequence with incorrect access of value x.

Theorem 3.4. *Algorithm 3 generates a correct total PR-Ordering across local snapshots in S.*

Proof. We first show that the generated PR-Ordering is a total ordering, and then show its correctness.

TOTAL ORDERING: Each of the local snapshots captured is assigned a unique timestamp via the distributed coordinator. Hence, $\forall s_i, s_j \in S$, their timestamps, t_{s_i} and t_{s_j} are ordered, i.e., either $t_{s_i} < t_{s_j}$ or $t_{s_j} < t_{s_i}$. By mapping this timestamp ordering between t_{s_i} and t_{s_j} to the PR-Ordering between s_i and s_j, we achieve either $s_i \leq_{PR} s_j$ or $s_j \leq_{PR} s_i$. Since this mapping is done for every pair of snapshots in S, S is totally ordered under $\leq_{PR}$.

CORRECT PR-ORDERING: We prove this by contradiction. Let us assume that $s_i \leq_{PR} s_j$ is an incorrect PR-Ordering. This ordering is a result of mapping from timestamp relation $t_{s_i} < t_{s_j}$. Since the logical timestamps are monotonically increasing in the order of arrival of requests, the timestamp request from node c_i should have arrived before than that from node c_j in real time space.

Without loss of generality, Figure 4 shows the sequence of events representing our current case. Note that r_a through r_j indicate real time points in the global real time space. We know the following orderings are valid.

$$r_a < r_b \quad \text{(send-receive ordering)} \tag{1}$$

$$r_b < r_f \quad \text{(request arrival ordering)} \tag{2}$$

$$r_f < r_g \quad \text{(causality ordering)} \tag{3}$$

$$r_g < r_h \quad \text{(send-receive ordering)} \tag{4}$$

Moreover, since our assumption is that $s_i \nleq_{PR} s_j$, there should be a value x which is read from c_j to c_i (indicated via dotted arrow) with the following ordering constraints:

$$r_j < r_a \quad \text{(prepare phase ordering)} \tag{5}$$

$$r_h < r_i \quad \text{(prepare phase ordering)} \tag{6}$$

$$r_i < r_j \quad \text{(send-receive ordering)} \tag{7}$$

Combining Equations 1-6 leads to $r_j < r_i$ which contradicts Equation 7. Hence, our assumption is false, i.e., $s_i \leq_{PR} s_j$ is a correct PR-Ordering. □

Theorem 3.5. *Algorithm 3 generates a strict total PR-Ordering across local snapshots in S, i.e.,* $\forall s_i, s_j \in S$, *if* $s_i \leq_{PR} s_j$, *then* $s_j \nleq_{PR} s_i$.

Proof. The $\leq_{PR}$ ordering is mapped from the ordering of logical timestamps assigned using GETNEXTLOGICALTIMESTAMP() function which atomically provides monotonically increasing timestamps. □

Theorem 3.5 indicates two things: first, the locally consistent checkpointing process generates local snapshots that are considered to be inconsistent with other local snapshots; even if the snapshots captured are truly globally consistent, the monotonic nature of timestamps assigned to snapshots does not capture this information. Secondly, the schedule for recovery from multiple failures is deterministic.

Missing Snapshots. Failures can occur even before the first set of snapshots from the affected machines are available. Recovery from such failures is done from the initial state of the affected machine's workload. To ensure PR-Semantics is adhered to during the recovery process, a total PR-Ordering must be available across the initial states for different machines' workload and the captured snapshots. Such PR-Ordering is available naturally by viewing the initial states to have read no values from other workloads. Let $I = \{i_0, i_1, ..., i_{k-1}\}$ represent the set of initial local states of machines in cluster C.

Corollary 3.2. $\forall i_i \in I$ *and* $\forall s_j \in S$, $i_i \leq_{PR} s_j$.

Moreover, the total PR-Ordering among the individual initial states can be captured as follows.

Corollary 3.3. $\forall i_i, i_j \in I$, $i_i \leq_{PR} i_j$ *and* $i_j \leq_{PR} i_i$. *This means,* $i_i =_{PR} i_j$.

Corollary 3.3 captures the PR-equivalence across initial states which means, processing an initial state using values from other initial states adheres to PR-Semantics. For simplicity, we consider the initial states as snapshots captured at the beginning of processing and assume the PR-Ordering based on the ordering of machine ids, i.e., if $\forall c_i, c_j \in C$, if $i < j$ then $i_i \leq_{PR} i_j$.

4. Evaluation

System Design. We incorporated CoRAL in an asynchronous iterative graph processing system based on ASPIRE [25] as shown in Figure 5. A fault tolerant layer is designed to handle distributed coordination across multiple machines and provide asynchronous communication. The distributed coordinator is based upon Apache Zookeeper [10]. It manages membership of machines, detects machine failures, and invokes callbacks to CoRAL module. It also provides atomic timestamp service required for capturing PR-Ordering, and synchronization primitives like barriers for programmability. The asynchronous communication layer is built using non-blocking primitives provided by ZeroMQ [30].

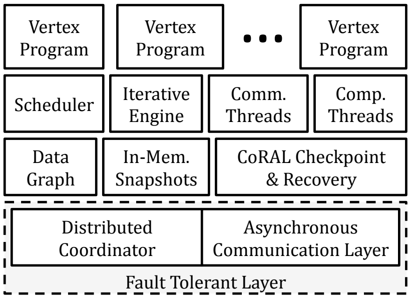


Figure 5: System design.

The application layer includes the graph processing engine which operates over the given graph. The graph is partitioned using GraphLab's partitioner that greedily minimizes the edge-cut. The CoRAL checkpointing and recovery module periodically captures locally consistent snapshots of the graph state and ensures total ordering by coordinating with Zookeeper. Upon failures, it recovers the lost values of the graph state using confined recovery.

Experimental Setup. Our evaluation uses four algorithms: PageRank (PR) [16], MultipleSourceShortestPaths (MSSP), ConnectedComponents (CC) [32], and KCoreDecomposition (KC) taken from different sources [12, 25]. The algorithms are oblivious to the underlying fault tolerance mechanisms used in our evaluation and hence, no modifications were done to their implementations. They were evaluated using three

			Checkpoint Frequency (sec)			
Graphs	#Edges	#Vertices	PR	MSSP	CC	KC
Twitter (TT) [11]	1.5B	41.7M	200	30	100	200 (k = 10)
UKDoman (UK) [2]	1.0B	39.5M	100	30	100	50 (k = 20)
LiveJournal (LJ) [1]	69M	4.8M	10	2	1	10 (k = 50)

Table 7: Real world input graphs and benchmarks used.

	LJ	UK	TT
PR	22.8	301.6	474.9
MSSP	4.1	54.9	50
CC	4.2	102.2	78.3
KC	30.8	162.7	364.5

Table 8: Execution times (sec).

real-world graphs listed in Table 7 by allowing them to run until convergence. The k parameter for KC is also listed.

To evaluate the effectiveness of locally consistent checkpointing and recovery mechanism, we set the checkpointing frequency in our experiments such that 3-6 snapshots are captured over the entire execution lifetime. The checkpoint frequencies used in our evaluation are shown in Table 7. While capturing locally consistent snapshots allows relaxing the time at which different local checkpoints can be saved, for KC we limit this relaxation to within the same k so the snapshot is fully captured within the same engine invocation.

All experiments were conducted on a 16-node cluster on Amazon EC2. Each node has 8 cores, 64GB main memory, and runs 64-bit Ubuntu 14.04 kernel 3.13.

Techniques Compared. We evaluate CoRAL using ASPIRE distributed asynchronous processing framework. ASPIRE guarantees PR-Semantics and it performs well compared to other frameworks as shown in [25]. For comparison with other systems, the raw execution times (in seconds) for ASPIRE are shown in Table 8.

Our experiments compare two fault tolerant versions that are described below:

– **CoRAL** captures locally consistent snapshots and performs confined recovery to PR-Consistent state.

– **BL** is the baseline technique used by asynchronous frameworks like GraphLab [12]. It captures globally consistent snapshots based on the the Chandy-Lamport snapshot algorithm [3] and recovers by rolling back all machines to the most recent checkpoint.

To ensure correct comparison between the two versions, failures are injected to bring down the same set of machines when same amount of progress has been achieved by the iterative algorithms. We check the execution state that is present immediately after failure to confirm that the vertex values are essentially the same (within tolerable bound for floating point values) for BL and CoRAL so that recovery starts from the same point. We also evaluate our recovery mechanism by starting the program assuming that failure has already occurred; we do this by feeding the same execution state (vertex values) as initializations and starting the recovery process; the performance results are same in this case too. CoRAL guarantees correctness of results; thus, final results of BL and CoRAL for each experiment are 100% accurate.

4.1 Recovery Overhead

Single Failure. We measured the execution times of CoRAL and BL when a single failure occurs during the program run. The execution times after the occurrence of failures (i.e., recovery and post recovery), normalized with respect to execution time for BL, are shown in Figure 6. The complete execution times (in seconds) including the execution prior to failure are given in Table 9.

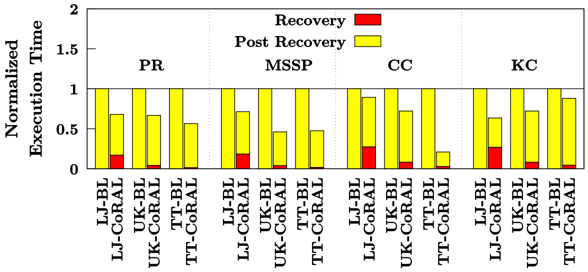


Figure 6: CoRAL vs. BL: Single failure execution times normalized w.r.t. BL.

		LJ	UK	TT
PR	BL	31.24	334.78	603.12
	CoRAL	24.85	322.71	398.92
MSSP	BL	8.73	69.72	57.00
	CoRAL	6.19	53.25	40.50
CC	BL	6.80	121.62	173.04
	CoRAL	6.80	102.20	84.56
KC	BL	64.68	195.24	612.36
	CoRAL	44.35	157.82	539.46

Table 9: CoRAL vs. BL execution times (sec) for single machine failure.

We observe that CoRAL quickly recovers and performs faster compared to BL in all cases – on an average across inputs, CoRAL is 1.6×, 1.7×, 1.3× and 2.3× faster than BL for PR, MSSP, CC, and KC respectively. We also found that the recovery process of CoRAL is lightweight – on an average across benchmarks, the recovery process takes 22.5%, 3.5%, and 3.3% of the total execution time for inputs LJ, UK, and TT. More importantly, the percentage time taken by the recovery process reduces as graph size increases.

Furthermore, in some cases we observed that for CoRAL, the overall execution time starting from the beginning of the iterative processing goes below the original execution time – for example, for both PR and MSSP on TT, the overall execution time reduces by 15.7% and 18.8%. This is because the CoRAL recovery constructs a PR-Consistent state with fresher values that is closer to the final solution, so the convergence is achieved faster.

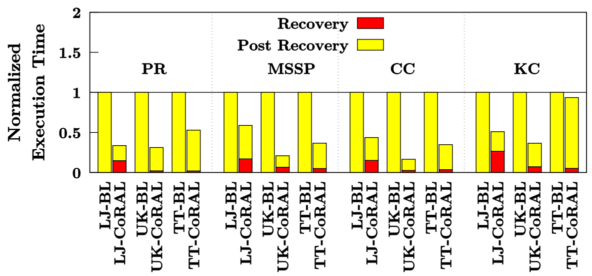


Figure 7: CoRAL vs. BL: Recovery for single failure from initial state. Execution times normalized w.r.t. BL.

The preceding experiment showed the combined benefit of lightweight checkpointing and confined recovery. Next we conducted an experiment to determine the benefit of confined recovery alone. We turned off checkpointing and upon (single) failure rolled back execution to initial default values – CoRAL only rolls back state of failed machine while BL must roll back states of all machines. Figure 7 shows that on an average across inputs, CoRAL executes for only 0.4×, 0.4×, 0.3× and 0.6× compared to BL for PR, MSSP, CC, and KC respectively. While BL recovery is fast as it simply reverts back to the initial state, the computation performed gets discarded and much time is spent on performing redundant computations. Moreover, we observed an increase in the overall execution time (starting from the beginning of processing) for BL by 1.1–1.9× which is due to the same amount of work being performed by fewer machines after failure. CoRAL, on the other hand, does not discard the entire global state, and hence finishes sooner.

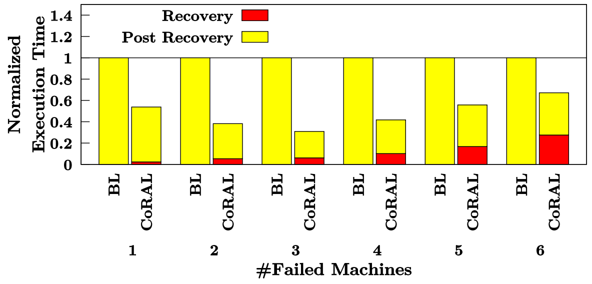


Figure 8: CoRAL vs. BL: Varying number (1 to 6) of machine failures. Execution times for PR on UK normalized w.r.t. BL.

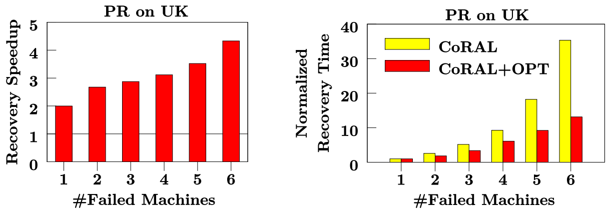
Multiple Failures. Figure 8 shows the performance for multiple failures for PR benchmark on UK graph. We simultaneously caused failure of 1 through 6 machines to see the impact of our strategy. As we can see, CoRAL performs 1.5–3.2× faster than BL. We observed that the overall execution times for BL after occurrence of failures increase up to 1.2×, whereas CoRAL takes only 0.2–0.8× of time to recover and finish processing. This is because CoRAL does not discard the progress of machines that are not impacted by failure.

Note that the execution times increase as the number of simultaneously failing machines increase. This is due to two reasons. First, the remaining (non-failed) state becomes smaller and the lost states become larger, causing more work during recovery and post-failure. Second, after failure, the processing continues on fewer leftover machines, i.e. computation resources decrease. It is also interesting to note that CoRAL's recovery time also increases with increase in the number of simultaneously failed machines due to the linear nature of our recovery strategy.

4.2 Partitioning Snapshots: Impact on Recovery

During checkpointing, a local snapshot can be saved by partitioning them and placing individual chunks on different machines, or by placing the entire snapshot on a single machine. Based upon the manner in which snapshots are saved, only the machines on which snapshots are locally available can quickly perform recovery. While partitioning the snapshot allows more machines to participate in the recovery process, placing the snapshots without partitioning reduces the communication during the recovery process – for example, for a single machine failure, communication is not required during the recovery process where PR-Consistent state is constructed by iteratively processing until convergence.

Figure 9a evaluates this design choice by showing the speedups achieved during recovery using the partitioning strategy over maintaining the snapshot as a whole. The speedups show that allowing multiple machines to take part during recovery process overshadows the communication increase and accelerates recovery. Moreover, the speedups are higher when greater number of machines fail. While not partitioning leads to an increase in the recovery workload by only a constant factor (i.e., size of a single snapshot), when more machines fail, the communication required to process the workload increases which limits the speedups.



(a) Speedup in CoRAL recovery due to partitioning snapshots.

(b) Recovery time with and without optimization normalized w.r.t. single failure case.

Figure 9

4.3 Optimizing Recovery from Multiple Failures

Various works suggest users often disable checkpointing (GraphX [7], Giraph [4], and Distributed GraphLab [12]) to eliminate its overheads. The PR-Consistent state can be constructed even when no snapshots are captured using initial state, i.e., default values. Moreover, Corollary 3.3 suggests that the execution state using default values is already PR-Consistent. Hence, the recovery process can be further optimized to incorporate the states of all the failed machines together, instead of adding them one by one. When the entire failed state is fully processed, it becomes PR-Consistent with the available current execution states of the machines not impacted by failure. Hence, computation can resume using this PR-Consistent state.

Figure 9b shows the time taken by the CoRAL recovery using initial state, with and without the above optimization. The optimization further speeds up the recovery process by an order of magnitude. This observation can be incorporated in the checkpointing strategy itself – if checkpointing guarantees subsets of local snapshots to be PR-Consistent, the snapshots in those subsets can be incorporated together during the recovery process, instead of adding them one by one.

4.4 Checkpointing: Impact on Network Bandwidth

We now evaluate the benefits of using locally consistent checkpointing. During checkpointing, the captured snapshots are saved remotely so that they are available upon failure. This leads to an increase in network usage. In Figure 10, we measure the 99th percentile [1] network bandwidth for BL and CoRAL by varying the replication factor (RF) from 1 to 6, normalized w.r.t. no replication [2].

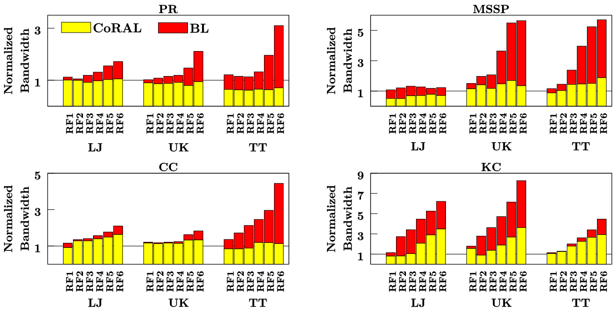


Figure 10: BL vs. CoRAL: 99th percentile network bandwidth for varying RF (1 to 6) normalized w.r.t. no checkpointing case.

As we can see, the peak bandwidth consumption increases rapidly with increase in RF for BL because the consistent checkpointing process saves all the snapshots at the same time, which leads to simultaneous bulk network transfers. The peak bandwidth consumption for CoRAL does not increase as rapidly – this is because CoRAL staggers the capturing of different snapshots over time, and hence, the snapshots are transferred to remote machines at different points in time at which they become available. On an average across all benchmark-input configurations, there is a 22% to 51% reduction in 99th percentile bandwidth using CoRAL as RF is varied from 1 to 6.

There is a noticeable increasing trend for CoRAL on KC – this is mainly because the checkpointing process in KC can be relaxed only during computation for a given core, which in certain cases became a rather narrow window over which all the snapshots had to be transferred.

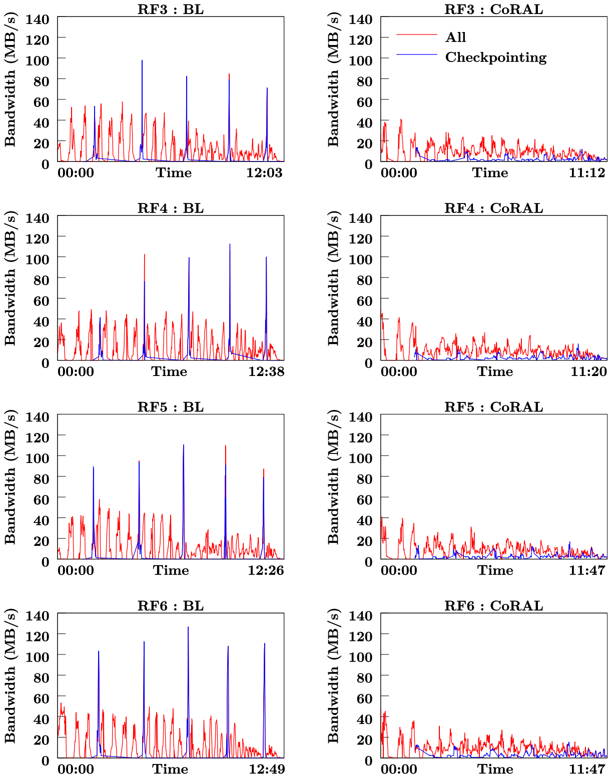


Figure 11: BL vs. CoRAL: Network usage for PR on UK.

Figure 11 shows the bandwidth consumption for PR on UK with RF varying from 3 to 6. Here All (red lines) indicate total bandwidth usage while Checkpointing (blue lines) indicate bandwidth consumed due to checkpointing alone. As we can see, for BL, the bandwidth periodically increases; moreover, the intermittent spikes are due to the checkpointing process. This is mainly because the local snapshots from all machines are sent and received at the same time. For CoRAL, the checkpointing process on different machines can take place at different times and hence, the transfer of local snapshots is spread over time, reducing the effect of bulk transfer across multiple machines. This reduces network contention.

5. Conclusion

The semantics of asynchronous distributed graph processing enables supporting fault tolerance at reduced costs. Following failures, PR-Consistent state is constructed without discarding any useful work performed on non-failing machines, i.e. confined recovery is achieved. CoRAL uses locally consistent snapshots that are captured at reduced peak network bandwidth usage for transferring snapshots to remote machines. Our experiments confirm reductions in checkpointing and recovery overhead, and low peak network bandwidth usage.

[1] The performance trend is similar for higher percentile values.

[2] The network statistics were measured using tcpdump.

Acknowledgments

We would like to thank our shepherd Lidong Zhou as well as the anonymous reviewers for their valuable and thorough feedback. This work is supported in part by NSF grants CCF-1524852 and CCF-1318103 to UC Riverside.

References

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 44–54, 2006.

[2] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW*, pages 595–601, 2004.

[3] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM TOCS*, 3(1):63–75, Feb. 1985.

[4] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: graph processing at facebook-scale. In *Proc. VLDB Endowment*, 2015.

[5] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002.

[6] A. Farahat, T. LoFaro, J. C. Miller, G. Rae, and L. A. Ward. Authority rankings from hits, pagerank, and salsa: Existence, uniqueness, and effect of initialization. *SIAM Jornal of Scientific Computing*, 27(4):1181–1201, Nov. 2005.

[7] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *USENIX OSDI*, pages 599–613, 2014.

[8] M. Han and K. Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proc. VLDB Endowment*, 8(9):950–961, May 2015.

[9] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. Kla: A new algorithmic paradigm for parallel graph computations. In *PACT*, pages 27–38, New York, NY, 2014.

[10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, pages 11–11, Berkeley, CA, 2010.

[11] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[12] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endowment*, 5(8):716–727, Apr. 2012.

[13] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, and G. Inc. Pregel: A system for large-scale graph processing. In *ACM SIGMOD*, pages 135–146, 2010.

[14] D. Manivannan and M. Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE TPDS*, 10(7):703–713, 1999.

[15] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *ACM SOSP*, pages 29–41, New York, NY, USA, 2011. ACM.

[16] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[17] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *USENIX OSDI*, pages 293–306, Berkeley, CA, USA, 2010.

[18] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *ACM SoCC*, pages 195–208, 2015.

[19] S. Salihoglu and J. Widom. GPS: A graph processing system. In *Scientific and Statistical Database Management Conference*, pages 22:1–22:12, 2013.

[20] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *ACM SIGMOD*, pages 505–516, 2013.

[21] Y. Shen, G. Chen, H. V. Jagadish, W. Lu, B. C. Ooi, and B. M. Tudor. Fast failure recovery in distributed graph processing systems. *Proc. VLDB Endowment*, 8(4):437–448, Dec. 2014.

[22] L. G. Valiant. A bridging model for parallel computation. *CACM*, 33(8):103–111, Aug. 1990.

[23] H. Cui, J. Cipar, Q. Ho, J.K. Kim, S, Lee, A. Kumar, J. Wei, W. Dai, G.R. Ganger, P.B. Gibbons, G.A. Gibson, and E.P. Xing. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *USENIX ATC*, pages 37–48, 2014.

[24] K. Vora, G. Xu, and R. Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX ATC*, pages 507–522, 2016.

[25] K. Vora, S. C. Koduru, and R. Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms using a Relaxed Consistency based DSM. In *OOPSLA*, pages 861–878, 2014.

[26] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *Conference on Innovative Data Systems Research* (CIDR), 2013.

[27] P. Wang, K. Zhang, R. Chen, and H. Chen. Replication-based fault-tolerance for large-scale graph processing. In *IEEE/IFIP DSN*, pages 562–573, 2014.

[28] J. W. Young. A first order approximation to the optimum checkpoint interval. *CACM*, 17(9):530–531, Sept. 1974.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*, pages 2–2, 2012.

[30] ZeroMQ. http://zeromq.org/.

[31] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *ScienceCloud*, 2012.

[32] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical Report CALD-02-107, Carnegie Mellon University, 2002.