Parallelizing Recovery in Distributed Graph Processing Systems

Xuan Liu[†], Wei Lu[†], Yanyan Shen[†], Gang Chen[§], Beng Chin Ooi[†]
[†]National University of Singapore, Singapore
[§]Zhejiang University, China
[†]{liuxuan,luwei1,shenyanyan,ooibc}@comp.nus.edu.sg, [§]cg@cs.zju.edu.cn

ABSTRACT

Distributed graph processing has attracted a great deal of research interest in recent years. Fueled by the needs for processing huge graphs, the number of compute nodes in the distributed graph processing systems is growing fast. However, an increasingly large number of compute nodes will inevitably increase the chance of node failures. Therefore, provisioning an efficient failure recovery strategy is very important for distributed graph processing systems.

In this paper, we propose a novel partition based failure recovery method. Instead of traditional checkpoint based recovery or confined recovery, our recovery method distributes the recovery tasks to multiple compute nodes to accelerate failure recovery. Our recovery method pre-computes the recovery index for each of the compute nodes. The recovery index is used to distribute the recovery jobs of the corresponding failed compute nodes to other nodes. Furthermore, our proposed failure recovery method is also applicable to multiple node failures. We implement our recovery method and conduct extensive experiments on the widely use Giraph system to validate the performance of our proposed method.

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. Map-Reduce was proposed as a programming model for the Big Data about a decade ago, and since then, many Map-Reduce based distributed systems have been designed for Big Data applications such as large-scale data analytics. However, in recent years, Map-Reduce has been shown to be ineffective for handling graph data, and several new systems such as Pregel [16], Giraph [1], GraphLab [15, 9], and Trinity [21] have been recently proposed and designed for scalable distributed graph processing.

With the explosion in graph size and increasing demand of complex analytics, distributed graph processing systems have to continuously scale out by increasing the number of compute nodes in order to handle the load. Naturally, an increasingly large number of compute nodes will inevitably lead to an increase of node failures. Therefore, provisioning an efficient failure recovery strategy is very important for distributed graph processing systems. In existing distributed processing systems, two recovery strategies, namely checkpoint based recovery method and confined recovery method, are commonly used.

In the checkpoint based recovery method, each compute node is required to periodically and synchronously write its data to a stable storage like distributed file system (DFS) as a checkpoint. It requires all the active compute nodes to rollback to the most recent checkpoint when there is a failure, and uses an unused active compute node to replace the failed node and let all the nodes synchronously re-execute all the supersteps. The failure is recovered when all the nodes finish all the supersteps before the failure occurs.

In contrast to the checkpoint based recovery method, the confined recovery method will not rollback all active compute nodes. Instead, it requires every compute node to cache all messages sent to other nodes. The confined recovery method also uses an unused active node to replace the failed node, but only the new node will re-execute all supersteps. Other active nodes would just resend the messages to the new node. The failure is recovered when the new node finishes all the supersteps before the failure occurs.

Although these two failure recovery methods are effective in existing distributed graph processing systems, they still have two major weaknesses. First, both failure recovery methods require a very long recovery time, especially for computational intensive graph processing applications, such as LDA [4] and restricted Boltzmann machine [18]. For these computational intensive applications, both failure recovery methods take very long time to re-execute each superstep. Second, both failure recovery methods require a strict condition to guarantee that the recovery processing converges when there are multiple node failures. In fact when an active node fails during the recovery processing, both failure recovery methods need to rollback to the most recent checkpoint and re-execute the recovery procedure. As a result, they can terminate only if there are no more failures during the recovery processing. However, this condition may not be satisfied when the number of compute nodes in the distributed graph processing system is large enough such that there may be some failed nodes at every superstep.

To alleviate the above two problems, we propose a novel partition based failure recovery method in this paper. Our method partitions the subgraph in the failed compute node to distribute the complex computations to multiple active nodes such that they can compute concurrently. By partitioning and distributing the failed subgraph, the total time of failure recovery can be significantly reduced. In addition, our failure recovery method does not require any active compute node to rollback when there are multiple node failures. Instead, our method determines the cascading failures within these multiple node failures and iteratively recover all

Table 1: Table of Notations

$G = (V, E)$	computation graph
N	compute node
P	partition on N
v	vertex on the node
t	average computation time for a vertex
m	average size of messages
B	bandwidth of the network
C	number of executed supersteps when the failure occurs
N_f	failed node
V_f	vertices on the failed node
$G^* = (V^*, E^*)$	weighted computation graph
$w(v_i, v_j)$	weight on edge (v_i, v_j)
$A(v_i)$	associated vertex for v_i

the failures from the most recent failure. Since there is no rollback on the compute nodes, our method requires a much weaker condition to guarantee that the recovery process converges. Therefore, with a large probability, our proposed recovery method finishes the recovery processing before the two existing recovery methods.

The contributions of our work proposed in this paper are as following:

- We propose a processing time based cost model to analyze the performance of distributed graph processing systems.
- We design a novel partition based failure recovery method that can efficiently recover a single node failure using precomputed recovery index.
- We extend our partition based failure recovery method such that it handles multiple node failures. Furthermore, our method requires a much weaker condition to guarantee the convergence of the recovery processing than existing methods.
- We implement our proposed recovery method on the widely used Giraph system.
- We conduct extensive experimental studies to validate the performance of our proposed method.

The rest of this paper is organized as follows. Section 2 introduces the preliminaries of distributed graph processing. Section 3 presents the cost model for the distributed graph processing systems with respect to the processing time. Section 4 discusses our partition based failure recovery method for a single node failure. Section 5 extends the recovery method presented in Section 4 for multiple node failures, including both cascading failures and non- cascading failures. Section 6 presents the implementation details of our method on top of the widely used open source Giraph system. Section 7 presents the experimental results. We discuss related works in Section 8, and conclude in Section 9.

2. PRELIMINARIES

In this section, we explain some preliminaries related to distributed graph processing and failure recovery.

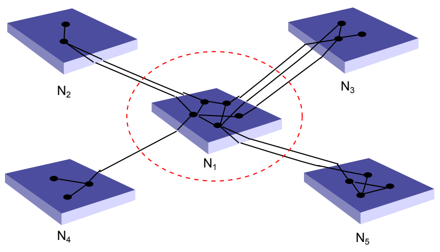

Figure 1: Partitioned Computation Graph

2.1 Computation Graph

In distributed graph processing, the computation in each superstep (iteration) could be modeled as a connected undirected graph G, where the vertices V represent the computational unit and the edges E represent the dependency between computational units. Edge e_{ij} represents the computation of vertex v_i that relies on the message of vertex v_j and vice versa. The computation graph is partitioned such that the vertices in a partition are located on a single physical compute node. We denote ith compute node as N_i and the partition on N_i as P_i. Furthermore, the vertices in compute node N_i is denoted as V_i and the edges across compute node N_i and N_j is represented as E_{ij}. The communication delay between two vertices in the same partition is negligible and hence could be ignored. For example, Figure 1 shows a computational graph distributed on five compute nodes, namely N_1 to N_5.

In this paper, we do not make assumptions on the prior distributions of the edges as our method can be applied to any general graphs. Furthermore, we also do not make assumptions on the method that the computation graph is partitioned. In fact, the computation graph and the partitions are provided by the user of the distributed graph processing system. However, our method could be further optimized if each partition of the computation graph follows the power law distribution.

2.2 Computation Model

The computational model of distributed graph processing in this paper follows the Bulk Synchronous Parallel (BSP) model, which has been adapted for the Google's Pregel. In the Bulk Synchronous Parallel model, the computation consists of a number of supersteps. In each superstep, every node starts to send messages to its neighbors after receiving a coordinating message from the master node. Then the nodes that have received all the messages can start the computation job. After completing the computation, each node notifies the master node with an acknowledgement message, indicating it has finished this superstep. The procedure of a superstep specifies the behavior on a single node and the supersteps are managed by the master node using global synchronization.

3. COST MODEL OF DISTRIBUTED GRAPH PROCESSING

Since we are motivated by efficient failure recovery in distributed graph processing systems, we shall introduce the cost model of distributed graph processing in this section in order to illustrate our idea.

The cost of distributed graph processing in this paper focuses on the execution time of graph processing rather than the amount of migrated data or the energy consumptions etc. This means that the objective of our failure recovery method is to minimize the total amount of failure recovery time.

Our cost model is proposed based on the synchronized distributed graph processing systems that could be modeled using the Bulk Synchronous Parallel model. In order to simplify the problem, we assume the following properties.

Assumption 1 (General Property of Bulk Synchronous Parallel model). *Distributed graph processing systems obey*

1. *The computation ability of each compute node is invariant during the supersteps.*
2. *The network between a pair of compute nodes is reliable.*
3. *The communication bandwidth B of the network between a pair of compute nodes is invariant during the supersteps.*

Note that these three general assumptions are required to guarantee that the performance of the system is stable while a distributed graph processing job is being executed. Given the above assumptions on the stableness of the performance, we denote the average computation time of each vertex as t and the average amount of data in a message as m. Therefore, the computation time T_{comp} of compute node N_i is

$$T_{comp} = |V_i| * t$$

and the message passing communication time T_{comm} between compute node N_i and N_j is

$$T_{comm} = |E_{ij}| * m/B$$

In each superstep, the compute node N_i needs to communicate with all its neighbors. Therefore, the total message passing time depends on the topological structure of the network in the distributed graph processing system. To generally support all the distributed graph processing systems, we model the network transportation among compute nodes as types, namely serialized communication and concurrent communication.

The serialized communication is described in the Assumption 2.

Assumption 2 (Serialized Communication Property). *The transportation of all the data from one compute node to all its neighbors through the physical network must be serialized.*

Using the serialized communication assumption, the average processing time T of a superstep for a compute node N_i in the Bulk Synchronous Parallel model is computed as

$$T_i = |V_i| * t + \sum_{j \neq i} |E_{ij}| * m/B$$

In contrast to the serialized communication, the concurrent communication is described in the Assumption 3.

Assumption 3 (Concurrent Communication Property). *The transportation of all the data from one compute node to all its neighbors through the physical network can be concurrently executed.*

Using the concurrent communication assumption, the messages can be concurrently transported using the physical network. Thus, the average processing time T of a superstep for a compute node N_i is derived as

$$T_i = |V_i| * t + \max_{j \neq i} |E_{ij}| * m/B$$

Obviously, when t or m is increased, the total processing time is increased. Our model assumes that t and m are fixed and they are determined by the architecture of distributed graph processing systems. Thus, the average processing time of a superstep can be minimized by properly partitioning the computation graph. In the distributed graph processing systems, failure recovery is usually designed as partial re-execution of the supersteps from the most recent checkpoint. Therefore, we apply our average processing time model to analyze the time consumption in the failure recovery.

4. SINGLE NODE FAILURE RECOVERY

In this section, we present a graph re-partitioning based method that reduces the failure recovery processing time. We first discuss the case that only one compute node fails in this section and we extend our algorithm in Section 5 to solve the failure recovery problem of multiple nodes.

4.1 Re-partition based Failure Recovery

Traditional failure recovery methods in existing distributed graph processing systems like Google's Pregel only let the failed compute node re-execute all the supersteps after the most recent checkpoint after it recovers. All the other compute nodes have to keep the current intermediate results in memory and resend all the messages in each superstep to the failed compute node. As our main objective is to reduce the processing time of failure recovery of computation intensive applications, our idea is to distribute the computation job of the failed node to several other compute nodes such that these computation jobs can be executed concurrently. Intuitively, for computation intensive applications, the time spent on the node computation could be much larger than the time spent on message passing. Therefore, the total time of processing a superstep might become smaller if the computation job is split into several concurrently executed jobs. The computation structure on the failed node is represented by a partition of the computation graph, and hence the main recovery process is to re-partition and distribute the partition on the failed node to other healthy compute nodes.

Figure 2, 3 and 4 show the recovery process of our re-partitioning based method. In Figure 2, the failed node is re-partitioned into small partitions. Each partition is distributed to the active node that has most number of edges as shown in Figure 3. By distributing the small partitions, the recovery can be concurrently executed by those nodes. After these nodes finish the recovery jobs, they send the data back to the failed node, shown in Figure 4.

To analyze the time cost of the failure recovery procedure, we first formally define the problem. We denote the failed node as N_f and the partition on N_f as P_f. The corresponding vertices and edges of P_f are represented as V_f

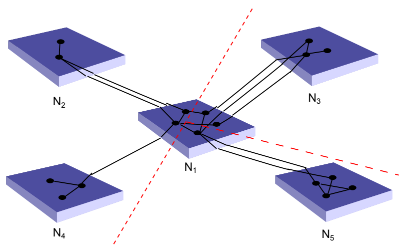

Figure 2: Re-partitioning the failed node

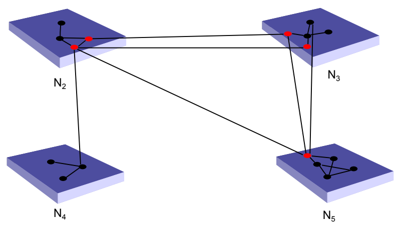

Figure 3: Distributing recovery jobs

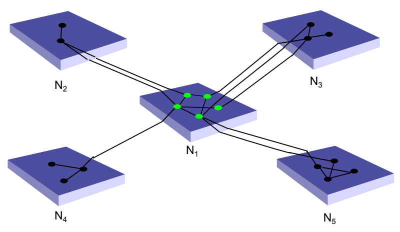

Figure 4: Recoverng the failed node

and E_f. Instead of using the original computation graph G, we construct a simplified weighted computation graph $G^* = (V^*, E^*)$ to formally define our problem. The vertices $V^* = V_f \cup \mathbb{V}$ of G^* consist of two parts, namely the vertices V_f in P_f and the abstract vertices $\mathbb{V}$ where each $v_i \in \mathbb{V}$ represents a compute node N_i other than N_f. The edges $E^* = \bar{E}_f \cup \mathbb{E}$ of G^* also contain two parts, namely the internal edges $\bar{E}_f$ in P_f and the edges $\mathbb{E}$ between each $v_i \in V_f$ and each $v_j \in \mathbb{V}$. Note that the edge set E_f may contain the edges across compute nodes. The internal edges $\bar{E}_f$ is defined as the edges between vertices in V_f, namely $\bar{E}_f = \{(v_i, v_j)|(v_i, v_j) \in E_f, v_i \in V_f, v_j \in V_f\}$. For each $v_i \in V_f$ and each $v_j \in \mathbb{V}$, we define the edge set between vertex v_i and the vertices in V_j as $\mathbb{E}_{ij}$. The weight function $w()$ of each edge is defined as

$$w(v_i, v_j) = \begin{cases} 1 & \text{if } (v_i, v_j) \in \bar{E}_f & \text{(1a)} \\ |\mathbb{E}_{ij}| & \text{if } (v_i, v_j) \in \mathbb{E} & \text{(1b)} \\ 0 & \text{otherwise} & \text{(1c)} \end{cases}$$

We then define the failure recovery problem as an optimal weighed k-partition problem (assuming the serialized communication property).

Problem Definition (Optimal Weighed k-Partition Problem). *Given the simplified weighted computation graph $G^* = (V^*, E^*)$ and the weight function $w()$ of the edges E^*, the optimal weighted k-partition problem is to find the best k-partition $G^* = P_1^* \cup P_2^* \cdots \cup P_k^*$ such that*

1. $\forall i \neq j$, $P_i^* \cap P_j^* = \emptyset$.
2. *For any partition P_j^*, there exists a unique vertex a vertex $v_s \in V_j^* - V_f$ such that very vertex $v_i \in V_j^* \cap V_f$ is associated with v_s, denoted as $v_s = A(v_i)$. The set of associated vertex is denoted as $\mathbb{A} = \{v_s | \exists v_i, v_s = A(v_i)\}$.*
3. *The time*

$$\begin{aligned} T &= C * \max_{v_s \in \mathbb{A}} \{|V^*(v_s) \cap V_f| * t \\ &+ \sum_{A(v_i)=v_s} \sum_{A(v_j) \neq v_s} w(v_i, v_j) * m/B\} + 2|V_f|m/B \end{aligned}$$

is minimized, where C is the number of executed supersteps before the failure since the most recent checkpoint and $V^*(v_s)$ refers to the vertices in the partition containing v_s.

4. *The workload of each associated node is about the same, i.e.*

$$\exists \epsilon > 0, \forall v_s \in \mathbb{A}, |\{v_i | v_i \in V_f, v_s = A(v_i)\}| < (1+\epsilon)\lceil \frac{|V_f|}{k} \rceil$$

In our definition of the optimal weighed k-partition problem, the processing time of the failure recovery includes three components, namely:

1. Data migration from the failed node N_f to the k associated helper nodes in each partition P_i^*: The time cost is $\sum_{v_i \in V_f} m/B = |V_f| m/B$.
2. Recomputation of each superstep from the most recent checkpoint: The number of supersteps that need to be recomputed is also C and the average time cost of each superstep is $\max_{v_s \in \mathbb{A}} \{|V^*(v_s) \cap V_f| * t + \sum_{A(v_i)=v_s} \sum_{A(v_j) \neq v_s} w(v_i, v_j) * m/B\}$.
3. Data migration from the k associated helper nodes back to the failed node N_f: The time cost is also $|V_f| m/B$.

We shall prove that the optimal weighed k-partition problem falls into the NP-Hard complexity class, i.e.

Theorem 1 (NP-Hardness). *The optimal weighed k-partition problem is NP-Hard.*

Proof. We prove the theorem by reducing the k-balanced partitioning problem [8] to our problem, which is an NP-Hard problem. For each instance graph $\mathcal{G}$ of k-balanced partitioning problem, we construct an instance graph G of the optimal weighed k-partition problem. The vertices V and edges E of G are the same as those in $\mathcal{G}$ and the weight $w(e) = 1$ for every edge e. V_f is an arbitrary set of vertices with constant size. We let $t = 0$ such that

$$T = C * \max_{v_s \in \mathbb{A}} \{ \sum_{A(v_i)=v_s} \sum_{A(v_j) \neq v_s} w(v_i, v_j) * m/B\} + 2|V_f|m/B$$

Note that $|V_f|m/B$ is a constant. Therefore,

$$\begin{aligned} & \min T \\ \Rightarrow\ & \min C * \max_{v_s \in \mathbb{A}} \{ \sum_{A(v_i)=v_s} \sum_{A(v_j) \neq v_s} w(v_i, v_j) * m/B\} \\ & +2|V_f|m/B \\ \Rightarrow\ & \min \max_{v_s \in \mathbb{A}} \{ \sum_{A(v_i)=v_s} \sum_{A(v_j) \neq v_s} w(v_i, v_j) * m/B \\ \Rightarrow\ & \min |\{(v_i, v_j) | A(v_i) \neq A(v_j)\}| \end{aligned}$$

The solution of the optimal weighed k-partition problem in this configuration finds the partition that minimizes the number of edges across the partitions. As a result, the solution of the optimal weighed k-partition problem also solves the k-balanced partitioning problem. As the k-balanced partitioning problem has been proven to be NP-Hard in [8], the

Algorithm 1: Recovery Index Computation

```
    Input: Graph partitions $P_i$ on each worker node $N_i$
    Output: Recovery Index $RI$
1   foreach partition $P_i = (V_i, E_i)$ do
2       $min\_cost\_map \leftarrow \emptyset$
3       $min\_cost \leftarrow +\infty$
4       Construct weighted graph $G^*(P_i)$ for $P_i$
5       foreach possible $k$ do
6           $map \leftarrow \emptyset$
7           Partition $G^*(P_i)$ into $k$-weighted partition using METIS algorithm
8           foreach $P_j^*$ in the $k$-weighted partition do
9               $v_{max} \leftarrow \arg\max_{v_m \in V_j^* - V_f} \sum_{v_n \in V_j^* \cap V_f} w(v_m, v_n)$
10              foreach $v_n \in V_j^* \cap V_f$ do
11                  $map \leftarrow map \bigcup < v_n, v_{max} >$
12          Compute the time cost $T(map)$ of the $k$-weighted partition
13          if $T(map) < min\_cost$ then
14              $min\_cost \leftarrow T(map)$
15              $min\_cost\_map \leftarrow map$
16      $RI_i \leftarrow min\_cost\_map$
17  return $RI$
```

optimal weighed k-partition problem is therefore also NP-Hard. □

We shall further prove that the optimal weighed k-partition problem does not have any effective approximate solutions:

Theorem 2 (Inapproximability). *There does not exist a fully polynomial time (with respect to $\frac{|V^*|}{\epsilon}$) approximate algorithm that solves the optimal weighed k-partition problem with an approximate ratio $\frac{|V^*|^\alpha}{\epsilon^\beta}$ for any constants α and β where $\alpha < \frac{1}{2}$, unless P=NP.*

Proof. We again prove the theorem by reducing the k-balanced partitioning problem to our problem. Using similar techniques in the proof of Theorem 1, we construct an instance graph G of the optimal weighed k-partition problem for each instance graph $\mathcal{G}$ of the k-balanced partitioning problem. The vertices V and edges E of G are the same as those in G and the weight $w(e) = 1$ for every edge e. We let $V_f = \emptyset$ and choose proper parameters C, m and B such that $\frac{Cm}{B} \geqslant 1$. Therefore, the time

$$\begin{aligned} T &= C * \max_{v_s \in \mathbb{A}} \{ \sum_{A(v_i)=v_s} \sum_{A(v_j) \neq v_s} w(v_i, v_j) * m/B \} \\ &= \frac{Cm}{B} |\{(v_i, v_j) | A(v_i) \neq A(v_j)\}| \end{aligned}$$

We denote the edge set $\{(v_i, v_j) | A(v_i) \neq A(v_j)\}$ as E.

Suppose there exists a a fully polynomial time (with respect to $\frac{|V^*|}{\epsilon}$) approximate algorithm that solves the optimal weighed k-partition problem with an approximate ratio $\frac{|V^*|^\alpha}{\epsilon^\beta}$ for some constants α and β where $\alpha < \frac{1}{2}$. The time cost T of this approximate algorithm satisfies

$$T \leqslant (1 + \frac{|V^*|^\alpha}{\epsilon^\beta}) T_{OPT}$$

Algorithm 2: Single Node Failure Recovery

```
    Input: Graph partitions $P_i$ on each worker node $N_i$,
           the failed worker node $N_f$ with the partition $P_f$ on $N_f$
    Output: Recovered node $N_f$
1   Master node reads the recovery index $RI_f$ from its hard disk
2   Master node sends each of the vertex $v$ in $V_f$ and the neighbors of $v$ to the worker node $N_{RI_f(v)}$
3   Master node sends the starting recovery message to each of the worker node in $RI_f$ and waits for the response of finishing recovery from the worker nodes
4   foreach worker node $N_i$ in $RI_f$ do
5       foreach recovery superstep $s_j$ do
6           $N_i$ does the computation of the $j$th superstep for its received vertices
7           $N_i$ sends messages to the neighbors of its received vertices
8       $N_i$ sends the computed data to $N_f$
9   return $N_f$
```

That leads to

$$|E| \leqslant (1 + \frac{|V^*|^\alpha}{\epsilon^\beta}) |E_{OPT}|$$

which contradicts with the fact that there does not exist a a fully polynomial time approximate algorithm that solves the k-balanced partitioning problem with an approximate ratio $\frac{n^\alpha}{\epsilon^\beta}$. □

4.2 Recovery Index based Failure Recovery

Theorem 2 implies that effective approximate algorithms for the optimal weighed k-partition problem does not exist. To solve our problem, we therefore need to look for heuristic solutions. We adapt the widely used METIS [11] graph partition algorithm for the optimal weighed k-partition problem. Using the METIS graph partition algorithm, we can compute strategy of distributing the recovery jobs offline. We store the strategy in the hard disks of the master node of the recovery index. The recovery index RI is actually a mapping function for each of the partition P_i. Using RI_i, $v_j \in V_i$ should be migrated to the compute node $N_{RI_i(v_j)}$ once node N_i fails. The recovery index computation algorithm is outlined in Algorithm 1.

Having computed the recovery index, we now proceed to outline the algorithm of failure recovery using the index in Algorithm 2.

5. MULTIPLE NODES FAILURE RECOVERY

It is common for distributed graph processing systems that have a large number of compute nodes to have multiple nodes failing at the same time. All of traditional failure recovery methods like checkpoint based recovery and confined recovery require all the compute nodes to rollback to the most recent checkpoints to recompute the supersteps. However, these failure recovery methods suffer from the long recovery time. In this section, we present a novel failure recovery method for multiple node failures by extending our single node failure recover in Algorithm 2.

Following Theorem 1 and 2, we shall get the corollary that the corresponding optimal k-weighted partition problem for

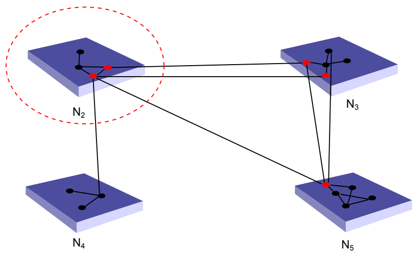

Figure 5: Cascading Failure Example

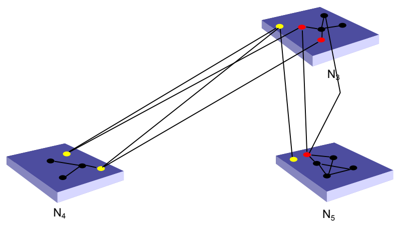

Figure 6: Iterative Recovery for Cascading Failures

the multiple node failure recovery is also NP-Hard and inapproximable, i.e.

Theorem 3 (Hardness of Multiple Node Failure Recovery Problem)**.** *Given a optimal k-weighted partition problem with respect to the multiple node failure recovery problem, we have*

1. *The optimal k-weighted partition problem for multiple node failures is NP-Hard.*

2. *There does not exist a fully polynomial time (with respect to $\frac{|V^*|}{\epsilon}$) approximate algorithm that solves the optimal weighed k-partition problem for multiple node failures with an approximate ratio $\frac{|V^*|^\alpha}{\epsilon^\beta}$ for any constants α and β where $\alpha < \frac{1}{2}$, unless P=NP.*

Proof. This theorem can be shown by proving that the solution of multiple node failure recovery problem implies a solution of single node failure recovery problem. This is obvious that for each instance of single node failure, it is also an instance of "multiple" node failures with only one failed node. □

Therefore, we also need to apply the heuristic algorithms to solve the multiple node failure recovery problem. We modify and extend our recovery algorithm (Algorithm 2) for single node failure to recover multiple node failures. We present the modification to Algorithm 2 by discussing the two types of multiple node failures, namely cascading failures and non-cascading failures.

5.1 Cascading Failures

Cascading failures are failures that occur when the recovery nodes themselves also fail when they are performing a recovery job for some other failed nodes. Suppose the first failed node is N_i and the second one is N_j. In this case, unlike traditional failure recovery methods, our method would not rollback those nodes that have not failed, as they would have already partially recovered the first failed node N_i. Instead, we let all active nodes that are running the recovery job for N_i stop and wait for the node N_j to recover. Using our method, we do not need to wait until N_j is fully recovered. In contrast, suppose N_j fails when the recovery of N_i is executed by s supersteps, we only need to recover N_j for the first s supersteps and then recover N_i and N_j in the same time. Note that for cascading failures, all failed nodes require the same number of supersteps to recover. Therefore, some supersteps of the recovery process can be concurrently

```
Algorithm 3: Cascading Failure Recovery
  Input: Graph partitions $P_i$ on each worker node $N_i$,
         the stack of already failed worker nodes
         $\mathcal{N} = \{< N_i, s_i >\}$ with the number of executed
         recovery supersteps, the newly failed node $N_f$,
         nodes have been used to execute recovery jobs
         $\mathcal{N}_r$
  Output: Recovered nodes $\mathcal{N} \cup N_f$
1  Master node reads the recovery index $RI_f$ from its hard
   disk
2  if $\mathcal{N} = \emptyset$ then
3      return Single_Node_Failure_Recovery($\mathcal{P}, N_f$)
4  else
5      $s \leftarrow \mathcal{N}.top().s$
6      Master node sends the message to all worker nodes
       in $\mathcal{N}_r$ to pause their recovery job
7      recover_nodes $\leftarrow \emptyset$
8      Assign_Recover_Node($v_f$)
9      foreach worker node $N_i$ in recover_nodes do
10         foreach $1 \leqslant j \leqslant s$ do
11             $N_i$ does the computation of the $j$th
               superstep for its received vertices
12             $N_i$ sends messages to the neighbours of its
               received vertices
13         $N_i$ sends the computed data to $N_f$
14     $\mathcal{N}.push(< N_f, s >)$
15     return Non-Cascading_Failure_Recovery($\mathcal{P}, \mathcal{N}$)
```

executed. Obviously, the number of executed recovery supersteps of newly failed node is never larger than that of previously failed nodes.

For example, consider the example shown in Figure 2 where N_1 fails. The data on N_1 are distributed to N_2, N_3, N_4 and N_5 shown in Figure 3. Suppose that N_2 fails when it is executing the recover jobs for N_1 as shown in Figure 5. Our method will detect that the failure of N_2 is a cascading failure related to N_1. Therefore, we will first recover N_2 to the status before it fails and later N_2 will be used again to recover N_1 shown in Figure 6.

In this paper, we use a stack to store the cascading failure information. The last failed node will be recovered first. Our algorithm is presented in Algorithm 3. The process of assigning recovery node is shown in Algorithm 5.

In Algorithm 3, when there is a cascading failure, we first stop the processing of all active nodes (Line 6). We distribute the recovery jobs for the most recent failed node to

Algorithm 4: Non-Cascading Failure Recovery

```
   Input: Graph partitions $P_i$ on each worker node $N_i$,
          the stack of already failed worker nodes $\mathcal{N}$,
          nodes have been used to execute recovery jobs
          $\mathcal{N}_r$
   Output: Recovered nodes $\mathcal{N}$
1  Master node reads the recovery index $RI_f$ from its hard
   disk
2  Master node sends the message to all worker nodes in
   $\mathcal{N}_r$ to pause their recovery job
3  $recover\_nodes \leftarrow \emptyset$
4  foreach node $v_i \in \mathcal{N}$ do
5      Assign_Recover_Node($v_i$)
6  foreach worker node $N_i$ in recover_nodes do
7      $N_i$ continues the computation supersteps for its
       received vertices
8      $N_i$ sends messages to the neighbours of its received
       vertices
9      $N_i$ sends the computed data back to the nodes in $\mathcal{N}$
10 return $\mathcal{N}$
```

several active nodes by reading the recovery index (Line 8 and Algorithm 5). The most recently failed node will be recovered for the same number of supersteps as the the earlier failed node (Line 9-13). Then the latest two failed nodes can be recovered together as neither of them needs to wait for the other one. Therefore, the two latest failures can be viewed as if the failure is a non-cascading failure after Line 14. Subsequently, we call the Non-cascading failure algorithm to recover both nodes together in Line 15. Note that Algorithm 3 needs to be invoked once a cascading failure occurs.

In a cascading failure containing two nodes, we can concurrently recover the two failed nodes after the second failed node has been partially recovered by re-executing the supersteps after the first node fails. Therefore, we can recursively apply the failure recovery method for non-cascading failures to recover multiple nodes together, which will be shown in the next subsection.

5.2 Non-Cascading Failures

The non-cascading failure, in contrast to cascading failures, implies that the failed nodes are not executing recovery jobs for other failed nodes. In this case, all the failed nodes fail independently. Note that the recovery for these failed nodes may not be isolated, as some failed nodes may require the same active node to run the recovery jobs. Therefore, to solve the conflict on distributing the recovery jobs, we design the non-cascading failures recovery algorithm (shown in Algorithm 4). Unlike the recovery algorithm for cascading failures, in Algorithm 4, when there is a non-cascading failure, we just need to stop current jobs (Line 2), reassign active nodes (Line 4, 5) and recover all the failed nodes in one go (Line 6-9).

5.3 Convergence of the Recovery

The convergence of the failure recovery methods, i.e. whether the recovery is guaranteed to finish, is always a critical problem. For distributed graph processing systems built on unstable compute nodes, it is common that the recovery pro-

Algorithm 5: Assigning Recovering Node

```
  Input: Node $v_i$
1 foreach node $v \in RI_i$ do
2     if $v \notin \mathcal{N}_r \cup \mathcal{N}$ then
3         $v' \leftarrow v$
4     else
5         $v' \leftarrow$ a random unused active node
6     Master node sends the starting recovery message to
      $v'$ for $v_i$ waits for the response of finishing recovery
      from $v'$
7     $recover\_nodes \leftarrow recover\_nodes \cup \{v'\}$
8     $\mathcal{N}_r \leftarrow \mathcal{N}_r \cup \{v'\}$
```

cess is always being executed for the continuously failed nodes. Like the existing checkpoint based recovery and confined recovery, our proposed partition based failure recovery method cannot guarantee the convergence of the recovery neither. However, our partition based recovery method requires weaker condition to be converged than the checkpoint based recovery method and confined recovery method. Therefore, our partition based recovery method has a larger probability to converge.

Suppose the recovery needs C supersteps to be executed. Both checkpoint based recovery method and confined recovery method would let all the active compute node to rollback to the most recent checkpoint to re-execute all the supersteps. Thus, the condition of convergence for the existing two recovery methods is:

Theorem 4. *The Checkpoint based recovery and confined recovery converge if and only if there exist C supersteps in the recovery processing such that any active compute node does not fail.*

The correctness of Theorem 4 is obvious. Note that Theorem 4 requires that none of the active compute nodes fail during the recovery is processed.

In contrast to the checkpoint based recovery method and confined recovery, our proposed partition based recovery method requires a much weaker condition to be converged, i.e.:

Theorem 5. *Partition based recovery method is converged if there exist $2C - 1$ supersteps in the recovery processing such that there is at most one cascading failure.*

Note that in practice, a compute node actually cannot fail more than once in a short time, as typically the distributed graph processing systems will replace the failed node with an unused active node instead of waiting for the failed node to restart. However, in fact these two nodes execute the same jobs and as a result, we treat these two nodes as one compute node for simplicity. Theorem 5 seems to require more supersteps than Theorem 4 does. However, the execution time for a single superstep of the partition based recovery method is shorter than that of the existing two recovery methods, due to the fact that the recovery job is distributed to multiple compute nodes. Therefore, the total executing time in the condition of Theorem 5 should be shorter than Theorem 4.

Before we prove that Theorem 5 is correct, we first show three important lemmas:

Lemma 1. *Non-cascading failures do not affect the convergence of partition based recovery method.*

Lemma 1 actually indicates that the execution of recovery for non-cascading failures are isolated. Therefore, the convergence of our partition based recovery method is not affected by the non-cascading failures.

Lemma 2. *In the cascading failures, a failed node cannot be recovered before any node failed after it.*

Lemma 2 directly follows Algorithm 3. The later failed node will be pushed into the cascading failure stack after the previously failed nodes. Algorithm 3 is executed for the top failure in the cascading failure stack. Therefore, the most recent failed node will be recovered first.

Lemma 3. *When a failed compute node is recovered, the nodes failed before this node have been recovered at this superstep.*

Now we prove Theorem 5 using these lemmas.

Proof. By Lemma 1, if there are no cascading failures during the $2C-1$ supersteps, our partition based recovery method converges.

Suppose there is only one cascading failure in the $2C-1$ supersteps. We denote the two failed nodes as N_1, N_2 to represent the first and second failed nodes. We consider the number s of supersteps that have been executed between the failures of N_1 and N_2. Obviously $0 < s < C$, otherwise N_2 must fail at least C supersteps after N_1 fails. Due to that there are no other cascading failures, especially related to N_1. Therefore, N_1 should have been recovered in C supersteps and this conclusion conflicts with the fact that the failures of N_1 and N_2 are cascading failures. Because there are no other cascading failures, the recovery for both N_1 and N_2 takes $s + s + (C - s) = C + s$ supersteps after N_1 fails. Here we incur s supersteps to recover N_1 before N_2 fails. Next, we need to incur s supersteps to recover N_2 to the state just before N_2 fails, according to Lemma 2. Thereafter we use $C - s$ supersteps to recover both N_1 and N_2. Note that $s \leqslant C - 1$. Thus, $C + s \leqslant 2C - 1$, indicating that both N_1 and N_2 will be recovered during the $2C - 1$ supersteps by Lemma 3. As a result, our partition based recovery method converges. □

Theorem 5 indeed implies that our partition based recovery method requires a much weaker condition to be converged than checkpoint based recovery and confined recovery. The convergence of our method is mainly affected by cascading failures. However, any multiple node failures will lead to the divergence of checkpoint recovery and confined recovery, as they need to rollback all the active compute nodes.

6. IMPLEMENTATIONS

We implement our partition-based failure recovery method on Apache Giraph [1], an open-source implementation of Pregel. Before delving into the implementation details, we first introduce some background of Giraph.

6.1 Giraph Overview

Giraph is an iterative graph processing system that supports large-scale graph analysis applications. It consists of three components: master, worker and zookeeper. The master is responsible for the coordinations among the workers such as assigning data to the workers, and coordinating synchronization. The workers perform vertex computations as well as message passing for each iteration. Zookeeper maintains various statuses that are shared among the master and workers, including computation statuses and worker health statuses. Giraph system is built on top of Hadoop infrastructure and each Giraph program is executed as a Map-only job. Figure 7 shows the runtime of Giraph. When a job starts, both the master and workers become active. The workers load graph partitions from HDFS after the master has written the partition assignment into zookeeper. Workers will then exchange partitions based on the assignment and start computations iteratively. During each iteration, a worker processes the messages received in the last iteration, execute compute function for each vertex in its partitions and sends messages to other workers. After the computation of all vertices is completed, the worker then writes the computation status into zookeeper and waits for all other workers to finish their computations. The master checks the computation statuses of all workers via zookeeper periodically and coordinates the synchronization when all the workers finish an iteration. Finally, the workers flush the results to HDFS and the job is completed.

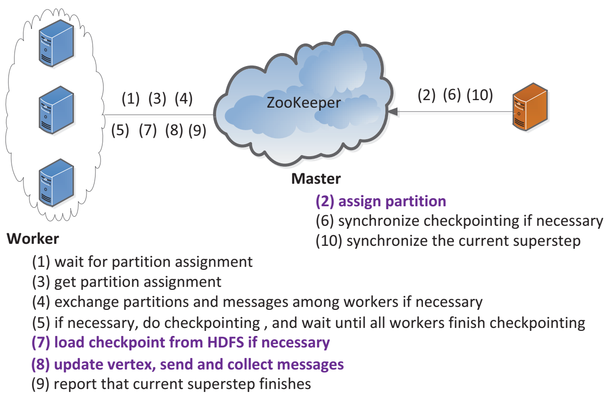
(1) (3) (4)
(5) (7) (8) (9)
ZooKeeper
(2) (6) (10)
Master
(2) assign partition
(6) synchronize checkpointing if necessary
(10) synchronize the current superstep
Worker
(1) wait for partition assignment
(3) get partition assignment
(4) exchange partitions and messages among workers if necessary
(5) if necessary, do checkpointing , and wait until all workers finish checkpointing
(7) load checkpoint from HDFS if necessary
(8) update vertex, send and collect messages
(9) report that current superstep finishes

Figure 7: Overview of Giraph System

Failure recovery in Giraph. The recovery mechanism implemented in Giraph is simply checkpoint based. At the beginning of each superstep, each worker will check whether it needs to do the checkpointing. If so, the workers will flush partitions as well as the messages received in the last superstep into HDFS as a checkpoint. When a worker fails, a new worker thread will be initiated to redo the computation for the current superstep, and the whole job can proceed smoothly. However, when a compute node fails, all the workers residing on that node become inactive and the master will terminate the whole job. During the recovery, a new compute node will be involved as a substitute for the failed one. A new job will then be launched where the master requests the workers to load partitions and messages from the latest checkpoint and continue the computation afterwards. In the current implementation of Giraph, the new job cannot be scheduled automatically. In fact, we have to manually launch a new job and request it to load data from the latest checkpoint.

6.2 Our Extensions to Giraph

We make two major extensions to Giraph to support our partition based recovery. First, we extend Giraph to support confined recovery. Confined recovery requires each worker to flush all its sending messages to the local disk at the end of each superstep. However, we require the workers to flush their own partitions as well. This is because the recovery in Giraph is performed as a new job and all the partitions are lost regardless whether they reside on a failed node or not. In the experiments, we observe the overhead of writing to local disk is negligible compared to the whole processing time. Suppose the latest checkpoint is made at the beginning of superstep i and a node failure occurs in superstep j ($j > i$). During the recovery, we divide the workers into two groups, and denote by NEW and OLD the workers in the newly-added node and original ones, respectively. Workers in NEW will load partitions and messages from the latest checkpoint stored in HDFS. Essentially, these workers rollback to the beginning of superstep i and restart the computation from that superstep. Workers in OLD read from local disk the partitions flushed at the end of superstep j in the original job as there is no need for them to rollback. When restarting from superstep i, the workers in NEW perform as per normal. Namely, they process the messages, execute compute function for their own vertices and send messages to the other workers. For the workers in OLD, the messages that will be sent from superstep i to j are already stored in the local disk. Hence, they read messages from the disk and forward them directly without performing any computation. After superstep j, all the workers will perform as usual.

The above processing ignores an important factor that there is no need to forward messages to the workers in OLD from superstep i to j as their partitions are the ones produced in superstep j. To achieve this, we maintain a list L of partition status in the zookeeper where each element is a pair of partition id and the most recent superstep accomplished by the partition. The list will be updated by the master. For any superstep k, each worker will first check partition status and forward a message only if its destination vertex belongs to partition p and there exists $s \leq k$ such that $(p, s) \in L$.

Second, we implement our partition based recovery on Giraph. The major difference between our approach and confined recovery is that we assign the partitions in a failed node to the workers based on our pre-computed recovery index. Therefore, we modify the partition assignment strategy for the master in Giraph. During the recovery, the master will assign a partition p to worker w iff the index indicates that. We note that our proposed recovery method can be applied to other distribute graph processing platforms, and we chose Giraph mainly as a platform to illustrate the idea and validate the performance.

7. EXPERIMENTAL STUDIES

We compare our partition based failure recovery mechanism with checkpoint based failure recovery mechanism on top of Giraph graph processing engine. The open source implementation of Giraph system is provided in Apache website [1] and we use the latest version-1.0.0.

7.1 Experiment Setup

The experimental study was conducted on our in-house cluster. The cluster includes 72 compute nodes, each of which is equipped with one Intel X3430 2.4GHz processor, 8GB of memory, two 500GB SATA hard disks and gigabit ethernet. 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 [1]. All the nodes are connected via three high-speed switches. Since Giraph runs as a MapReduce job on top of Hadoop, to adapt the Giraph environment to our application, we made the following changes to the default Hadoop configurations: (1) the size of virtual memory for each task is set to 4GB; (2) each node was configured to run one map task. We chose 40 nodes out of the 72 nodes in our experiment and among them, one node acted as the master for running Hadoop's NameNode, JobTracker daemons.

7.2 Benchmark Tasks and Datasets

We evaluate our partition based recovery via the execution of PageRank algorithm. As PageRank algorithm is not a traditional computation intensive task, we extend it into PageRankX where the computation of each vertex is repeated by X times. We conduct the experiments on two real-life large graphs.

• **LiveJournal.** LiveJournal is an online social network, in which the vertices represent the users and the edges represent the friendship relationship. LiveJournal graph [14] contains more than 4 million vertices and about 70 million directed edges.

• **Twitter.** Twitter dataset [2] is a segment of Twitter graph with more than 10 million vertices and 60 million edges.

We compare our proposed partition based recovery method (PBR) with the checkpoint based recovery method (CBR) using the benchmark tasks on these two datasets. By default, we evaluate the performance in term of recovery time and message passing over LiveJournal dataset by running PageRank algorithm.

7.3 Effect of Message Caching

In this study, we report the performance of using message caching mechanism in each superstep. We show that similar to the confined recovery method, our partition based recovery method only incurs a small overhead over the checkpoint based recovery method. Figure 8 illustrates the average running time of each superstep for both PBR and CBR for different number of compute nodes. PBR takes slightly more time compared with CBR as PBR needs to allow all compute nodes to cache all the sent messages. Moreover, when the number of used nodes is increased, the average running time is also increased for both methods. This is caused by the fact that communication among the nodes is also increased when there are more compute nodes.

7.4 Single Failure Recovery

We show the experimental results on the single failure recovery in this subsection. Figure 9 and 10 present the experimental results on the LiveJournal dataset.

Figure 9 presents the total number of messages that have been sent in order to recover a single failed node. The number of sent messages in CBR is almost 7 times than that in PBR. This is because in PBR, each active compute node only needs to send messages to the failed node. On the contrary, in CBR, all the computation and message sending are

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

[2]http://grafia.cs.ucsb.edu/sedge/dataset.html

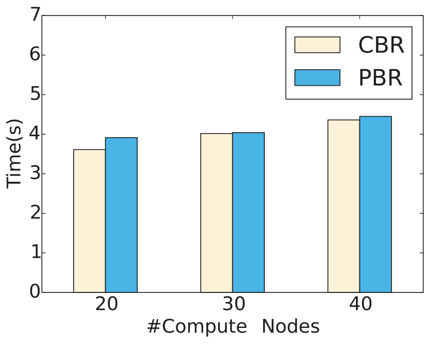

Figure 8: Average running time of each superstep

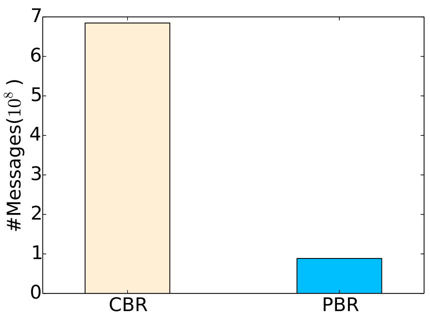

Figure 9: Number of messages incurred for recovering a single node failure

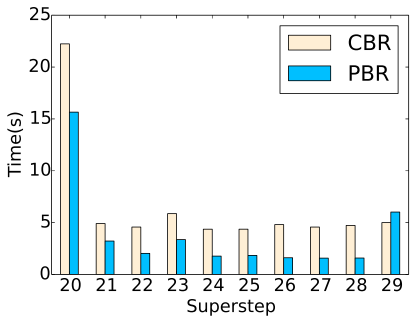

Figure 10: Recovery time in each superstep for single node failure

required to be re-executed. Therefore, CBR needs to send a lot more messages to recover a single node than PBR.

Figure 10 reports the recovery in each superstep for both PBR and CBR. In Figure 10, the failure occurs at superstep 29. As the checkpoint interval is 10 supersteps, all active nodes recover the failed node from superstep 20. Thus, both PBR and CBR read the data of checkpoint at superstep 20. In PBR, as partitions that were assigned to the failed node are lost, workers that are responsible for such partitions need to read them from the HDFS while other partitions are loaded from local disks. However, In CBR, all partitions are required to fetch from HDFS. Hence, PBR spends less time than CBR in superstep 20. During supersteps 21 to 28, PBR distributes the recovery tasks to multiple active nodes while CBR only recovers the failed node by assigning a new node only. As a result, the processing time of PBR is much less than that of CBR. In superstep 29, all the active nodes finish the recovery tasks. Since PBR requires the recovered data to be migrated back to the failed node, regarding the communication cost is comparable to the computation cost, we find that PBR spends fairly more time than CBR in this superstep. However, the total processing time of PBR is much less than that of CBR, due to the distribution of recovery processing.

7.5 Computation-Intensive Task

In this subsection, we show that our proposed PBR method can be used for a variety of computational intensive tasks. As it is fairly difficult to design real world tasks with a specific CPU computation time, we vary the computational intensity by repeatedly running the computational task on each node for multiple times. In this experiments, we execute the computation function over each vertex for 100, 500, 1000 and 2000 times on both LiveJournal dataset and Twitter dataset.

In Figure 11, the total processing time of PBR is much less compared with that of CBR. Furthermore, when the number of times the computational tasks being run increases, the gap between the total recovery time of PBR and CBR grows larger. This is because our PBR method can benefit more by distributing the recovery task when the CPU computation time is significantly larger than the communication time. PBR is about 7 times faster than the CBR method on the LiveJournal dataset.

Figure 12 shows similar trends as shown in Figure 11. The total processing time of PBR is also less compared with that of CBR for most of the cases. PBR is about 6 times faster than the CBR method on the Twitter dataset.

7.6 Non-Cascading Failure Recovery

In this subsection, we compare the performance of our proposed PBR method to the CBR method on recovering multiple non-cascading failures.

Figure 13 shows the comparison between PBR and CBR for the recovery processing time when multiple nodes fail at the same superstep. We have two observations. First, PBR spends less time to recover non-cascading failures than CBR. The reason is similar to that of Figure 11 and 12 as the multiple non-cascading failures on several compute nodes can be viewed as a large failed compute node that contains all the data on those failed nodes. Second, when the number of non-cascading failures increases, the processing time of both PBR and CBR almost remains flat. This can be explained by the fact that for both PBR and CBR, the non-cascading failures are recovered concurrently on those failed nodes. Therefore, the time of the recovery processing should be almost independent to the number of failed nodes. In fact, the recovery processing time depends on the number of supersteps to recover these failed nodes and the communication time between the nodes.

Figure 14 shows the number of sent messages during the recovery processing for both PBR and CBR. We can observe that the number of sent messages for CBR remains quite flat while the number of sent messages for PBR increase slightly as the number of failed nodes increases. CBR requires all the active nodes to rollback to the most recent checkpoint and every node to send the message to all its neighbors. Therefore, no matter how many non-cascading failures occur, the total amount of messages needed to recovery these failed nodes are the same. However, for PBR, the messages are sent to the failed nodes, and hence, the number of messages almost linearly increases with the number of failed non-cascading nodes.

7.7 Cascading Failure Recovery

We compare the performance of our proposed PBR method with CBR method on cascading node failures. Figure 15 shows the recovery time while Figure 16 presents the num-

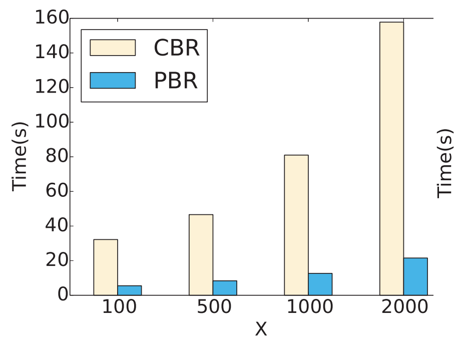

Figure 11: Total recovery time on LiveJournal dataset

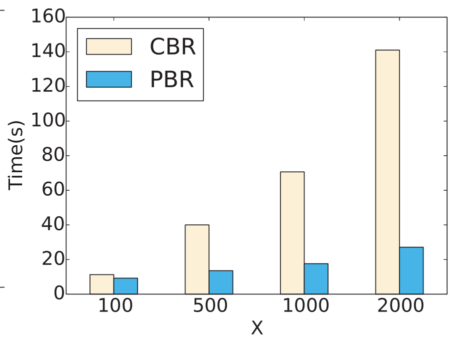

Figure 12: Total recovery time on Twitter dataset

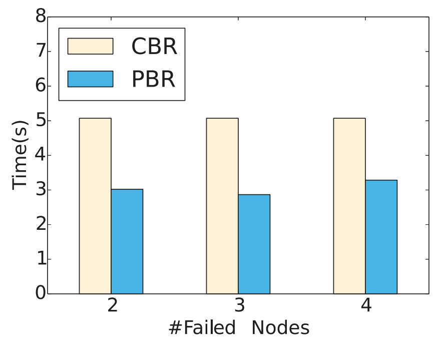

Figure 13: Average Recovery time in each recovery superstep for non-cascading failures

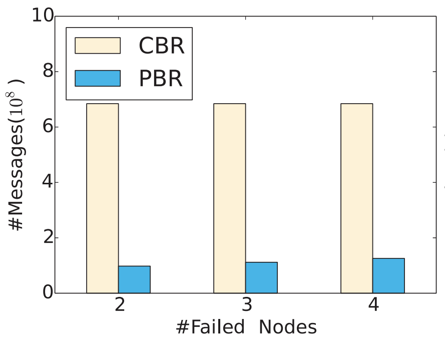

Figure 14: Number of messages sent for recovery of non-cascading failures

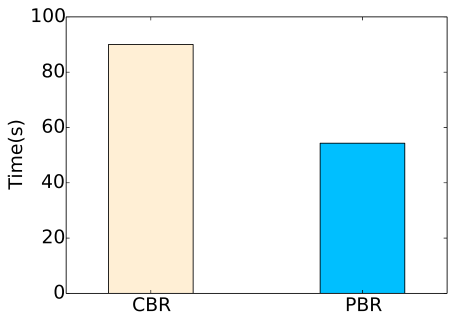

Figure 15: Recovery time for cascading failures

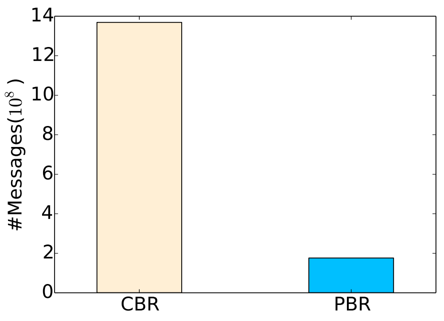

Figure 16: Number of messages sent for recovery of cascading failures

ber of sent messages for recovering a cascading failure with two node failures. In our case, the first node N_1 fails at superstep 29. During the recovery to superstep 29 from the latest checkpoint (superstep 20) for N_1, the second node N_2 fails. Let $\mathcal{P}_1$ be the partitions that are assigned to N_1 before the first failure, and $\mathcal{P}_2$ be the partitions that are assigned to N_2 before the second failure. When the failure of node N_1 emerges, both PBR and CBR rollback to superstep 20, and execute a single node failure recovery. For the second failure of node N_2, as partitions in $\mathcal{P}_1| - \mathcal{P}_2$ have been recovered to superstep 29, PBR only recovers data from partitions $\mathcal{P}_2 - \mathcal{P}_1$. When a cascading failure happens, although the overall recovery time increases, the time gap in cascading failure recovery basically follows the same trend in the single node failure recovery, that is, the recovery time of PBR is nearly half of the recovery time of CBR for this cascading failure (shown in Figure 15).

Figure 16 shows the total amount of sent messages to recover the cascading failures. As PBR only needs to send messages to all the failed nodes, the number of sent messages for PBR is significantly less than that of CBR, which reduces the communication time in PBR.

8. RELATED WORK

Recently, a wide range of parallel graph processing systems such as Pregel [16], Trinity [21], GraphLab [15, 9], GraphX [23] have been introduced to support graph computations over large-scale graph data. Both Pregel and Trinity follow the Bulk Synchronous Parallel (BSP) computation model, where a graph analysis job is executed in a sequence of iterations, i.e., supersteps, and a synchronization among all the compute nodes is performed at the end of each superstep. Alternatively, GraphLab adopts asynchronous processing model where compute nodes have shared access to other nodes during the computation.

Existing systems aim to support large-scale graph analysis applications over hundreds or even thousands of compute nodes. To achieve this goal, one of the major challenges is to handle the node failure, which is inevitable during the execution of a long-running graph application. Various recovery mechanisms are proposed to tolerate the failure of compute nodes in a distributed environment, among which rollback recovery is widely adopted [5, 6, 10, 3, 17, 22]. As surveyed in [7], rollback recovery is classified into two categories: checkpoint based and log based.

To the best of our knowledge, most existing distributed graph processing systems adopt checkpoint based rollback recovery such as Giraph [1], GraphLab [15], PowerGraph [9], GPS [19], Mizan [12]. Pregel proposes confined recovery which is a hybrid recovery mechanism of checkpoint based 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 adopts log (called lineage) based failure recovery, and utilizes a novel mechanism called resilient distributed dataset (RDD) to speedup failure recovery. However, when a node fails, graph data lying in this node still need to be recovered. Regarding our partition based recovery mechanism that distributes graph data residing on failed nodes to different compute nodes for parallel processing, our method is orthogonal to these systems.

Another direction closely related to our work is the partitioning methods on large-scale graphs. In this paper, we adopt a widely used partitioning algorithm called METIS [11]. METIS has been extended in several aspects, including partitioning power-law graph [2], multi-threaded graph partitioning [13] and dynamic multi-constraint graph partitioning [20]. In practice, METIS has been adopted in other distributed graph processing systems such as PowerGraph.

9. CONCLUSION

In this paper, we present a novel partition based failure recovery method to accelerate failure recovery processing. Different from traditional checkpoint based recovery and confined recovery, our recovery method distributes the recovery jobs to multiple compute nodes such that the recovery processing can be executed concurrently. We have proven that the partition based failure recovery problem is both NP-Hard and inapproximatable even for a single node failure. We adapt the METIS algorithm [11] based heuristic method to pre-compute the recovery index for each of the compute nodes. The recovery index is used to distribute the recovery jobs of the corresponding failed compute nodes to other nodes. Furthermore, our proposed failure recovery method is also applicable to multiple node failures including both cascading failures and non-cascading failures. We implement our recovery method and conduct extensive experiments on the widely used Giraph system to validate the performance of our proposed method. The experimental results show that our proposed partition based failure recovery method outperforms existing recovery methods especially on computational intensive tasks.

10. REFERENCES

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

[2] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDPS*, pages 10–pp. IEEE, 2006.

[3] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1):3:1–3:44, Mar. 2008.

[4] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[5] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *SIGMOD*, pages 265–276, 2011.

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

[7] 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):375–408, Sept. 2002.

[8] A. Feldmann. Fast balanced partitioning is hard even on grids and trees. In *Mathematical Foundations of Computer Science 2012*, volume 7464, pages 372–382. 2012.

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

[10] N. Gupta, A. Demers, J. Gehrke, P. Unterbrunner, and W. White. Scalability for virtual worlds. In *ICDE*, pages 1311–1314, 2009.

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

[12] 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 *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[13] D. LaSalle and G. Karypis. Multi-threaded graph partitioning. In *27th IEEE International Parallel & Distributed Processing Symposium*, 2013.

[14] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[15] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.

[16] 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 Conference*, pages 135–146, 2010.

[17] J.-A. Quiané-Ruiz, C. Pinkel, J. Schad, and J. Dittrich. Raft at work: Speeding-up mapreduce applications under task and node failures. In *SIGMOD*, pages 1225–1228, 2011.

[18] R. Salakhutdinov, A. Mnih, and G. E. Hinton. Restricted boltzmann machines for collaborative filtering. In *ICML*, pages 791–798, 2007.

[19] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Technical Report (Stanford)*, 2012.

[20] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

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

[22] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(3):204–226, 1985.

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