ACF2: Accelerating Checkpoint-Free Failure Recovery for Distributed Graph Processing

Chen Xu[1,2](✉), Yi Yang[1,2], Qingfeng Pan[1,2], and Hongfu Zhou[3]

[1] East China Normal University, Shanghai, China
cxu@dase.ecnu.edu.cn, {yiyang,qfpan}@stu.ecnu.edu.cn
[2] Shanghai Engineering Research Center of Big Data Management, Shanghai, China
[3] Shanghai Ruanzhong Information Technology Company Limited, Shanghai, China
hfzhou@softline.sh.cn

Abstract. Iterative computation in distributed graph processing systems typically incurs a long runtime. Hence, it is crucial for graph processing to tolerate and quick recover from intermittent failures. Existing solutions can be categorized into checkpoint-based and checkpoint-free solution. The former writes checkpoints periodically during execution, which leads to significant overhead. Differently, the latter requires no checkpoint. Once failure happens, it reloads input data and resets the value of lost vertices directly. However, reloading input data involves repartitioning, which incurs additional overhead. Moreover, we observe that checkpoint-free solution cannot effectively handle failures for graph algorithms with topological mutations. To address these issues, we propose ACF2 with a *partition-aware backup strategy* and an *incremental protocol*. In particular, the partition-aware backup strategy backs up the sub-graphs of all nodes after initial partitioning. Once failure happens, the partition-aware backup strategy recovers the lost sub-graphs from the backups, and then resumes computation like checkpoint-free solution. To effectively handle failures involving topological mutations, the incremental protocol logs topological mutations during normal execution which would be exploited for recovery. We implement ACF2 based on Apache Giraph and our experiments show that ACF2 significantly outperforms existing solutions.

Keywords: Failure recovery · Graph processing · Checkpoint-free

1 Introduction

Graph processing is widely employed in various application (e.g., social network analysis and spatial data processing). In big data era, to efficiently process big graph data, a set of large-scale distributed graph processing systems such as Pregel/Giraph [9], GraphLab [7] and PowerGraph [3] has emerged. Distributed graph processing usually involves iterative computation, where each iteration is regarded as a *superstep* [9]. Typically, the computation with a serial of supersteps leads to a long execution time. During this prolonged time span, certain nodes

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
B. Li et al. (Eds.): APWeb-WAIM 2022, LNCS 13421, pp. 45–59, 2023.
https://doi.org/10.1007/978-3-031-25158-0_5

of a distributed graph processing system may encounter failures due to network disconnection, hard-disk crashes, etc. Hence, it is vital that distributed graph processing systems tolerate and recover from failures automatically.

A common solution in systems like GraphLab [7] and Pregel/Giraph [9] to handle failures is to periodically write checkpoints, i.e., checkpoint-based solution. However, it consumes runtime costs to write checkpoints even though no failure happens. In contrast, some studies propose the checkpoint-free solutions, e.g., Phoenix [2] and Zorro [11], to tolerate failure. Once failure happens, it reloads input data to recover the lost sub-graphs on the failed nodes, and recovers the values of vertices on these sub-graphs by applying a user-defined compensation function. Hence, checkpoint-free solution outperforms checkpoint-based solution by saving the overhead cost of checkpointing.

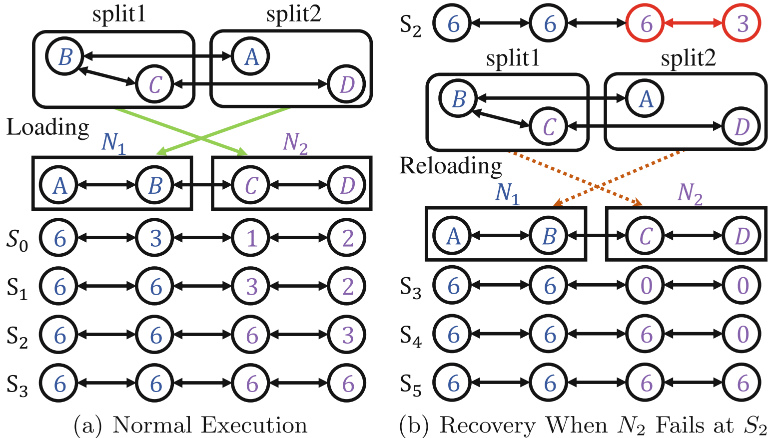
Nonetheless, checkpoint-free solution still requires further improvement. First, it has to rebuild the lost sub-graphs from input data once failure happens, which involves repartitioning. Particularly, the repartitioning incurs additional recovery overhead, since it shuffles graph data during recovery. Moreover, checkpoint-free solution fails to handle the failures involving topological mutations (e.g., deletions of edges). Once failures occur, checkpoint-free solution re-initializes the topology on the failed nodes from input data, but keeps the topology on the normal nodes unchanged. This may lead to result inaccuracy, since checkpoint-free solution discards topological mutations of vertices on failed nodes.

In this work, we propose a prototype system, namely ACF2, with a *partition-aware backup strategy* to reduce the recovery overhead, and an *incremental protocol* to handle failures involving topological mutations. The partition-aware backup strategy backs up the sub-graphs of all nodes into reliable storage after partitioning. Once failure happens, the partition-aware backup strategy recovers the lost sub-graphs from the backup and then restarts computation like checkpoint-free solution. Our experimental results show that, in case of failures, the partition-aware backup strategy is able to reduce the overall execution time by 29.4% in comparison to Phoenix [2], a state-of-the-art checkpoint-free solution. The incremental protocol logs the topological mutations during execution. Upon failure, our protocol utilizes these logs to recover the topology on all nodes to a certain superstep before failure. In our experiments, we find the incremental protocol improves the result accuracy by up to 50% compared to Phoenix.

In the rest of this paper, we introduce the background of the checkpoint-free solution and our motivation in Sect. 2, and make the following contributions.

- We propose a *partition-aware backup strategy* in Sect. 3 that decreases the overhead imposed by checkpoint-free solution on the lost sub-graphs.
- We devise an *incremental protocol* in Sect. 4, so as to support fault tolerance with graph topological mutations.
- We describe the implementation of ACF2 based on Giraph in Sect. 5 and demonstrate the ACF2 outperforms state-of-the-art solutions via experimental evaluation in Sect. 6.

We discuss the related work in Sect. 7 and conclude this paper in Sect. 8.



(a) Normal Execution (b) Recovery When N_2 Fails at S_2

Fig. 1. The working process of checkpoint-free solution

2 Background and Motivation

Many graph processing systems such as GraphX [4] employ a checkpoint-based solution to tolerate failures. During normal execution, the checkpoint-based solution writes checkpoints periodically. Once failure happens, the systems read the latest checkpoint for rollback and start recomputation. Clearly, this checkpoint-based solution incurs a high overhead even though no failure happens.

To avoid the checkpointing overhead, some studies propose checkpoint-free solution, such as Phoenix [2], optimistic recovery [12] and Zorro [11]. Differently, the solution does not write any checkpoint during normal execution. Once failure happens, it loads input data to recover the lost sub-graphs on failed nodes via partitioning. After that, the checkpoint-free solution employs a user-defined compensation function to recover the value of vertices on lost sub-graphs and continues computation. In particular, it does not rollback the value of vertices and starts recomputation, but utilizes semantic properties of graph algorithms to set the value of vertices, so as to decrease the overhead of recomputation [2].

Figure 1 takes maximum value calculation as an example to illustrate the checkpoint-free solution. Here, the job executed on two nodes N_1 and N_2 consists of four supersteps from superstep S_0 to S_3. As shown in Fig. 1(a), before executing S_0, N_1 and N_2 load sub-graphs from input data splits split1 and split2, respectively. However, according to partition function, the vertex C on the sub-graph in split1 does not belong to N_1. Likewise, A does not belong to N_2. Hence, N_1 and N_2 exchange vertices with each other during loading, called *shuffle*. After that, the job executes computation from S_0 to S_3. During computation, the checkpoint-free solution achieves zero overhead, since it does not write checkpoints. Once N_2 fails at S_2, as shown in Fig. 1(b), the sub-graph consisting of C and D on N_2 is lost. To recover the lost sub-graph, the checkpoint-free solution requires N_1 and N_2 to reload the sub-graphs from split1 and split2.

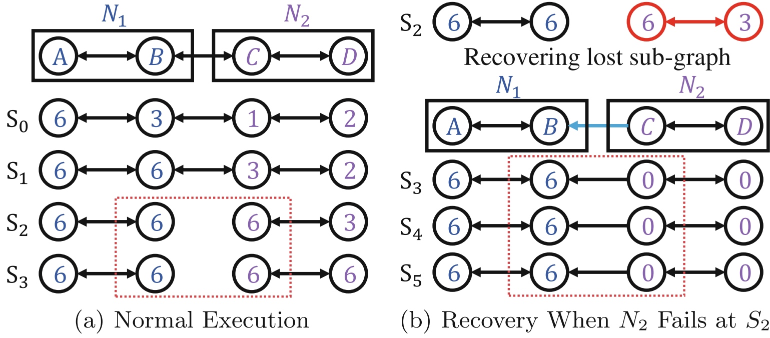


Fig. 2. Impact of topological mutations

Then, the checkpoint-free solution compensates the value of vertices C and D to minimum value, i.e., 0. This avoids the recomputation overhead on updating the value of D to 3. After that, the job continues the computation and terminates the iteration at S_5.

Recovery Overhead. Once failure occurs, the checkpoint-free solution has to reload the splits of input data and executes partitioning. However, the partitioning shuffles data via the network during recovery, which introduces additional overhead. We take Fig. 1(b) as an example to illustrate this. Once N_2 fails at S_2, the checkpoint-free solution asks N_1 and N_2 to reload input data. Then, N_1 recovers the vertex C by shuffle, i.e., N_1 sends C via the network to N_2 after reading C from split1. Clearly, the checkpoint-free solution avoids the additional overhead incurred by shuffle if N_2 can read the vertex C directly.

Inaccuracy. Besides recovery overhead, the checkpoint-free solution leads to a decrease in the result accuracy to recover graph algorithms with topological mutations. To elaborate this, Fig. 2 shows the normal execution and failure recovery of maximum value graph job with topological mutations. As shown in Fig. 2(a), the vertex B sends the maximum value 6 as a messages to C by edge $\langle B, C\rangle$ during normal execution. Then, the vertex C updates its value to 6 at S_2. Meanwhile, the job removes the edges $\langle B, C\rangle$ and $\langle C, B\rangle$. Subsequently, the job updates the value of vertex D to 6 and finishes the computation at S_3. Once N_2 fails at S_2, as shown in Fig. 2(b), the checkpoint-free solution employs the same way as Fig. 1(b), i.e., reloading input data, to recover the topology of N_2 to initial topology at S_0. Moreover, for normal node N_1, the checkpoint-free solution keeps its topology unchanged. Hence, at superstep S_3, there is an edge $\langle C, B\rangle$ but no edge $\langle B, C\rangle$ in the graph. Without the edge $\langle B, C\rangle$, vertex B is unable to send the maximum value 6 as a messages to C again. Hence, C and D cannot update their values to the maximum value 6, so that the job eventually outputs inaccurate results.

Further, combining the above example, we summarize the reasons for the inaccuracy introduced by the checkpoint-free solution. There two reasons for the inaccuracy: (a) computation on the failed nodes requires messages from

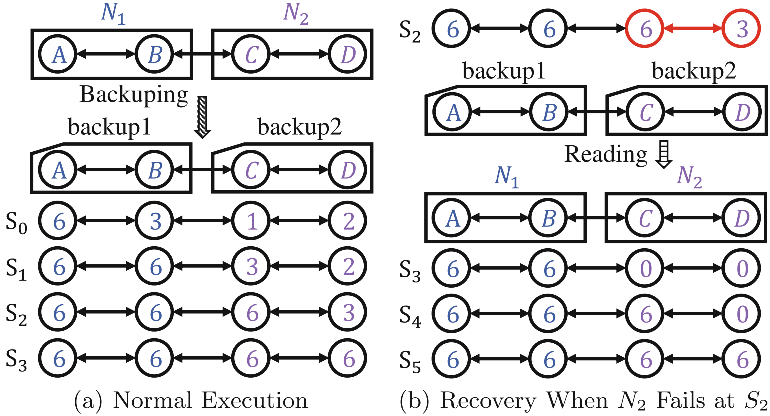


Fig. 3. Partition-aware backup strategy

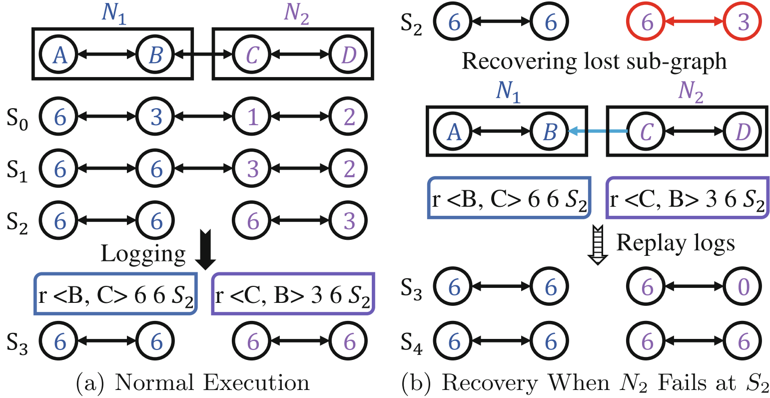
the normal nodes, while topological mutations break the link used to transfer message from the vertices on normal nodes to the vertices on failed nodes; (b) the checkpoint-free solution does not fix these links, since it keeps the topology of the normal nodes unchanged during recovery.

3 Partition-aware Backup Strategy

To avoid the additional overhead caused by the checkpoint-free solution during recovery, we propose partition-aware backup strategy which saves the sub-graphs of all nodes to distributed file system (DFS) during normal execution. Once failures occur, our strategy recovers the lost sub-graphs directly from the backups instead of from the input data. Then, we are able to apply a user-defined function following the checkpoint-free solution.

During normal execution, the design of the partition-aware backup strategy involves two questions: *how many* times to backup; and *when* to backup. For the first question, the times of backuping depend on whether the system repartitions the sub-graphs on the nodes during computation. However, repartitioning sub-graphs does not offer significant performance improvements except under particular conditions [6,10]. Based on this observation, we assume that the system does not repartition the sub-graphs, and therefore back up the sub-graphs of all nodes only once. Moreover, we employ an unblocking manner to minimize the influence of backup on computation. This manner backs up the sub-graphs in parallel with the computation. For the second question, our strategy immediately backs up sub-graphs after partitioning, i.e., at superstep S_0, so as to make the backup available as soon as possible. As an example in Fig. 3(a), at superstep S_0, our strategy asks the sub-graph writers on N_1 and N_2 to back up their sub-graphs. Here, the backup is in parallel with the iterative computation, which involves a low overhead cost.

Once failure happens, our strategy checks the availability of backup. Then, our strategy allows the sub-graph readers on the failed nodes to recover the



(a) Normal Execution (b) Recovery When N_2 Fails at S_2

Fig. 4. Incremental protocol

lost sub-graphs from the backup when the backup is available. Otherwise, it lets these readers recover the lost sub-graphs from the input data. Then, our strategy employs the same way as checkpoint-free solution to recover the values of the vertices. As illustrated by the example in Fig. 3(b), the backup is available when N_2 fails at superstep S_2. Hence, the sub-graph reader on N_2 recovers the lost sub-graph from the backup rather than the input data, and resets the values of vertices following checkpoint-free solution.

4 Incremental Protocol

In this section, we propose an incremental protocol to deal with failure involving topological mutations, so as to ensure the result accuracy. To improve the result accuracy, a naïve solution is to reset the topology of normal nodes to their initial topology at S_0, which ensures that normal nodes can resend messages to failed nodes. However, this solution forces the job to recompute from the initial graph topology, wasting the computation.

To reduce the overhead on recomputation, we propose an incremental protocol, which logs topological mutations of all nodes during execution. Once failure occurs, our protocol utilizes these logs to redo mutations on failed nodes, so as to avoid the overhead of recomputation involving message transmission and processing. In detail, our protocol consists of the following two phases, i.e., normal execution and failure recovery.

During normal execution, once the node N_i modifies the topology of graph at superstep S_j, our protocol lets the logger on N_i log these modifications to DFS. In particular, the logger also stores the type of modifications, e.g., adding or removing edges, as well as the superstep where the mutations occur. Moreover, when the modifications of all nodes at superstep S_j is fully stored, our protocol creates a log flag file of S_j on DFS. This log flag file indicates that it is viable to redo the mutations from S_0 to S_j on failed nodes once failure happens.

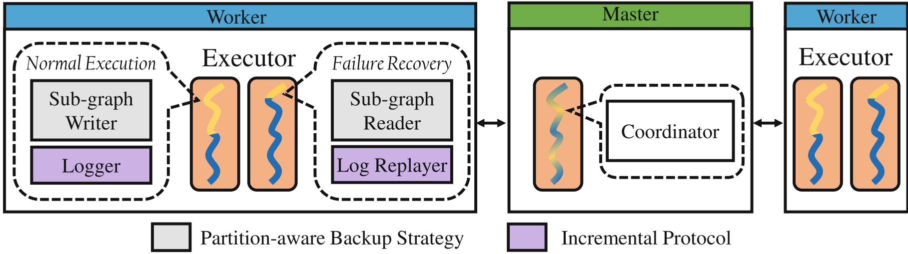


Fig. 5. System architecture

Figure 4(a) provides an example on how the incremental protocol works during normal execution. At superstep S_2, the vertex C on N_2 updates the value to 6 and removes the edge $\langle C, B\rangle$ after receiving the message from B on N_1. Meanwhile, the logger on N_2 logs the information related to the modification "r $\langle C, B\rangle$ 3 6 S_2" to DFS. These information contains the type of modification r, the removed edges $\langle C, B\rangle$, the change in the value of vertex from 3 to 6 as well as the superstep S_2. Likewise, the logger on N_1 logs the information related to the modification. Once these information at superstep S_2 is fully stored, our protocol creates a log flag file of S_2 on DFS.

Once failure occurs at superstep S_f, our protocol loads all log flag files and generates a list L consisting of the supersteps in these flag files. Then, our protocol requires all nodes to load their respective log from DFS. For failed nodes, our protocol obtains the superstep S_t of each modification in log and redoes the modification when S_t belongs to L. In contrast, our protocol does not redo the modification when S_t does not belong to L, since our protocol may not have wrote fully the mutations on failed nodes to DFS during execution. To redo these mutations, the failed nodes still requires messages from the normal nodes. Hence, our protocol undoes the modification when S_t does not belong to L for normal nodes, so as to fix the link used to transfer messages.

Figure 1(b) depicts how the incremental protocol works during recovery. As shown in Fig. 4, once N_2 fails at superstep S_2, our protocol generates a list L including S_2 and lets the log replayers on N_1 and N_2 read logs from the DFS. Then, the log replayer on N_2 redoes the modification "r $\langle C, B\rangle$ 3 6 S_2", since L contains S_2. Meanwhile, the log replayer on N_1 does not redo the modification involving B, since it is normal node and L contains S_2. It is worth noting that the log replayer undoes the modification involving B if L does not contain S_2, so as to fix the link from B to C. Then, the vertex D receives the message from C and performs the subsequent computation. Eventually, the computation will produce accurate results.

Algorithm 1: Normal Execution of the Worker n

```
1  while iteration is not terminated do
2      i ← current superstep
3      G_n ← get the sub-graph maintained by the worker n
4      if i = 0 then // Sub-graph Writer
5          backuping G_n
6      initialize set L // Logger
7      for v ∈ G_n do
8          if v modifies topology then
9              R_n^i ← topological mutations
10             add R_n^i into L
11     write L to DFS
12     i ← i + 1
```

5 System Implementation

In this section, we present the implementation of ACF2 which is based on Apache Giraph. Figure 5 shows the architecture of our proposed ACF2, which inherits the master/worker architecture of the existing systems. Specifically, the master is responsible for coordinating worker activity, whereas the worker executes computation via a series of executor threads. Moreover, the worker integrates the failure recovery mechanism implemented through an executor thread. To achieve the partition-aware backup strategy, we introduce a new component *Sub-graph Writer* for the worker and integrate it into the thread responsible for the computation. Also, we modify the execution logic of the thread responsible for failure recovery on the worker, so as to integrate the new component *Sub-graph Reader*. Likewise, to achieve the incremental protocol, we add two new components to worker, i.e., *Logger* and *Log Replayer*. Moreover, we modify the execution logic of the *Coordinator* component on the master to record the failed supersteps, so as to coordinate these new components in the worker.

Algorithm 1 illustrates the implementation details of one worker (say worker n) during normal execution. Our partition-aware backup strategy employs sub-graph writer obtains and backs up the sub-graph of each node after partitioning, i.e., superstep S_0 (line 3–5). The logger in incremental protocol monitors the behavior of each vertex v at each superstep. When v modifies the graph topology, the logger logs the topological mutations R_n^i in superstep S_i of worker n, and adds it to a set L that stores all topological mutations (line 6–10). Then, it uploads logs to DFS (line 11).

Algorithm 2 describes the implementation details of worker n upon failure. The partition-aware strategy adopts sub-graph reader to load lost sub-graphs from the backup if it exists (line 2). Otherwise, it reloads the input data (line 4). The log repalyer in incremental protocol get a list F of failed supersteps from master and read logs from DFS into L (line 5–6). For each record R_n^i in logs,

Algorithm 2: Failure Recovery of the Worker n

```
1  if backup exists then // sub-graph reader
2      load lost sub-graphs from the backup
3  else
4      recover lost sub-graphs from input data
5  list F ← get failed supersteps from Master
6  L ← read logs for worker n
7  for each R_n^i in L do // log replayer
8      if i ∈ F then
9          undo R_n^i
10     else
11         redo R_n^i
```

the protocol obtains the superstep S_i saved in R_n^i and undoes it if S_i belongs to F (line 9). Otherwise, our protocol redoes R_n^i (line 11).

6 Experiments

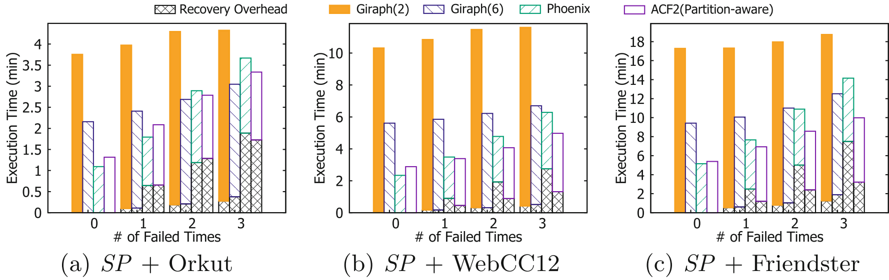
This section introduces our experimental setting and demonstrates the efficiency of the partition-aware backup strategy and the incremental protocol.

6.1 Experimental Setting

Cluster Setup. We conduct all experiments on a cluster with 13 compute nodes. Here, each compute node has a eight-core Intel Xeon CPUs, a 32GB RAM, a 300GB SSD and 1Gbps Ethernet. We deploy Hadoop 2.5.1 on this cluster, since the Giraph program is executed via a MapReduce job on the top of Hadoop. By default, we issue 13 map tasks and each task owns 24GB memory. Here, we follow previous studies [1,8,13] to set the size of task memory.

Algorithms. We choose two algorithms to evaluate the effectiveness of partition-aware backup strategy and incremental protocol. They are the single source *shortest path* without topological mutations and the *maximum weight matching* with topological mutations. These algorithms are popular in graph analysis applications [5]. In the rest of this paper, we refer to above algorithms as *SP* and *MWM*, respectively. We set the number of iterations of *MWM* to 50. Moreover, we conduct the experiments over three real-life graphs, Orkut[1], WebCC12[2] and Friendster[3], with million-scale or billion-scale edges.

[1] http://networkrepository.com/orkut.php.
[2] http://networkrepository.com/web-cc12-PayLevelDomain.php.
[3] https://snap.stanford.edu/data/com-Friendster.html.



(a) SP + Orkut (b) SP + WebCC12 (c) SP + Friendster

Fig. 6. Impact of the number of failures

Baselines. We take the original checkpoint-based recovery solution in Giraph as a baseline. The checkpoint interval for this solution imposes an impact on performance. For fairness, based on previous studies [12,15,17], we set the checkpointing interval to two and six, respectively. Also, we take the state-of-the-art checkpoint-free solution, i.e., Phoenix, as a baseline and implement it in Giraph. Compared to these baselines, we evaluate our proposed ACF2 consisting of a partition-aware backup strategy and an incremental protocol.

6.2 Efficiency of Partition-aware Backup Strategy

In this section, we evaluate performance of the partition-aware backup strategy which means that ACF2 does not enable the incremental protocol, denoted as ACF2(Partition-aware) in Fig. 6, 7, 8, 9 and 10. In particular, we consider the impact of two factors on performance: (i) the number of failures and (ii) the time at which they occur, i.e., failed superstep.

Number of Failures. We focus on the case where the number of failures is less than three, since checkpoint-free solution assumes that modern clusters has large mean time between failures of a machine [11]. Figure 6 depicts the execution time of failure recovery with different number of failures n ranging from zero to three. Particularly, the failure does not happen when $n = 0$. When $n = 1$, we issue one failure at superstep S_6. When $n = 2$, we issue two failures at S_6 and S_8. When $n = 3$, we issue three failures at S_6, S_8 and S_{10}.

As shown in Fig. 6, Giraph with a checkpointing interval of six, i.e., Giraph(6) in Fig. 6, always outperforms Giraph with a checkpointing interval of two denoted as Giraph(2). The reason is Giraph with a checkpointing interval of six writes fewer checkpoints. Based on this observation, in the following discussion, we only discuss the differences between Giraph with a checkpoint interval of 6 and other recovery solutions.

In general, Phoenix performs better than the Giraph. However, Phoenix performs worse than the Giraph in some cases because of its additional overhead involving repartitioning. As an example in Fig. 6(a), in comparison to Giraph, Phoenix decreases the overhead by up to 58.6% when $n = 0$, while they exceed the execution time of the Giraph when $n = 2$.

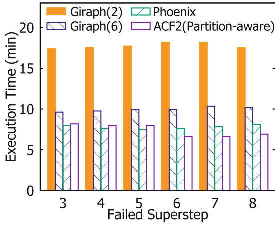
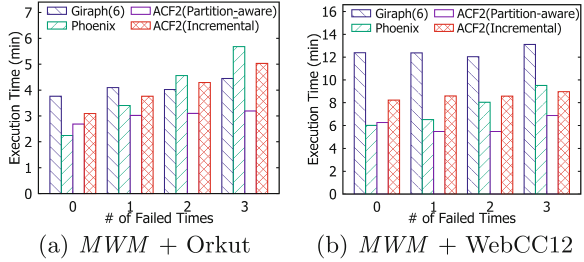


Fig. 7. Different failed supersteps



(a) MWM + Orkut (b) MWM + WebCC12

Fig. 8. Impact of the number of failures

The partition-aware backup strategy keeps low overhead under failure-free cases, so as to achieve similar performance to Phoenix when $n = 0$. As shown in Fig. 6(c), the overhead cost of the partition-aware backup strategy on the Orkut dataset only increases 4.6% as against Phoenix. In case of failure, the partition-aware backup strategy outperforms other recovery solutions, and achieves the best performance. For example, in Fig. 6(c), the partition-aware backup strategy decreases the overhead by 31% compared to the Giraph, and by up to 9% compared to Phoenix when $n = 1$. Notably, for individual cases depicted in Fig. 6(a), the partition-aware backup strategy performs worse than Giraph. The reason is that the backup is not completed in time when the *SP* algorithm is executed on the small dataset Orkut. This leads to the partition-aware backup strategy to recover lost partition data using a similar way to Phoenix.

As the number of failures n increases, the partition-aware backup strategy benefits more than Phoenix on the execution time. As shown in Fig. 6(c), the percentage of overhead cost reduced by the partition-aware backup strategy increases from 9% to 29.4% as against Phoenix when n increases from 1 to 3.

Overhead of Recovering Lost Sub-graphs. The change of n has an impact on the overhead of recovering lost sub-graphs, which decides the performance of different recovery solutions. Next, we further evaluate the overhead of recovering lost sub-graphs caused by these solutions.

In Fig. 6, the partition-aware backup strategy achieves a lower recovery overhead than Phoenix, since it recovers lost sub-graphs from backups which avoids the repartitioning during recovery. As depicted in Fig. 6(b), the Giraph decreases the overhead of recovering lost sub-graphs by up to 81.2%, compared to Phoenix when $n = 1$. In particular, Giraph obtains a similar recovery overhead as the partition-aware backup strategy, since it recovers lost sub-graphs from the checkpoint which also avoids the repartitioning.

Along with n increases, advantages of the partition-aware backup strategy during recovery phase becomes significant. As shown in Fig. 6(c), our strategy decreases the overhead by 67.2% as against Phoenix when $n = 1$. Moreover, as n increases, the overhead of recovering lost sub-graphs for Phoenix grows faster than that of the partition-aware backup strategy, since the cost on reloading the input data is higher than the cost on loading backup.

Different Failed Supersteps. Next, we study the performance of the partition-aware backup strategy when failure occurs in different supersteps, since failures always happen at certain supersteps in the aforementioned experiments. Figure 7 provides the execution time of *SP* on the Friendster dataset with one failure which happens in different supersteps. Here, we consider the case of one failure (i.e., $n = 1$), as we have already observed that the partition-aware backup strategy achieves more benefits as n increases. In this experiment, the failed superstep is varied from S_3 to S_8.

Clearly, as shown in Fig. 7, the partition-aware backup strategy outperforms other recovery solutions as long as the backup of sub-graphs is available, i.e., the failure occurs after superstep S_5. Otherwise, the partition-aware backup strategy behaves similarly to Phoenix. In other words, the partition-aware backup strategy degenerates to Phoenix when the backup is not available.

In summary, the partition-aware backup strategy outperforms Giraph and Phoenix in most cases. Moreover, with the increasing of the number of failures, the partition-aware backup strategy obtains more benefits as against Phoenix.

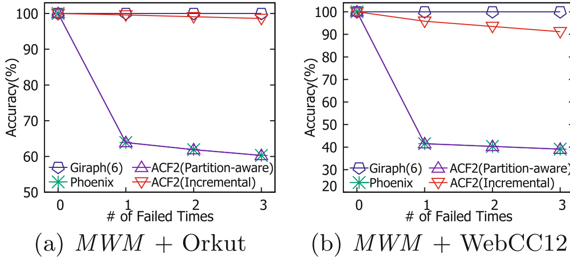
6.3 Efficiency of Incremental Protocol

In this section, based on the partition-aware backup strategy, we compare the performance of the incremental protocol against other recovery solutions. This means that ACF2 enables both the partition-aware backup strategy and the incremental protocol, denoted as ACF2(Incremental). In particular, we exclude Giraph with a checkpointing interval of two in this group experiments, since previous experiments have demonstrated that Giraph with a checkpointing interval of six outperforms it.

Number of Failures. Figure 8 reports the total execution time of different recovery solutions with the number of failures n ranging from zero to three. Here, we follow the previous failures setting and issue failures at the same supersteps. Moreover, due to space limitation, we only take *MWM* on Orkut and WebCC12 datasets as examples. However, similar results hold on Friendster.

As shown in Fig. 8, the incremental protocol outperforms Giraph in most cases. As an example depicted in Fig. 8(b), compared to Giraph, the incremental protocol reduces the overhead by 28.7% on the execution time when $n = 2$. Due to the logging overhead, our incremental protocol performs worse than the partition-aware backup strategy and Phoenix. For example, in Fig. 8(b), the execution time of incremental protocol is always longer than that of the partition-aware backup strategy, no matter how many times the failure occurs. Moreover, compared to Phoenix, the overhead of incremental protocol increases 32% when $n = 1$. However, incremental protocol saves the execution time up to 5% when $n = 3$.

Result Accuracy. Next, we focus on the result accuracy, since the incremental protocol affects it. To evaluate the accuracy of *MWM*, we define the *correct match* metric, which means the fraction of matched vertices with the same value as the original result, i.e., the result under the failure-free case. This metric is



(a) MWM + Orkut (b) MWM + WebCC12

Fig. 9. Result accuracy

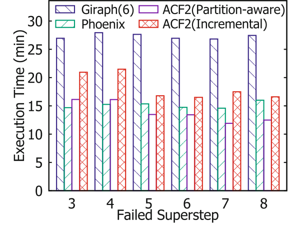


Fig. 10. Different failed supersteps on MWM

computed via $m_c(v)/m_a(v) * 100\%$, where $m_c(v)$ denotes the matched vertices with the same value as the original result and $m_a(v)$ denotes all matched vertices in the original result.

As shown in Fig. 9, Phoenix and the partition-aware backup strategy do not effectively handle failures when running graph algorithms with topological mutations. For example, in Fig. 9(b), Giraph achieves no accuracy loss in all cases. However, Phoenix and the partition-aware backup strategy achieve the maximum accuracy of around 40% when $n = 1$. Moreover, the accuracy of Phoenix and the partition-aware backup strategy continue to decline as n increases from 1 to 3.

The incremental protocol efficiently handles failures when running graph algorithms with topological mutations. As shown in Fig. 9(a), compared to Phoenix and the partition-aware backup strategy, the incremental protocol leads to an inaccuracy of 0.4% when $n = 1$. Further, when n increases from 1 to 3, the incremental protocol still ensures an accuracy over 98.6%, even though its accuracy decreases. The incremental protocol still sacrifices the accuracy of the results, which is because we limit the number of iterations. However, our protocol achieves 100% accurate results if we do no limit the number of iterations [2].

Different Failed Supersteps. Similarly, we continue to investigate the performance trend of the incremental protocol when failure occurs in different supersteps. Also, we follow the previous failure setting. Figure 10 provides the results on the Friendster dataset.

As shown in Fig. 10, no matter in which superstep the failure happens, the incremental protocol always outperforms Giraph. Moreover, the incremental protocol performs worse than Phoenix and the partition-aware backup strategy, when the failure occurs before S_5. This is because our protocol introduces logging overhead and employs a similar way as Phoenix to recover lost sub-graphs. However, the performance of the incremental protocol is close to that of Phoenix when the failure occurs after S_5, due to the backup of sub-graphs.

In summary, our incremental protocol outperforms Phoenix, since we effectively handle failures involving topological mutations. Also, our protocol achieves similar accuracy as Giraph and faster execution than Giraph.

7 Related Work

This section discusses the related work on recovery solutions of distributed graph processing systems, in term of checkpoint-based and checkpoint-free solutions.

The checkpoint-based solutions require the system to write checkpoints or maintain a certain number of replicas during normal execution. Once failure occurs, the system utilizes the checkpoints or replicas to recover. In general, distributed graph processing systems such as Pregel [9], GraphLab [7], GraphX [4] and PowerGraph [3] employ the checkpoint-based solution to tolerate failure. These systems periodically write the checkpoints during normal execution and reload the latest checkpoint upon failure. Rather than writing all the edges in each checkpoint, the lightweight checkpointing [17] reduces the data volume of checkpoint by saving the vertices and incrementally storing edges. Xu et al. [16] explore unblocking checkpoint to decrease the overhead of blocking checkpoint for graph processing on dataflow systems. CoRAL [14] applies unblocking checkpointing to asynchronous graph processing systems. Shen et al. [13] propose a repartition strategy to reduce the communication cost during recovery, so as to accelerate recovery. However, these works focus on checkpoint-based failure recovery, whereas our work targets checkpoint-free failure recovery. Also, Spark [18] employs checkpoint-based solutions to tolerate failure. In particular, Spark utilizes the lineage of RDD to accelerate recovery. During recovery, it replays the transformation to recompute the lost RDD sub-graphs. However, our work directly loads the lost sub-graphs from the backups and resets the vertex value by semantic properties of graph algorithms, which avoids the recomputation.

In contrast to the checkpoint-based solutions, the checkpoint-free solutions achieve failure recovery without any checkpoint or replica. Schelter et al. [12] propose the optimistic recovery which reloads the lost partitions from input data and applies the algorithmic compensations in the lost partitions once failure happens. As a variant of optimistic recovery, Zorro [11] utilizes the implicit replicas of vertices in graph systems to recover the value of lost vertices, so as to accelerate recovery. The optimistic recovery is applicable only to partial graph algorithms. Different from optimistic recovery, Phoenix [2] classifies existing graph algorithms into four classes and provides APIs for these different classes of algorithms to achieve failure recovery. However, these works do not consider the additional overhead caused by recovering lost sub-graphs, and the failure involving topological mutations.

8 Conclusions

This paper proposes ACF2 to mitigate the shortcoming of checkpoint-free solution. In specific, ACF2 includes a partition-aware backup strategy and an incremental protocol. Instead of reloading input data, the partition-aware backup strategy recovers the lost sub-graphs from the backups on DFS, so as to reduce the recovery overhead incurred by checkpoint-free solution. The incremental protocol which logs topological mutations during normal execution and employs

these logs for failure recovery. The experimental studies show that ACF2 outperforms existing checkpoint-free solutions in general. Presently, we implement ACF2 based on Giraph. Nevertheless, it is possible to integrate ACF2 in other distributed graph processing systems such as GraphLab.

Acknowledgments. This work has been supported by the National Natural Science Foundation of China (No. 61902128).

References

1. Ammar, K., et al.: Experimental analysis of distributed graph systems. Proc. VLDB Endow. **11**(10), 1151–1164 (2018)
2. Dathathri, R., et al.: Phoenix: a substrate for resilient distributed graph analytics. In: ASPLOS, pp. 615–630 (2019)
3. Gonzalez, J.E., et al.: Powergraph: distributed graph-parallel computation on natural graphs. In: OSDI, pp. 17–30 (2012)
4. Gonzalez, J.E., et al.: Graphx: graph processing in a distributed dataflow framework. In: OSDI, pp. 599–613 (2014)
5. Kalavri, V., et al.: High-level programming abstractions for distributed graph processing. IEEE Trans. Knowl. Data Eng. **30**(2), 305–324 (2018)
6. Li, B., et al.: : A trusted parallel route planning model on dynamic road networks. TITS (2022)
7. Low, Y., et al.: Distributed graphLab: a framework for machine learning in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
8. Lu, Y., et al.: Large-scale distributed graph computing systems: an experimental evaluation. Proc. VLDB Endow. **8**(3), 281–292 (2014)
9. Malewicz, G., et al.: Pregel: a system for large-scale graph processing. In: SIGMOD, pp. 135–146 (2010)
10. McCune, R.R., Weninger, T., Madey, G.: Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Comput. Surv. **48**(2), 1–39 (2015)
11. Pundir, M., et al.: Zorro: zero-cost reactive failure recovery in distributed graph processing. In: SoCC, pp. 195–208 (2015)
12. Schelter, S., et al.: “All roads lead to rome”: optimistic recovery for distributed iterative data processing. In: CIKM, pp. 1919–1928 (2013)
13. Shen, Y., et al.: Fast failure recovery in distributed graph processing systems. Proc. VLDB Endow. **8**(4), 437–448 (2014)
14. Vora, K., et al.: Coral: confined recovery in distributed asynchronous graph processing. In: ASPLOS, pp. 223–236 (2017)
15. Wang, P., et al.: Replication-based fault-tolerance for large-scale graph processing. In: DSN, pp. 562–573 (2014)
16. Xu, C., et al.: Efficient fault-tolerance for iterative graph processing on distributed dataflow systems. In: ICDE, pp. 613–624 (2016)
17. Yan, D., et al.: Lightweight fault tolerance in Pregel-like systems. In: ICPP, pp. 1–10 (2019)
18. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI, pp. 15–28 (2012)