A Fault-Tolerant Framework for Asynchronous Iterative Computations in Cloud Environments

Zhigang Wang[1] Lixin Gao[2] Yu Gu[1] Yubin Bao[1] Ge Yu[1]

[1]Northeastern University, China
[2]University of Massachusetts Amherst, USA
wangzhiganglab@gmail.com, lgao@ecs.umass.edu, {guyu,baoyubin,yuge}@mail.neu.edu.cn

Abstract

Many graph algorithms are iterative in nature and can be supported by distributed memory-based systems in a synchronous manner. However, an asynchronous model has been recently proposed to accelerate iterative computations. Nevertheless, it is challenging to recover from failures in such a system, since a typical checkpointing based approach requires many expensive synchronization barriers that largely offset the gains of asynchronous computations.

This paper first proposes a fault-tolerant framework that performs recovery by leveraging surviving data, rather than checkpointing. Our fault-tolerant approach guarantees the correctness of computations. Additionally, a novel asynchronous checkpointing method is introduced to further boost the recovery efficiency at the price of nearly zero overhead. Our solutions are implemented on a prototype system, *Faiter*, to facilitate tolerating failures for asynchronous computations. Also, *Faiter* performs load balancing on recovery by re-assigning lost data onto multiple machines. We conduct extensive experiments to show the effectiveness of our proposals using a broad spectrum of real-world graphs.

Categories and Subject Descriptors H.3.4 [*System and Software*]: Distributed systems

General Terms Algorithms, Design, Performance, Theory

Keywords fault-tolerance, iterative algorithms, distributed memory-based systems, asynchronous computations

1. Introduction

Iterative graph algorithms have been widely used in numerous applications with billion-node graphs. In order to efficiently handle large graphs, distributed systems have been developed [1, 2, 15, 23, 31], most of which mainly focus on memory-based computations [2, 15, 31], including Pregel, Spark and Maiter. Fault-tolerance is one of the core components in such systems due to the following two reasons. First, most distributed systems usually run on a cluster consisting of commodity machines, and failures on these machines may occur frequently, especially when a large number of computational nodes are required. Second, iterative computations typically require many iterations, which inevitably increases the likelihood of encountering failures.

For simplicity, existing systems usually employ a synchronous model to perform computations through a series of iterations separated by explicit global barriers. Within one iteration, computational nodes coordinate with each other to synchronize the progress. Nodes running faster thereby need to block themselves so that they can wait for others to complete computations. Because of the expensive synchronization overhead, asynchronous systems, such as Maiter [31] and its variants [26, 27], have been recently proposed to accelerate the convergence speed of iterative algorithms by removing synchronization barriers. The issue we investigate in this paper is thereby to find an efficient fault-tolerant solution tailored for Maiter.

Challenges: Currently, although many efforts have been devoted to fault-tolerance, most of them focus on synchronous computations, such as checkpointing [15] and its variants [20, 24, 25] used in Pregel-like systems, and lineage [28] employed in Spark. These techniques are far from ideal for asynchronous computations. First, the checkpointing solution typically archives data periodically and hence any failure can be recovered by rolling back to the most recent available checkpoint. However, archiving data requires an explicit barrier among nodes to coordinate the progress, and computing work must be suspended. That exposes the asynchronous engine to the same inefficiency of a synchronous engine that the former is trying to address. On the other hand, the basic idea behind lineage is tracking the coarse-grained dependency among data sets instead of data themselves, in order to save the storage space and network bandwidth re-

SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.
© 2016 ACM. ISBN 978-1-4503-4525-5/16/10...$15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987552

quired by checkpointing. It, however, lacks built-in support for fine-grained updates in asynchronous systems.

The well-known asynchronous system GraphLab [14] removes the explicit barrier requirement for checkpointing, but computations are still suspended when they are in conflict with the checkpointing operations.

Our contributions: This paper first proposes a novel failure recovery solution without rolling back, referred to as FR-WORB. Upon failures, FR-WORB preserves data on surviving nodes without rolling back. A special restarting point is automatically constructed where computations interrupted by failures can be continued. We prove the correctness of our solution, i.e., the continued computations will converge to the same point as if failures have never occurred. FR-WORB can leverage data on surviving nodes to potentially accelerate recovering lost data, while simultaneously keeping refining surviving data without pausing. Intrinsically, FR-WORB is a reactive approach where no checkpoint is written before failures. Thus, there is no overhead for checkpointing. There exist some recently published reactive approaches [7, 17, 19, 22]. However, they either consume large amounts of memory [32], or raise nontrivial challenges for users to design a complex recovery function [24].

Additionally, we design an improved solution with asynchronous checkpointing, termed FR-WAC, where recovery computations of the lost data can be performed from the most recent available checkpoint instead from scratch. As recomputing work is reduced, FR-WAC can further boost the recovery efficiency. In particular, archiving operations are separated from computations and hence neither synchronization barriers among nodes nor coordination between data persistence and computing work is required. The overhead caused by checkpointing is thereby nearly zero and the impact on asynchronous engine can be negligible.

Finally, we develop a prototype system *Faiter* on top of the asynchronous system Maiter, to implement our fault-tolerant methods. In the system design, we perform load balancing on recovery. By assigning recovery workload to multiple nodes, the recovery time can be significantly dropped. Key issues, such as the data assignment policy and how to efficiently route messages to correct targets after changing the data placement, are discussed in detail.

The major contributions are summarized as below.

- We present a novel fault-tolerant solution FR-WORB for asynchronous systems. Since no checkpoint is written before failures, the overhead caused is zero. Meanwhile, FR-WORB provides prominent performance because data on surviving nodes perform computations progressively without rolling back or pausing and can be utilized to accelerate recovering lost data on failed nodes.
- An improved solution FR-WAC is proposed to further boost the recovery performance by recomputing lost data from the most recent available checkpoint instead from scratch. FR-WAC employs an asynchronous checkpointing mechanism to reduce its overhead to nearly zero, which is naturally suitable for asynchronous systems.
- We propose a load balancing solution for recovering lost data by assigning the recovery workload on a failed node to multiple nodes for efficiency.

Paper organization: The remainder of this paper is organized as follows. Section 2 introduces preliminaries about asynchronous computations and the challenges of tolerating failures. Section 3 presents our failure recovery methods and proves the correctness. Section 4 describes the system design of *Faiter*, especially for load balance. Section 5 reports our experimental studies. Section 6 highlights the related work. Finally, we conclude this work in Section 7.

2. Preliminaries

Currently, distributed iterative graph processing systems usually keep data in main memory for I/O-efficiency [8, 9, 14, 15, 31], and many of them, like Maiter [31], employ an asynchronous computation model [8, 14, 31] to accelerate computations. In the following, we briefly review the asynchronous memory-based system taking Maiter as an example, and then discuss the challenges of tolerating failures.

2.1 Asynchronous Memory-based System: Maiter

We model a graph as a directed graph G=(V,E), where V is a set of vertices and E is a set of outgoing edges (pairs of vertices). Given an edge (i,j), i/j is the source/destination vertex. The in-neighbors of i as $\Gamma^{in}(i)$ is a set of vertices that have an edge linking to i, and out-neighbors as $\Gamma^{out}(i)$ is a set of vertices that i has an edge to link. The in/out-degree is thereby defined as the number of in/out-neighbors, i.e., $|\Gamma^{in}(i)|$/$|\Gamma^{out}(i)|$. In distributed systems, G is partitioned onto multiple computational nodes to be processed in parallel.

Iterative graph algorithms can be naturally implemented in a synchronous system through a sequence of iterations separated by explicit barriers. Typically, the workload at the (k+1)-th iteration consists of computing state v_j^{k+1} for any vertex $j \in V$ by consuming messages received at the k-th iteration, and sending new messages based on v_j^{k+1}. Eq. (1) shows it mathematically, where c_j is an algorithm-specified constant. $g_{\{i,j\}}$ and $\oplus$ are user-defined functions used to generate a message from i to j based on v_i and compute vertex states, respectively. In general, iterations terminate until a fixed point is reached, i.e., every v_j does not change between two consecutive iterations.

$$v_j^{k+1} = c_j \oplus \sum_{i \in \Gamma^{in}(j)} \oplus g_{\{i,j\}}(v_i^k) \quad (1)$$

Take the PageRank [4] algorithm as an example. PageRank computes a score, i.e., v_j, for every web page j to evaluate its importance. At the k-th iteration, every web page i sends its tentative score divided by its out-degree along

outgoing links, i.e., $g_{\{i,j\}}(v_i) = d \cdot \frac{v_i^k}{|\Gamma^{out}(i)|}$, where d is a user-defined decay factor and $0 < d < 1$. At the next iteration, a new score is computed by summing up received values, i.e., $v_j^{k+1} = (1-d) + \sum_{i \in \Gamma^{in}(j)} (d \cdot \frac{v_i^k}{|\Gamma^{out}(i)|})$. Here, '$\oplus$' is '+' and $c_j = (1-d)$ for any vertex j.

Maiter, on the other hand, employs a delta-based asynchronous computation model where messages are generated based on the "change" of v_i, to avoid repeatedly processing messages from unchanged v_i, and synchronization barriers are removed to eliminate blocking overheads.

Specifically, when $g_{\{i,j\}}(x)$ has the *distributed* property over '$\oplus$', and '$\oplus$' has the *communicative* and *associate* properties, iterative computations can be performed as follows. For vertex j on node $N(j)$ where j resides, it carries two values: v_j and Δv_j, where Δv_j indicates the "change" of v_j since j's last update. Δv_j is computed by accumulating delta-based messages $g_{\{i,j\}}(\Delta v_i)$ from in-neighbors in the $\oplus$ manner, as shown in Eq. (2). At anytime, as shown in Eq. (3), j is possibly scheduled to update its v_j by consuming Δv_j. Δv_j is further forwarded to out-neighbors and then reset to $\mathbf{0}$ so that it is cleared and can accumulate newly received messages. $\mathbf{0}$ is an abstract zero value satisfying that $x \oplus \mathbf{0} = x$ and $g_{\{i,j\}}(\mathbf{0}) = \mathbf{0}$. For PageRank, it is 0.

$$receive: \begin{cases} When\ receiving\ g_{\{i,j\}}(\Delta v_i)\ from\ N(i); \\ \quad \Delta v_j \leftarrow \Delta v_j \oplus g_{\{i,j\}}(\Delta v_i), i \in \Gamma^{in}(j); \end{cases} \tag{2}$$

$$update: \begin{cases} If\ \Delta v_j \neq \mathbf{0} \\ \quad v_j \leftarrow v_j \oplus \Delta v_j; \\ \quad \forall h \in \Gamma^{out}(j),\ If\ g_{\{j,h\}}(\Delta v_j) \neq \mathbf{0} \\ \quad\quad send\ g_{\{j,h\}}(\Delta v_j)\ to\ N(h); \\ \quad \Delta v_j \leftarrow \mathbf{0}; \end{cases} \tag{3}$$

In Maiter, $g_{\{i,j\}}(\Delta v_i)$ is always closer to $\mathbf{0}$ than Δv_i. Thus, after performing Eqs. (2) and (3), asynchronous computations converge when every $\Delta v_j = \mathbf{0}$. In particular, the initial input values of v_j and Δv_j are given as $v_j^0 = \mathbf{0}$ and $\Delta v_j^0 = c_j$, to guarantee that an algorithm can converge to the same fixed point as achieved in the synchronous model. Besides, it has been validated that priority scheduling, which uses a priority (i.e., Δv_j) to determine the order of updating v_j, can accelerate the convergence speed [30], because vertices with large "change" play an important role in determining v_j.

2.2 Fault-tolerance for Maiter

Failures may be encountered when running computations in a distributed system. Till now, most systems, including Maiter, employ a Master-Slave framework where a master node is in charge of monitoring the cluster health by periodically checking status information collected from slave nodes. Obviously, the master node, network and slave nodes may fail, but this paper focuses on the last two only because the whole system will crash when the master node fails. A slave node is marked as failed by the master if the elapsed time since its last report exceeds a threshold.

It is challenging to recover failures in an asynchronous memory-based system. Upon a failure event, the memory-resident data on failed nodes are lost. To recover them and continue computations, a conventional approach is to checkpoint by archiving data periodically beforehand and restart computations from the last available checkpoint [15, 20, 24, 25]. However, existing checkpointing methods largely degrade the performance of asynchronous computations, as vertex updates must be paused when archiving data, and global synchronization barriers are usually required.

3. Failure Recovery for Maiter

This section introduces two failure recovery methods for Maiter. In particular, we describe how to guarantee that they can converge to the same fixed point as *failure-free* (i.e., no failure occurs) execution.

3.1 Failure Recovery Methods

Upon any failure at time t_f, the master node immediately replaces failed nodes with standby ones and then notifies replacements to reload the lost vertices and edges. Afterwards, one of our failure recovery methods (Sections 3.1.1 and 3.1.2) is invoked to recover vertex states on failed nodes and then continue computations, until a fixed point is reached.

3.1.1 Failure Recovery without Rolling Back (FR-WORB)

Assume no checkpoint is archived before failures. In this scenario, a traditional way of tolerating failures is to roll back every vertex state value v_j to $v_j^0 = \mathbf{0}$ and hence $\Delta v_j^0 = c_j$. It is referred to as FR-Scratch, since every v_j is recomputed from scratch. FR-Scratch is inefficient because workload on surviving nodes before t_f is discarded, and re-performing it wastes computation and communication resources.

By contrast, we present a failure recovery method where lost vertices on failed nodes only are recomputed from scratch and updates over surviving vertices can keep going without rolling back, referred to as FR-WORB.

Compared with FR-Scratch, FR-WORB avoids recomputing surviving vertex states, but a key issue is how to guarantee the correctness, i.e., it can converge to the same fixed point as reached in FR-Scratch, because surviving vertex states are not rolled back to $\mathbf{0}$. We solve this problem by designing a special restarting point and the basic idea behind it is to utilize available data as much as possible. We denote by $\tilde{v}_j$ and $\Delta\tilde{v}_j$ the state value and delta value in FR-WORB, respectively, to distinguish them from ones before failures. In the restarting point, for $\tilde{v}_j^0$, it equals to v_j^f if j resides on surviving nodes, where v_j^f is the state value of j at t_f. Otherwise, it is $\mathbf{0}$. Let V_N stand for a set of vertices residing on

node N. Eq. (4) shows that mathematically, where $\mathbb{N}_F$ and $\mathbb{N}_S$ represent the failed node set and surviving node set, respectively. On the other hand, $\Delta\tilde{v}_j^0$ in the restarting point is given by $(\tilde{v}_j^1 \ominus \tilde{v}_j^0)$. Here, $\ominus$ is an abstract operation satisfying that $g_{\{i,j\}}(x)$ has the *distributed* property over '$\ominus$', $x \ominus x = \mathbf{0}$, and $(x \oplus y) \ominus z = x \oplus (y \ominus z)$. For PageRank, $\ominus$ is '-'. $\tilde{v}_j^1$ used above is derived from its newly initialized in-neighbor state $\tilde{v}_i^0$ using the synchronous model (Eq. (1)). Eq. (5) gives a mathematical description about constructing $\Delta\tilde{v}_j^0$.

$$\tilde{v}_j^0 = \begin{cases} \mathbf{0}, j \in \bigcup\limits_{N \in \mathbb{N}_F} V_N \\ v_j^f, j \in \bigcup\limits_{N \in \mathbb{N}_S} V_N \end{cases} \tag{4}$$

$$\Delta\tilde{v}_j^0 = \tilde{v}_j^1 \ominus \tilde{v}_j^0 = \left(c_j \oplus \Big(\sum_{i \in \Gamma_j^{in}} \oplus g_{\{i,j\}}(\tilde{v}_i^0) \Big) \right) \ominus \tilde{v}_j^0 \tag{5}$$

As analyzed above, $\tilde{v}_j^1$ can be easily calculated by running a synchronous iteration. However, in context of asynchronous computations, flushing operations are required to guarantee that operations can be performed in right order. That is, as a **preprocessing phase** before recovery, at t_f, any existing $g_{\{i,j\}}(\Delta v_i)$ in network and Δv_j on surviving nodes must be flushed. This is because $g_{\{i,j\}}(\tilde{v}_i^0)$ in Eq. (5) is different from $g_{\{i,j\}}(\Delta v_i)$ in Eq. (2) (normal asynchronous computation), and the two types of messages must be separated to get correct $\tilde{v}_j^1$. Also, the current Δv_j on surviving nodes perhaps stores the value based on $g_{\{i,j\}}(\Delta v_i)$, that needs to be cleared, i.e., being reset to $\mathbf{0}$, in order to accumulate newly received messages $g_{\{i,j\}}(\tilde{v}_i^0)$.

Flushing $g_{\{i,j\}}(\Delta v_i)$ needs a three-step operation. First, computing threads are suspended to stop generating new messages. Second, at each sender side, messages in sending buffer are flushed and then an EOF notification is broadcasted to any node subsequently. Third, each receiver side blocks itself until it has received all EOF notifications from sender sides or the waiting time exceeds a given threshold. After flushing $g_{\{i,j\}}(\Delta v_i)$, Δv_j is reset to $\mathbf{0}$ locally to clear its value. In particular, when calculating $\Delta\tilde{v}_j^0$, flushing $g_{\{i,j\}}(\tilde{v}_i^0)$ is also required to guarantee that each vertex has received all possible messages from in-neighbors. This can be done using the same way with flushing $g_{\{i,j\}}(\Delta v_i)$.

Now, FR-WORB can continue asynchronous computations as usual from the restarting point $(\tilde{v}_j^0, \Delta\tilde{v}_j^0)$. And we can prove its correctness in Theorem 1 in Section 3.2. Intuitively, FR-WORB is more efficient than FR-Scratch, since the former preserves completed workloads on surviving nodes and further uses them to benefit recovering lost ones on failed nodes, instead of abandoning them as the latter. Also, in FR-WORB, when recovering lost vertex states, vertices on surviving nodes can be refined continuously without pausing.

Last but not least, new failures, termed **cascading failures**, may occur when constructing $\tilde{v}_j^0$ and $\Delta\tilde{v}_j^0$. Let F_{cur} and F_{cas} be the current failures and cascading failures. In this scenario, current work, including partially constructed $\Delta\tilde{v}_j^0$ and possibly broadcasted $g_{\{i,j\}}(\tilde{v}_i^0)$ for F_{cur}, becomes invalid because new data are lost caused by F_{cas}. FR-WORB thereby abandons current work and then re-constructs $\tilde{v}_j^0$ and $\Delta\tilde{v}_j^0$ to recover F_{cur} and F_{cas} together. Note that it is unnecessary to perform additional flushing operations to discard current work, because it can be done in the **preprocessing phase** before re-constructions for F_{cur} and F_{cas}.

3.1.2 Failure Recovery with Asynchronous Checkpointing (FR-WAC)

Because no checkpoint is archived during *failure-free* execution, for FR-WORB, the only way of initializing lost vertex states is to reset them to the initial value $\mathbf{0}$. The heavy recovery workload impacts the performance, even though surviving vertex state values can be used.

Now we introduce a method to further boost the recovery efficiency by archiving data during *failure-free* execution. It is inspired by checkpointing. In general, the checkpointing method archives data as checkpoint periodically. Upon failures, all nodes load the last available checkpoint and then recover computations from that point, instead from scratch, to reduce the recovery workload. However, existing techniques need to block updating vertex state v_j when archiving it. This requires a synchronization barrier to coordinate the progress of each node, which largely degrades the performance of *failure-free* execution. Unlike them, our method asynchronously archives data, termed FR-WAC. FR-WAC avoids recovering lost data from scratch compared with FR-WORB, while leading to nearly zero performance degradation of *failure-free* execution.

In FR-WAC, each node individually archives its local vertex state values v_j based on a user-specified interval τ. There is no global synchronization barrier [15] or any other protocols [14] to coordinate the progress, which eliminates the expensive synchronization overheads. Furthermore, on one node, a separate thread is launched to accomplish the archiving operation, which runs in fully parallel with the message-receiving (Eq. (2)) and vertex-updating (Eq. (3)) threads. The asynchronous computation is thereby performed progressively without pausing.

Different from FR-WORB, FR-WAC initializes lost vertex states using checkpoint data, i.e., $\tilde{v}_j^0 = v_j^x$, if $j \in \cup_{N \in \mathbb{N}_F} V_N$. Here, v_j^x is kept in the most recent available checkpoint archived at t_x by node N. After that, we calculate $\tilde{v}_j^1$ and $\Delta\tilde{v}_j^0$ using the same way (including flushing operations) as FR-WORB. Since v_j^x is closer to the fixed point than $\mathbf{0}$, FR-WAC potentially outperforms FR-WORB.

For **cascading failures**, FR-WAC also abandons current work and re-constructs the restarting point. But new lost vertex states are reset to values in the checkpoint.

Usually, for existing work, τ is a key parameter to balance the tradeoff between the recovery efficiency and archiving

runtime, and it is a non-trivial task to set a reasonable value prior to computations. But in this paper, we will experimentally show that a quite large range of τ can allow FR-WAC to have prominent recovery efficiency and nearly zero performance degradation of asynchronous computations.

3.2 Correctness

Now we prove the correctness of our methods in Theorem 1.

Theorem 1. FR-WORB *and* FR-WAC *will converge to the same fixed point as* FR-Scratch.

Proof. We prove this theorem by analyzing the relationship between the value of vertex j in the fixed point ($\tilde{v}_j^{\infty}$) and the new input ($\tilde{v}_j^0$ and $\Delta\tilde{v}_j^0$). This is because the difference between FR-Scratch and FR-WORB/FR-WAC can be reduced to the restarting point after failures.

FR-WORB is first analyzed. As described in Eqs. (2) and (3), delta-based messages are transferred from one vertex to another along the edge between them, and used to update vertex states. Thus, at time t_k, the state of vertex j, $\tilde{v}_j^k$, has accumulated any received message value from j's direct and indirect neighbors. As the initial input, $\tilde{v}_j^0$ and $\Delta\tilde{v}_j^0$, of course are accumulated into $\tilde{v}_j^k$. Mathematically, this is shown in Eq. (6). Here, $\prod_{i_x,i_{x+1}\in p,\ and\ x=0}^{l-1} g_{\{i_x,i_{x+1}\}}(\Delta\tilde{v}_{i_0}^0)$ indicates the message value transferred from i_0 to j along one path p in the graph, and we replace it with $\prod_p(\Delta\tilde{v}_{i_0}^0)$ for simplicity.

$$\begin{aligned}\tilde{v}_j^k &= \tilde{v}_j^0 \oplus \Delta\tilde{v}_j^0 \\ &\oplus \sum_{l=1}^{k} \oplus \Big(\sum_{p\in P(j,l)} \oplus \big(\prod_{\substack{i_x,i_{x+1}\in p,\\ and\ x=0}}^{l-1} g_{\{i_x,i_{x+1}\}}(\Delta\tilde{v}_{i_0}^0) \big) \Big)\end{aligned} \tag{6}$$

Specifically, a path is defined as $p=\{i_0,i_1,...,i_x\}$ consisting of vertices that satisfy the following condition: there exists $0\le m_0<m_1<...<m_x\le k$ such that $i_h\in S_{m_h}$. S_k is a set of vertices performing the user-defined update logic at time t_k, and it is an element in an update sequence $\mathbb{S}=\{S_0,S_1,S_2,...,S_k\}$ corresponding to a continuous time instance sequence $\{t_0,t_1,t_2,...,t_k\}$. In particular, vertices in S_0 use input vertex and delta values to update themselves. The physical meaning of p is that some value as a message is sent from i_0 and then forwarded along $i_1,i_2,...$ to i_x. p is available for i_x when i_x has received the message originating from i_0. The number of p's hops equals to (|p|-1). We use $P(j,l)$ to stand for a set of l-hop available paths to j. Thus, for $p\in P(j,l)$, $\prod_p(\Delta\tilde{v}_{i_0}^0)$ is computed by recursively applying the message generating function $g_{\{i_x,i_{x+1}\}}$, i.e.,

$$\prod_p(\Delta\tilde{v}_{i_0}^0) = g_{\{i_{l-1},j\}}\Big(...g_{\{i_1,i_2\}}\big(g_{\{i_0,i_1\}}(\Delta\tilde{v}_{i_0}^0)\big)\Big)$$

Obviously, at the fixed point, to guarantee that all possible values have been accumulated into $\tilde{v}_j$, a vertex i_0 must perform Eqs. (2) and (3) an infinite number of times until its $\Delta\tilde{v}_{i_0}$ is $\mathbf{0}$. We thereby derive $\tilde{v}_j^{\infty}$ by accumulating all possible paths with hops from 0 to ∞, which is described in Eq (7).

$$\tilde{v}_j^{\infty} = \tilde{v}_j^0 \oplus \Delta\tilde{v}_j^0 \oplus \sum_{l=1}^{\infty} \oplus \Big(\sum_{p\in P(j,l)} \oplus \big(\prod_p(\Delta\tilde{v}_{i_0}^0) \big) \Big) \tag{7}$$

Based on Eq. (5), the expression of $\Delta\tilde{v}_{i_0}^0$ is

$$\Delta\tilde{v}_{i_0}^0 = \tilde{v}_{i_0}^1 \ominus \tilde{v}_{i_0}^0 = \Big(\sum_{n\in\Gamma_{i_0}^{in}} \oplus g_{\{n,i_0\}}(\tilde{v}_n^0) \Big) \ominus \tilde{v}_{i_0}^0 \oplus c_{i_0}$$

Thus, compared with FR-Scratch ($\tilde{v}_{i_0}^0=\mathbf{0}$, $\Delta\tilde{v}_{i_0}^0=c_{i_0}$), the difference in input is caused by $\tilde{v}_n^0/\tilde{v}_{i_0}^0$. In the following, we introduce how to eliminate this difference.

Without loss of generality, given an l-hop path p starting from v_{i_0}, by substituting the expression for $\Delta\tilde{v}_{i_0}^0$, we can easily expand $\prod_p(\Delta\tilde{v}_{i_0}^0)$ as

$$\begin{aligned}\prod_p(\Delta\tilde{v}_{i_0}^0) = &\Big(\sum_{n\in\Gamma_{i_0}^{in}} \oplus \Big(\prod_p \big(g_{\{n,i_0\}}(\tilde{v}_n^0) \big) \Big) \Big) \\ &\oplus \prod_p(c_{i_0}) \ominus \prod_p(\tilde{v}_{i_0}^0).\end{aligned}$$

$\forall n\in\Gamma_{i_0}^{in}$, an ($l$+1)-hop path can be formed by adding the in-neighbor n, i.e., $p'=\{n\}\cup p$. Like p, we have

$$\begin{aligned}\prod_{p'}(\Delta\tilde{v}_n^0) = &\Big(\sum_{m\in\Gamma_n^{in}} \oplus \Big(\prod_{p'} \big(g_{\{m,n\}}(\tilde{v}_m^0) \big) \Big) \Big) \\ &\oplus \prod_{p'}(c_n) \ominus \prod_{p'}(\tilde{v}_n^0)\end{aligned}$$

In particular, $\prod_{p'}(\tilde{v}_n^0) = \prod_p\big(g_{\{n,i_0\}}(\tilde{v}_n^0)\big)$. Hence, by accumulating $\prod_p(\Delta\tilde{v}_{i_0}^0)$ and $\prod_{p'}(\Delta\tilde{v}_n^0)$ as described in Eq. (7), the item $\prod_p\big(g_{\{n,i_0\}}(\tilde{v}_n^0)\big)$ is eliminated. Furthermore, when values along all (l+1)-hop paths starting from i_0's in-neighbors have been received, $\sum_{n\in\Gamma_{i_0}^{in}} \oplus \Big(\prod_p \big(g_{\{n,i_0\}}(\tilde{v}_n^0) \big) \Big)$ is eliminated. As a result, after accumulating values transferred along all paths to j, including $\tilde{v}_j^0$ and $\Delta\tilde{v}_j^0$, we can infer that

$$\begin{aligned}\tilde{v}_j^{\infty} = &c_j \oplus \Big(\sum_{l=1}^{\infty} \oplus \Big(\sum_{p\in P(j,l)} \oplus \big(\prod_p(c_{i_0}) \big) \Big) \Big) \\ &\oplus \Big(\sum_{p\in P(j,\infty)} \oplus \Big(\prod_p \big(\sum_{n\in\Gamma_{i_0}^{in}} \oplus g_{\{n,i_0\}}(\tilde{v}_n^0) \big) \Big) \Big)\end{aligned}$$

Since $g_{\{i,j\}}(x)$ is always closer to $\mathbf{0}$ than x, $\prod_{p\in P(j,\infty)}(x)\to\mathbf{0}$. Therefore, $\tilde{v}_j^{\infty}$ only depends on c_{i_0} which is the same as that in FR-Scratch, $i_0\in V$.

Consider that the only difference between FR-WORB and FR-WAC is the value of $\tilde{v}_n^0$ for every vertex on failed nodes. Since $\tilde{v}_j^{\infty}$ is independent of $\tilde{v}_n^0$, we can easily infer that FR-WAC also converges to the same fixed point as FR-Scratch. Then we have the claim. □

3.3 Example Graph Algorithms

We finally illustrate a series of well-known graph algorithms, the user-defined update functions of which can be converted into the required operations $g_{\{i,j\}}(x)$, $\oplus$ and $\ominus$ (as shown in Table 1). Since PageRank has been given as an example in Section 2.1), we describe other algorithms as follows.

Penalized Hitting Probability (PHP) [10, 29]: PHP is used to measure the proximity (similarity) between a given source vertex s and any other vertex j. As a random walk based algorithm, a walker at vertex i moves to i's out-neighbor j with probability proportional to an edge weight $w(i, j)$. The sum of transition probabilities indicates the proximity. In particular, s as the query vertex has constant proximity value 1. Both PageRank and PHP use a decay factor d ($0 < d < 1$) when computing messages. Without loss of generality, we set $d = 0.8$ in this paper.

Table 1. Example Graph Algorithms

Algorithms	c_j	$g_{\{i,j\}}(x)$	$\oplus$	$\ominus$	**0**
PageRank	(1-d)	$d \cdot \frac{x}{\lvert\Gamma^{out}(j)\rvert}$	+	−	0
PHP	1 (j=s) or 0 ($j \neq s$)	$d \cdot x \cdot w(i,j)$ ($j \neq s$) or 0 (j=s)	+	−	0
Katz	1 (j=s) or 0 ($j \neq s$)	$\beta \cdot x$	+	−	0
Adsorption	$p_j^{inj} \cdot I_j$	$p_j^{cont} \cdot w(i,j) \cdot x$	+	−	0

Katz metric (Katz) [11]: Katz is another proximity measure between two vertices in a graph. The score value is computed as the sum over the collection of paths between a given source vertex s and any other vertex j. Like d used in PHP, β is a user-defined damping factor.

Adsorption [3]: Adsorption is a graph-based label propagation algorithm used in personalized recommendation. Every vertex j iteratively refines a probability distribution on a label set which contains features of entities (e.g., the category of music/video). c_j is initialized by a user-defined distribution I_j. Afterwards, j is updated by the weighted average of label distributions from its in-neighbors. p_j^{inj} and p_j^{cont} are two parameters associated with j and $p_j^{inj} + p_j^{cont} = 1$.

4. Faiter: A Distributed Framework with FR-WORB and FR-WAC

Now we present *Faiter*, a distributed implementation of our fault-tolerant methods, built on top of Maiter. Fig. 1 illustrates the major difference between *Faiter* and Maiter. Newly added components in *Faiter* are *RecoveryCoordinator* on the master node and *RecoveryExecutor* on every slave node. The two components work together to recover failures. *Faiter* also performs load balancing on recovery to further improve the recovery efficiency.

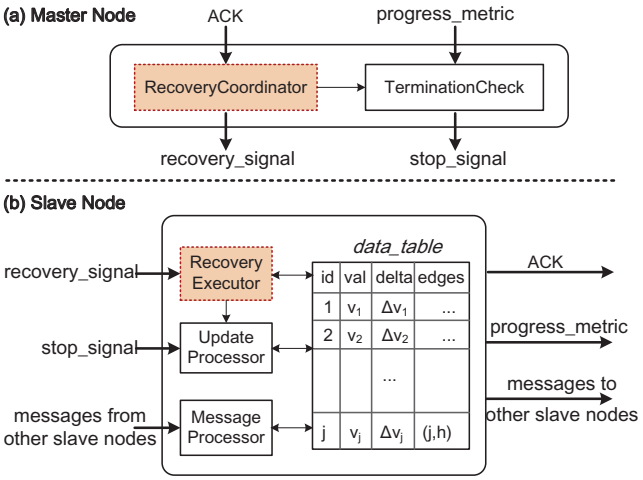


Figure 1. Implementing *Faiter* on top of Maiter

4.1 Maiter Architecture

As shown in Fig. 1 (ignore *RecoveryCoordinator* and *RecoveryExecutor* for now), at the very beginning of computations, an input graph is divided into partitions, each of which is kept on a slave node as *data_table* so that they can be processed in parallel. *data_table* consists of a series of quadruples, i.e., vertex id *id*, vertex state value *val*, delta-based value *delta* and outgoing edges *edges*. After that, by individually accessing different columns in *data_table*, *MessageProcessor* and *UpdateProcessor* can respectively perform Eq. (2) and Eq. (3), to continuously consume received messages, update vertex values and send new messages. Nodes communicate with each other through MPI.

Given an algorithm, *TerminationCheck* on the master node monitors its computation progress in a passive way where *UpdateProcessor* on every slave node periodically reports the *progress_metric* to *TerminationCheck*, and then the latter accumulates reported information to obtain the global progress. Generally, when the difference between two consecutive global progress values is less than a user-specified threshold, *TerminationCheck* sets a flag *stop_signal* as "true" and then broadcasts it to all slave nodes to stop computations in *UpdateProcessor* and dump final results.

4.2 Design of *Faiter*

Upon failures, the process of employing FR-WORB or FR-WAC for recovery consists of three phases: loading lost graph data where $\tilde{v}_j^0$ is initialized, followed by flushing messages, and constructing $\Delta\tilde{v}_j^0$. The three phases are performed by new components *RecoveryCoordinator* on the master node and *RecoveryExecutor* on every slave node. The details are given in Algorithm 1 (*RecoveryCoordinator*) and Algorithm 2 (*RecoveryExecutor*). Here, $\mathbb{N}$ is the set of nodes.

When encountering failures, the process in Algorithm 1 is first triggered. Consider that $\tilde{v}_j$ keeps stable when constructing the restarting point in FR-WORB/FR-WAC and the dif-

ference between two consecutive *progress_metric* values is zero. *TerminationCheck* thereby makes a false positive judgment and then terminates the algorithm. To solve this problem, *TerminationCheck* must be suspended (Line 1).

Algorithm 1: RecoveryCoordinator()

```
1  suspending TerminationCheck
2  sending recovery_signal of "load" to RecoveryExecutor
   on N ∈ ℕ
3  while Number of ACKs of "load" ≠ |ℕ| do
4      waiting for new ACK
5  sending recovery_signal of "flush" to
   RecoveryExecutor on N ∈ ℕ
6  while Number of ACKs of "flush" ≠ |ℕ| do
7      waiting for new ACK
8  sending signal_recovery of "construct" to
   RecoveryExecutor on N ∈ ℕ
9  while Number of ACKs of "construct" ≠ |ℕ| do
10     waiting for new ACK
11 waking TerminationCheck
12 sending signal_recovery of "restart" to
   RecoveryExecutor on N ∈ ℕ
13 return;
```

Subsequently, *RecoveryCoordinator* notifies all *RecoveryExecutor*s to load lost graph data if necessary by the *recovery_signal* of "*load*", and then blocks itself to wait for acknowledgements (i.e., *ACK*) from *RecoveryExecutor*s (Lines 2-4 in Algorithm 1). Once receiving the notification, as shown in Algorithm 2, *RecoveryExecutor* first suspends *UpdateProcessor* to avoid updating vertex state values (Line 2). For a new employed node, its *data_table* is empty and can be filled up by raw graph data (FR-WORB) or checkpoint data (FR-WAC) (Lines 3-4). An acknowledgement is sent to indicate that the loading work is done (Line 5). Note that lost data on one node can be re-assigned to multiple nodes to accelerate the recovery efficiency. We discuss this problem in Section 4.3.

Once every acknowledgement of "*load*" has been received, *RecoveryCoordinator* starts the flushing operation by broadcasting the *recovery_signal* of "*flush*", to discard existing messages in the system (Lines 5-7 in Algorithm 1). *RecoveryExecutor*, as shown in Algorithm 2, performs flushing by first flushing local messages (Line 8) and then broadcasting an *EOF* flag to each other to clear messages flying on network (Line 9). When all *EOF*s are available, that means *MessageProcessor* on this node has received every possible message and accumulated its value into Δv_j (Lines 10-11). Thus, the next step is to reset Δv_j as $\mathbf{0}$, to discard accumulated values (Lines 12-13). The acknowledgement of "*flush*" tells *RecoveryCoordinator* that the corresponding node has finished flushing operations (Line 14).

After flushing messages, *RecoveryCoordinator* broadcasts another *recovery_signal* of "*construct*" to construct $\Delta\tilde{v}_j$ (Lines 8-10 in Algorithm 1). In Algorithm 2, *RecoveryExecutor* accesses *data_table* and then sends messages $g_{\{j,h\}}(\tilde{v}_j)$ (Lines 17-19). Like flushing operations, *EOF* is exchanged to guarantee that all messages have been received by targets (Lines 20-22). Afterwards, we can calculate $\Delta\tilde{v}_j$ using Eq. 5 (Lines 23-24).

Algorithm 2: RecoveryExecutor()

```
   Input : recovery signal received from
           RecoveryCoordinator: recovery_signal
1  if recovery_signal is load then
2      suspending UpdateProcessor
3      if data_table is empty then
4          loading graph data and initializing ṽ_j^0
5      sending ACK of "load" to RecoveryCoordinator
6      return
7  else if recovery_signal is flush then
8      flushing messages in sending buffer
9      sending EOF to RecoveryExecutor on N ∈ ℕ
10     while Number of EOFs ≠ |ℕ| do
11         waiting for new EOF
12     foreach j ∈ V_N do
13         Δv_j ← 0
14     sending ACK of "flush" to RecoveryCoordinator
15     return;
16 else if recovery_signal is construct then
17     foreach j ∈ V_N do
18         foreach h ∈ Γ^out(j) do
19             sending g_{j,h}(ṽ_j), j ∈ V to target
               MessageProcessor
20     sending EOF to RecoveryExecutor on N ∈ ℕ
21     while Number of EOFs ≠ |ℕ| do
22         waiting for new EOF
23     foreach j ∈ V_N do
24         calculating Δṽ_j using Eq. 5
25     sending ACK of "construct" to
       RecoveryCoordinator
26     return;
27 else if recovery_signal is restart then
28     waking UpdateProcessor
29 return;
```

Finally, *TerminationCheck* and *UpdateProcessor*s are waked (Lines 11-12 in Algirihtm 1 and Line 28 in Algorithm 2), and then computations are restarted. In particular, if cascading failures are encountered during the three phases, the two algorithms will be re-executed.

4.3 Recovery with Load Balancing

In restarting computations after failures, every vertex value is refined iteratively to the fixed point as *failure-free* execution. However, computing work on lost vertices must be performed from scratch (FR-WORB) or the most recent available checkpoint (FR-WAC), instead from the point just before failures (like surviving vertices). In another word, compared with surviving nodes, replacements of failed nodes need to handle the lost computing work until failures, indicating a heavy load imbalance problem.

We solve this problem by distributing lost computations on m failed nodes onto n replacements, where $n=m+x$, and $x \geq 1$. The basic idea behind it is to leverage the additional x nodes to balance the load on replacements and then accelerate recovery. When recovery is done, data on these additional x nodes will be sent back to m replacements, in order to release resources.

Intuitively, data can be re-assigned onto n replacements using the simple "HASH" policy, i.e., *v_id mod n*. However, a key issue is how to avoid collisions, because the original data assignment usually uses the same policy in most existing systems [1, 15, 31], i.e., *v_id mod* $|\mathbb{N}|$, where $|\mathbb{N}|$ is the number of employed nodes before failures. For example, assume that $|\mathbb{N}|$=16, $n = 2$, and vertex ids on the failed node *node_0* are {0,16,32,48,...}. In re-assignment, the value of *v_id mod 2* is always zero, that is, vertices on *node_0* are still re-assigned onto a single replacement and another one is idle. This paper designs a new hash function as shown in Eq. (8). By using $(|\mathbb{N}| \cdot x)$ instead of n, we can evenly distribute data. Here, *mapTable* is a lookup table mapping *idx* into a new unique node id *node_id*.

$$node\_id = mapTable[idx], idx = j \ mod \ (|\mathbb{N}| \cdot x) \quad (8)$$

The new hash function also facilitates the message addressing problem, i.e., routing a message to the correct target node after data placement has been changed. Existing work [12] maintains a distributed lookup table storing the ownership of every vertex. Searching a large table is obviously time-consuming. Instead, our method computes the address dynamically. That is, given a destination vertex id *v_id*, its target node id *node_id* = *v_id mod* $|\mathbb{N}|$. If the node fails, *v_id* is hashed again using Eq. (8) to get a new node id. Apparently, the only requirement in our method is to maintain a *mapTable* with n entries, which is much smaller than the table with $|V|$ entries used in [12].

There are advanced partitioning techniques, but none of them is suitable for our re-assignment scenario. For example, multi-level partitioning [13] is not cost-effective as its expensive runtime will be counted in our online processing time. By contrast, streaming partitioning [21] requires to be run on a single machine, that impacts the scalability.

5. Evaluation

In this section, we evaluate our fault-tolerant methods on *Faiter* to show their effectiveness.

Solutions for comparison: We analyze the performance of our FR-WORB and FR-WAC in comparison with the baseline method FR-Scratch. Existing checkpoint-based methods are not involved in our study due to the heavy performance degradation caused by expensive synchronization barriers [9] or overheads of blocking computations [14].

Experimental cluster: We conduct testing on the Amazon EC2 cluster which consists of 32 t2.micro instances/nodes with one additional node as Master. Each node running Ubuntu Server 14.04 is equipped with 1 virtual core, 1GB of RAM, and 8GB SSD storage.

Workload and datasets: Limited by the manuscript length, we perform evaluation on two representative graph algorithms only, PageRank and PHP. Other algorithms listed in Table 1 exhibit the similar performance.

All tests are done over real graphs as shown in Table 2. The web graph Wiki[1] has a large diameter, and LiveJ[2] and Wiki are sparser than Orkut[3].

Table 2. Real Graph Datasets (M: million)

Graph	Vertices	Edges	Degree	Type
LiveJ	4.8M	68M	14.2	Social networks
Wiki	5.7M	130M	22.8	Web graphs
Orkut	3.1M	234M	75.5	Social networks

Evaluation metrics: Two metrics are evaluated in experiments, *Recovery time* and *Runtime*.

Assume that failures occur at t_f. *Recovery time* is defined as the elapsed time from t_f to the time t_r where lost vertex states have been recovered. In order to evaluate whether the recovery is completed, we need to define a progress metric (*PM*). Given input as shown in Table 1, each vertex state v_j is monotonously increased to a fixed point for both PageRank and PHP in *failure-free* execution. We thereby define *PM* as the sum of every v_j, i.e., at time t_k, $PM_k = \sum_{j \in V} \oplus v_j^k$. Then, at t_r, for the partial progress metric $PM_k^{'} = \sum_{j \in V_N} \oplus v_j^k, N \in \mathbb{N}_F$, we have $PM_r^{'} = PM_f^{'}$. $\mathbb{N}_F$ is the set of failed nodes.

On the other hand, for both PageRank and PHP, *Runtime* is the elapsed time from t_f to the point where $|PM_\infty - PM_k| \leq \delta$, where PM_∞ is the sum of vertices at the fixed point and δ is the user-specified difference threshold. Specifically, PM_∞ is given by running *failure-free* algorithms offline. On the other hand, surviving vertices are still updated progressively during recovery, which continuously contributes to *PM*. If δ is large, it's possible that the termination is triggered but many restarted vertices are still far from convergence. Based

[1] http://haselgrove.id.au/wikipedia.htm

[2] http://snap.stanford.edu/data/soc-LiveJournal1.html

[3] http://socialnetworks.mpi-sws.org/data-imc2007.html

on our experience, $\delta = 10^{-3}$ is small enough to solve this problem for graphs used in this paper.

Experiment design: Consider that no explicit barrier exists in asynchronous systems. To test the effectiveness of our methods when failures occur at different phases in computations, we thereby state three scenarios, T_1, T_2, and T_3. That is, failures occur at times T_1, T_2, and T_3, respectively. Specifically, we run an algorithm without failures beforehand to record the total runtime t and then set the specific values of T_1, T_2, T_3 as $0.1t$, $0.5t$, and $0.9t$, respectively. Besides, the priority queue size is set to $q = 0.2|V|$ to achieve prominent performance of *failure-free* execution based on priori tests.

5.1 Recovery Time

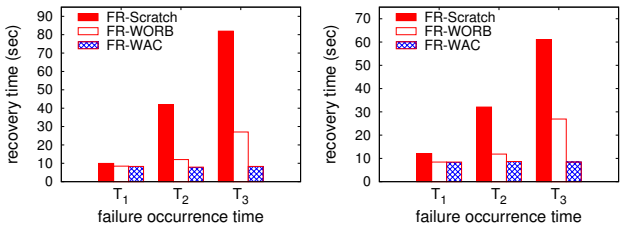
Suppose that 16 slave nodes are used and only a single one fails. Unless otherwise specified, we replace failed nodes with the same number of standby nodes. This suite of experiments shows the recovery time of FR-Scratch, FR-WORB and FR-WAC, by running PageRank and PHP over all datasets. In particular, τ in FR-WAC is set to 8 seconds for PageRank on Orkut, and for others, it equals to 4 seconds. A detailed discussion about τ is given in Section 5.6.

All tests are done in three scenarios: T_1 where the three methods exhibit similar performance, and T_2 and T_3 where our FR-WORB and FR-WAC are supposed to be better. Specifically, as plotted in Fig. 2-Fig. 4, the speedup of FR-WORB compared with FR-Scratch is 3.03x (PageRank over LiveJ, T_3) at most. For FR-WAC, it is even up to 9.88x (PageRank over LiveJ, T_3).

In T_1, compared with FR-Scratch, the recovery performance using FR-WORB and FR-WAC is not improved significantly, and even slightly worse in some cases. That can be explained from two perspectives. First, in the early phase of computations, accumulated workloads on surviving nodes are not so many that the cost of recomputing them from scratch (FR-Scratch) can be negligible. Second, there exist some synchronization barriers in FR-WORB and FR-WAC to execute flushing operations, which incurs overheads.

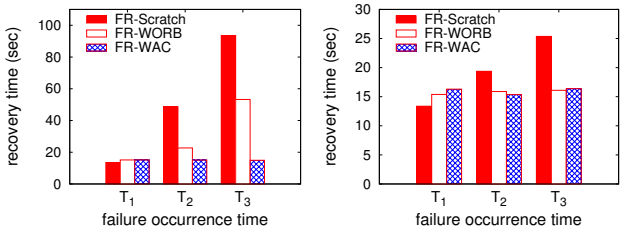
By contrast, in T_2 and T_3, FR-Scratch takes much time to recompute workloads on surviving nodes. However, FR-WORB and FR-WAC avoid this problem and can leverage them to accelerate the recovery speed of lost data. FR-WORB and FR-WAC thereby have superior performance to FR-Scratch. FR-WAC usually outperforms FR-WORB, especially for T_3. This is because at the end of computations, recomputing lost data from scratch (FR-WORB) is still time-consuming, even though we can utilize preserved workloads. FR-WAC solves this problem by providing the most recent available checkpoint data.

An interesting observation we can make in Fig. 3(b) is that FR-WORB and FR-WAC perform similarly. PHP traverses a graph starting from a given source vertex, and hence for the large diameter graph Wiki, there exists a long convergent stage where only a few vertices are updated and others are convergent. Thus, when some data are lost, convergent vertices on surviving nodes can greatly accelerate the recovery, which largely narrows the performance gap between FR-WORB and FR-WAC.



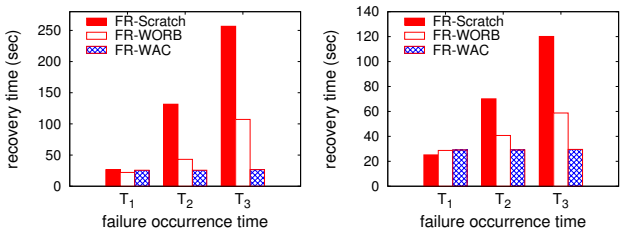
(a) PageRank (b) PHP

Figure 2. Recovery time on LiveJ



(a) PageRank (b) PHP

Figure 3. Recovery time on Wiki



(a) PageRank (b) PHP

Figure 4. Recovery time on Orkut

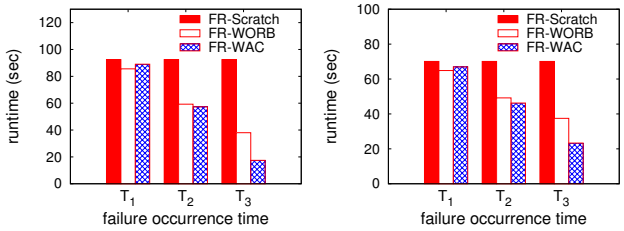
5.2 Runtime

We test runtime using the same setting in Fig. 2-Fig. 4. As shown in Fig. 5-Fig. 7, FR-WORB is 2.43 times faster than FR-Scratch at most (PageRank over LiveJ, T_3). FR-WAC can offer up to 5.57x speedup in comparison with FR-Scratch (PageRank over Wiki, T_3). Note that the gain is not so large as that in recovery time, because the time of running remaining computations after recovering failures may occupy a large proportion of the overall performance, especially for T_1 and T_2.

Different from recovery time, runtime of FR-WORB and FR-WAC is inversely proportional to the failure occurrence time, because they can avoid rolling back completed computations. However, for FR-Scratch, it is a constant since FR-Scratch always recomputes from scratch.

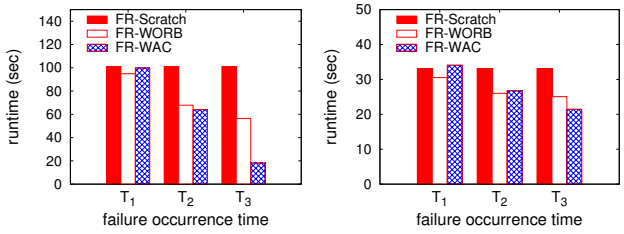
Another observation we can make is that when failures occur in T_1 and T_2, compared with FR-WORB, checkpointing in FR-WAC brings marginal benefit, and even slightly performance degradation in some cases. The reason is that runtime counts the cost of archiving data. Thus, benefits about reducing recovery time is partially offset, especially

for T_1 (more checkpoint data are archived in restarting computations). From the runtime perspective, FR-WORB is a preferred solution when failures occur during the early computations, while FR-WAC is more suitable for the scenario where the process is interrupted at the end of computations.



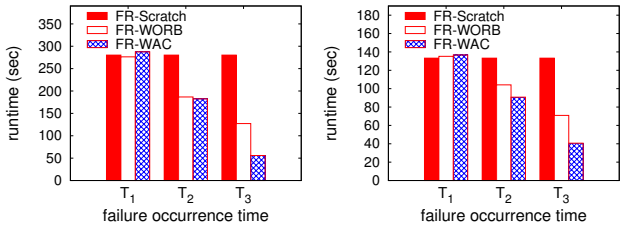
(a) PageRank (b) PHP

Figure 5. Runtime on LiveJ



(a) PageRank (b) PHP

Figure 6. Runtime on Wiki



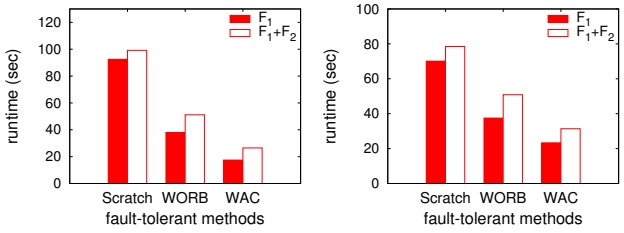
(a) PageRank (b) PHP

Figure 7. Runtime on Orkut

5.3 Runtime with Cascading Failures

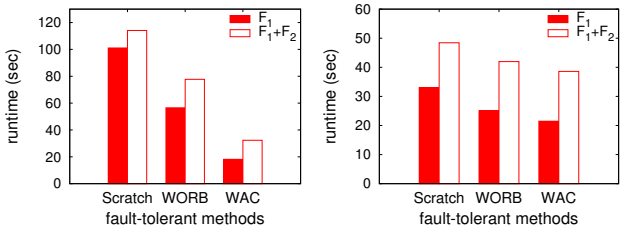
Figs. 8-10 report runtime of FR-Scratch, FR-WORB and FR-WAC when encountering cascading failures. All tests are run in T_3 and the other setting is the same as that used in Fig. 2-Fig. 4. Here, F_1 indicates a single failed node, and $(F_1 + F_2)$ means that another node fails (cascading failure) at time $t_f^{'}$, when constructing the restarting point for F_1. Since there is no construction in FR-Scratch, we simulate cascading failures by setting another node failed at $t_f^{'}$.

Although FR-WORB and FR-WAC exhibit the similar reconstructing operations, in some cases, as shown in Fig. 10, FR-WORB performs slightly worse than FR-WAC, from the performance degradation perspective. This is because more failed nodes means more data are lost, which increases the recomputing workload for FR-WORB. But for FR-WAC, the increase is not so much due to checkpoint data. Nevertheless, both of them still outperform FR-Scratch.



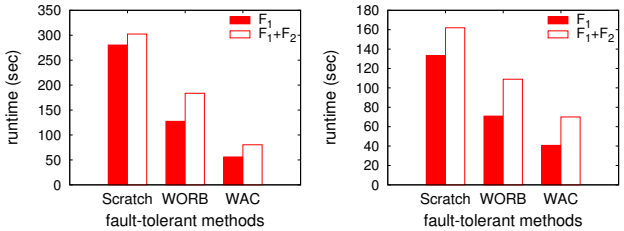
(a) PageRank (b) PHP

Figure 8. Runtime on LiveJ



(a) PageRank (b) PHP

Figure 9. Runtime on Wiki



(a) PageRank (b) PHP

Figure 10. Runtime on Orkut

5.4 Impact of Number of Failed Nodes

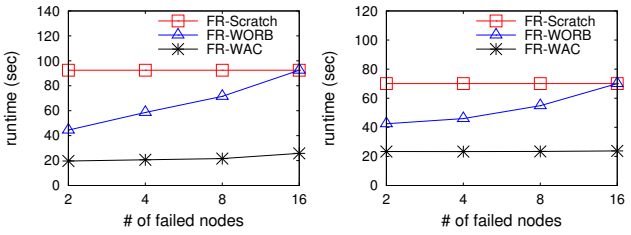
We explore the features of FR-Scratch, FR-WORB, and FR-WAC, when varying the number of failed nodes. Using the same setting in Fig. 2-Fig. 4, Figs. 11-13 report the runtime for PageRank and PHP in T_3.

Not surprisingly, the performance of FR-Scratch keeps stable since it always forgets about completed workloads on all nodes. Benefitting from the checkpoint data, a similar trend can also be observed for FR-WAC, but the performance is prominent when compared to FR-Scratch, because computations on surviving nodes are preserved.

On the contrary, the performance of FR-WORB gradually degrades with the increase of failed nodes, as more lost data are required to be recomputed from scratch and the benefits brought by surviving vertex states become less significant. Nevertheless, even when 50% of nodes fail, FR-WORB is still up to 1.29 times faster than FR-Scratch.

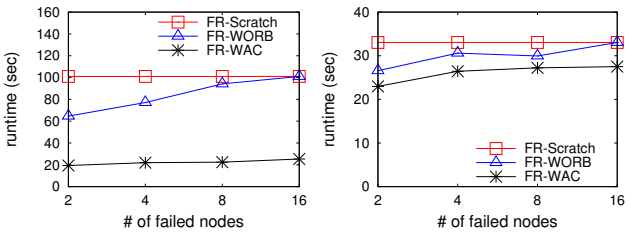
5.5 Recovery with Load Balancing

This group of experiments validates that balancing the load of recovery (Section 4.3) can significantly drop the recovery time. PageRank over Wiki and PHP over LiveJ as two cases are tested. The similar trends can be observed in other cases but we omit them due to the limited manuscript space.



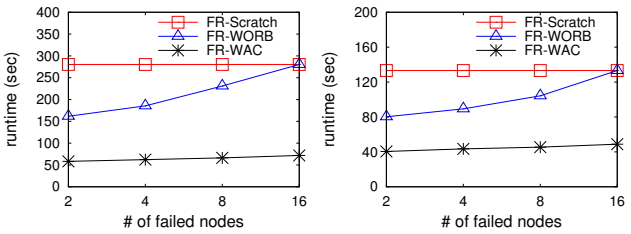
(a) PageRank (b) PHP

Figure 11. Runtime on LiveJ



(a) PageRank (b) PHP

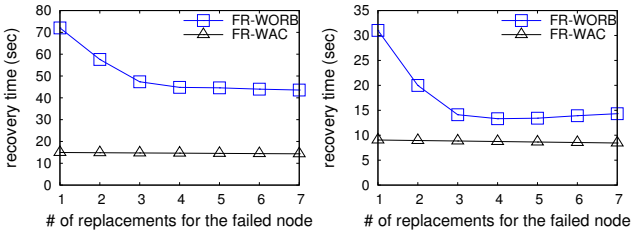
Figure 12. Runtime on Wiki



(a) PageRank (b) PHP

Figure 13. Runtime on Orkut

Specifically, in order to show how FR-WORB and FR-WAC scale, we run the two cases using 10 nodes but when one node fails in T_3, the number of replacements varies from 1 to 7. Fig. 14 plots the recovery time. FR-WORB has a significant improvement. The recovery time is reduced by up to 39.5% and 57.3% in the two cases. This is because recomputing from scratch is extremely time-consuming and then leads to a heavy load imbalance. By contrast, increasing the number of replacements brings marginal benefit for FR-WAC, as checkpoint data reduce recomputing workload, which has already alleviated the load imbalance problem.



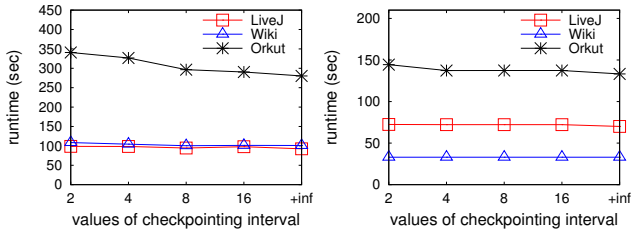
(a) PageRank over Wiki (b) PHP over LiveJ

Figure 14. Recovery with load balancing

5.6 Determining τ for FR-WAC

FR-WAC usually exhibits prominent performance but requires to archive data periodically. Now we compare the runtime of *failure-free* execution when archiving data with different τ values. $\tau = +inf$ means that no checkpoint is archived. A reasonable τ used above is also determined here.

As shown in Fig. 15, different from existing synchronous checkpointing techniques, a quite large range of τ can guarantee that the runtime overhead of archiving data is nearly zero because of the efficient asynchronous mechanism. On the other hand, a smaller τ means a more recent checkpoint is available, which can reduce the recomputing workload. Thus, in experiments above, we set τ= 8 seconds for PageRank over Orkut and τ= 4 seconds for other cases.

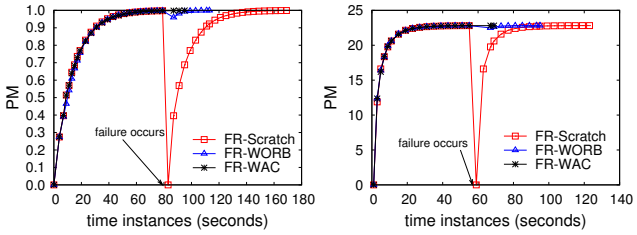


(a) PageRank (b) PHP

Figure 15. Impact of archiving data

5.7 Empirical Validation of Correctness

Section 3.2 analyzes the correctness of FR-WORB and FR-WAC. Now we empirically validate it by continuously monitoring the progress metric PM_k at sampled time instances. Without loss of generality, in Fig. 16, we report the results of PageRank and PHP over LiveJ, using the same setting in Fig. 5. More specifically, we assume that $t_f = 87/57$ (second) for PageRank/PHP.



(a) PageRank (b) PHP

Figure 16. Correctness of FR-WORB and FR-WAC

Let $error(x) = \frac{|PM(x) - PM(FR-Scratch)|}{PM(FR-Scratch)}$. When algorithms converge, $error(FR-WORB)/error(FR-WAC)$ is less than 0.012% which can be negligible. Thus, FR-WORB/FR-WAC converges to the same fixed point as achieved in FR-Scratch.

6. Related Work

Many representative techniques have been developed to provide fault-tolerance in graph processing systems. All of them basically fall into three categories, checkpoint-based, lineage-based, and reactive paradigms. We summarize them and distinguish our work from them as follows. Fault-tolerant techniques in high performance computing (HPC) area are also discussed briefly.

Checkpoint-based solutions: Checkpointing as an early technique proposed in [15] has been extensively used in Pregel-like systems [1, 15, 18], due to its simplicity. It works well in synchronous systems, but results in suboptimal performance in asynchronous scenarios, as archiving checkpoint data needs expensive global barriers among different computational nodes, which largely offsets the benefit brought by asynchronous computations.

In particular, a non-blocking approach used in [24] makes checkpoint written along with the generation/computation of vertex values. Although barriers are still required, it partially overlaps CPU processing and I/O processing when archiving data, in order to avoid suspending computations as much as possible. However, the benefit is limited in many cases as reported by authors because the data generating/consuming rate is usually greater than the checkpoint writing rate, i.e., the computing thread must wait for writing data.

GraphLab [14] introduces a variant of the Chandy-Lamport method [6] to archive data without global barriers for its asynchronous engine. That significantly reduces the checkpointing overhead, but workloads on surviving nodes still need to be rolled back to the most recent checkpoint. Instead, our methods preserve these completed workloads to avoid the expensive recomputation overhead.

Besides, it greatly drops the overhead by reducing the serialized data to vertex values [25], that is also used in our methods. Some other researchers focus on improving the efficiency by confining re-computations only to lost data on failed nodes and performing these re-computations in parallel [20, 24]. However, vertex updates on surviving nodes are still paused to wait for the failure recovery. Differently, in our methods, surviving vertices can keep performing updates. Also, we design new data re-assignment function which is simple yet efficient due to the small lookup table.

Lineage-based solutions: Spark [2] employs a lineage method to track the dependency of its coarse-grained data structure [28]. The lost data can be recomputed by analyzing lineage. Unlike checkpointing, lineage can save storage space and network bandwidth since the volume of dependency information is much smaller than that of algorithm-specified data. However, directly injecting lineage into asynchronous engines cannot work well as expected. This is because fine-grained updates makes the dependency relationship prohibitively complicated, incurring expensive overheads in terms of space and runtime.

Reactive solutions: Recently, reactive recovery solutions without checkpointing have attracted a lot of attention.

Some techniques focus on utilizing redundant data. Specifically, Z. Chen tries to recover lost vector data for sparse matrix-vector multiplication by utilizing inherent redundant backup on other computational nodes [7]. M. Pundir et al. design Zorro [17] where lost vertices can be recovered by exploiting data replications naturally provided by today's systems, such as PowerGraph [8]. J. Wang et al. implement the asynchronous computation model in a Datalog system for general purpose applications and employ an existing technique, i.e., keeping outgoing messages in memory at the producer side until computations terminate, to re-send them without re-generation upon failures [22]. However, redundant information and replicated data may quickly exhaust memory resources as reported in [32]. Thus, they are not always feasible to scale to massive data sets. Besides, Zorro is an approximate solution, i.e., incurring a slight loss of accuracy when all replicated values are lost.

S. Schelter et al. [19] tolerate failures in a synchronous system by designing a restarting point, resembling our solutions. However, their technique requires users to carefully design an algorithm-specified compensation function, which is usually a nontrivial task [24]. On the contrary, our solutions replace vertex values automatically, which largely eases the burden of users. Also, our asynchronous checkpointing mechanism can further boost the efficiency.

Fault-tolerance in HPC: High-performance computing systems typically decompose an application into multiple tasks and then run these tasks in parallel for efficiency. Recently proposed techniques [5, 16] utilize the task dependency graph to recover a failed task by re-executing its predecessors. However, tracking this dependency graph is particularly prohibitive in asynchronous graph processing systems. This is because every vertex as an independent computing unit performs computations, which is complicated.

7. Conclusion

This paper proposes two novel fault-tolerant methods FR-WORB and FR-WAC for a typical asynchronous system Maiter. Unlike the most widely used checkpointing techniques, our methods remove expensive synchronous barriers and hence are naturally suitable for asynchronous computations. Both FR-WORB and FR-WAC perform computations on surviving data progressively without rolling back, and can leverage surviving data to accelerate recovering lost data. FR-WAC as an improved solution can provide the most recent checkpoint data through archiving data asynchronously, to reduce recovery workloads. An optimization technique about load balancing is also designed to boost the recovery efficiency, especially for FR-WORB. Evaluation studies on real-world graphs validate the effectiveness of our fault-tolerant solutions and load balancing technique.

Acknowledgments

This work was partially supported by the U.S. NSF grant CNS-1217284, and the National Natural Science Foundation of China (61433008, 61472071, and 61528203). Zhigang Wang was a visiting student at UMass Amherst, supported by China Scholarship Council, when this work was performed. Authors also would like to thank anonymous reviewers for their constructive comments.

References

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

[2] Apache spark. http://spark.apache.org/.

[3] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly. Video suggestion and discovery for youtube: taking random walks through the view graph. In *Proc. of the 17th international conference on World Wide Web*, pages 895–904. ACM, 2008.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. 7th Intl. Conf. on the World Wide Web*, pages 107–117. Elsevier, 1998.

[5] C. Cao, T. Herault, G. Bosilca, and J. Dongarra. Design for a soft error resilient dynamic task-based runtime. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 765–774. IEEE, 2015.

[6] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[7] Z. Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proc. of HPDC*, pages 73–84. ACM, 2011.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. of OSDI*, volume 12, page 2, 2012.

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

[10] Z. Guan, J. Wu, Q. Zhang, A. Singh, and X. Yan. Assessing and ranking structural correlations in graphs. In *Proc. of SIGMOD*, pages 937–948. ACM, 2011.

[11] L. Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[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 *Proc. of Eurosys*, pages 169–182. ACM, 2013.

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

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

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

[16] T. Martsinkevich, O. Subasi, O. Unsal, F. Cappello, and J. Labarta. Fault-tolerant protocol for hybrid task-parallel message-passing applications. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 563–570. IEEE, 2015.

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

[18] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proc. of SSDBM*, page 22. ACM, 2013.

[19] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. All roads lead to rome: optimistic recovery for distributed iterative data processing. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 1919–1928. ACM, 2013.

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

[21] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proc. of SIGKDD*, pages 1222–1230. ACM, 2012.

[22] J. Wang, M. Balazinska, and D. Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *Proc. of the VLDB Endowment*, 8(12):1542–1553, 2015.

[23] Z. Wang, Y. Gu, Y. Bao, G. Yu, and J. X. Yu. Hybrid pulling/pushing for i/o-efficient distributed and iterative graph computing. In *Proc. of SIGMOD*, pages 479–494. ACM, 2016.

[24] C. Xu, M. Holzemer, M. Kaul, and V. Markl. Efficient fault-tolerance for iterative graph processing on distributed dataflow systems. In *Proc. of ICDE*, pages 613–624. IEEE, 2016.

[25] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai. Seraph: an efficient, low-cost system for concurrent graph processing. In *Proc. of HPDC*, pages 227–238. ACM, 2014.

[26] J. Yin and L. Gao. Scalable distributed belief propagation with prioritized block updates. In *Proc. of CIKM*, pages 1209–1218, 2014.

[27] J. Yin and L. Gao. Asynchronous distributed incremental computation on evolving graphs. In *ECML/PKDD'16*, 2016.

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

[29] C. Zhang, L. Shou, K. Chen, G. Chen, and Y. Bei. Evaluating geo-social influence in location-based social networks. In *Proc. of CIKM*, pages 1442–1451. ACM, 2012.

[30] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: A distributed framework for prioritizing iterative computations. *Parallel and Distributed Systems, IEEE Transactions on*, 24(9):1884–1893, 2013.

[31] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation. *TPDS*, 25(8):2091–2100, 2014.

[32] C. Zhou, J. Gao, B. Sun, and J. X. Yu. Mocgraph: Scalable distributed graph processing using message online computing. *Proc. of the VLDB Endowment*, 8(4):377–388, 2014.