Fast Failure Recovery in Distributed Graph Processing Systems

Yanyan Shen[§], Gang Chen[†], H. V. Jagadish[‡], Wei Lu[¶*], Beng Chin Ooi[§], Bogdan Marius Tudor[§]

[§]National University of Singapore, [†]Zhejiang University, [‡]University of Michigan, [¶]Renmin University
[§]{shenyanyan,ooibc,bogdan}@comp.nus.edu.sg, [†]cg@cs.zju.edu.cn, [‡]jag@umich.edu, [¶]lu-wei@ruc.edu.cn

ABSTRACT

Distributed graph processing systems increasingly require many compute nodes to cope with the requirements imposed by contemporary graph-based Big Data applications. However, increasing the number of compute nodes increases the chance of node failures. Therefore, provisioning an efficient failure recovery strategy is critical for distributed graph processing systems. This paper proposes a novel recovery mechanism for distributed graph processing systems that parallelizes the recovery process. The key idea is to partition the part of the graph that is lost during a failure among a subset of the remaining nodes. To do so, we augment the existing checkpoint-based and log-based recovery schemes with a partitioning mechanism that is sensitive to the total computation and communication cost of the recovery process. Our implementation on top of the widely used Giraph system outperforms checkpoint-based recovery by up to 30x on a cluster of 40 compute nodes.

1. INTRODUCTION

Graphs capture complex relationships and data dependencies, and are important to Big Data applications such as social network analysis, spatio-temporal analysis and navigation, and consumer analytics. MapReduce was proposed as a programming model for Big Data about a decade ago, and since then, many MapReduce-based distributed systems have been designed for Big Data applications such as large-scale data analytics [16]. However, in recent years, MapReduce has been shown to be ineffective for handling graph data, and several new systems such as Pregel [20], Giraph [1], GraphLab [10, 18], and Trinity [24] have been recently proposed for scalable distributed graph processing.

With the explosion in graph size and increasing demand of complex analytics, graph processing systems have to continuously scale out by increasing the number of compute nodes, in order to handle the load. But scaling the number of nodes has two effects on the failure resilience of a system. First, increasing the number of nodes will inevitably lead to an increase in the number of failed nodes. Second, after a failure, the progress of the entire system is halted until the failure is recovered. Thus, a potentially large number of nodes will become idle just because a small set of nodes have failed. In order to scale out the performance continuously when the number of nodes increases, it is becoming crucial to provision the graph processing systems with the ability to handle failures effectively.

The design of failure recovery mechanisms in distributed systems is a nontrivial task, as they have to cope with several adversarial conditions. Node failures may occur at any time, either during normal job execution, or during recovery period. The design of a recovery algorithm must be able to handle both kinds of failures. Furthermore, the recovery algorithm must be very efficient because the overhead of recovery can degrade system performance significantly. To a certain extent, due to the long recovery time, failures may occur repeatedly before the system recovers from an initial failure. If so, the system will go into an endless recovery loop without any progress in execution. Finally, the system must cope with the failures while maintaining the recovery mechanism transparent to user applications. This implies that the recovery algorithm can only rely on the computation model of the system, rather than any computation logic applied for specific applications.

The usual recovery method adopted in current distributed graph processing systems is *checkpoint-based* [17, 20, 26]. It requires each compute node to periodically and synchronously write the status of its own subgraph to a stable storage such as the distributed file system as a checkpoint. Upon any failure, checkpoint-based recovery employs an unused healthy compute node to replace each failed node and requires all the compute nodes to load the status of subgraphs from the most recent checkpoint and then synchronously re-execute all the missing workloads. A failure is recovered when all the nodes finish the computations that have been completed before the failure occurs. Note that the recomputation will be replayed again whenever a further failure occurs during recovery.

Although checkpoint-based recovery is able to handle any node failures, it potentially suffers from high recovery latency. The reason is two-fold. First, checkpoint-based recovery re-executes the missing workloads over the whole graph, residing in both failed and healthy compute nodes, based on the most recent checkpoint. This could incur high computation cost as well as high communication cost, including loading the whole checkpoint, performing recomputation and passing the messages among all compute nodes during the recovery. Second, when a further failure occurs during the recovery, the lost computation caused by the previous failure may have been partially recovered. However, checkpoint-based recovery will forget about all of this partially completed workload, rollback every compute node to the latest checkpoint and replay the computation since then. This eliminates the possibility of performing the recovery progressively.

In this paper, we propose a new recovery scheme to enable fast failure recovery. The key idea is to 1) *restrict the recovery workload*

[*]This work was done at the National University of Singapore.

Proceedings of the VLDB Endowment, Vol. 8, No. 4
Copyright 2014 VLDB Endowment 2150-8097/14/12.

Table 1: Symbols and Their Meanings

Symbol	Definition
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$
N	compute node
$\mathcal{V}_N$	vertices that reside in node N
$\mathcal{P}$	graph partitions
$\mathcal{N}_\mathrm{f}$	failed nodes
s_f	superstep that a failure occurs
F	failure
F^i	i-th cascading failure for F
S	state
φ	vertex to partition mapping
ϕ_p	partition to node mapping
ϕ_r	failed partition to node mapping (reassignment)

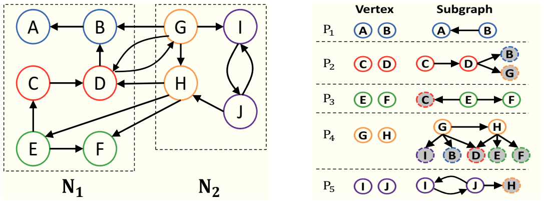
to the subgraphs residing in the failed nodes using locally logged messages; 2) distribute the subgraphs residing in the failed nodes among a subset of compute nodes to *redo the lost computation concurrently*. In our recovery scheme, in addition to global checkpointing, we require every compute node to log their outgoing messages locally. Upon a failure, the system first replaces each failed node with a new one. It then divides the subgraphs residing in the failed nodes into partitions, referred to as *failed* partitions, and distributes these partitions among a subset S of compute nodes. During recovery, every node in S will hold its original subgraph and load the status of its newly received partitions from the latest checkpoint. When the system re-executes missing workloads, the recomputation is confined to the failed partitions by nodes in S concurrently, using logged messages from healthy subgraphs and recalculated ones from failed partitions. To distribute the lost subgraphs effectively, we propose a computation and communication cost model to quantify the recovery time, and according to the model, we split the lost subgraphs among a subset of compute nodes such that the total recovery time is minimized.

To the best of our knowledge, this is the first parallel recovery mechanism proposed for distributed graph processing. In contrast with traditional checkpoint-based recovery, our approach eliminates the high recomputation cost for the subgraphs residing in the healthy nodes due to the fact that failures often occur among a small fraction of compute nodes. Note that the subgraph in a healthy node can include both its original subgraph (whose computation is never lost) and a set of newly received partitions (whose computation is partially recovered) due to previous failures. Furthermore, we distribute the recomputation tasks for the subgraphs in the failed nodes among multiple compute nodes to achieve better parallelism. Thus, our approach is not a replacement for checkpoint-based recovery methods. Instead, it complements them because it accelerates the recovery process through simultaneous reduction of recovery communication costs and parallelization of the recovery computations.

Our proposed recovery scheme is an important component of our epiCG project: a scalable graph engine on top of epiC [12]. In this paper, we use Giraph [1] as the underlying graph engine for the experiments as epiCG was under development when our recovery method was being designed. We have fully implemented the proposed recovery mechanism onto epiCG and would report the results in the future.

Contributions. Our contributions are summarized as follows.

- We formally define the failure recovery problem in distributed graph processing systems and introduce a partition-based recovery method that can efficiently handle any node failures, either during normal execution or during the recovery period (Section 2 and 3).
- We formalize the problem of distributing recomputation tasks for subgraphs residing in the failed nodes as a reassignment generation problem: find a reassignment for failed partitions with minimized recovery time. We show the problem is NP-hard and propose a cost-sensitive reassignment algorithm (Section 4).



(a) $\mathcal{G}(\mathcal{V}, \mathcal{E})$ (b) $\mathcal{P}$

Figure 1: Distributed Graph and Partitions

- We implement our proposed parallel recovery method on top of the widely used Apache Giraph graph processing system, and release it as open source[1] (Section 5).
- We conduct extensive experiments on real-life datasets using both synthetic and real applications. Our experiments show our proposed recovery method outperforms traditional checkpoint-based recovery by a factor of 12 to 30 in terms of recovery time, and a factor of 38 in terms of the network communication cost using 40 compute nodes (Section 6).

2. BACKGROUND AND PROBLEM

In this section, we provide some background of distributed graph processing systems (DGPS), define our problem and discuss the challenges of failure recovery in DGPS. Table 1 lists the symbols and their meanings used throughout this paper.

2.1 Background of DGPS

Distributed graph. The input to distributed graph processing systems is a directed graph[2] $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ are the sets of vertices and edges, respectively. Every vertex in the graph has a unique vertex identifier. In distributed graph processing, the set of vertices is divided into partitions. A partition of $\mathcal{G}$ is formally denoted by $P_i = (V_i, E_i)$, where $V_i \subseteq \mathcal{V}$ and $E_i = \{\langle v_i, v_j \rangle \in \mathcal{E} | v_i \in V_i\}$. Note that E_i includes all the outgoing edges from vertices in V_i, which may cross partitions. All the partitions are distributed among compute nodes, i.e., physical machines.

Let $\mathcal{P}$ and $\mathcal{N}$ respectively be the set of partitions and the set of compute nodes. Typically, the number of partitions is larger than that of compute nodes (i.e., $|\mathcal{P}| > |\mathcal{N}|$), to achieve a better load balance. For ease of illustration, we denote by φ, ϕ_p two mappings, where (vertex-partition mapping) $\varphi : \mathcal{V} \to \mathcal{P}$ records which vertex belongs to which partition and (partition-node mapping) $\phi_\mathrm{p} : \mathcal{P} \to \mathcal{N}$ records which partition resides in which compute node. For any node $N \in \mathcal{N}$, we denote by $\mathcal{V}_N$ the set of vertices residing in N.

Figure 1(a) shows a distributed graph $\mathcal{G}$ over two nodes N_1, N_2. $\mathcal{G}$ is divided into 5 partitions P_1 to P_5, as shown in Figure 1(b). We use colors to differentiate vertices in different partitions.

Computation model. The computation model in Pregel-like DGPS follows the Bulk Synchronous Parallel (BSP) model [25]. Typically, the computation consists of an *input* phase, where a graph is distributed among the compute nodes, followed by a set of iterations, called *supersteps*, separated by global synchronization points, and finally an *output* phase. Every vertex carries two states: *active* and *inactive*. Initially (at the beginning of superstep 1), all the vertices are active. A vertex can deactivate itself by *voting to halt*. Once a vertex becomes inactive, it has no further work to do in the following supersteps unless activated by incoming messages from

[1] http://www.comp.nus.edu.sg/∼epic/recovery/

[2] Undirected graphs can be represented as directed graphs where for every edge $\langle u, v \rangle$ there is a corresponding edge $\langle v, u \rangle$

other vertices. Within each superstep, only active vertices participate in computation: process messages sent by other vertices in the previous superstep, update its value or the values of its outgoing edges and send messages to other vertices (to be processed in the next superstep). This kind of computation logic is expressed by a user-defined function. All the active vertices in the same compute node execute the function sequentially, while the execution in each compute node is performed in parallel with other nodes. After all the active vertices finish their computation in a superstep, a global synchronization point is reached.

Basic architecture. Pregel-like DGPS follows a master/slave architecture. The master is responsible for coordinating the slaves, but is not assigned any graph partitions. The slaves are in charge of performing computation over its assigned partitions in each superstep. More details on this architecture can be found in [20].

2.2 Failure Recovery in DGPS

2.2.1 Checkpointing Scheme

We consider synchronous checkpointing to be performed every $\mathbb{C}(\in \mathbb{N}^+)$ supersteps. At the beginning of superstep $i\mathbb{C}+1(i \in \mathbb{N}^+)$, we flush the complete graph status into reliable storage such as a distributed file system, including the graph structure, vertex values, vertex status(active/inactive), edge values, incoming messages received in the previous superstep, and other auxiliary information. The saved status is called a *checkpoint*. In short, a checkpoint made in superstep $i\mathbb{C}+1$ records the graph status after the completion of superstep $i\mathbb{C}$. We assume that no failures occur during checkpointing.

2.2.2 Problem Statement

We consider a graph job that is executed on a set $\mathcal{N}$ of compute nodes from superstep 1 to s_{max}. A compute node may fail at any time during the normal job execution. Let $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$ denote a failure that occurs on a set $\mathcal{N}_\mathrm{f}(\subseteq \mathcal{N})$ of compute nodes when the job performs normal execution in superstep $s_\mathrm{f}(\in [1, s_{max}])$. We associate with F two states S_F and S_F^*, which record the statuses of vertices before and after the recovery of F, respectively.

Definition 1 (State)**.** *The state S is a function: $\mathcal{V} \rightarrow \mathbb{N}^+$ recording the latest superstep that has been completed by each vertex (no matter it is active or inactive in that superstep) at a certain time.*

After F is detected, all the vertices residing in the failed nodes are lost and their latest statuses are stored in the latest checkpoint. The recovery for F is initiated after all the healthy nodes finish their execution in superstep s_f. Let $c+1$ be the superstep when the latest checkpoint is made. We have:

$$S_F(v) = \begin{cases} c & v \in \bigcup_{N \in \mathcal{N}_\mathrm{f}} \mathcal{V}_N \\ s_\mathrm{f} & \text{Otherwise} \end{cases} \tag{1}$$

In general, the recovery for F is to re-execute the computation from the latest checkpointing superstep to superstep s_f. Hence, we have:

$$S_F^*(v) = s_\mathrm{f}, \qquad \forall v \in \mathcal{V} \tag{2}$$

We now formalize the failure recovery problem as follows.

Definition 2 (Failure recovery)**.** *Given $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$, the recovery for F is to transform the statuses of all the vertices from S_F to S_F^*.*

Example 1 (Running example)**.** *Consider graph $\mathcal{G}$ distributed over compute nodes N_1, N_2 in Figure 1(a) and failure $F(\{N_1\}, 12)$, i.e., N_1 fails during the normal execution of superstep 12. Assume that every vertex is active and sends messages to all its neighbors in normal execution of each superstep, and the latest checkpoint was made in the beginning of superstep 11. S_F and S_F^* are the following.*

- *$\forall v \in \{A, B, C, D, E, F\}$, $S_F(v) = 10$ and $S_F^*(v) = 12$;*
- *$\forall v \in \{G, H, I, J\}$, $S_F(v) = 12$ and $S_F^*(v) = 12$;*

The recovery for F is to transform the status of each vertex to the one achieved after the completion of superstep 12.

2.2.3 Challenging Issues

Consider a failure $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$ that occurs during normal execution of a graph job. During the recovery for F, compute nodes may fail at any time. More specifically, multiple failures may occur sequentially before the system achieves state S_F^*. We refer to these failures as the cascading failures for F.

Definition 3 (Cascading failure)**.** *Given $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$, a cascading failure for F is a failure that occurs during the recovery for F, i.e., after F occurs but before F is recovered.*

Let $\mathbb{F}$ be a sequence of all the cascading failures for F. We denote by F^i the i-th cascading failure in $\mathbb{F}$.

Challenge 1. The key challenge of recovering F is to handle cascading failures for F. To the best of our knowledge, we are not aware of any previous works that provide details on how to handle cascading failures in distributed graph processing systems.

Our goal is to speed up the recovery process for F. Informally, the time of recovering F is contributed by three main tasks:

- re-execute computation such that the status of every vertex is updated to the one achieved after the completion of superstep s_f.
- forward inter-node messages during recomputation.
- recover cascading failures for F.

A naive recovery method re-runs the job from the first superstep upon the occurrence of each failure. Obviously, such an approach incurs long recovery time as the execution of every superstep can be costly in many real-world graph jobs. In the worst case, the failure occurs during the execution of the final superstep and the system needs to redo all the supersteps. Furthermore, it is more likely that a cascading failure will occur as the recovery time becomes longer.

Challenge 2. Given a failure F, we denote by $\Gamma(F)$ the recovery time for F, i.e., the time span between the start and the completion of recovering F. The objective of this paper is to recover F with *minimized* $\Gamma(F)$.

In what follows, we first describe how locally logged messages can be utilized for failure recovery, which is the basis of our recovery algorithm, and then discuss its limitations.

2.2.4 Utilizing Locally Logged Messages

Besides checkpoints, we require every compute node to log its outgoing messages at the end of each superstep. The logged messages are used to reduce recovery workload. Consider a failure $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$. Following the checkpoint-based recovery method, for any failed node $N \in \mathcal{N}_\mathrm{f}$, we employ a new available node to replace N and assign partitions in N to it. We require all the replacements to load the status of received partitions from the latest checkpoint, and all the healthy nodes hold their original partitions. In superstep $i \in [c+1, s_\mathrm{f}]$ during recovery, only the replacements perform computation for the vertices in failed partitions, while every healthy node forwards locally logged messages to the vertices in failed partitions without any recomputation. For $i \in [c+1, s_\mathrm{f})$, the vertices in failed partitions forward messages to each other, but for $i = s_\mathrm{f}$, they send messages to those in healthy nodes as well.

Example 2. *Continue with Example 1. To recover $F(\{N_1\}, 12)$, we first employ a new node to replace N_1 and then re-execute supersteps 11, 12. We refer to the new node by N_1. N_1 loads the*

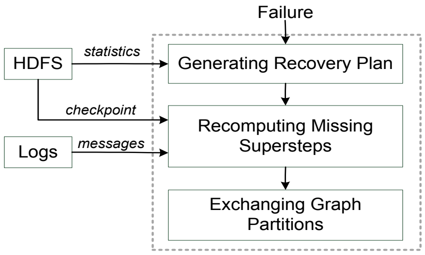


Figure 2: Failure Recovery Executor

statuses of P_1, P_2, P_3 from the latest checkpoint. During recovery, only N_1 performs computation for 6 vertices A-F in two supersteps, while the recomputation for vertices G-J is avoided. Superstep 11 incurs 5 logged inter-node messages $G \to B$, $G \to D$, $H \to D$, $H \to E$, $H \to F$. Superstep 12 incurs 6 inter-node messages: the above five logged ones plus a recalculated one $D \to G$.

Utilizing locally logged messages helps to *confine* recovery workload to the failed partitions (in terms of both recomputation and message passing), thus reducing recovery time. Moreover, the overhead of locally logging is negligible in many graph applications as the execution time is dominated by computation and network message passing (see details in Section 6). However, the recomputation for the failed partitions is shared among the nodes that replace the failed ones and this achieves limited parallelism as the computation in one node can only be executed sequentially. This inspires us to *reassign* failed partitions to multiple nodes to achieve parallelism of recomputation.

3. PARTITION-BASED RECOVERY

We propose a partition-based method to solve the failure recovery problem. Upon a failure, the recovery process is initiated by the recovery executor. Figure 2 shows the workflow of our partition-based failure recovery executor. Let $c+1$ be the latest checkpointing superstep for the failure. The recovery executor is responsible for the following three tasks.

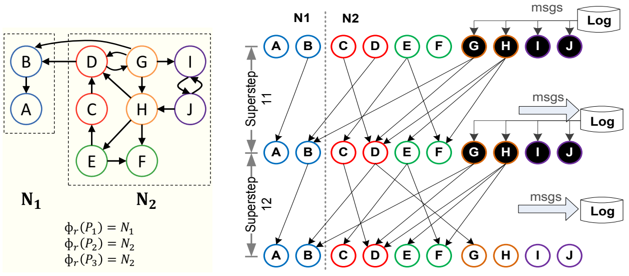
• **Generating partition-based recovery plan.** The input to this task includes the state before recovery starts and the *statistics* stored in reliable storage, e.g., HDFS. We collect statistics during checkpointing, including:
(1) computation cost of each partition in superstep c.
(2) partition-node mapping ϕ_{p} in superstep c.
(3) for any two partitions in the same node, the size of messages forwarded from one to another in superstep c.
(4) for each partition, the size of messages from an outside node (where the partition does not reside) to the partition in superstep c.
The statistics require a storage cost of $\mathrm{O}(|\mathcal{P}| + |\mathcal{P}||\mathcal{N}| + |\mathcal{P}|^2)$, which is much lower than that of a checkpoint.

The output recovery plan is represented by a *reassignment* for failed partitions, which is formally defined as follows.

Definition 4 (Reassignment)**.** *For any failure, let $\mathcal{P}_{\mathrm{f}}$ be the set of partitions residing in the failed nodes. The reassignment for the failure is a function ϕ_{r}: $\mathcal{P}_{\mathrm{f}} \to \mathcal{N}$.*

Figure 3(a) shows a reassignment for $F(\{N_1\}, 12)$ in Example 1. We assign P_1 to N_1 (the replacement) and P_2, P_3 to N_2.

• **Recomputing failed partitions.** This task is to inform every compute node of the recovery plan ϕ_{r}. Each node N checks ϕ_{r} to see whether a failed partition is assigned to it. If so, N loads the partition status from the latest checkpoint. The status of a partition includes (1) the vertices in the partition and their outgoing edges; (2) values of the vertices in the partition achieved after the completion of superstep c; (3) the status (i.e., active or inactive) of every vertex in the partition in superstep $c+1$; (4) messages received by the vertices in the partition in superstep c (to be processed in superstep $c+1$). Every node then starts recomputation for failed partitions. The details are provided in Section 3.1.



(a) Reassignment

(b) Recomputation

Figure 3: Recovery for $F(\{N_1\}, 12)$

Algorithm 1: Recomputation

```
   Input: $S$, the state when failure occurs
          $i$, current superstep
          $N$, a compute node
1  $M$ ←logged outgoing messages in superstep $i$;
2  for $v \in \mathcal{V}_N$ do
3      for $v$.Active= $True$ and $S(v) < i$ do
4          Perform computation for $v$;
5          $M \leftarrow M \cup v.Sendmsgs$;
6  for $m \in M$ do
7      $v_s \leftarrow m.Getsrc(); v_d \leftarrow m.Getdst()$;
8      if $S(v_d) < i$ or $(S(v_s) < i \wedge S(v_d) = i)$ then
9          Send $m$ to $v_d$;
10 Flush $M$ into local storage;
```

• **Exchanging graph partitions.** This task is to re-balance the workload among all the compute nodes after the recomputation of the failed partitions completes. If the replacements have different configurations than the failed nodes, we allow a new partition assignment (different from the one before failure occurs) to be employed for a better load balance, following which, the nodes might exchange partitions among each other.

3.1 Recomputing Failed Partitions

Consider a failure $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$ that occurs during normal execution. The recomputation for the failed partitions starts from the most recent checkpointing superstep $c+1$. After all the compute nodes finish superstep i, they proceed to superstep $i+1$ synchronously. The goal of recovery is to achieve state S^*_F (see Equation 2). Therefore, the recomputation terminates when all the compute nodes complete superstep s_{f}.

Algorithm 1 provides recomputation details in a superstep during recovery. Consider a node N. In superstep i ($\in [c+1, s_{\mathrm{f}}]$), N maintains a list M of messages that will be sent by vertices residing in N in the current superstep. Initially, M contains all the locally logged outgoing messages for superstep i if any (line 1). N then iterates through all the active vertices residing in it and for each active vertex, N executes its computation and appends all the messages sent by this vertex to M if the vertex value has not been updated to the one achieved in the end of superstep i (line 2-5). After that, N iterates over messages in M. A message m in M is forwarded if m is needed by its destination vertex to perform recomputation in the next superstep (line 6-9). Finally, N flushes M into its local storage (line 10), which will be used in case there are further cascading failures.

Example 3. *Figure 3(b) illustrates recomputation for $F(\{N_1\}, 12)$, given ϕ_r in Figure 3(a). We use directed edges to represent the forwarding messages. In superstep 11, N_1 and N_2 respectively perform 2 and 4 vertex computations for A-F; 2 inter-node messages $D \to B$, $G \to B$ are forwarded. N_2 retrieves 4 logged messages sent by G in normal execution of superstep 11 but only re-sends messages to B, D because H, I belongs to healthy partition P_4. Further, N_1, N_2 will log 5 messages sent by A-F locally as they have not yet been included in the log. Superstep 12 performs similarly, except for an additional message $D \to G$. Compared with the approach in Example 2, our algorithm achieves more parallelized recomputation and incurs less network communication cost.*

Note that the messages received by the vertex during recomputation might have a different order compared with those received during normal execution. Therefore, the correctness of our recomputation logic implicitly requires the vertex computation is insensitive to message order. That is, given messages with an arbitrary order, the *effect* of a vertex computation (new vertex value and its sending messages) remains the same. This requirement is realistic since a large range of graph applications are implemented in a message-ordering independent manner. Example includes PageRank, breadth first search, graph keyword search, triangle counting, connected component computation, graph coloring, minimum spanning forest computation, k-means, shortest path, minimum cut, clustering/semi-clustering. While we are not aware of any graph algorithms that are nondeterministic with respect to the message order, our recovery method can be extended easily to support such algorithms if there is any. Specifically, we can assign a unique identifier to each message. Recall that all the messages to be processed in a superstep must be completely collected by graph processing engine before any vertex computation starts. In each superstep (either during normal execution or recovery), for every active vertex v, we can sort all these messages received by v based on their identifiers, before initiating the computation. The sorting ensures the messages for a vertex computation during normal execution follow the same order as those for recomputation during recovery.

3.2 Handling Cascading Failures

We now consider cascading failures for $F(\mathcal{N}_f, s_f)$, which occur before F is recovered. A useful property of our partition-based recovery algorithm is that for any failure, the behavior of every compute node *only* relies on the reassignment for the failure and the state after the failure occurs. That is, in our design, given the reassignment and state for the failure, the behavior of every node is independent of what the failure is. The failure can be F itself or any of its cascading failures. Therefore, whenever a cascading failure for F occurs, the currently executing recovery program is terminated and the recovery executor can start a new recovery program for the new failure using the same recovery algorithm.

In practice, the occurrence of failures is not very frequent and hence we expect at least one recovery program to complete successfully. F is recovered when a recovery program exits normally. That is, all the vertices complete superstep s_f and S^*_F is achieved. Further, due to cascading failures, a compute node may receive new partitions during the execution of each recovery program. After recomputation finishes, nodes may exchange partitions to re-balance the workload. The following example illustrates how our recovery algorithm is used to handle cascading failures.

Example 4. *We start with a recovery program for $F(\{N_1\}, 12)$ in Example 3. Suppose a cascading failure F^1 occurs in N_2 when the program is executing in superstep 12. Vertices C-J residing in N_2 are lost due to F^1, while A, B in healthy node N_1 are recovered. Hence, the state S_{F^1} after F^1 occurs satisfies: $S_{F^1}(A) = S_{F^1}(B) = 12$ and $S_{F^1}(v) = 10$ for $v =$ C-J. A new recovery program is initiated for F^1. Suppose the reassignment for F^1 assigns P_2, P_3 to N_1 and P_4, P_5 to N_2 (replacement). N_1, N_2 load the statuses of newly assigned partitions from the latest checkpoint and start recomputation as shown in Figure 4.*

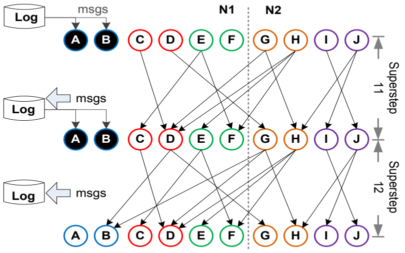


Figure 4: Recomputation for Cascading Failure F^1

Since $S_{F^1}(A) = S_{F^1}(B) = 12$, we only perform recomputation for vertices C-J in newly failed partitions P_2-P_5 when re-executing superstep $11, 12$. In superstep 11, C-J forward messages to each other. In superstep 12, these vertices send messages to A, B as well. Suppose there is no further cascading failure after F^1. The recovery for F is accomplished upon the completion of the new recovery program triggered by F^1.

Example 4 considers cascading failures that occur during recomputation. In practice, failures may occur at any time. If a failure occurs during the period of generating a recovery plan for the previous failure, we treat both failures as one *bigger* failure and the union of their failed nodes as the failed node set. If a failure occurs during the exchanging phase, we treat it as the one that occurs in superstep s_f. Our recovery approach can be applied to both cases.

Without loss of generality, in the rest of this paper, we only consider cascading failures that occur in a recomputation phase.

3.3 Correctness and Completeness

We first focus on the *correctness* of our recovery method.

Definition 5 (Correctness). *Let* $\textsc{Value}(v, i)$ *and* $\textsc{Msg}(v, i)$ *denote the value and the set of received messages for vertex v in the end of superstep i during normal execution, respectively. A recovery algorithm is* correct if for any failure $F(\mathcal{N}_f, s_f)$, after the recovery algorithm finishes, the value of every vertex v equals to $\textsc{Value}(v, s_f)$ and the set of messages received by v equals to $\textsc{Msg}(v, s_f)$.

The above definition is under the assumption that the vertex computation is deterministic with respect to message ordering, which is the case in real-world graph applications. The correctness of our recovery algorithm is based on two properties of Algorithm 1.

Lemma 1. *Algorithm 1 has the following properties:*
(1) A vertex performs computation in superstep i iff the vertex status has not been updated to the one achieved at the end of superstep i during normal execution, $\forall i \in [c+1, s_f]$.
(2) Vertex v sends messages to u in superstep i iff u will perform a computation in superstep $i+1$, $\forall i \in [c+1, s_f]$.

Lemma 1 holds for the recovery program triggered by any failure. In essence, Lemma 1 guarantees: i) the input (vertex value and received messages) of a vertex computation in any superstep during recomputation is exactly the same as that during normal execution; ii) for failure $F(\mathcal{N}_f, s_f)$, when our recovery algorithm finishes successfully, each vertex completes superstep s_f and receives the same

Algorithm 2: CostSensitiveReassign

```
Input  : $S$, state after the failure occurs
         $\mathcal{P}_{\mathrm{f}}$, failed partitions
         $\mathcal{I}$, statistics
         $\mathcal{N}$, a set of compute nodes
Output: $\phi_{\mathrm{r}}$: reassignment
$\phi_{\mathrm{r}} \leftarrow$ RandomAssign($\mathcal{P}_{\mathrm{f}}, \mathcal{N}$);
$T_{low} \leftarrow$ ComputeCost($\phi_{\mathrm{r}}, S, \mathcal{I}$);
while true do
    $\phi_{\mathrm{r}}' \leftarrow \phi_{\mathrm{r}}$; $\mathcal{P} \leftarrow \mathcal{P}_{\mathrm{f}}$; $i \leftarrow 0$;
    while $\mathcal{P} \neq \emptyset$ do
        $i \leftarrow i + 1$;
        $\mathcal{L}_i \leftarrow$ NextChange($\phi_{\mathrm{r}}', \mathcal{P}, S, \mathcal{I}$);
        foreach $P \in \mathcal{L}_i.\phi.Keys()$ do
            $\phi_{\mathrm{r}}'(P) \leftarrow \mathcal{L}_i.\phi(P)$;
            $\mathcal{P} \leftarrow \mathcal{P} - \{P\}$;
    $l \leftarrow \arg\min_i \ \mathcal{L}_i.Time$;
    if $\mathcal{L}_l.Time < T_{low}$ then
        for $j = 1$ to $l$ do
            foreach $P \in \mathcal{L}_j.\phi.Keys()$ do
                $\phi_{\mathrm{r}}(P) \leftarrow \mathcal{L}_j.\phi(P)$;
        $T_{low} \leftarrow \mathcal{L}_l.Time$;
    else
        break;
```

set of messages as it does at the end of superstep s_{f} during normal execution. These properties ensure the correctness of our approach.

Furthermore, our recovery algorithm is *complete* in that the recovery logic is independent of high-level applications. That is, any node failure can be correctly recovered using our algorithm.

Theorem 1. *Our partition-based recovery algorithm is correct and complete.*

4. REASSIGNMENT GENERATION

In this section, we present how to generate a reassignment for a failure. Consider a failure $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$. The reassignment for F is critical to the overall recovery performance, i.e., the time span of recovery. In particular, it decides the computation and communication cost during recomputation. Our objective is to find a reassignment that minimizes the recovery time $\Gamma(F)$.

Given a reassignment for F, the calculation of $\Gamma(F)$ is complicated by the fact that $\Gamma(F)$ depends not only on the reassignment for F, but also on the cascading failures for F and the corresponding reassignments. However, the knowledge of cascading failures can hardly be obtained beforehand since F and its cascading failures do not arrive as a batch but come sequentially. Hence, we seek an *online* reassignment generation algorithm that can react in response to any failure, without knowledge of future failures.

Our main insight is that when a failure (either F or its cascading failure) occurs, we prefer a reassignment that can benefit the *remaining* recovery process for F by taking into account all the cascading failures that have already occurred. More specifically, we collect the state S after the failure occurs and measure the minimum time T_{low} required to transform from S to S_F^*, i.e., the time of performing recomputation from superstep $c+1$ to s_{f} without further cascading failures. We then aim to produce a reassignment that minimizes T_{low}. Essentially, S encapsulates all the useful information about previous failures and the corresponding reassignments performed, and T_{low} provides a lower bound of remaining recovery time for F. In what follows, we introduce how to compute T_{low} and then provide our cost-driven reassignment algorithm.

Algorithm 3: NextChange

```
Input  : $\phi_{\mathrm{r}}$, reassignment        $\mathcal{P}$, a set of partitions
         $\mathcal{I}$, statistics             $\mathcal{N}$, a set of compute nodes
Output: $\mathcal{L}$: exchange
$\phi \leftarrow \emptyset$; $\mathcal{L}_i.Time \leftarrow +\infty$;
foreach $P \in \mathcal{P}$ do
    foreach $P' \in \mathcal{P} - \{P\}$ do
        $\phi_{\mathrm{r}}' \leftarrow \phi_{\mathrm{r}}$;
        Swap $\phi_{\mathrm{r}}'(P)$ and $\phi_{\mathrm{r}}'(P')$;
        $t' \leftarrow$ ComputeCost($\phi_{\mathrm{r}}', S, \mathcal{I}$);
        if $\mathcal{L}_i.Time > t'$ then
            $\mathcal{L}_i.\phi \leftarrow \{(P, \phi_{\mathrm{r}}(P')), (P', \phi_{\mathrm{r}}(P))\}$;
            $\mathcal{L}_i.Time \leftarrow t'$;
    foreach $N \in \mathcal{N} - \{\phi_{\mathrm{r}}(P)\}$ do
        $\phi_{\mathrm{r}}' \leftarrow \phi_{\mathrm{r}}$; $\phi_{\mathrm{r}}'(P) \leftarrow N$;
        $t' \leftarrow$ ComputeCost($\phi_{\mathrm{r}}', S, \mathcal{I}$);
        if $\mathcal{L}_i.Time > t'$ then
            $\mathcal{L}_i.\phi \leftarrow \{(P, N)\}$;
            $\mathcal{L}_i.Time \leftarrow t'$;
```

4.1 Estimation of T_{low}

For any failure, T_{low} is determined by the total amount of computation cost and network communication cost required during recomputation, which is formally defined as follows.

$$T_{low} = \sum_{i=c+1}^{s_{\mathrm{f}}} \left(\mathbf{T}_{\mathrm{p}}[i] + \mathbf{T}_{\mathrm{m}}[i]\right) \tag{3}$$

where $\mathbf{T}_{\mathrm{p}}[i]$ and $\mathbf{T}_{\mathrm{m}}[i]$ denote the time for vertex computation and that for inter-node message passing required in superstep i during recomputation, respectively.

Equation 3 ignores the *downtime* period for replacing failed nodes and *synchronization* time because they are almost invariant w.r.t. the recovery methods discussed in this paper. We also assume the cost of *intra-node message passing* is negligible compared with network communication cost incurred by inter-node messages.

We now focus on how to compute $\mathbf{T}_{\mathrm{p}}[i]$ and $\mathbf{T}_{\mathrm{m}}[i]$ in Equation 3. Let S_i and ϕ_{p_i} denote the state and the partition-node mapping in the beginning of superstep i (during recomputation), respectively. We find that $\mathbf{T}_{\mathrm{p}}[i]$ and $\mathbf{T}_{\mathrm{m}}[i]$ can be computed based on S_i and ϕ_{p_i}. Therefore, we first describe how to compute S_i, ϕ_{p_i}, and then define $\mathbf{T}_{\mathrm{p}}[i]$ and $\mathbf{T}_{\mathrm{m}}[i]$ based on S_i, ϕ_{p_i}.

Compute S_i, ϕ_{p_i}. For $i = c+1$, S_{c+1} is the state right after the failure occurs. Let φ be the vertex-partition mapping, ϕ_{p} be the partition-node mapping before the failure occurs, ϕ_{r} be the reassignment for the failure and $\mathcal{P}_{\mathrm{f}}$ be the set of failed partitions.

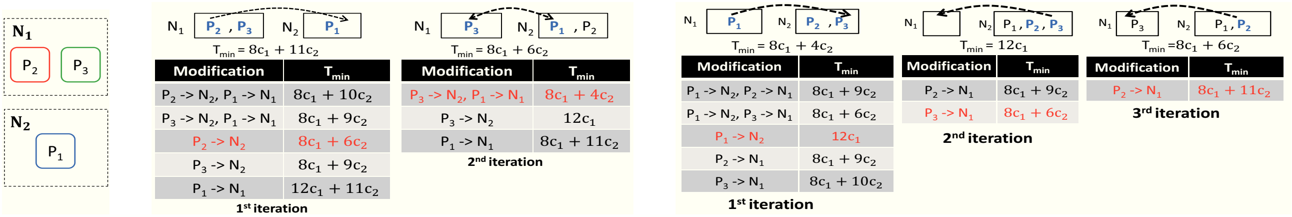
$$\phi_{\mathrm{p}_{c+1}}(v) = \begin{cases} \phi_{\mathrm{r}}(v) & \text{If } \varphi(v) \in \mathcal{P}_{\mathrm{f}} \\ \phi_{\mathrm{p}}(v) & \text{Otherwise} \end{cases} \tag{4}$$

For any $i \in (c+1, s_{\mathrm{f}}]$, we have:

$$\phi_{\mathrm{p}_i} = \phi_{\mathrm{p}_{c+1}}, \quad S_i(v) = \begin{cases} S_{c+1}(v) & \text{If } S_{c+1}(v) \geq i \\ i-1 & \text{Otherwise} \end{cases} \tag{5}$$

Compute $\mathbf{T}_{\mathrm{p}}[i]$, $\mathbf{T}_{\mathrm{m}}[i]$. We now formally define $\mathbf{T}_{\mathrm{p}}[i]$ and $\mathbf{T}_{\mathrm{m}}[i]$. According to the computation model in Section 2.1, computation time required in a superstep is determined by the *slowest* node, i.e., maximum computation time among all the nodes. Let $\mathcal{A}(i)$ be the set of vertices that perform computation during the re-execution of superstep i. Let $\tau(v,i)$ denote the computation time of v in the normal execution of superstep i.

$$\mathbf{T}_{\mathrm{p}}[i] = \max_{N \in \mathcal{N}} \sum_{\tau(v,i)} \{v \in \mathcal{A}(i) \mid \phi_{\mathrm{p}_i}(\varphi(v)) = N\} \tag{6}$$



(a) Initial

Modification	T_{min}
P_2 -> N_2, P_1 -> N_1	$8c_1 + 10c_2$
P_3 -> N_2, P_1 -> N_1	$8c_1 + 9c_2$
P_2 -> N_2	$8c_1 + 6c_2$
P_3 -> N_2	$8c_1 + 9c_2$
P_1 -> N_1	$12c_1 + 11c_2$

1st iteration

Modification	T_{min}
P_3 -> N_2, P_1 -> N_1	$8c_1 + 4c_2$
P_3 -> N_2	$12c_1$
P_1 -> N_1	$8c_1 + 11c_2$

2nd iteration

(b) Pass 1

Modification	T_{min}
P_1 -> N_2, P_2 -> N_1	$8c_1 + 9c_2$
P_1 -> N_2, P_3 -> N_1	$8c_1 + 6c_2$
P_1 -> N_2	$12c_1$
P_2 -> N_1	$8c_1 + 9c_2$
P_3 -> N_1	$8c_1 + 10c_2$

1st iteration

Modification	T_{min}
P_2 -> N_1	$8c_1 + 9c_2$
P_3 -> N_1	$8c_1 + 6c_2$

2nd iteration

Modification	T_{min}
P_2 -> N_1	$8c_1 + 11c_2$

3rd iteration

(c) Pass 2

Figure 5: Example of Modifications

Due to simplicity, we assume computations for vertices in one node are performed sequentially. A more accurate estimation for $\mathbf{T}_{\mathrm{p}}[i]$ can be applied if the computation within a node can be parallelized using machines with multithreaded and multicore CPUs.

To compute $\mathbf{T}_{\mathrm{m}}[i]$, we adopt the Hockney's model [11], which estimates network communication time by the total size of inter-node messages divided by network bandwidth. Let $\mathcal{M}(i)$ be the set of messages forwarded when re-executing superstep i. Let $m.u$, $m.v$ and $\mu(m)$ be the source vertex, destination vertex and size of message m, respectively. Suppose the network bandwidth is B.

$$\mathbf{T}_{\mathrm{m}}[i] = \sum_{\mu(m)/B} \{m \in \mathcal{M}(i) \mid \phi_{\mathrm{P}_i}(\varphi(m.u)) \neq \phi_{\mathrm{P}_i}(\varphi(m.v))\} \quad (7)$$

Note that $\mathcal{A}(i)$, $\tau(v, i)$, $\mathcal{M}(i)$ and $\mu(m)$ in Equation 6 and 7 can only be obtained during the runtime execution of the application. A *perfect knowledge* of these values requires a detailed bookkeeping of graph status in every superstep, which incurs high maintenance cost. Therefore, we refer to *statistics* (See Section 3) for approximation. Specifically, we can learn from S_i, ϕ_{P_i} whether a partition will perform computation and forward messages to another partition during the re-execution of superstep i, and based on the statistics, we know the computation cost and communication cost among these partitions in superstep c. We then approximate the costs in superstep i by those in superstep c.

Example 5. *Consider $F(\{N_1\}, 12)$ in Example 2 and ϕ_{r} in Figure 5(a). Let c_1 and c_2 be the time for each vertex computation and that for sending an inter-node message, respectively. To compute T_{low} under ϕ_{r}, we calculate the re-execution time of superstep $11, 12$ without further cascading failures. In both supersteps, computation time is $4c_1$ caused by P_1, P_2 in N_1. Communication time in superstep 11 is $5c_2$ caused by 5 inter-node messages: 1 from P_2 to P_1, 4 from P_4 to P_2, P_3, and that in superstep 12 is $6c_2$ following the 6 cross-node edges. Hence, T_{low} under ϕ_{r} is $8c_1 + 11c_2$.*

Theorem 2. *Given a failure, finding a reassignment ϕ_{r} for it that minimizes T_{low} in Equation 3 is NP-hard.*

Theorem 2 can be proven by reducing the graph partitioning problem to the problem of finding reassignment with minimized T_{low}. We omit proof due to space constraint.

4.2 Cost-Sensitive Reassignment Algorithm

Due to the hardness result in Theorem 2, we develop a cost-sensitive reassignment algorithm. Before presenting our algorithm, we shall highlight the differences between our problem and traditional graph partitioning problems. First and foremost, the traditional graph partitioning problems focus on partitioning a static graph into k components with the objective of minimizing the number of cross-component edges. In our case, we try to minimize the remaining recovery time T_{low}. T_{low} is independent of the original graph structure but relies on the vertex states and message-passing during the execution period. Second, graph partitioning outputs k components where k is predefined. On the contrary, our reassignment is required to dynamically allocate the failed partitions among the healthy nodes without the knowledge of k. Further, besides the partitioning, we must know the node to which a failed partition will be reassigned. Third, traditional partitioning always requires k components to have roughly equal size, while we allow unbalanced reassignment, i.e., assign more partitions to one node but fewer to another, if a smaller value of T_{low} can be achieved.

Algorithm 2 outlines our reassignment algorithm. We first generate a reassignment ϕ_{r} by randomly assigning partitions in $\mathcal{P}_{\mathrm{f}}$ among compute nodes $\mathcal{N}$, and then calculate T_{low} under ϕ_{r} (line 1-2). We next make a copy of ϕ_{r} as $\phi_{\mathrm{r}}{}'$ and improve $\phi_{\mathrm{r}}{}'$ iteratively (line 3-18). In the i-th iteration, the algorithm chooses some partitions and modifies their reassignments (line 7-9). The modification information is stored in $\mathcal{L}_i$. $\mathcal{L}_i$ is in the form of $(\phi, Time)$, where ϕ is a partition-node mapping recording which partition is modified to be reassigned to which node, and $Time$ is T_{low} under the modified reassignment. The selected partitions are removed for further consideration (line 10). The iteration terminates when no more failed partitions are left. After that, we check list $\mathcal{L}$ and find l such that $\mathcal{L}_l.Time$ is minimal (line 11), i.e.,

$$l \leftarrow \arg\min_i \ \mathcal{L}_i.Time$$

If $\mathcal{L}_l.Time$ is smaller than T_{low} achieved by the initial reassignment ϕ_{r}, we update ϕ_{r} by sequentially applying all the modifications in $\mathcal{L}_1, \cdots, \mathcal{L}_l$ (line 12-16), and start another pass of iterations. Otherwise, the algorithm outputs ϕ_{r} as the result.

Algorithm 3 describes how to generate modification $\mathcal{L}_i$ (line 7 in Algorithm 2) in the i-th iteration. We focus on two types of modifications: i) exchanging the reassignments between two partitions; ii) changing the reassignment for one partition. Given a reassignment ϕ_{r}, NEXTCHANGE iterates over all the partitions (line 2) and for each partition P, it enumerates all of the possible modifications, i.e., exchanging the reassignment of P with another partition (line 3-9) as well as assigning P to another node instead of $\phi_{\mathrm{r}}(P)$ (line 10-15). NEXTCHANGE computes the corresponding T_{low} achieved by each modification and chooses the one with minimized value of T_{low} as the modification $\mathcal{L}_i$.

Example 6. *Continuing from Example 5, suppose $\frac{c_1}{c_2} = 1.1$. Figure 5(a) shows the initial reassignment with $T_{low} = 8c_1 + 11c_2$. Figure 5(a) provides enumerated modifications and their T_{low} in the first pass. In iteration 1, assigning P_2 to N_2 achieves minimum T_{low}: $8c_1 + 6c_2$. In iteration 2, we only consider modifications for P_1, P_3 as P_1 has been considered. Exchanging reassignments for P_1, P_3 produces T_{low} of $8c_1 + 4c_2$. After that, all the partitions have been considered. We apply the first two modifications to the initial reassignment because the minimal T_{low} (i.e., $8c_1 + 4c_2$) is achieved after the second modification.*

Figure 5(c) shows enumerated modifications and their T_{low} in pass 2. The minimal T_{low} (i.e., $12c_1$) in three iterations is achieved after the first modification, which is larger than $8c_1 + 4c_2$. Hence,

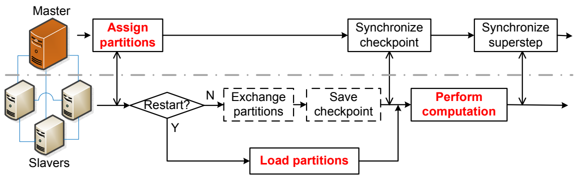


Figure 6: Processing a Superstep in Giraph

the algorithm terminates and reports the reassignment produced by pass 1*, i.e., assigning P_1 to N_1 and P_2, P_3 to N_2.*

5. IMPLEMENTATION

We implement our partition-based failure recovery method on Apache Giraph [1], an open-source implementation of Pregel. It is worth mentioning that our proposed recovery method can be integrated to other distributed graph processing platforms such as Hama [2], in a similar way.

Giraph overview. Giraph distributes a graph processing job to a set of workers. One worker is selected as the *master* that coordinates the other *slave* workers, which perform vertex computations. One of the slaves acts as *zookeeper* to maintain various statuses shared among the master and slaves, e.g., notifying slaves of partitions assigned by the master, doing synchronization after accomplishing a superstep. Figure 6 shows the processing logic of workers in one superstep. Initially, the master generates partition assignment indicating which partition is processed by which slave, and writes the partition-to-slave mapping into zookeeper. Slaves fetch the mapping from zookeeper and exchange partitions along with their receiving messages based on the mapping. They then check whether the current superstep is a checkpointing superstep. If so, each slave saves the status of its partitions to a stable storage. After that, every slave performs computation for the vertices residing in it, sends messages and collects messages sent to its vertices. Finally, the master synchronizes the completion of the superstep.

Failure recovery. Node failures are detected by the master at the end of each superstep, before synchronization. The master checks the healthy status registered periodically by every slave and considers a slave as failed if it has not registered its status over a specified interval. Giraph adopts checkpoint-based recovery mechanism. We refer to the first superstep performed upon a failure as *restart* superstep. In the restart superstep, after the master generates the recovery plan and writes it to the zookeeper, slaves will load failed partitions that are assigned to them from the latest checkpoint and start recomputation. Recovery details are omitted to avoid redundancy.

Major APIs. To support partition-based failure recovery, we introduce several APIs to Giraph, as shown in Figure 7. We utilize `PartitionOwner` class to maintain ownership of each partition. `setRestartSuperstep()` sets the next superstep when a partition needs to perform computation; `setWorkerInfo()` and `setPreviousWorkerInfo()` set information (e.g., IP address) for current and previous slaves in which a partition resides, respectively. To shuffle a partition from slave 1 to slave 2, we can simply set the previous, current workers to slave 1 and 2, respectively; the workers can retrieve this information via the three interfaces: `getRestartSuperstep()`, `getPreviousWorkerInfo()` and `getWorkerInfo()`. To generate the ownership of every partition, we introduce a new class `FailureMasterPartitioner`. This class will be initialized in the beginning of each superstep, with two major functions: `createInitialPartitionOwners()` generates reassignment for newly failed partitions and retains original ownership for healthy ones. `genChangedPartitionOwners()` is applied to exchange failed partitions after recovery finishes.

```
PartitionOwner()  //metadata about ownership of a partition
void setRestartSuperstep(long superstep)
long getRestartSuperstep()
void setPreviousWorkerInfo(WorkerInfo workerInfo)
void getPreviousWorkerInfo()
void setWorkerInfo(WorkerInfo workerInfo)
void getWorkerInfo(WorkerInfo workerInfo)

FailureMasterPartitioner<I,V,E,M>  //generate partition assignment
Collection<PartitionOwner> createInitialPartitionOwners
    (Collection<WorkerInfo>, int max)  //for restart
Collection<PartitionOwner> genChangedPartitionOwners
     (Collection<PartitionStats>, Collection<WorkerInfo>,
      int max, long superstep)

FailureMasterPartitioning  //generate reassignment for failed partitions
void doCostSensitivePartitioning();
```

Figure 7: Major APIs

Our extensions. As illustration, we consider a failure (can be a cascading failure) that occurs in executing superstep s_f and latest checkpointing superstep is $c + 1$. We extend Giraph mainly in the following three aspects.

Partition assignment. This is performed by the master in the beginning of each superstep.

(1) During superstep 1 or the restart superstep, the master invokes `createInitialPartitionOwners()` to generate a partition assignment and set the current worker for each partition accordingly. In superstep 1, we set the previous worker for a partition to be the same as its current worker and the restart superstep for each partition to 1. In the restart superstep, we set the previous worker for each partition to be the one before failure occurs. For newly failed partitions, we set $c + 1$ as their restart supersteps; for the other partitions, their restart supersteps are set to be one after the last superstep in which their computation are performed.

(2) In the other supersteps, `genChangedPartitionOwners()` is invoked by the master to dynamically reassign partitions among the slaves. This is achieved by setting the previous worker of a partition as its current one and modifying its current worker to the new one.

Loading partitions. After the master computes the partition assignment, it writes the partition-to-slave mapping to the zookeeper. Since all slaves are listening to the changes of this mapping information, every slave can fetch and parse this mapping and then load the corresponding failed partitions from the latest checkpoint if necessary. Note that in the checkpoint, partitions residing in the same slave are stored in the same file named with the slave host name, and within each file, there is a pointer to indicate which offset a partition starts. In this way, a slave can quickly load a partition using this implicit two-level index.

Performing computation. For recomputation, every slave invokes the function `processGraphPartitions()` to execute the vertex compute function, and invokes `sendMessageRequest()` to forward messages. During recovery, we adjust these two functions to avoid unnecessary computation and communication, as follows.

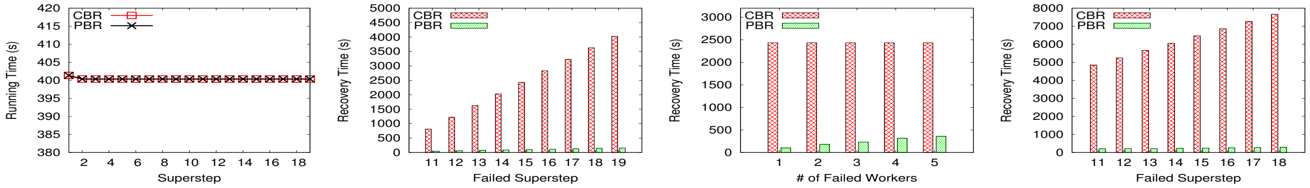
(1) `processGraphPartitions()` iterates over the partitions and check whether `PartitionOwner.getRestartSuperstep()` is less than the current superstep. If so, the slave loops over all the vertices residing in the partition and perform computation by invoking `Vertex.Compute()`;

(2) During the computation from superstep $c{+}1$ to $s_\text{f}{-}1$, a message is omitted if it is sent to a vertex residing in the partition whose restart superstep is less than the current superstep;

(3) At the end of each superstep, every slave loads its locally logged messages. For supersteps in $[c + 1, s_\text{f} - 1]$, only messages to the partitions whose restart supersteps are less than the current superstep are forwarded. For superstep s_f, all the messages are sent via `sendMessageRequest()` to the corresponding slaves.

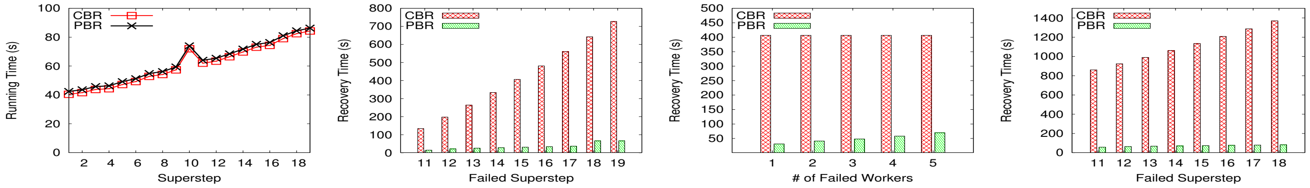
6. EXPERIMENTAL STUDIES

We compare our proposed recovery method with the checkpoint-based method on top of Giraph graph processing engine. We use



(a) Logging Overhead (b) Single Node Failure (c) Multiple Node Failure (d) Cascading Failure

Figure 8: k-means



(a) Logging Overhead (b) Single Node Failure (c) Multiple Node Failure (d) Cascading Failure

Figure 9: Semi-clustering

the latest version-1.0.0 of Giraph that is available in [1].

6.1 Experiment Setup

The experimental study was conducted on our in-house cluster. The cluster consists of 72 compute nodes, each of which is equipped with one Intel X3430 2.4GHz processor, 8GB of memory, two 500GB SATA hard disks. All the nodes are hosted on two racks. The nodes within one rack are connected via 1 Gbps switch and the two racks are connected via a 10 Gbps cluster switch. On each compute node, we installed CentOS 5.5 operating system, Java 1.6.0 with a 64-bit server VM and Hadoop 0.20.203.0[3]. Giraph runs as a MapReduce job on top of Hadoop, hence we made the following changes to the default Hadoop configurations: (1) the size of virtual memory for each task was set to 4GB; (2) each node was configured to run one map task. By default, we chose 42 nodes out of the 72 nodes for the experiments and among them, one node acted as the master running Hadoop's NameNode and JobTracker daemons while the other 41 nodes ran TaskTracker daemons and Giraph jobs. Among the 41 nodes that ran Giraph jobs, one node acted as the master and zookeeper, and the others were slaves.

6.2 Benchmark Tasks and Datasets

We study the failure recovery over three benchmark tasks: k-means, semi-clustering [20] and PageRank.

- **k-means.** We implement k-means in Giraph[4]. In our experiments, we set $k = 100$.
- **Semi-clustering.** A semi-cluster in a social graph consists of a group of people who interact frequently with each other and less frequently with others. We port the implementation in Hama [2] into Giraph. We use the same parameter values as in Hama, i.e., each cluster contains at most 100 vertices, a vertex is involved in at most 10 clusters, and the boundary edge score factor is set to 0.2.
- **PageRank.** The PageRank algorithm is contained in the Giraph package and we simply use it to run PageRank tasks.

Without loss of generality, we run all the tasks for 20 supersteps, and perform a checkpoint at the beginning of superstep 11. For all experiments, the results are averaged over ten runs.

We evaluate benchmark tasks over one vector dataset and two real-life social network graphs (Table 2 provides dataset details and the two graph datasets are downloaded from the website[5]).

Table 2: Dataset Description

Dataset	Data Size	#Vertices	#Edges	#Partitions
Forest	2.7G	58,101,200	0	160
LiveJournal	1.0G	3,997,962	34,681,189	160
Friendster	31.16G	65,608,366	1,806,067,135	160

- **Forest.** Forest dataset[6] predicts forest cover type from cartographic variables. It originally contains 580K objects, each of which is associated with 10 integer attributes. To evaluate the performance on large datasets, we increase the size of Forest to 58,101,200 while maintaining the same distribution of values over each dimension using the data generator from [19]. We use this dataset to evaluate the execution of k-means tasks.
- **LiveJournal.** LiveJournal is an online social networking and journaling service that enables users to post blogs, journals, and dairies. It contains more than 4 million vertices (users) and about 70 million directed edges (friendships between users). We use this dataset to evaluate the execution of semi-clustering tasks.
- **Friendster.** Friendster is an online social networking and gaming service. It contains more than 60 millions vertices and 1 billion edges. We use it to evaluate the execution of PageRank tasks.

We compare our proposed partition-based recovery method (PBR) with the checkpoint-based recovery method (CBR) over two **metrics**: recovery time and communication cost.

6.3 k-means

We first study the overhead of logging outgoing messages at the end of each superstep in PBR. Figure 8(a) shows the running time. PBR takes almost the same time as CBR. The reason is that in k-means tasks, there does not exist any outgoing messages among different vertices, and in this case, PBR performs exactly the same as CBR during normal execution. Another interesting observation is that the checkpointing superstep 10 does not incur higher running time compared with other supersteps. This is because compared with computing the new belonging cluster for each observation, the time of doing checkpointing is negligible.

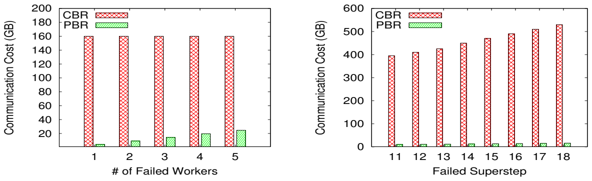
We then evaluate the performance of recovery methods for single node failures by varying the failed superstep from 11 to 19. Figure 8(b) plots the results. The recovery time of both CBR and PBR increases linearly when the failed superstep varies. Since there are no messages passing among different workers, computing the new belonging clusters for failed partitions can be accelerated by using all

[3] http://hadoop.apache.org/

[4] https://github.com/tmalaska/Giraph.KMeans.Example/

[5] http://snap.stanford.edu/data/index.html

[6] http://archive.ics.uci.edu/ml/datasets/Covertype



(a) Multiple Node Failure

(b) Cascading Failure

Figure 10: Communication Cost of Semi-clustering

available workers, i.e., recomputation is parallelized over 40 workers for recovery. We find that PBR outperforms CBR by a factor of 12.4 to 25.7 and there is an obvious gain when the failed superstep increases. The speedup is less than 40x due to the overhead of loading the checkpoint in the beginning of a recovery.

Next, we investigate the performance of recovery methods for multiple node failures. The number of failed nodes is varied from 1 to 5 and the failed superstep is set to 15. Figure 8(c) plots the results. When the number of failed nodes increases, the recovery time increases linearly for PBR while that remains constant for CBR. No matter how many nodes fail, CBR will redo all computation from the latest checkpoint, while PBR recomputes the new belonging clusters for observations in the failed nodes and hence the recovery time becomes longer for a larger number of failed nodes. On average, PBR outperforms CBR by a factor of 6.8 to 23.9.

Finally, we focus on cascading failure by setting the first failed superstep to 19 and varying the second failed superstep from 11 to 18. Figure 8(d) plots the results. When the second failed superstep increases, the recovery time increases linearly for both CBR and PBR. On average, PBR can reduce recovery time by a factor of 23.8 to 26.8 compared with CBR.

6.4 Semi-clustering

Figure 9(a) plots the running time of each superstep for semi-clustering. PBR takes slightly longer time than CBR during normal execution. This is because compared with the computation and communication costs, the overhead of logging outgoing messages to local disks is relatively insignificant. Moreover, in semi-clustering, the size of each message from a vertex to its neighbors increases linearly with the superstep. Hence, both CBR and PBR runs slower in larger supersteps. In superstep 10, there is an obvious increment in the running time due to performing a checkpoint.

We evaluate the performance of recovery methods for single node failures, multiple node failures and cascading failures using the same settings as k-means. Figure 9(b), 9(c), 9(d) show the respective results. Basically, the trends of the running time of PBR and CBR in semi-clustering follows those in k-means. Specifically, PBR outperforms CBR by a factor of 9.0 to 15.3 for single node failures, by a factor of 13.1 to 5.8 for multiple node failures, and by a factor of 14.3 to 16.6 for cascading failures.

Besides the benefit of parallelizing computation, we also show the communication cost incurred by PBR and CBR in Figure 10. Since messages sent to the vertices residing in the healthy nodes can be omitted in PBR, we observe that in multiple node failure, PBR incurs 6.5 to 37.9 times less communication cost than CBR. For cascading failures, PBR can reduce communication cost by a factor of 37.1 compared with CBR.

6.5 PageRank

To study the logging overhead for PageRank tasks, we report the running time of every superstep in Figure 11(a). Compared with k-means and semi-clustering, PBR takes slightly more time than CBR in PageRank. This is because PageRank is evaluated over the Friendster dataset which has a power-law link distribution and each superstep involves a huge number of forwarding messages that should be logged locally via disk I/O. However, the overhead is still negligible, only 3% increase in running time. Due to doing checkpointing, there is an obvious increment of running time in superstep 10. In each superstep, the worker basically does the same task and hence the running time of each superstep almost remains the same. We also evaluate the performance of recovery methods for single node failures, multiple node failures and cascading failures. Figure 11(b), 11(c), 11(d) provide the recovery time, respectively. Figure 12 shows the corresponding communication cost. The performance of PBR and CBR follow the same trends as those in semi-clustering and k-means tasks. This further verifies the effectiveness of PBR, which parallelizes computation and eliminates unnecessary computation and communication cost.

Table 3: Parameter Ranges for Simulation Study

Parameter	Description	Range
n	number of failed partitions	$20, \underline{40}, 50$
m	number of healthy nodes	$20, \underline{40}, 50$
k	number of partitions (or healthy nodes) with high communication cost	$\underline{2}, 4, 8$
γ	comp-comm-ratio	$0.1, \underline{1}, 10$

6.6 Simulation Study

We perform a simulation study to evaluate the effectiveness and efficiency of our cost-sensitive reassignment algorithm CostSen in partition-based recovery. As a comparison, we consider a random approach Random by balancing computation among the nodes.

Data preparation. We investigate the effect of the following parameters that potentially affect the performance of the reassignment algorithms:

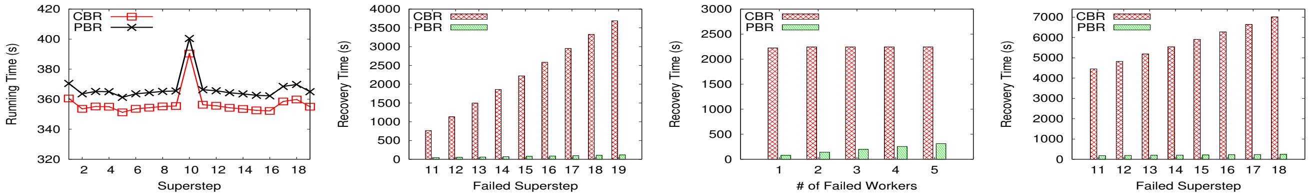
- n: the number of failed partitions
- m: the number of healthy compute nodes
- computation cost per failed partition during recovery
- communication cost between every two failed partitions during recovery
- communication cost between failed partitions and healthy compute nodes during recovery

We generate communication cost by simulating two categories of graph partitioning, *random-partitioning* and *well-partitioning*. In *random-partitioning*, there is no obvious difference in the connections of two partitions lying in the same node or across two nodes; in *well-partitioning*, the number of edges connecting two partitions within the same node is much larger than that across two nodes. For simulation, we generate communication cost using two distributions *uniformly-distributed* and *well-distributed* corresponding to *random-partitioning* and *well-partitioning*, as follows.

1) In *uniformly-distributed*, the communication cost between two failed partitions and that between a healthy node and a failed partition, is uniformly drawn from the range $[1, low]$.

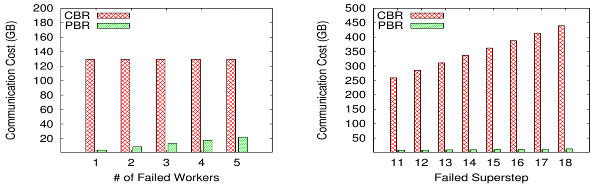
2) In *well-distributed*, for each failed partition, we randomly select k failed partitions. The communication cost from the partition to each of the selected ones is uniformly drawn from range $[1, high]$, and that from the partition to any other failed partition is uniformly drawn from range $[1, low]$. The communication cost between a partition and healthy node is generated in the same way. By default, we set p, low, $high$ to 0.6, 100, 40000, respectively.

We generate comparable computation cost for each failed partition based a *comp-comm-ratio*, γ. Let S_P be the total communication cost from healthy nodes to a failed partition P. We use γ to adjust the ratio between the computation cost of P and S_P. The computation cost of P is randomly drawn from the range $[1, \gamma S_P]$. A larger γ implies that the job is more computation-intensive. Table 3 summarizes the ranges of our tuning parameters. Unless otherwise specified, we use the underlined default values.



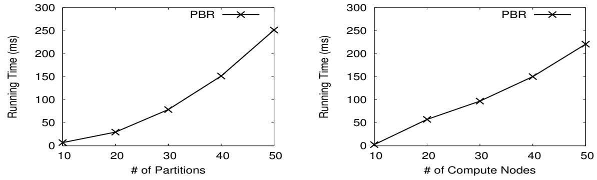
(a) Logging Overhead (b) Single Node Failure (c) Multiple Node Failure (d) Cascading Failure

Figure 11: PageRank



(a) Multiple Node Failure (b) Cascading Failure

Figure 12: Communication Cost of PageRank



(a) # of Healthy Nodes (b) # of Healthy Nodes

Figure 13: Running Time (well-distributed)

Table 4: Effects of Comp-comm-ratio γ (uniformly-distributed)

γ	0.1		1		10	
	Random	CostSen	Random	CostSen	Random	CostSen
CompCost	0.4	8.2	3.9	5.8	38.9	38.9
CommCost	152.6	75.2	152.4	143.5	152.6	147.1
TotalCost	153.0	83.4	156.3	149.3	191.5	186.0
Used nodes	40	1	40	19.9	40	24.1

Table 5: Effects of the Number of Partitions (or Healthy Nodes) with High Communication Cost k (well-distributed)

k	2		4		8	
	Random	CostSen	Random	CostSen	Random	CostSen
CompCost	3.9	17.4	3.9	46.0	3.8	76.7
CommCost	1603.5	690.5	2830.1	1422.5	5190.9	2558.8
TotalCost	1607.4	707.9	2834.0	1468.5	5194.7	2635.5
Used nodes	40	11.75	40	7.63	40	2.79

Measure. We measure the performance of reassignment algorithms via five metrics: maximum computation cost (CompCost), total inter-node communication cost (CommCost), sum of CompCost and CommCost (TotalCost), running time and the number of nodes to which failed partitions are reassigned. All the costs are measured in *seconds* by default.

Effects of comp-comm-ratio. Table 4 shows the results of CostSen and Random by varying γ in *uniformly-distributed* scenario. On average, CostSen produces reassignments with lower TotalCost and CommCost than Random over all the ratios. For $\gamma = 0.1$, CostSen outperforms Random with 2x lower TotalCost and CommCost. As γ increases, the advantage of CostSen in TotalCost and CommCost becomes less significant. The reason is that a larger γ makes the job more computation-intensive; this requires more nodes to parallelize the computation, while CommCost can hardly be reduced due to the uniform distribution. For smaller γ (e.g., 0.1), CostSen assigns failed partitions to a small number of nodes (< 5) due to insignificant CompCost, hence it reports reassignments with higher CompCost than Random. For larger γ (e.g., 10), CostSen performs similarly as Random in terms of the three costs, but it requires 2x fewer nodes for recovery. This saving is desirable in practice. We observe similar results for the *well-distributed* scenario and omit the results to avoid duplication.

Effects of High Communication Partition (Healthy Node) Number. Table 5 shows the results of both methods when we vary the number of partitions (nodes) with high communication cost (k). For all values of k, CostSen outperforms Random with 2x lower TotalCost and CommCost. CostSen produces reassignments with higher CompCost which is relatively insignificant compared with CommCost. Furthermore, CostSen always involves fewer nodes for recovery. For $k = 8$, it uses 14x fewer nodes than Random.

Effects of the number of failed partitions. Table 6 provides the results by varying the number of failed partitions (n). For each n, CostSen outperforms Random by 2.5x lower TotalCost and CommCost. Again, the reassignments reported by CostSen require higher CompCost, which is much smaller than CommCost. Furthermore, CostSen uses 3x fewer nodes for recovery. Figure 13(a) shows the running time of CostSen. It requires less than 250ms to generate reassignments. The running time increases quadratically with the number of failed partitions.

Effects of the number of healthy nodes. Table 7 provides the results by varying the number of healthy nodes (m). CostSen produces reassignments with 3x lower TotalCost and CommCost over all values of m. Furthermore, it employs fewer healthy nodes for recovery. For larger m (e.g., $40, 50$), the number of nodes involved in the reassignments from CostSen is 3x fewer than Random. Figure 13(b) shows the running time of CostSen. The running time increases linearly with the number of healthy nodes. For $m = 50$, CostSen generates reassignments over 40 nodes within 250ms.

Summary. Our simulation study show that the cost-sensitive reassigning algorithm achieves 2x speedup in our partition-based recovery framework compared with random assignment. It also outperforms random approach by wisely choosing a smaller number of compute nodes to handle failed partitions. Furthermore, our reassigning algorithm is efficient; the running time grows quadratically with the number of failed partitions and linearly with the number of healthy nodes.

7. RELATED WORK

Designing efficient failure recovery methods has long been a goal of distributed systems. As our proposed approach accelerates the failure recovery by parallelizing the re-execution of failed graph partitions, we split the discussion on the existing work on failure recovery in distributed graph processing systems into two categories: (i) methods for accelerating the failure recovery and (ii) graph partitioning methods.

Accelerating failure recovery. Failure recovery approaches are typically split into three categories: *checkpoint-based*, *log-based* and *hybrid* approaches [9]. Most popular distributed graph processing systems such as Giraph [1], GraphLab [18], PowerGraph [10], GPS [22], Mizan [14] adopt checkpoint-based recovery. Pregel [20] proposes confined recovery which is a hybrid mechanism of the checkpoint-based and log-based recovery. Specifically, only the

Table 6: Effects of the Number of Partitions n (well-distributed)

n	20		40		50	
	Random	CostSen	Random	CostSen	Random	CostSen
CompCost	3.7	14.7	3.8	17.8	4.6	18.8
CommCost	775.3	291.9	1605.2	689.0	2005.0	876.9
TotalCost	779.0	306.6	1609.0	706.8	2009.6	895.7
Used nodes	20	7.05	40	11.69	40	15.19

Table 7: Effects of the Number of Healthy Nodes m (well-distributed)

m	20		40		50	
	Random	CostSen	Random	CostSen	Random	CostSen
CompCost	5.9	20.5	3.9	15.6	3.8	16.5
CommCost	1463.3	572.1	1582.0	686.8	1617.9	695.8
TotalCost	1469.2	592.6	1585.9	702.4	1621.7	712.3
Used nodes	20	10.21	40	12.27	50	12.97

newly-added node that substitutes the failed one has to rollback and repeats the computations from the latest checkpoint. GraphX [26] adopts log (called lineage) based recovery, and utilizes resilient distributed datasets (RDD) to speedup failure recovery. However, when a node fails, graph data lying in this node still need to be recovered. Checkpointing and logging operations are the backbones of recovery methods [5, 6, 7]. Many works focused on accelerating checkpoint-based or the log-based recovery [9]. Location and replication independent recovery proposed by Bratsberg et al. employed replicas for recovery [8]. The algorithm partitions the data into fragments and replicates fragments among multiple nodes which can takeover in parallel upon failures. However, the recovery task for the failed node is still performed in a centralized manner after the node finishes internal recovery. Instead, we focus on accelerating the task of recovering by parallelizing the computations required to recompute the lost data. Another recovery method that presents similarities with ours is present in RAMCloud [21]. RAMCloud backs up the data across many distributed nodes, and during recovery, it reconstructs in parallel the lost data. However, as RAMCloud is a distributed storage, it does not need to track the dependencies among the scattered data. In contrast, in distributed processing systems, understanding how the program dependencies affect both the communication and the computation time is of utmost importance [4].

Graph partitioning. METIS [13], which performs offline partitioning of a distributed unstructured graph, is most relevant to our approach for partitioning the failed subgraph. Several extensions have been proposed such as for power-law graphs [3], multi-threaded graph partitioning [15] and dynamic multi-constraint graph partitioning [23]. In practice, it has been adopted in other distributed graph processing systems such as PowerGraph. However, METIS does not partition based on a cost model that includes both communication and computation.

8. CONCLUSION

This paper presents a novel partition-based recovery method to parallelize failure recovery processing. Different from traditional checkpoint-based recovery, our recovery method distributes the recovery tasks to multiple compute nodes such that the recovery processing can be executed concurrently. Because partition-based failure recovery problem is NP-Hard, we use a communication and computation cost model to split the recovery among the compute nodes. We implement our recovery method on the widely used Giraph system and observe that our proposed parallel failure recovery method outperforms existing checkpoint-based recovery methods by up to 30 times when using 40 compute nodes.

Acknowledgments. This work was supported by the National Research Foundation, Prime Minister's Office, Singapore under Grant No. NRF-CRP8-2011-08. We thank Xuan Liu for his contributions on the earlier work of this paper. We would also like to thank anonymous reviewers for their insightful comments and suggestions, which helped improve the quality of our paper.

9. REFERENCES

[1] `http://giraph.apache.org/.`

[2] `https://hama.apache.org/.`

[3] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDPS*, pages 124–124, 2006.

[4] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *ICDE*, pages 52–63, 2014.

[5] J. F. Bartlett. A nonstop kernel. In *SOSP*, pages 22–19, 1981.

[6] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *SOSP*, pages 90–99, 1983.

[7] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under unix. In *ACM Trans. Comput. Syst.*, pages 1–24, 1989.

[8] S. E. Bratsberg, S.-O. Hvasshovd, and O. Torbjørnsen. Location and replication independent recovery in a highly available database. In *BNCOD*, pages 23–37, 1997.

[9] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3), 2002.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[11] R. W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. In *Parallel Computing*, pages 389–398, 1994.

[12] D. Jiang, G. Chen, B. C. Ooi, K.-L. Tan, and S. Wu. epic: an extensible and scalable system for processing big data. In *PVLDB*, pages 541–552, 2014.

[13] G. Karypis and V. Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[14] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, pages 169–182, 2013.

[15] D. LaSalle and G. Karypis. Multi-threaded graph partitioning. In *IPDPS*, pages 225–236, 2013.

[16] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using mapreduce. *ACM Comput. Surv.*, 46(3):31:1–31:42, 2014.

[17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, pages 340–349, 2010.

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. In *PVLDB*, pages 716–727, 2012.

[19] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. In *PVLDB*, pages 1016–1027, 2012.

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

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

[22] S. Salihoglu and J. Widom. Gps: a graph processing system. In *SSDBM*, pages 22:1–22:12, 2013.

[23] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. In *CCPE*, pages 219–240, 2002.

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

[25] L. G. Valiant. A bridging model for parallel computation. In *Commun. ACM*, pages 103–111, 1990.

[26] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: a resilient distributed graph system on spark. In *GRADES*, pages 2:1–2:6, 2013.