Fast Failure Recovery in Vertex-centric Distributed Graph Processing Systems

Wei Lu, Yanyan Shen,Tongtong Wang, Meihui Zhang, H. V. Jagadish, Xiaoyong Du

Abstract—There is a growing need for distributed graph processing systems to have many more compute nodes processing graph-based Big Data applications. However, increasing the number of compute nodes inevitably raises the chance of node failures. Therefore, provisioning an efficient failure recovery scheme is critical for distributed graph processing systems. Fueled by this need, we propose a novel recovery scheme that aims to accelerate the recovery process by parallelizing the recomputation. In this recovery scheme, once a failure occurs, on one hand, all recomputations to recover the failure are confined to subgraphs originally residing in the failed compute nodes $\bar{S}$; on the other hand, when the recovery starts, these subgraphs are reassigned to another set S of compute nodes, and the recomputations over these subgraphs are parallelized using the compute nodes in S. While the computation cost can be minimized when S includes all the compute nodes in the cluster, the communication cost may increase as a side effect. For this reason, to minimize the recovery latency, we then develop a reassignment strategy, from these subgraphs to the replaced compute nodes, by properly leveraging the computation and communication cost. We elaborate the integration of our recovery scheme into Giraph system, a widely used graph processing system. The experimental results over a variety of real graph datasets demonstrate that our proposed recovery scheme outperforms existing recovery methods by up to 30x on a cluster of 40 compute nodes.

Index Terms—Distributed Graph Processing Systems, Failure Recovery, Checkpoint, Log, Compression, Partition-based Recovery

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. In recent years, MapReduce has been shown to be ineffective for handling graph data, and several new systems such as Pregel [21], Pregel++ [27], Pregelix [10], Giraph [1], GraphLab [12], [19], and Trinity [25] 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 workload. But scaling the number of nodes results in 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 the failures effectively and efficiently.

- *Wei Lu, Tongtong Wang and Xiaoyong Du are with DEKE, MOE and School of Information, Renmin University of China.*
  E-mail: {lu-wei,wangtongtong,duyong}@ruc.edu.cn
- *Yanyan Shen is with Shanghai Jiaotong University, China.*
  E-mail: shenyy@sjtu.edu.cn
- *Meihui Zhang is with Singapore University of Technology and Design, Singapore.*
  E-mail:meihui_zhang@sutd.edu.sg
- *H. V. Jagadish is with University of Michigan, US*
  E-mail:jag@umich.edu

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. An effective 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 legacy recovery method in current distributed graph processing systems follows the *checkpoint-restart* strategy [18], [21], [29]. It requires each compute node to periodically and synchronously create a checkpoint by flushing the current status of its own subgraphs to a persistent storage such as the Hadoop distributed file system (HDFS). Upon any failure, an (in-use or unused) healthy compute node is employed to replace the failed node. All the compute nodes reload their subgraph statuses from the most recent checkpoint and redo all the computations. A failure is recovered when all the nodes finish the computations that have been completed before the failure occurs.

Although checkpoint-restart recovery is able to handle any node failures, it potentially suffers from high recovery latency. The reason is two-fold. First, checkpoint-restart recovery re-executes the missing computations over the whole graph, originally 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 recovery, the lost computation caused by the previous failure may have been partially recovered. However, checkpoint-restart recovery will forget about all of this partially completed computation, rollback every compute node to the most recent checkpoint and replay the computation since then. This eliminates the possibility of performing recovery progressively.

To enable fast failure recovery, in this paper, we augment the checkpoint-restart recovery with logging mechanism and propose a new recovery scheme that is encapsulated with the following three features to scale out the performance.

• **Checkpoint+Log**. The key idea of our approach is to confine the recomputation to the subgraphs originally residing in failed nodes. Note that graph computation is conducted iteratively. During an iteration, every compute node examines its residing vertices and performs vertex-centric computation sequentially. The computation over a vertex in an iteration typically takes the computed vertex value and receiving messages from the last iteration as input. Hence, to recover failed vertices, it is imperative to request all vertices to resend messages to the failed ones when recomputing the lost iterations. For this purpose, in addition to global checkpointing, we require every compute node to locally log their outgoing messages at the end of each iteration during execution. By augmenting checkpoint-restart recovery with logging, we are able to show that graph recomputation is confined to the failed vertices, and healthy vertices are responsible for resending logged messages without recomputation. We also provide a column-wise message compression method to reduce the logging overhead.

• **Parallel recovery**. When the recovery starts, we redistribute the subgraphs originally residing in the failed nodes over a subset S of compute nodes to parallelize the recovery process. Typically, the size of S is set to be much larger than the number of failed nodes to scale out the performance. While the computation cost can be minimized for S covering all the available nodes, the communication cost may increase as a side effect. We favor a good repartitioning method over failed subgraphs that reduces the overall recovery time by taking both computation cost and communication cost into consideration.

• **Optimization**. We formally quantify the recovery time under a given repartitioning by measuring its computation and communication costs. Our goal is to find a repartitioning for failed subgraphs with minimized recovery time. We prove the complexity of the problem is NP-complete and develop a heuristic algorithm to tackle this problem practically.

We conduct extensive experiments on synthetic and real-life datasets, showing our proposed recovery method outperforms traditional checkpoint-restart 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.

In contrast with traditional checkpoint-restart recovery, our approach eliminates high recomputation cost for the subgraphs residing in the healthy nodes. Furthermore, we distribute the recomputation tasks for the subgraphs originally in the failed nodes over multiple compute nodes to achieve better parallelism. Thus, our approach is not a replacement for checkpoint-restart recovery. Instead, it complements the existing method by accelerating the recovery process via simultaneous reduction on both recovery communication cost and computation cost. Several strategies are introduced to reduce the logging overhead and ensure the correctness of our recovery method when performing graph repartitioning at runtime.

A preliminary version of this work was published in [26]. Here in this paper, (1) we formally define the correctness of a failure recovery scheme, and propose a theorem that guides the design of the scheme by maintaining complete outgoing messages from every vertex in each iteration. We prove that our preliminary version strictly follows the correctness theorem; (2) To reduce potentially expensive I/O and communication cost caused by the logging scheme in [26], we propose a column-wise message compression scheme and a lazy decompression technique to avoid unnecessary message decompression during the recovery execution. We experimentally show the log compression and decompression scheme can reduce the recovery latency and communication cost by a factor up to 25% and 2.4X, respectively.

2 PRELIMINARIES

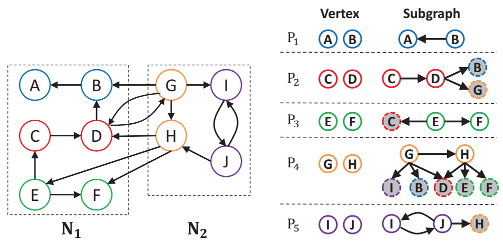
This section provides background and our problem definition.

2.1 Distributed Graph Processing Systems (DGPS)

• **Graph model.** The input to the DGPS is a labeled, directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{L}, \mathcal{F})$, where (1) $\mathcal{V}$ is the set of vertices, (2) $\mathcal{E}$ is the set of edges, (3) $\mathcal{L}$ is the set of labels in forms of strings or numerical values, and (4) $\mathcal{F}$ is a labeling function that maps every vertex/edge to a label in $\mathcal{L}$. For an edge $\langle v_i, v_j \rangle \in \mathcal{E}$, v_i and v_j are referred to as source and target vertices, respectively; we say v_j is a *direct successor* of v_i, and v_i is a *direct predecessor* of v_j. In DGPS, labels of vertices and edges are utilized in the computation logic specified by users. As our recovery scheme only relies on the computation model of DGPS rather than any computation logic, we simplify our graph model to be $\mathcal{G}(\mathcal{V}, \mathcal{E})$ when the context is clear.

• **Graph partitioning.** In DGPS, the set of vertices is divided into *partitions*. Each partition of $\mathcal{G}$ is represented as $P(V, E)$, where $P.V \subseteq \mathcal{V}$ and $P.E = \{\langle v_i, v_j \rangle \in \mathcal{E} | v_i \in P.V\}$. That is, $P.E$ includes all the edges with source vertices in $P.V$. For simplicity, we use P and $P(V, E)$ interchangeably. All partitions are distributed among compute nodes. Let $\mathcal{P}$ and $\mathcal{N}$ respectively be the set of partitions for $\mathcal{G}$ and the set of compute nodes in the cluster. Typically, the number of partitions is set to be much larger than that of compute nodes (i.e., $|\mathcal{P}| \gg |\mathcal{N}|$), to achieve a better workload balance. Note that partitions can be dynamically redistributed across compute nodes during the graph computation.

We also denote by $\varphi, \phi_{\mathrm{p}}$ two mappings, where (1) vertex-to-partition mapping $\varphi : \mathcal{V} \rightarrow \mathcal{P}$ records the belonging partition of each vertex; (2) partition-to-node mapping $\phi_{\mathrm{p}} : \mathcal{P} \rightarrow \mathcal{N}$ records the residing compute node of each partition. Figure 1(a) shows a distributed graph $\mathcal{G}$ over two nodes N_1, N_2.



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

Fig. 1. Distributed Graph and Partitions

$\mathcal{G}$ is divided into 5 partitions P_1-P_5, as shown in Figure 1(b). We use colors to differentiate vertices in different partitions.

• **Basic architecture.** Pregel-like DGPS follows a master/slave architecture. The master is responsible for coordinating the slaves, but typically does not manage 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 [21].

• **Computation model.** The computation model in Pregel-like DGPS follows the Bulk Synchronous Parallel (BSP) model [28]. Typically, the computation consists of (1) an *input* phase, which splits graph into partitions and distributes them across compute nodes, (2) followed by a set of iterations, called *supersteps*, each of which is separated by a global synchronization point, and (3) finally an *output* phase, where values of vertices are typically taken as the output.

Every vertex carries two statuses: *active* and *inactive*. All vertices are active at the beginning of superstep 1. 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 other vertices. In each superstep, only active vertices participate in computation: process messages sent by other vertices in the last superstep, update its value or its outgoing edges' values and send messages to other vertices (to be processed in the next superstep). Such computation logic is expressed by a user-defined `compute` function. All active vertices in the same compute node execute the function sequentially, while the executions in different compute nodes are performed in parallel. After all the active vertices finish their computation in a superstep, a global synchronization point is reached.

2.2 Problem Statement

Let $\mathbb{C}(\in \mathbb{N}^+)$ be the checkpointing interval. A synchronous checkpoint is performed at end of superstep $i\mathbb{C}$ or at the beginning of superstep $i\mathbb{C}+1$ ($i \in \mathbb{N}^+$). Without loss of generality, we adopt the latter strategy. We consider a graph job, 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 $F(\mathcal{N}_\text{f}, s_\text{f})$ denote a failure that occurs on a set $\mathcal{N}_\text{f}(\subseteq \mathcal{N})$ of compute nodes when the job executes in superstep $s_\text{f}(\in [1, s_{max}])$. For simplicity, we use $F(\mathcal{N}_\text{f}, s_\text{f})$ and F interchangeably in the rest of this paper. Let $\mathbb{C}+1$ be the latest checkpointing superstep when a failure F occurs. We associate a state with each vertex, as defined below.

Definition 1 (State). *$\forall v \in \mathcal{V}$, the state of v, denoted as $S(v,i)$ in the normal job execution and as $S_F(v,i)$ during the recovery of failure F, is a function recording the latest superstep that has been completed for v at the beginning of superstep i ($\in [1, s_{max}]$).*

Based on Definition 1, we have $S(v,i) = i-1$ during normal job execution. However, upon a failure, all the vertices originally residing in the failed nodes are lost while the others are still maintained in healthy nodes. For this reason, when the recovery starts, for every vertex v originally residing in the failed nodes (i.e. $\phi_\text{p}(\varphi(v)) \in \mathcal{N}_\text{f}$), its state is $\mathbb{C}$ while the state for any of the other vertices is s_f. Formally, we have:

$$S_F(v, \mathbb{C}+1) = \begin{cases} \mathbb{C} & \text{If } \phi_\text{p}(\varphi(v)) \in \mathcal{N}_\text{f} \\ s_\text{f} & \text{Otherwise} \end{cases} \tag{1}$$

As discussed later, when recovery starts, partitions originally residing in the failed nodes will be reassigned to other compute nodes for better parallelism, i.e., the partition-to-node mapping ϕ_p, might be adjusted. To avoid abuse of notations, we use ϕ_p to particularly represent the partition-to-node mapping during normal job execution. For each superstep i ($\in [\mathbb{C}+1, s_\text{f}+1]$) during the recovery, we have:

$$S_F(v, i) = \begin{cases} i-1 & \text{If } \phi_\text{p}(\varphi(v)) \in \mathcal{N}_\text{f} \\ s_\text{f} & \text{Otherwise} \end{cases} \tag{2}$$

We now formalize the failure recovery problem as follows.

Definition 2 (Failure recovery). *Given a failure $F(\mathcal{N}_\text{f}, s_\text{f})$, the recovery for F is to transform the state for each $v \in \mathcal{V}$ from $S_F(v, \mathbb{C}+1)$ to $S_F(v, s_\text{f}+1)$ (i.e., s_f).*

Example 1 (Running example). *Consider the $\mathcal{G}$ in Figure 1(a) and a failure $F(\{N_1\}, 12)$. Assume that every vertex is active and sends messages to all its direct successors in each superstep. $\mathbb{C}$ is set to 10[1], i.e., the latest checkpoint is made at the beginning of superstep 11. We have: (1) $\forall v \in \{A,B,C,D,E,F\}$, $S_F(v,11) = 10$, $S_F(v,11) = 12$; (2) $\forall v \in \{G,H,I,J\}$, $S_F(v,11) = 12$, $S_F(v,11) = 12$.*

The recovery for F is to transform the state of each vertex to the one achieve after the completion of superstep 12. □

It is possible that new failures occur during the recovery for failure $F(\mathcal{N}_\text{f}, s_\text{f})$. Particularly, multiple failures may occur sequentially before all vertices achieves state s_f. We refer to these failures as the *cascading failures* for F.

Definition 3 (Cascading failure). *Given $F(\mathcal{N}_\text{f}, s_\text{f})$, 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 $F_i(\mathcal{N}_{\text{f}i}, s_{\text{f}i})$ the i-th cascading failure in $\mathbb{F}$.*

Upon any cascading failure, analogous recomputation from the latest checkpointing superstep $\mathbb{C}+1$, to s_f will be performed to recover the state of each vertex. Note that when a cascading failure $F_i(\mathcal{N}_{\text{f}i}, s_{\text{f}i})$ occurs, the states of the vertices failed by previous failures may have been partially recovered. For these vertices, $S_{F_i}(v, \mathbb{C}+1) > \mathbb{C}$. However, the recovery for cascading failures are still to transform the states of all vertices to s_f and hence Definition 2 is also applicable to

1. Unless otherwise specified, $\mathbb{C}$ is set to 10 in the rest of this paper.

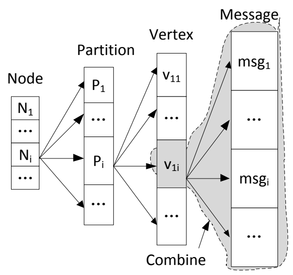


Fig. 2. Message Combination

cascading failure recovery. The goal of our paper is to recover failure F with minimized recovery time.

Problem Statement: Given a failure $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$, 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* $\Gamma(F)$.

To solve this problem, we not only design a failure recovery scheme that is able to handle both single failures and cascading failures, but also develop a recovery algorithm that is able to recover failures efficiently in order to prevent the system from entering an endless recovery loop.

3 Checkpoint and Log based Recovery

This section introduces the checkpoint and log based recovery scheme and our efficient partition-based recovery algorithm.

3.1 Checkpoint and Log based Recovery Scheme

A naïve approach to recovering F correctly is to rollback the state of each vertex to $\mathbb{C}$ based on the latest checkpoint, and redo missing supersteps from $\mathbb{C}+1$ to s_f.

Definition 4 (Correctness). *Let* $\textsc{Value}(v,i)$ *(*$\textsc{Value}^*(v,i)$*) and* $\textsc{Msg}(v,i)$ *(*$\textsc{Msg}^*(v,i)$*) be the value and the set of received messages for vertex v at the beginning of superstep $i+1$ during normal execution (recovery), respectively. A recovery algorithm is* correct if and only if for any failure $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$, after the recovery algorithm finishes, $\textsc{Value}(v,s_\mathrm{f})$ = $\textsc{Value}^*(v,s_\mathrm{f})$ and $\textsc{Msg}(v,s_\mathrm{f})$ = $\textsc{Msg}^*(v,s_\mathrm{f})$.

Recall the computation model of DGPS, for any $v \in \mathcal{V}$, $\textsc{Value}(v,i+1)$ relies on $\textsc{Value}(v,i)$ and $\textsc{Msg}(v,i)$. Hence, we have the following theorem.

Theorem 1. *Given a failure $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$, for any $v \in \mathcal{V}$ and $i \in [\mathbb{C}, s_\mathrm{f}]$, if* $\textsc{Value}(v,i)$ = $\textsc{Value}^*(v,i)$ *and* $\textsc{Msg}(v,i)$ = $\textsc{Msg}^*(v,i)$*, then* $\textsc{Value}(v,i+1)$ = $\textsc{Value}^*(v,i+1)$.

Based on Theorem 1, it is trivial to prove that the above naïve approach can perform failure recovery correctly because $\forall i \in [\mathbb{C}, s_\mathrm{f}]$, $\textsc{Value}(v,i)$ = $\textsc{Value}^*(v,i)$ and $\textsc{Msg}(v,i)$ = $\textsc{Msg}^*(v,i)$ holds for any vertex v. Note that if v resides in healthy nodes $\mathcal{N}-\mathcal{N}_\mathrm{f}$, we have $\textsc{Value}(v,s_\mathrm{f})$ = $\textsc{Value}^*(v,s_\mathrm{f})$ and $\textsc{Msg}(v,s_\mathrm{f})$ = $\textsc{Msg}^*(v,s_\mathrm{f})$ and hence it is practically unnecessary to redo computations for v. This inspires us to perform failure recovery that is confined to the vertices residing in failed nodes. Intuitively, if we collect complete $\textsc{Msg}(v,i)$ for each $v \in \mathcal{N}_\mathrm{f}$ ($i \in [\mathbb{C}, s_\mathrm{f}]$), then we can guarantee that $\textsc{Value}(v,s_\mathrm{f})$ = $\textsc{Value}^*(v,s_\mathrm{f})$ and $\textsc{Msg}(v,s_\mathrm{f})$ = $\textsc{Msg}^*(v,s_\mathrm{f})$. Following this intuition, we propose a checkpointing

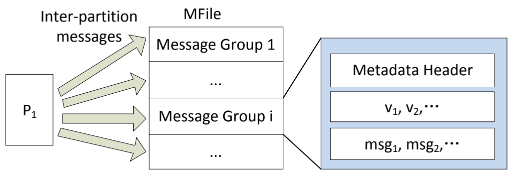


Fig. 3. MFile Layout Example

plus logging recovery mechanism that is able to perform the failure recovery correctly and efficiently.

In particular, during normal job execution, we request each compute node to log the inter-partition messages locally (due to the fact that partitions could be transferred among compute nodes for load balance purpose). Upon a failure $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$, we first rollback the state of each vertex v residing in $\mathcal{N}_\mathrm{f}$ to $\mathbb{C}$, and then redo the missing supersteps by replaying the computation, while vertices in $\mathcal{N}-\mathcal{N}_\mathrm{f}$ do not participate in any computation by merely re-sending the logged messages to vertices in failed nodes. Based on the logged messages, we can collect complete $\textsc{Msg}(v,i)$ for each $v \in \mathcal{N}_\mathrm{f}$, and hence correctly recover the state of v from $\mathbb{C}$ to s_f. Since the maintenance of logged messages may suffer from expensive I/O and communication (messages from healthy nodes to failed nodes) cost, we propose a column-wise message compression and a lazy decompression technique to avoid unnecessary message decompression during the recovery execution.

• **Column-wise message compression.** The inter-partition messages in each compute node are typically organized in a three-level hash table, as shown in Figure 2. The first, second and third levels organize messages to different groups by target compute nodes, target partitions and target vertices, respectively. Based on this organization, if the graph computation is commutative and associative, then one compression strategy is to *combine* all messages intended for a vertex into a single message (see v_{1i} and its messages in Figure 2). To further compress the messages, one naive approach is to employ some regular compression method such as Gzip, over the whole messages directly. Nevertheless, Gzip-like compression can hardly achieve a high data compression ratio due to vertices and messages from different domains. Hence, we leverage the column-wise organization to compress the messages.

In a compute node N, we store messages sent from the same partition into a single file, named as MFile. MFile stores messages using the layout in Figure 3. Specifically, messages sent to the same partition compose a *message group*. A message group contains two sections. The first section is a metadata header for the message group. The metadata header stores the information on the number of vertices (or messages) in this message group, total bytes in each column, and total bytes of each field in a column. The second section contains two columns, target vertices and messages, stored separately. The metadata header, target vertices and messages are compressed independently. We use RLE (Run Length Encoding) algorithm [13] to compress metadata headers where the RLE algorithm can find long runs of repeated data values, especially for fixed field length. Target vertices and messages are independently compressed using Gzip.

• **Lazy decompression.** During the recovery, it is not necessary for each healthy node to read all logged messages into memory. Instead, it can read the messages that are sent to failed compute nodes. Thus, we can skip unnecessary messages and gain I/O advantages. For example, if N_1 in Figure 1(a) fails, for each MFile, N_2 only reads message groups sent to partitions in $\{P_1, P_2, P_3\}$. All these messages will be loaded into memory in the compressed format. We employ lazy decompression technique for message passing to obtain communication advantages. Specifically, compressed messages are directly sent to target compute nodes, and the target compute nodes decompress the messages accordingly.

3.2 Partition-based Recovery Algorithm

Under the checkpoint and log based recovery mechanism, we propose a partition-based method to recover the failures. Upon a failure, the recovery process is initiated by the recovery executor that is responsible for the following three tasks.

• **Generating the partition-based recovery plan.** The input to this task includes the state of each vertex before the recovery starts, and the *statistics* stored in HDFS. We collect statistics during the checkpointing, including:

(1) computation cost of each partition in superstep $\mathbb{C}$.

(2) partition-node mapping ϕ_{p} in superstep $\mathbb{C}$.

(3) for any two partitions in the same compute node, the size of messages forwarded from one to another in superstep $\mathbb{C}$.

(4) for each partition P, total message size from an outside node (where P does not reside) to P in superstep $\mathbb{C}$.

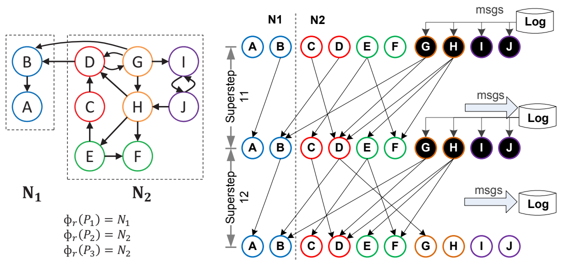
The statistics require a storage cost of $O(|\mathcal{P}| + |\mathcal{P}||\mathcal{N}| + (\frac{|\mathcal{P}|}{|\mathcal{N}|})^2)$, which is much lower than that of a checkpoint. The output recovery plan is represented by a *reassignment* for failed partitions, which is formally defined as follows.

Definition 5 (Reassignment)**.** *Let $\mathcal{P}_{\mathrm{F}}$ (or $\mathcal{P}_{\mathrm{F_i}}$) be the set of partitions residing in the failed nodes, for a failure $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$ (or its cascading failure $F_i(\mathcal{N}_{\mathrm{f}i}, s_{\mathrm{f}i})$). The reassignment for the failure F (or its cascading failure F_i) is a function ϕ_{F}: $\mathcal{P}_{\mathrm{F}} \to \mathcal{N}$ (or $\phi_{\mathrm{F_i}}$: $\mathcal{P}_{\mathrm{F_i}} \to \mathcal{N}$). For simplicity, ϕ_{F} is collectively represented as ϕ_{F} or $\phi_{\mathrm{F_i}}$ when the context is clear.*

Figure 4(a) lists a reassignment for $F(\{N_1\}, 12)$ in Example 1, where 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_{F}. Each node N checks ϕ_{F} 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 $\mathbb{C}$; (3) the status (i.e., active or inactive) of every vertex in the partition in superstep $\mathbb{C}+1$; (4) messages received by the vertices in the partition in superstep $\mathbb{C}$ (to be processed in superstep $\mathbb{C}+1$). Every node then starts recomputation for failed partitions. The details are provided in Section 3.2.1.

• **Exchanging graph partitions.** This task is to rebalance 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.



(a) Reassignment (b) Recomputation

Fig. 4. Recovery for $F(\{N_1\}, 12)$

3.2.1 *Recomputing Failed Partitions*

Consider a failure $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$. The recomputation for the failed partitions starts from the most recent checkpointing superstep $\mathbb{C} + 1$. 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}, i.e., the recomputation terminates when all the compute nodes complete superstep s_{f}.

Algorithm 1 provides recomputation details of node N in a superstep during recovery. In superstep i, N maintains a three-level hash data structure M (shown in Figure 2) that maintains the messages from node N to other nodes (line 1). N iterates through all its belonging partitions. For each of its belonging partitions, P, if its state[2] is no less than the current superstep, we do not need to recover the status of vertices in P (line 2); otherwise, N sequentially checks all the vertices in P and for each active vertex, we redo the vertex computation and store the out-going messages in M. A message $m \in M$ is forwarded if m is needed by its destination vertex (line 10) to perform recomputation in the next superstep (line 3-10). In order to recover cascading failures correctly, we still need to maintain the outgoing messages that are produced during the recovery (line 10). Finally, we resend messages in MFile to the vertices in the failed partitions (line 12-14).

Example 2. *Figure 4(b) illustrates recomputation for $F(\{N_1\}, 12)$, given ϕ_{F} in Figure 4(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 job 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 $D \to G$.*

Note that the messages received by a vertex during recomputation might have a different order compared with those received during normal execution. Hence, the correctness of our recomputation logic implicitly requires the vertex computation is insensitive to message order. This requirement is realistic since a large range of graph applications are implemented in a message-ordering independent manner. Example includes PageRank, BFS, triangle counting, connected component computation, etc. While we are not aware of any

2. Consider that vertext-to-partition mapping is invariant during the entire job execution. The states of vertices in the same partition are the same. We use $S_F(P, i)$ to denote the state of any vertex in partition P in superstep i.

Algorithm 1: Recomputation

```
Input: $i$, current superstep
       $N$, a compute node
1  $M \leftarrow \emptyset$;
2  foreach partition $P \in N$ do
3      if $S_F(P, i) \geq i$ then continue;
4      foreach $v$ in partition $P$ do
5          if $v$.Active= $True$ then
6              Perform computation for $v$;
7              $M.add(v.sendMsg)$;
8              for $m \in v.sendMsg$ do
9                  $v_d \leftarrow$ the target vertex of the message $m$;
10                 if $S_F(v_d, i) = i$ then Send $m$ to $v_d$
11 Compress and Flush $M$ into local storage;
12 for $P \in \mathcal{P}_{\mathrm{F}}$ do
13     $M' \leftarrow$ load compression messages from MFiles sent to $P$ in superstep $i$;
14     send $M'$ to the compute node that $P$ resides in;
```

graph algorithms that are nondeterministic with respect to the message order, our recovery method can be extended easily to support such algorithms if 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 job execution or recovery), for every active vertex v, we can sort all messages received by v based on their identifiers, before initiating the computation. The sorting ensures the messages for a vertex computation during normal job execution follow the same order as those for recomputation during recovery.

3.2.2 Handling Cascading Failures

We now consider cascading failures for $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$. 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 current executing recovery program is terminated and the recovery executor can start a new recovery program for the new failure using the same recovery algorithm that is given in Algorithm 1.

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}. 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 handles cascading failures.

Example 3. *We start with a recovery program for $F(\{N_1\}, 12)$ in Example 2. Suppose there occurs a cascading failure $F_1(\{N_2\}, 12)$ of F. 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 states of vertices after F_1 are: $S_{F_1}(A, 11) = S_{F_1}(B, 11) = 12$ and $S_{F_1}(v, 11) = 10$ 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. Since $S_{F_1}(A, 11) = S_{F_1}(B, 11) = 12$, 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 recovery program triggered by F_1.*

Example 3 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.

Theorem 2. *Our partition-based recovery algorithm is correct and complete.*

Proof: We first prove the correctness of the partition-based recovery algorithm (Algorithm 1). The correctness of a recovery algorithm is given in Definition 4 in which a recovery algorithm is correct if and only if for every vertex v, $\textsc{Value}(v, s_{\mathrm{f}})$ = $\textsc{Value}^*(v, s_{\mathrm{f}})$ and $\textsc{Msg}(v, s_{\mathrm{f}})$ = $\textsc{Msg}^*(v, s_{\mathrm{f}})$. That is, when the recovery completes, the value ($\textsc{Value}(v, s_{\mathrm{f}})$) and the messages ($\textsc{Msg}(v, s_{\mathrm{f}})$) collected of every vertex v must be the same as that during the normal job execution. In Theorem 1, we find that in every superstep during the failure recovery, if $\textsc{Value}(v, s_{\mathrm{f}})$ = $\textsc{Value}^*(v, s_{\mathrm{f}})$ and $\textsc{Msg}(v, s_{\mathrm{f}})$ = $\textsc{Msg}^*(v, s_{\mathrm{f}})$, then the algorithm is correct. To prove the correctness, we then demonstrate that Algorithm 1 strictly follows Theorem 1.

• Under any single failure $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$, Algorithm 1 has the following properties in every superstep i during the recovery:
(1) For any vertex v in a partition $P \in \mathcal{P}_{\mathrm{F}}$, based on Equation 2, its state is less than i. We then perform the recomputation over v (lines 2–6).
(2) Given any vertex v in a partition $P \in \mathcal{P}_{\mathrm{F}}$, the messages collected by v consists of two parts. The first part is sent from vertices in failed partitions as well (lines 8–10) and the other part is from vertices residing in healthy partitions of the other compute nodes (lines 12–14).

The above properties guarantee: i) the vertex value and received messages of a vertex computation in any superstep during recomputation is exactly the same as that during normal job execution; ii) for failure $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$, when our recovery algorithm finishes successfully, each vertex completes superstep s_{f} and receives the same set of messages as it does at the end of superstep s_{f} during normal job execution. Based on Theorem 1, these properties ensure the correctness of our approach.

• Under any cascading failure $F_i(\mathcal{N}_{\mathrm{f}i}, s_{\mathrm{f}i})$, Algorithm 1 has:
(1) It recovers F_i as a single failure (lines 2–6).

Algorithm 2: CostSensitiveReassign

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

(2) After recovering the state of every vertex v to the state that F_i occurs, F_i and F_{i-1} (if any) are integrated into a single failure $F'(\mathcal{N}_{\mathrm{f}i} \cup \mathcal{N}_{\mathrm{f}i-1}, s_{\mathrm{f}i-1})$, and Algorithm 1 continues to recover F' instead of F_{i-1}.
(3) The above recovery continues until the state of every vertices is updated to F.

Analogous recovery of every single failure is performed under any cascading failures, and hence the correctness of our proposed algorithm can be guaranteed. 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. □

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 states for all the vertices after the failure occurs and measure the minimum time T_{low} required to transform their states to s_{f}, i.e., the time of performing recomputation from the beginning of superstep $\mathbb{C}+1$ to that of $s_{\mathrm{f}}+1$ without further cascading failures. We then aim to produce a reassignment that minimizes T_{low}. Essentially, 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.

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.

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

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. Given a failure $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$, 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 $\tau(v,i)$ denote the computation time of v in the normal job execution of superstep i. Regarding that the recovery is confined to the vertices residing in the failed partitions, we then compute $\mathbf{T}_{\mathrm{p}}[i]$ by collecting the sum of computations for vertices in failed partitions in each compute node, and $\mathbf{T}_{\mathrm{p}}[i]$ is quantified below:

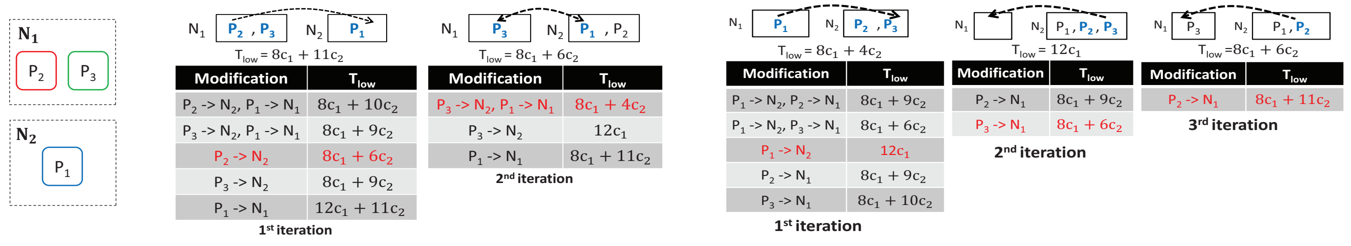
$$\mathbf{T}_{\mathrm{p}}[i] = \max_{N \in \mathcal{N}} \sum_{P \in \mathcal{P}_{\mathrm{F}} \wedge \phi_{\mathrm{F}}(P) = N} \sum_{v \in P} \tau(v,i) \tag{4}$$

For 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 [14], which estimates network communication time by the total size of inter-node messages divided by network bandwidth. Let $\mathcal{M}(P,P')$ be the size of compressed messages from P to P' forwarded when re-executing superstep i. Suppose the network bandwidth is B. The communication cost consists of two parts: (1) messages between every two failed partitions that span two compute nodes, and (2) messages from partitions residing in the other healthy compute nodes to every failed partition. Let $\overline{\phi_F}$ be the partition-to-node mapping (including partitions in failed and healthy nodes) when the recovery starts and hence mapping ϕ_F is included in $\overline{\phi_F}$. We can compute $\mathbf{T}_{\mathrm{m}}[i]$ below:

$$\mathbf{T}_{\mathrm{m}}[i] = \sum_{P \in \mathcal{P}_{\mathrm{F}}} \sum_{P' \in \mathcal{P} \wedge \phi_{\mathrm{F}}(P) \neq \overline{\phi_F}(P')} \mathcal{M}(P,P')/B \tag{5}$$

Note that $\tau(v,i)$ and $\mathcal{M}(i)$ in Equation 4 and 5 can only be obtained during the runtime execution of the application.



Modification	T_{low}
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_{low}
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

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

2nd iteration

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

3rd iteration

(a) Initial (b) Pass 1 (c) Pass 2

Fig. 5. Example of Modifications

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 3.2) for approximation. Specifically, we compare the status of each partition with the current superstep (line 3 in Algorithm 1) to examine whether 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 $\mathbb{C}+1$. We then approximate the costs in superstep i by those in superstep $\mathbb{C}+1$.

Example 4. *Consider $F(\{N_1\}, 12)$ in Example 3 and ϕ_{F} 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_{F}, 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_{F} is $8c_1 + 11c_2$.*

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

Proof: We use a reduction from the $(k,1)$-balanced graph partitioning problem ($k \geq 2$) defined as follows. Given an undirected graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{W}_e)$ where $\mathcal{W}_e : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}^*$ represents non-negative edge weights, the task is to divide $\mathcal{V}$ into k components (each of size less than $\frac{|\mathcal{V}|}{k}$) so that the total weight of edges between different components is minimized. The complexity of the problem is NP-complete [5].

Our reassignment generation problem can be viewed as finding a partitioning ϕ_{F} for an undirected weighted reassignment graph where every failed partition $P \in \mathcal{P}_{\mathrm{F}}$ is condensed into a vertex v_p with weight of: $\mathcal{W}_v(v_p) = \sum_{i=\mathbb{C}+1}^{s_{\mathrm{f}}} \sum_{v \in P} \tau(v,i)$ and the weight of an edge $(v_p, v_{p'})$ satisfies: $\mathcal{W}_e(v_p, v_{p'}) = \sum_{i=\mathbb{C}+1}^{s_{\mathrm{f}}} \frac{\mathcal{M}(P,P',i)}{B}$ for $\phi_{\mathrm{F}}(P) \neq \phi_{\mathrm{F}}(P')$ and $\mathcal{W}_e(v_p, v_{p'}) = 0$ for $\phi_{\mathrm{F}}(P) = \phi_{\mathrm{F}}(P')$, that minimizes:

$$\sum_{P,P' \in \mathcal{P}_{\mathrm{f}}} \mathcal{W}_e(v_p, v_{p'}) + \max_{N \in \mathcal{N}} \{ \sum_{P \in \mathcal{P}_{\mathrm{f}}, \phi_{\mathrm{F}}(P) = N} \mathcal{W}_v(v_p) \}.$$

Given $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{W}_e)$, we can construct a reassignment graph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}, \mathcal{W}_e, \mathcal{W}_v)$ where vertices in $\mathcal{V}$ have equal weights of w_M. w_M is set to a sufficiently large number, e.g., total edge weight, to ensure the optimal repartitioning for $\mathcal{G}'$ is $(k,1)$-balanced. Suppose ϕ_{F} is the optimal reassignment for $\mathcal{G}'$ over k compute nodes. It is easy to verify that i) ϕ_{F} must produce k perfectly balanced partitions, where $k - n\%k$ partitions contain $\lfloor \frac{n}{k} \rfloor$ vertices and $n\%k$ partitions include $\lfloor \frac{n}{k} \rfloor + 1$ vertices; ii) ϕ_{F} is the optimal $(k,1)$-balanced partitioning for $\mathcal{G}$. That is, the optimization objectives of two problems are identical. The reassignment generation problem can be solved iff the $(k,1)$-balanced graph partitioning is solved. □

Algorithm 3: NextChange

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

    foreach $N \in \mathcal{N} - \{\phi_{\mathrm{F}}(P)\}$ do
        $\phi'_F \leftarrow \phi_{\mathrm{F}}$; $\phi'_F(P) \leftarrow N$;
        $t' \leftarrow$ComputeCost($\phi'_F, \mathcal{I}, \mathcal{N}$);
        if $\mathcal{L}_i.Time > t'$ then
            $\mathcal{L}_i.\overline{\phi_F} \leftarrow \{(P, N)\}$;
            $\mathcal{L}_i.Time \leftarrow t'$;
```

4.2 Cost-Sensitive Reassignment Algorithm

Due to the hardness result in Theorem 3, 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_{F} by randomly assigning partitions in $\mathcal{P}_{\mathrm{F}}$ among compute nodes $\mathcal{N}$, and then calculate T_{low} under ϕ_{F} (line 1-2). We next make a copy of ϕ_{F} as ϕ'_F and improve ϕ'_F 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 $(\overline{\phi_F}, Time)$, where $\overline{\phi_F}$ 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., $l = \arg\min_i\{\mathcal{L}_i.Time\}$. If $\mathcal{L}_l.Time$ is smaller than T_{low} achieved by the initial reassignment ϕ_{F}, we update ϕ_{F} by sequentially applying all modifications in $\mathcal{L}_1, \cdots, \mathcal{L}_l$ (line 12-16), and start another pass. Otherwise, the algorithm outputs ϕ_{F} 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) moving one partition from one compute node to another compute node. Given a reassignment ϕ_{F}, 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 $\phi_{\mathrm{F}}(P)$ (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 5. *Continue with Example 4. Suppose* $\frac{c_1}{c_2} = 1.1$. *Figure 5 shows the* T_{low} *in two passes under different modifications. The final reassignment with minimum* T_{low} $(8c_1{+}4c_2)$ *is achieved by 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

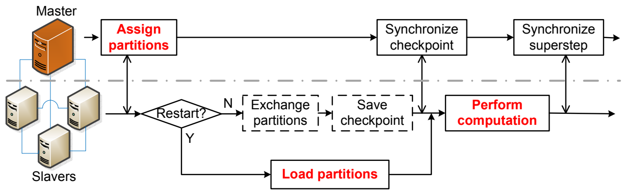


Fig. 6. Processing a Superstep in Giraph

and considers a slave as failed if it has not registered its status over a specified interval. Giraph adopts checkpoint-restart 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.

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) `genChangedPartitionOwners()` is invoked in the other supersteps, 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

```
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();
```

Fig. 7. Major APIs

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) Every slave iterates over the partitions using function `processGraphPartitions()` 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 $\mathbb{C}+1$ to s_{f}, a message is omitted if it is sent to a vertex residing in a 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 $[\mathbb{C}+1, s_{\mathrm{f}}]$, 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-restart method on top of Giraph graph processing engine. We use version-1.1.0 of Giraph that is available in [1].

6.1 Experiment Setup

The experimental study was conducted on our in-house cluster with 42 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. Giraph runs as a Map-only job on top of Hadoop, hence we made the following changes to the default Hadoop configurations: (1) the size of virtual memory was set to 4GB; (2) each node was configured to run one map task.

TABLE 1
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

6.2 Benchmark Tasks and Datasets

We study the failure recovery over three benchmark tasks: k-means, semi-clustering [21] and PageRank. We set $k = 100$ in k-means. We port the implementation of semi-clustering from Hama [2] into Giraph and use the same configurations: 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. 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 a vector dataset and two real-life graphs[3], as described in Table 1.

• **Forest.** Forest dataset[4] predicts forest cover type from cartographic variables. It originally contains 580K objects, each of which is associated with 10 integer attributes. We enlarge the size of Forest by 10X using the data generator from [20].

• **LiveJournal.** LiveJournal is an online social networking and journaling service. It contains 4 million vertices (users) and 70 million directed edges (friendships between users).

• **Friendster.** Friendster is an online social networking and gaming dataset with 60 millions vertices and 1 billion edges.

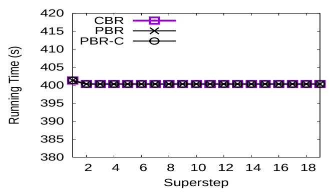
We compare our proposed partition-based recovery method (PBR), with the checkpoint-restart recovery method (CBR) over two **metrics**: recovery time and communication cost. For PBR, we also evaluate the effectiveness of our logging compression scheme, which we refer to as PBR-C.

6.3 k-means

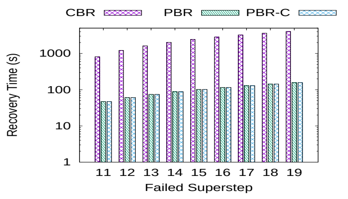
We first study the overhead of logging outgoing messages. Figure 8(a) shows the running time. PBR and PBR-C take almost the same time as CBR. This is because in k-means tasks, there is no outgoing messages among different vertices, and in this case, PBR and PBR-C perform exactly the same as CBR. Another 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

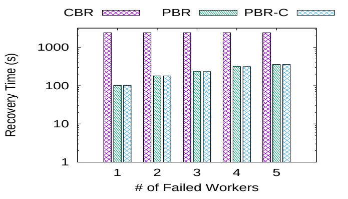
3. http://snap.stanford.edu/data/index.html
4. http://archive.ics.uci.edu/ml/datasets/Covertype



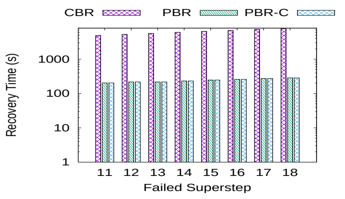
(a) Logging Overhead



(b) Single Node Failure

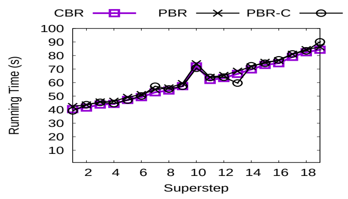


(c) Multiple Node Failure

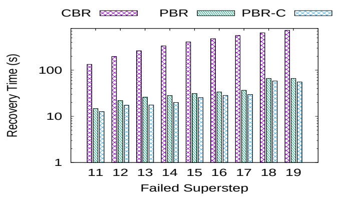


(d) Cascading Failure

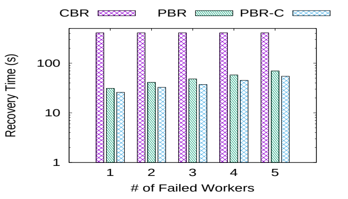
Fig. 8. k-means



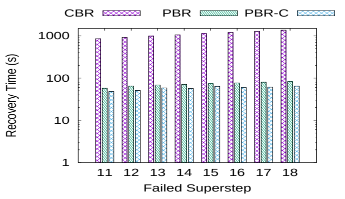
(a) Logging Overhead



(b) Single Node Failure

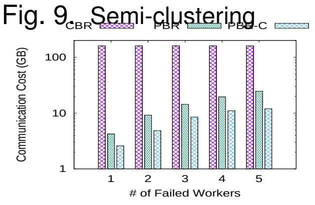


(c) Multiple Node Failure

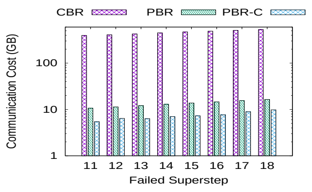


(d) Cascading Failure

Fig. 9. Semi-clustering



(a) Multiple Node Failure



(b) Cascading Failure

Fig. 10. Communication Cost of Semi-clustering

19. Figure 8(b) plots the results. The recovery time of all three methods increases linearly when the failed superstep varies. Since there are no messages passing among different workers, computing the new belonging nodes for failed partitions can be accelerated by using all available workers, i.e., recomputation is parallelized over 40 workers for recovery. We find that PBR and PBR-C outperform 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 and PBR-C while that remains constant for CBR. No matter how many nodes fail, CBR will redo all computation from the latest checkpoint, while PBR and PBR-C redo the computation for observations in the failed nodes and hence the recovery time becomes longer with the increase of failed nodes. On average, PBR and PBR-C outperform 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. On average, PBR and PBR-C can reduce recovery time by a factor of 23.8 to 26.8 compared with CBR.

6.4 Semi-clustering

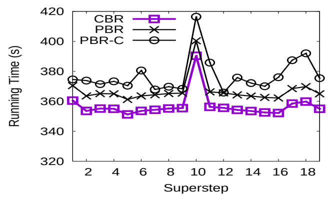
Figure 9(a) plots the running time of each superstep for semi-clustering. PBR and PBR-C takes slightly longer time than CBR during normal job execution due to the overhead of logging outgoing messages. PBR and PBR-C almost take the same time. The reason, as we discussed before, is that the I/O cost saved by compressing the logged messages is comparable with compression cost itself. Moreover, in semi-clustering, the size of each message from a vertex to its neighbors increases linearly with the superstep. Hence, all three methods runs slower in larger supersteps. In superstep 10, there is an obvious increment in the running time due to performing a checkpoint.

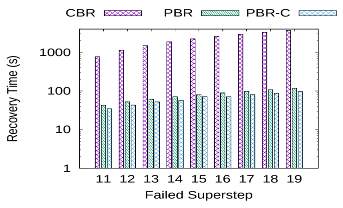
Figure 9(b), 9(c), 9(d) show the results for single node failures, multiple node failures and cascading failures under the same settings as k-means. Basically, the trends of the running time of PBR (PBR-C) and CBR are similar to that in k-means. Specifically, PBR outperforms CBR by a factor of 9.0 to 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. PBR-C outperforms PBR by a factor of 20%. PBR-C applies a lazy-decompression scheme and hence saving the communication cost by directly transmitting the compression messages.

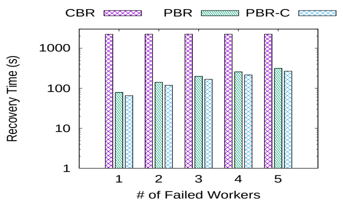
Besides the benefit of parallelizing computation, we also show the communication cost incurred by PBR, PBR-C and CBR in Figure 10. Since messages sent to vertices residing in healthy nodes can be omitted in PBR and PBR-C, we observe that in multiple node failure, PBR incurs 6.5 to 37.9 times less communication cost than CBR. For cascading failures, PBR can reduce communication cost by a factor of 37.1 compared with CBR. Due to the compressed messages, PBR-C can reduce the communication cost by a factor of 1.7 to 2.4.

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 and PBR-C take slightly more time than CBR in PageRank. This is because PageRank is evaluated over the Friendster 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. By compressing the messages, we can see that PBR-C can achieve 1.5% reduction of running time due to the I/O cost reduction compared with PBR. Due to doing checkpointing, there is an obvious increment of running time in superstep 10. In each superstep, the


(a) Logging Overhead


(b) Single Node Failure


(c) Multiple Node Failure

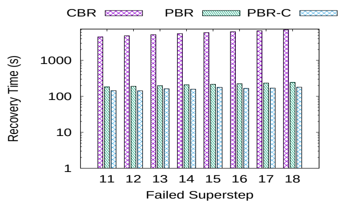

(d) Cascading Failure

Fig. 11. PageRank

TABLE 2
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

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(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. Due to the same reason discussed in semi-clustering, PBR-C can reduce the communication cost by a factor of 1.68 to 2.23.

6.6 Simulation Study

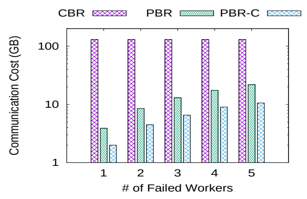
We perform a simulation study to evaluate the effectiveness and efficiency of our cost-sensitive reassignment algorithm CostSen in partition-based recovery. As a comparison, we consider a random approach Random by balancing computation among the nodes.

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 partition and failed one, is uniformly drawn from the range $[1, low]$.


(a) Multiple Node Failure

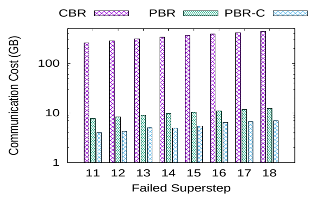

(b) Cascading Failure

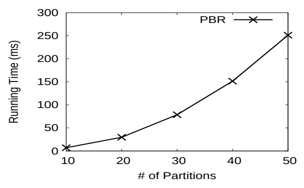
Fig. 12. Communication Cost of PageRank

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.

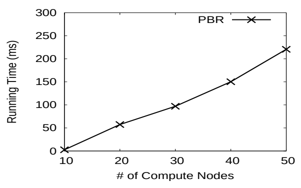
We generate comparable computation cost for each failed partition based a *comp-comm-ratio*, γ. Let S_P be the total communication cost from healthy nodes to a failed partition P. We use γ to adjust the ratio between the computation cost of P and S_P. The computation cost of P is randomly drawn from the range $[1, \gamma S_P]$. A larger γ implies that the job is more computation-intensive. Table 2 summarizes the ranges of our tuning parameters. Unless otherwise specified, we use the underlined default values.

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 3 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 analysis.



(a) # of Healthy Nodes



(b) # of Healthy Nodes

Fig. 13. Running Time (well-distributed)

TABLE 3
Varying 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 4
Varying 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

Effects of High Communication Partition (Healthy Node) Number. Table 4 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 5 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 n.

Effects of the number of healthy nodes. Table 6 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.

7 Related Work

Our work is related to: (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-restart*, *log-based* and *hybrid* approaches [11]. Most popular distributed graph processing systems such as Giraph [1], GraphLab [19], PowerGraph [12], GPS [23], Mizan [16] adopt checkpoint-restart recovery. Pregel [21] proposes confined recovery which is a hybrid mechanism of the checkpoint-restart and log-based recovery. Specifically, only the newly-added node that substitutes the failed one has to rollback and repeats the computations from the latest checkpoint. GraphX [29] 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 [6], [7], [8]. Many works focused on accelerating checkpoint-restart or the log-based recovery [11]. Location and replication independent recovery proposed by Bratsberg et al. employed replicas for recovery [9]. 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 [22]. 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 [15], 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 for power-law graphs [3], multi-threaded graph partitioning [17] and dynamic multi-constraint graph partitioning [24]. 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.

TABLE 5
Varying 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 6
Varying 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

8 Conclusion

This paper presents a novel partition-based recovery method to parallelize failure recovery processing. Different from traditional checkpoint-restart 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-restart recovery methods by up to 30 times when using 40 compute nodes.

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*, 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] K. Andreev and H. Räcke. Balanced graph partitioning. SPAA, pages 120–124, 2004.

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

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

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

[9] 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.

[10] Y. Bu, V. R. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.

[11] 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.

[12] 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.

[13] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *ICDE*, pages 1199–1208, 2011.

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

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

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

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

[23] S. Salihoglu and J. Widom. Gps: a graph processing system. In *SSDBM*, page 22, 2013.

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

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

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

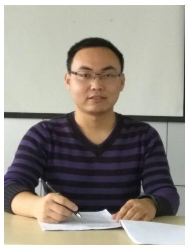
[27] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.

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

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

Wei Lu is currently an associate professor at Renmin University of China. He received his Ph.D degree in computer science from Renmin University of China in 2011. His research interest includes query processing in the context of spatiotemporal, cloud database systems and applications.

Yanyan Shen is currently an assistant professor at Shanghai Jiao Tong University, China. She received her BSc degree from Peking University, China, in 2010 and her Ph.D. degree in computer science from National University of Singapore in 2015. Her research interests include distributed systems, efficient data processing techniques and data integration.



Tongtong Wang is current a Ph.D student at Renmin University of China. He received his master degree in computer science from Renmin University of China in 2011. His research interests include distributed graph processing systems and efficient data processing techniques.

Meihui Zhang received the BEng degree in computer science from the Harbin Institute of Technology, China, in 2008 and the PhD degree in computer science from the National University of Singapore in 2013. She is currently an assistant professor at the Singapore University of Technology and Design. Her research interest includes crowdsourcing-powered data analytics, massive data integration, and spatio-temporal databases. She is a member of the IEEE.

H. V. Jagadish is currently the Bernard A Galler Collegiate Professor of Electrical Engineering and Computer Science at the University of Michigan. He received his Ph.D. from Stanford University in 1985. His research interests include databases and Big Data.

Xiaoyong Du is a professor at Renmin University of China. He received his Ph.D. degree from Nagoya Institute of Technology in 1997. His research focuses on intelligent information retrieval, high performance database and unstructured data management.