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

Zhigang Wang, Lixin Gao, *Fellow, IEEE*, Yu Gu, Yubin Bao, and Ge Yu, *Member, IEEE*

**Abstract**—Most graph algorithms are iterative in nature. They can be processed by distributed systems in memory in an efficient asynchronous manner. However, it is challenging to recover from failures in such systems. This is because traditional checkpoint fault-tolerant frameworks incur expensive barrier costs that usually offset the gains brought by asynchronous computations. Worse, surviving data are rolled back, leading to costly re-computations. This paper first proposes to leverage surviving data for failure recovery in an asynchronous system. Our framework guarantees the correctness of algorithms and avoids rolling back surviving data. Additionally, a novel asynchronous checkpointing solution is introduced to accelerate recovery at the price of nearly zero overheads. Some optimization strategies like message pruning, non-blocking recovery and load balancing are also designed to further boost the performance. We have conducted extensive experiments to show the effectiveness of our proposals using real-world graphs.

**Index Terms**—Fault-tolerance, asynchronous model, iterative graph algorithm, distributed memory-based systems

## 1 Introduction

Iterative graph algorithms have been widely used in numerous applications with billion-vertex graphs. To efficiently handle large graphs, many distributed systems have been developed [1], [2], [3], [4], [5], [6], most of which focus on memory-based computations, such as Pregel, Spark and GraphLab. Distributed systems run on a cluster consisting of a large number of commodity machines (also called "nodes" in this paper). When processing iterative graph algorithms, node failures may occur very frequently. Therefore, fault-tolerance is particularly important.

For simplicity, existing systems usually employ a synchronous model for graph computations through a series of iterations which are separated by explicit global barriers. Within one iteration, computational nodes coordinate with each other to synchronize the progress. Nodes running faster thereby block themselves to wait for stragglers, increasing synchronization overheads. Recently, asynchronous systems like GraphLab [4] and Maiter [5], have been proposed to eliminate the synchronization overheads by removing barriers. In particular, GraphLab can run most graph algorithms. However, its distributed lock contention for general purpose incurs expensive costs which can completely offset the gain achieved by asynchronous computations [7]. Different from GraphLab, Maiter is a light-weight asynchronous system with support for a limited set of important and often used algorithms, such as PageRank and Shortest Path. The issue we investigate in this paper is then to find an efficient fault-tolerant solution tailored for the asynchronous model described in Maiter.

*Challenges*: Although many efforts have been devoted into fault-tolerance, most of them focus on synchronous engines, such as checkpointing [1] used in many Pregel-like systems, and lineage [8] employed by Spark. Both are far from ideal for asynchronous engines. First, the checkpointing solution archives data periodically and then any failure can be recovered from the most recent checkpoint. However, archiving data requires an explicit barrier to coordinate the progress where underlying computations must be suspended. That exposes asynchronous engines to the same inefficiency of synchronous engines that the former are trying to address. On the other hand, the lineage solution tracks the coarse-grained dependency among data sets instead of data themselves, in order to save the storage space and network bandwidth. It, however, lacks built-in support for fine-grained updates in asynchronous systems. Note that researchers have designed an asynchronous checkpointing solution based on the Chandy-Lamport technique [9] for GraphLab. However, both surviving vertices and lost vertices are rolled back. The re-computation overhead is so large that this functionality is no longer offered.

*Our Contributions*: This paper first proposes a new failure recovery solution without rolling back, termed FR-WORB. Upon failures, FR-WORB automatically constructs a special restarting point where computations interrupted by failures can be continued. We can prove that the correctness is independent of vertex states provided for the restarting point, i.e., the continued computations always converge to the

- *Z. Wang, Y. Gu, and Y. Bao are with the School of Computer Science and Engineering, Northeastern University, Shenyang, Liaoning 110819, China. E-mail: wangzhiganglab@gmail.com, {guyu, baoyubin}@mail.neu.edu.cn.*
- *G. Yu is with the School of Computer Science and Engineering, Northeastern University, Shenyang, Liaoning 110819, China, and also with the College of Information, Liaoning University, Shenyang, Liaoning 110036, China. E-mail: yuge@mail.neu.edu.cn.*
- *L. Gao is with the Department of Electrical and Computer Engineering, University of Massachusetts Amherst, Amherst, MA 01003. E-mail: lgao@ecs.umass.edu.*

*Manuscript received 5 Sept. 2017; revised 1 Feb. 2018; accepted 16 Feb. 2018. Date of publication 22 Feb. 2018; date of current version 13 July 2018. (Corresponding author: Ge Yu.) Recommended for acceptance by X. Gu. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2018.2808519*

same point as if no failure has happened. Then FR-WORB can preserve vertex states on surviving nodes without rolling back. Further, it leverages surviving vertices to potentially accelerate the recovery of lost data, while simultaneously keeping refining surviving vertices. Intrinsically, FR-WORB is a reactive approach because no checkpoint is written before failures. We are aware of some recently published reactive approaches [10], [11], [12], [13]. However, they either consume a large amount of memory due to data replications, or raise nontrivial challenges when designing the recovery function.

Second, we design a variant of FR-WORB with an asynchronous checkpointing technique, referred to as FR-WAC, where lost data can be recomputed from the last checkpoint instead of scratch. It reduces the number of re-computations and then improves the recovery efficiency. In particular, the independence between correctness and vertex states provided for the restarting point, enables us to separate archiving operations from underlying computations. Then, neither the barrier among nodes nor the coordination between vertex persistence and updates on one node is required. The overhead of checkpointing is thereby nearly zero.

Finally, we design three important optimization strategies for FR-WORB and FR-WAC. (1) Message pruning. For correctness, both FR-WORB and FR-WAC broadcast messages to purge surviving vertex states of contributions from failed vertices, because later compensation messages will be propagated again. To support this, the underlying vertex update operation must be reversible. Our pruning technique can reduce the number of purge and compensation messages by controlling the communication behaviors. Also, it removes the reversibility requirement. (2) Non-blocking recovery. Till now, we assume that replacements of failed nodes can be immediately used upon failures. However, this is not always feasible because users may not want to pay for such standby nodes in the *failure-free* execution. In the absence of standby nodes, we design a non-blocking strategy so that surviving nodes can continue computations instead of waiting for acquiring new resources. (3) In contrast, if there are sufficient standby nodes, we can assign recovery workloads to multiple nodes for load balancing. A new data re-assignment policy is given to efficiently route messages after the vertex-assignment is changed.

This paper extends a preliminary work [14] in the following aspects. First, we analyze the communication behaviors of FR-WORB and FR-WAC in detail, and then propose a simple yet efficient message pruning strategy to further boost the recovery efficiency. Second, we consider the scenario without standby nodes and then design a new mechanism to provide non-blocking fault-tolerance service. We also devote some engineering efforts towards high-availability, i.e., writing checkpoint data onto the distributed file system HDFS, rather than local disk.

*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 fault-tolerance methods and proves the correctness. Section 4 proposes some optimizations for fast recovery. Section 5 reports experimental studies. Section 6 highlights the related work. Finally, we conclude this work in Section 7.

## 2 Preliminaries

Among asynchronous systems, Maiter [5] is a lightweight engine and can support many important yet often used graph algorithms. In the following, we briefly review its asynchronous model, followed by discussions of the challenges of tolerating failures and example algorithms supported by our recovery solutions.

### 2.1 Maiter: An Asynchronous Memory-Based System

We model a graph as a directed graph $G = (V, E)$, where $V$ is a set of vertices and $E$ is a set of edges. Given an edge $(i, j)$, $i/j$ is the source/target vertex. The set of in/out-neighbors of $i$ is denoted by $\Gamma_i^{in}/\Gamma_i^{out}$. $G$ is partitioned onto multiple computational nodes as different partitions to be processed in parallel.

Iterative algorithms can be naturally implemented in a synchronous system through a sequence of iterations separated by explicit barriers. The workloads at the $(k+1)$th iteration consist 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 to generate a message from $i$ to $j$ based on $v_i$ and compute vertex states, respectively. Iterations terminate until a fixed point is reached, i.e., $v_j$ does not change between two consecutive iterations.

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

Taking PageRank [15] (PR) as an example, it computes a score, i.e., $v_j$, for every web page $j$ to evaluate $j$'s importance. At the $k$th iteration, every page $i$ sends its tentative score divided by its out-degree along outgoing links, i.e., $g_{\{i,j\}}(v_i^k) = d \cdot \frac{v_i^k}{|\Gamma_i^{out}|}$, 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., $v_j^{k+1} = \frac{1-d}{|V|} + \sum_{i \in \Gamma_j^{in}} (d \cdot \frac{v_i^k}{|\Gamma_i^{out}|})$. Here, '$\oplus$' is '+' and $c_j = \frac{1-d}{|V|}$ for $j$. Theoretically, when the summation over all scores is 1, the fixed point is reached. In practice, we usually terminate iterations in advance when the summation is close enough to 1.

Maiter [5], on the other hand, employs a delta-based asynchronous model. Messages in Maiter are generated based on the "change" of $v_i$ (delta), to avoid repeatedly processing messages from unchanged $v_i$. Also, synchronization barriers are completely 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 $\Delta v_j$, where $\Delta v_j$ indicates the "change" of $v_j$ since $v_j$'s last update. $\Delta v_j$ 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 $\Delta v_j$. $\Delta v_j$ is further forwarded to out-neighbors and then reset to **0** to accumulate newly received messages. **0** is an abstract

TABLE 1
Example Graph Algorithms

| Alg. | $c_j$ | $g_{\{i,j\}}(x)$ | $f_{\{j\}}(x)$ | $\oplus$ | $\ominus$ | 0 |
|---|---|---|---|---|---|---|
| PR | $(1-d)$ | $d \cdot \frac{x}{\lvert\Gamma^{out}(j)\rvert}$ | $x > 0$ | $+$ | $-$ | 0 |
| PHP | $1\ (j = s)$ or $0\ (j \neq s)$ | d$\cdot x \cdot w(i,j)$ $(j \neq s)$ or $0\ (j = s)$ | ″ | ″ | ″ | ″ |
| Katz | ″ | $d \cdot x$ | ″ | ″ | ″ | ″ |
| SSSP | $0\ (j = s)$ or $\infty\ (j \neq s)$ | $w(i,j) + x$ $(x < v_i)$ or $\infty\ (x \geq v_i)$ | $x < v_i$ | min | $x \ominus y = x$ | $\infty$ |
| CC | $j$ | x $(x > v_i)$ or $-1\ (x \leq v_i)$ | $x > v_i$ | max | $x \ominus y = x$ | $-1$ |

zero value satisfying that $x \oplus \mathbf{0} = x$ and $g_{\{i,j\}}(\mathbf{0}) = \mathbf{0}$. For PR, it is 0.

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

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

$g_{\{i,j\}}(\Delta v_i)$ is always closer to $\mathbf{0}$ than $\Delta v_i$. 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 $\Delta v_j$ 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 prioritizing the update order of vertices can accelerate the convergence speed [16], because vertices with large "change" (i.e., $\Delta v_j$) play an important role in determining $v_j$.

## 2.2 Challenges of Fault-Tolerance

Maiter employs a widely used Master-Slave framework design where a master node is in charge of monitoring the cluster health by periodically checking the status information collected from slave nodes. Although Master, Network and Slaves may fail, this paper focuses on the last two only because the whole system will crash when Master fails.

It is challenging to recover failures because the memory-resident data on failed nodes are immediately lost upon failures. To recover them and continue computations, a conventional approach is to archive data periodically beforehand and roll back vertex states to the last available checkpoint [1], [4], [17], [18], [19]. However, existing checkpoint-based methods suffer from expensive archiving overheads and/or costly rolling back operations.

## 2.3 Example Algorithms

Table 1 lists a series of well-known graph algorithms that can be supported by our fault-tolerance solutions. Here, $\ominus$ and $f_{\{j\}}(x)$ are operators required to tolerate failures. We will introduce them in Sections 3 and 4, respectively. Since PR has been given as an example in Section 2.1, we now describe other algorithms.

*Penalized Hitting Probability* (PHP) [20]: 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 a 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 a constant proximity value 1. Both PR 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.

*Katz Metric* (Katz) [21]: Katz is another proximity measure. The score value is computed as the sum over the collection of paths between a given source $s$ and any other vertex $j$.

*Single Source Shortest Path* (SSSP) [1]: SSSP finds the shortest distance between a given source to any other one. Initially, the source has the shortest distance 0 as its value which is broadcasted to out-neighbors. In remaining computations, a vertex minimizes its distance based on the values of in-neighbors.

*Connected Components* (CC) [22]: CC finds all connected components in an undirected graph. The component id of each vertex is initialized by its own unique vertex id. Then a vertex maximizes its component id based on the values of its direct neighbors.

## 3 Fast Failure Recovery

This section introduces two failure recovery methods for the delta-based asynchronous model. In particular, we prove correctness and give performance analysis.

## 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 partition. Afterwards, one of our recovery methods (Sections 3.1.1 and 3.1.2) is invoked to restart computations.

### *3.1.1 Failure Recovery without Rolling Back*

In the scenario where no checkpoint is archived, a traditional way of tolerating failures is to recompute from scratch ($v_j^0 = \mathbf{0}$, $\Delta v_j^0 = c_j$), that is referred to as FR-Scratch. FR-Scratch is inefficient because workloads on surviving nodes are discarded, and re-performing them wastes resources. By contrast, we present a failure recovery method called FR-WORB where only lost vertices on failed nodes are recomputed from scratch while updates of surviving vertices can keep going without rollback.

Compared with FR-Scratch, FR-WORB avoids recomputing surviving vertex states. However, because surviving vertex states are not rolled back to $\mathbf{0}$, a key issue is how to guarantee the correctness, i.e., an algorithm under FR-WORB can converge to the same fixed point as reached under FR-Scratch. We solve this problem by designing a special restarting point. 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 a surviving node, 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 set of failed nodes and the set of surviving nodes, 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 PR, $\ominus$ is '-'. $\tilde{v}_j^1$ 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 $\Delta\tilde{v}_j^0$. Once every $\Delta\tilde{v}_j^0$ is ready, we can take $(\tilde{v}_j^0, \Delta\tilde{v}_j^0)$ as input to resume computations.

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

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

In the context of asynchronous computations, flushing operations are required to correctly run a synchronous iteration to compute $\tilde{v}_j^1$. That is, as a *preprocessing phase* before building the restarting point, at $t_f$, $g_{\{i,j\}}(\Delta v_i)$ in network and $\Delta v_j$ 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). The two types of messages must be separated to correctly compute $\tilde{v}_j^1$. Also, the current $\Delta v_j$ on surviving nodes perhaps stores the value based on $g_{\{i,j\}}(\Delta v_i)$. It needs to be cleared, i.e., being reset to $\mathbf{0}$, in order to accumulate the newly received message $g_{\{i,j\}}(\tilde{v}_i^0)$.

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 the sending buffer are flushed and then an EOF notification is broadcasted to all nodes 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 $g_{\{i,j\}}(\Delta v_i)$, $\Delta v_j$ is reset to $\mathbf{0}$ locally to clear its value. In particular, when calculating $\tilde{v}_j^1$, flushing $g_{\{i,j\}}(\tilde{v}_i^0)$ is also required to guarantee that each vertex has received all possible messages from in-neighbors.

Note that before the *preprocessing phase*, many regular messages $g_{\{i,j\}}(\Delta v_i)$ from failed nodes have already been received and accumulated into the vertex state $v_j$ on surviving nodes. "Changes" indicated by such messages will be propagated again by the replacements of failed nodes. Clearly, accumulating these "changes" twice may affect correctness. For example, in PR, the summation over all scores will be greater than 1. Flushing on-going messages and resetting the message-receiving variable $\Delta v_j$ cannot work on the vertex state $v_j$. However, the newly constructed delta value $\Delta\tilde{v}_j^0$ in our restarting point can eliminate the impact on correctness. Intrinsically, $\Delta\tilde{v}_j^0$ given by Eq. (5) indicates the opposite number of the accumulated value (*diff*) of messages which are sent out by failed nodes and have already been processed by surviving nodes. When resuming computations, $\Delta\tilde{v}_j^0$ as input is accumulated into the surviving $v_j$. The latter then immediately removes *diff* from itself to avoid repeatedly accumulating it. However, values of messages among surviving nodes are preserved without rollback. This is the key factor that our solution can guarantee correctness. Theorem 1 in Section 3.2.1 will give a formal proof.

### *3.1.2 Failure Recovery with Asynchronous Checkpointing*

In FR-WORB, no checkpoint is archived during computations. As a result, the only way of initializing lost states is to reset them to the initial value $\mathbf{0}$. The heavy recovery workloads impact the performance, even though surviving vertices are leveraged.

Now we introduce an improved method to further boost the recovery efficiency by archiving data during the *failure-free* execution. It is inspired by checkpointing. In general, the checkpointing method archives state $v_j$ as checkpoint periodically. Upon failures, all nodes load the lost partition and then resume computations from the last available checkpoint, instead of scratch. The recovery workloads are thereby reduced. However, existing techniques need to block updating $v_j$ when archiving it. This requires a synchronization barrier to coordinate the progress of each node, which largely degrades the performance of the *failure-free* execution. Unlike them, our method asynchronously archives $v_j$, termed FR-WAC. It avoids recovering lost data from scratch. Meanwhile the impact of archiving data on performance can be negligible.

In FR-WAC, each node individually archives its local vertex state value $v_j$ based on a user-specified interval $\tau$. There is no global synchronization barrier [1] or any other protocol [4] to coordinate the progress. Hence, FR-WAC can quickly perform a complete checkpoint. Furthermore, on one node, a separate thread is launched to accomplish the archiving operation. It runs in fully parallel with the message-receiving (Eq. (2)) and vertex-updating (Eq. (3)) threads. Then the underlying computation is performed progressively without pausing. Upon failures, 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 checkpoint archived at $t_x$, and $t_x \leq t_f$.

In synchronous checkpointing solutions, $\tau$ is a key parameter to balance the tradeoff between the recovery efficiency and the archiving costs. It is a non-trivial task to set a reasonable value. However, we will experimentally show that a quite large range of $\tau$ can allow FR-WAC to strike a good balance.

## 3.2 Recovery Analysis

### *3.2.1 Correctness*

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

**Theorem 1.** *An algorithm using* FR-WORB *can converge to the same fixed point as that using* FR-Scratch*.*

**Proof.** The difference between FR-WORB and FR-Scratch can be reduced to the different restarting points, i.e., $(\tilde{v}_j^0, \Delta\tilde{v}_j^0)$ and $(\mathbf{0}, c_j)$, respectively. Let $\tilde{v}_j^\infty$ be the state value of vertex $j$ in the fixed point. Then we can prove this theorem if $\tilde{v}_j^\infty$ computed by $(\tilde{v}_j^0, \Delta\tilde{v}_j^0)$ is the same with that by $(\mathbf{0}, c_j)$. Below, we first derive $\tilde{v}_j^\infty$ based on $(\tilde{v}_j^0, \Delta\tilde{v}_j^0)$.

We first analyze the state value of $j$ at time $t_k$, $\tilde{v}_j^k$. As shown in Eqs. (2) and (3), delta-based messages are transferred from one vertex to another along the edge between them, and are used to update vertex states. The received messages are forwarded again to propagate information. Thus, at $t_k$, $\tilde{v}_j^k$ has received some $\Delta\tilde{v}^0$-based messages, where every $\Delta\tilde{v}^0$ is originally

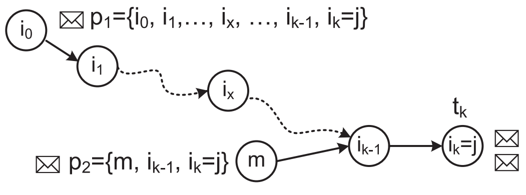


Fig. 1. Illustration of two paths from different sources to $i_k = j$.

provided by one of $j$'s direct or indirect neighbors. To demonstrate this process mathematically, we introduce "path". A path $p$ is a sequence of vertices where a message is transferred. Vertices in $p$ are sorted in an ascending order of the message-receiving times. There exists an edge between any two adjacent vertices in $p$. Fig. 1 shows two paths $p_1 = \{i_0, i_1, \ldots, i_x, \ldots, i_{k-1}, i_k = j\}$ and $p_2 = \{m, i_{k-1}, i_k = j\}$ from different sources $i_0$ and $m$ to the same target $i_k = j$. Taking $p_1$ as an example, a message $g(\Delta\tilde{v}^0_{i_0})$ is sent from $i_0$ at time $t_{i_0}$ and then forwarded along $i_1, i_2, \ldots$, to $i_k = j$ by recursively applying $g$, at times $t_{i_1}, t_{i_2}, \ldots$, and $t_{i_k} = t_k$. Here $t_{i_0} < t_{i_1} < \ldots < t_k$. Similarly, for $p_2$, we have $t'_m < t'_{i_{k-1}} < t_k$. At time $t_k$, $p_1$ is available for $j$ if $j$ has received the message originating from $i_0$. The number of a path $p$'s hops equals to $(|p| - 1)$. Then we can use $P(j, l)$ to stand for a set of $l$-hop available paths to $j$. Based on Eqs. (2) and (3), $\forall p \in P(j, l)$, the message value transferred from $i_0$ to $j$ along $p$ can be computed by recursively applying the message generating function $g_{\{i_x, i_{x+1}\}}$:

$$\begin{aligned}\prod_p(\Delta\tilde{v}^0_{i_0}) &= \prod_{\substack{i_x, i_{x+1} \in p \\ and\, x=0}}^{l-1} g_{\{i_x, i_{x+1}\}}(\Delta\tilde{v}^0_{i_0}) \\ &= g_{\{i_{l-1}, j\}}\Big(\ldots g_{\{i_1, i_2\}}\big(g_{\{i_0, i_1\}}(\Delta\tilde{v}^0_{i_0})\big)\Big)\end{aligned}$$

At $t_k$, the maximum value of $l$ is $k$ according to the definition of $p$. Besides, at $t_0$, $j$ accumulates the initial input $\Delta\tilde{v}^0_j$ into its $\tilde{v}^0_j$. Together, we have the expression of $\tilde{v}^k_j$ in Eq. (6).

$$\tilde{v}^k_j = \tilde{v}^0_j \oplus \Delta\tilde{v}^0_j \oplus \sum_{l=1}^{k} \oplus \left( \sum_{p \in P(j,l)} \oplus \left( \prod_p (\Delta\tilde{v}^0_{i_0}) \right) \right) \tag{6}$$

Further, to guarantee that all messages have been received at the fixed point, a vertex $i_0$ must perform Eqs. (2) and (3) an infinite number of times until its $\Delta\tilde{v}^\infty_{i_0}$ is $\mathbf{0}$. We thereby derive $\tilde{v}^\infty_j$ by considering all possible paths with hops from 0 to $\infty$, as described in Eq. (7).

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

We then explore the relationship between $\tilde{v}^\infty_j$ and ($\tilde{v}^0_j$, $\Delta\tilde{v}^0_j$). Based on Eq. (5), the expression for $\Delta\tilde{v}^0_{i_0}$ is

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

Compared with FR-Scratch ($\tilde{v}^0_{i_0} = \mathbf{0}, \Delta\tilde{v}^0_{i_0} = c_{i_0}$), the difference in inputs is caused by the values of $\tilde{v}^0_n$ and $\tilde{v}^0_{i_0}$. In the following, we will show that the difference can always be eliminated by messages along some pairs of paths $(p, p')$.

- Path $p$: $g_{\{i,j\}}$ has the *distributed* property over '$\oplus$' and '$\ominus$', and the latter two operations have the *communicative*and the *associate* properties. Given an $l$-hop path $p = \{i_0, i_1, ..., i_l = j\}$, we can substitute the expression for $\Delta\tilde{v}^0_{i_0}$ to expand $\prod_p(\Delta\tilde{v}^0_{i_0})$ as

$$\begin{aligned}\prod_p(\Delta\tilde{v}^0_{i_0}) = &\left( \sum_{n \in \Gamma^{in}_{i_0}} \oplus \Big( \prod_p \big(g_{\{n, i_0\}}(\tilde{v}^0_n)\big)\Big)\right) \\ &\oplus \prod_p(c_{i_0}) \ominus \prod_p(\tilde{v}^0_{i_0}).\end{aligned} \tag{8}$$

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

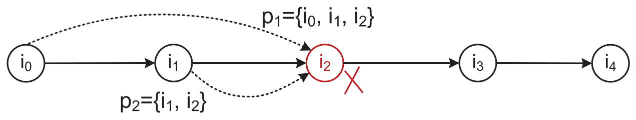
$$\begin{aligned}\prod_{p'}(\Delta\tilde{v}^0_n) = &\left( \sum_{m \in \Gamma^{in}_n} \oplus \Big( \prod_{p'} \big(g_{\{m,n\}}(\tilde{v}^0_m)\big)\Big)\right) \\ &\oplus \prod_{p'}(c_n) \ominus \prod_{p'}(\tilde{v}^0_n)\end{aligned} \tag{9}$$

Note that $\prod_p \big(g_{\{n,i_0\}}(\tilde{v}^0_n)\big)$ in Eq. (8) equals $\prod_{p'}(\tilde{v}^0_n)$ in Eq. (9). Besides, the two messages $\prod_p(\Delta\tilde{v}^0_{i_0})$ and $\prod_{p'}(\Delta\tilde{v}^0_n)$ corresponding to $(p, p')$ are definitely accumulated into $\tilde{v}^\infty_j$, as shown in Eq. (7). Because of the reversibility property of '$\oplus$' and '$\ominus$', the item $\prod_p(\tilde{v}^0_{i_0})$ is immediately eliminated. Further, when values along all $(l+1)$-hop paths starting from $i_0$'s in-neighbors have been received, $\sum_{n \in \Gamma^{in}_{i_0}} \oplus (\prod_p \big(g_{\{n,i_0\}}(\tilde{v}^0_n)\big))$ can be removed. As a result, after accumulating values transferred along all paths to $j$, including $\tilde{v}^0_j$ and $\Delta\tilde{v}^0_j$, we can infer $\tilde{v}^\infty_j$ as shown in Eq. (10).

$$\begin{aligned}\tilde{v}^\infty_j = c_j \oplus &\left( \sum_{l=1}^{\infty} \oplus \left( \sum_{p \in P(j,l)} \oplus \left( \prod_p (c_i) \right) \right) \right) \\ \oplus &\left( \sum_{p'(n) \in P(j,\infty)} \oplus \left( \prod_{p'(n)} \left( \sum_{n \in \Gamma^{in}_i} \oplus g_{\{n,i\}}(\tilde{v}^0_n) \right) \right) \right)\end{aligned} \tag{10}$$

Here, $p'(n) = \{n, i, ..., j\}$. Since $g_{\{i,j\}}(x)$ is always closer to $\mathbf{0}$ than $x$, $\prod_{p'(x) \in P(j,\infty)}(x) \to \mathbf{0}$. Hence, $\tilde{v}^\infty_j$ only depends on $c_x$ ($x \in V$) which is the same as that in FR-Scratch. Then we have the claim. □

We use a graph with five vertices $\{i_0, i_1, i_2, i_3, i_4\}$ to demonstrate the correctness guarantee of PR. Let $d = 0.8$. As shown in Fig. 2, the initial input at $t_0$ is ($v^0 = 0, \Delta v^0 = c = \frac{1-d}{|V|} = 0.04$). At time $t_k$, for example, $i_2$ has accumulated the message values along two paths $p_1$ ($0.04 \times 0.8 \times 0.8 = 0.0256$) and $p_2$ ($0.04 \times 0.8 = 0.032$), and its local $\Delta v^0_{i_2} = 0.04$. Now assume that a failure happens and then $v_{i_2}$ is lost. Using FR-WORB, it is re-initialized by $\tilde{v}^0_{i_2} = 0$ at time $\tilde{t}_0$, and other vertex states keep invariant. After running a synchronous iteration, at $\tilde{t}_1$, we can re-construct $\Delta\tilde{v}^0$ by $(\tilde{v}^1 - \tilde{v}^0)$. Taking $i_2$ as an example, $\Delta\tilde{v}^0_{i_2} = d \cdot \tilde{v}^0_{i_1} - \tilde{v}^0_{i_2} + c$. "$-\tilde{v}^0_{i_2}$" can be immediately eliminated when $\Delta\tilde{v}^0_{i_2}$ is accumulated into $\tilde{v}^0_{i_2}$. Therefore, the difference between $\Delta\tilde{v}^0_{i_2}$ and the initial input $\Delta v^0_{i_2} = c$ is $(d \cdot \tilde{v}^0_{i_1})$.



| time | $i_0$ | $i_1$ | $i_2$ | $i_3$ | $i_4$ |
|---|---|---|---|---|---|
| $t_0$ | (0, 0.04) | (0, 0.04) | (0, 0.04) | (0, 0.04) | (0, 0.04) |
| ... | | | | | |
| $t_k$ | (0.04, 0) | (0.072, 0) | (0.0976, 0) | (0.11808, 0) | (0.134464, 0) |
| $\tilde{t}_0$ | (0.04, 0) | (0.072, 0) | (0, 0) | (0.11808, 0) | (0.134464, 0) |
| $\tilde{t}_1$ | (0.04, 0) | (0.072, 0) | (0.0976, 0) | (0.04, 0) | (0.134464, 0) |
| | (0.04, 0) | (0.072, 0) | (0, 0.0976) | (0.11808, -0.07808) | (0.134464, 0) |

Fig. 2. Failure recovery of PR.

That can be eliminated after $i_2$ receives the message from its in-neighbor $i_1$ along $p_2$, i.e., $d \cdot \Delta\tilde{v}_{i_1}^0 = d\big(d \cdot \tilde{v}_{i_0}^0 - \tilde{v}_{i_1}^0 + c\big)$. Similarly, $d^2 \cdot \tilde{v}_{i_0}^0$ can be removed once the message from $i_0$ along $p_1$ is available. Clearly, like FR-Scratch, the algorithm convergence depends on $c$ only.

The only difference between FR-WORB and FR-WAC is the value of $\tilde{v}_n^0$ for every failed vertex. Since $\tilde{v}_j^\infty$ is independent of $\tilde{v}_n^0$, we can easily infer that algorithms under FR-WAC also converge.

### 3.2.2 Performance Analysis

Asynchronous models have removed barriers so as to allow vertices to freely update themselves. Then vertex update behaviors are not deterministic. When restarting computations, the vertex update sequence is not exactly the same as before failures. Therefore, it is difficult to compare the performance of different fault-tolerance solutions with a strict theoretical analysis. However, taking PR as an example, we can analyze the advantages of our methods.

We first compare FR-WORB against FR-Scratch. Both of them recompute lost states from scratch, i.e., $\tilde{v}_j^0 = \mathbf{0} = 0$. However, their input delta values are very different. That is, $\Delta\tilde{v}_j^0$ equals $c$ for FR-Scratch, while $\big(c + d\sum_{i\in\Gamma_j^{in}}(\tilde{v}_i^0/|\Gamma_i^{out}|)\big)$ for FR-WORB based on Eq. (5). For $j$, if at least one in-neighbor (e.g., $i$) resides on a surviving node, then we have $\tilde{v}_i^0 > 0$ for FR-WORB. Therefore, $\Delta\tilde{v}_j^0$ used in FR-WORB is greater than or at least equal to that in FR-Scratch. The goal of PR is continuously accumulating $\Delta\tilde{v}_j$ into $\tilde{v}_j^0$ so that the latter can converge to $\tilde{v}_j^\infty$. Clearly, a large initial $\Delta\tilde{v}_j^0$ used in FR-WORB can potentially speed up the convergence.

By contrast, FR-WAC provides a non-negative $\tilde{v}_j^0$ for $j$ on failed nodes. Then $\Delta\tilde{v}_j^0 = \big(c + d\sum_{i\in\Gamma_j^{in}}(\tilde{v}_i^0/|\Gamma_i^{out}|)\big) - \tilde{v}_j^0$. However, once resuming computations, by accumulating $\Delta\tilde{v}_j^0$ into $\tilde{v}_j^0$, "$-\tilde{v}_j^0$" in $\Delta\tilde{v}_j^0$ is immediately eliminated. FR-WAC can enjoy the same initial delta value as FR-WORB. Further, given any out-neighbor $m$ of $j$, based on Eq. (5), its $\Delta\tilde{v}_m^0$ is greater than that used in FR-WORB, because $\tilde{v}_j^0 \geq 0$. $m$ in FR-WAC potentially converges faster than that in FR-WORB. If there exist paths from $m$ to vertices on failed nodes, then such vertices can also converge faster. As a result, FR-WAC beats FR-WORB if we ignore the cost of archiving data.

## 4 Optimizations

This section introduces three optimizations to further boost the recovery efficiency, including message pruning (Section 4.1), non-blocking recovery (Section 4.2) and load balancing (Section 4.3).

### 4.1 Message Pruning

We first point out there exist some messages that can be pruned after restarting computations upon failures, and then present our pruning technique in detail.

*Analyzing Communication Behaviors*: In FR-WORB and FR-WAC, $\tilde{v}_j^0$ is given by Eq. (4) to preserve vertex states on surviving nodes. However, some of contributions accumulated into $\tilde{v}_j^0$ come from in-neighbors on failed nodes. We should purge $\tilde{v}_j^0$ of such contributions because now they are not available. Towards this end, we need to construct the new delta value $\Delta\tilde{v}_j^0$ carefully. Indeed, $\Delta\tilde{v}_j^0$ given by Eq. (5) logically can be divided into two parts: $(\sum_{i\in\Gamma_j^{in}} \oplus\, g_{\{i,j\}}(\tilde{v}_i^0) \ominus \tilde{v}_j^0)$ and $c_j$. The former figures out how many contributions come from failed in-neighbors and then it is propagated for a through purge. Meanwhile, vertices as in-neighbors will broadcast $c_j$ again, the second part in $\Delta\tilde{v}_j^0$, to compensate lost contributions. Theorem 1 tells us that the impact of purge and compensation messages will be definitely offset at the message-receiving vertex $j$, as shown in Eqs. (8), (9), and (10), which guarantees the algorithm convergence. But before that, the two kinds of messages possibly exist in a system for quite a while. This is because for both FR-WORB and FR-WAC, the purge message is immediately broadcasted once $\Delta\tilde{v}_j^0$ is constructed. It will take some time for compensation messages to catch up with purge messages to offset the impact of the latter. Processing the two kinds of messages of course consumes network bandwidth and CPU resources. Fortunately, as we will show later, some of such messages can be pruned for efficiency.

*Pruning Strategy*: The property that purge and compensation can be offset naturally motives us to keep all purge messages in their corresponding generation locations to wait for compensation messages. In this way, the latter can catch up with the former as soon as possible, but more importantly, there is no communication cost caused by purge. Further, before compensation is accomplished, two special designs are necessary for efficiency and correctness. First, we do not update the local vertex state $\tilde{v}_j$, because the operation of purging $\tilde{v}_j$ of lost contributions will be offset later by compensation. Then we can save CPU time. Second, all received compensation messages will not be forwarded since they are used to offset the impact of the local cached purge message.

*Correctness*: Theorem 2 establishes the correctness.

**Theorem 2.** *An algorithm under* FR-WORB *with pruning converges to the same fixed point as that under* FR-Scratch.

**Proof.** Different from FR-WORB, before compensation is accomplished, the pruning variant blocks all processing operations on purge and compensation messages, including propagation among vertices and consumption involved in vertex updates. Hence, this theorem can be proved if, for each vertex $j$, there exists a time instance when the impacts of the two kinds of blocked messages are offset (i.e., compensation is done) and then subsequent messages will be processed in the same way as FR-WORB.

Below, we first mathematically give the lost contributions for $j$ by analyzing elements in $\tilde{v}_j^0$ and $\sum_{i\in\Gamma_j^{in}} \oplus\, g_{\{i,j\}}(\tilde{v}_i^0)$. Suppose that $j$ resides on a surviving node when failures occur at $t_f$. Then at the restarting point,

$\tilde{v}_j^0 = v_j^f$ based on Eq. (4), where $v_j^f$ is the state value of $j$ right before failures. Following Eq. (6) in Theorem 1, given the initial input ($v_j^0 = \mathbf{0}$, $\Delta v_j^0 = c_j$), $v_j^f$ is

$$v_j^f = c_j \oplus \sum_{l=1}^{f} \oplus \left( \sum_{p \in P(j,l)} \oplus \left( \prod_p (c_{i_0}) \right) \right)$$

By Eq. (3), we know any message $\prod_p (c_{i_0})$ in $\tilde{v}_j^0$ or $v_j^f$ is forwarded by one of $j$'s direct in-neighbor $i$, by applying the function $g$ on $i$'s received message $\prod_{p'}(c_{i_0})$. Here, $p$ is the path passing through $i$, i.e., $p = \{i_0, i_1, \ldots, i_{l-2}, i_{l-1} = i, i_l = j\}$, and $p' = p/\{j\}$. $\prod_{p'}(c_{i_0})$ has already been accumulated into $v_i^f$. If $i$ also resides on a surviving node, then $\tilde{v}_i^0 = v_i^f$ and hence $g_{\{i,j\}}(\tilde{v}_i^0)$ contains $g_{\{i,j\}}(\prod_{p'}(c_{i_0}))$ which equals $\prod_p(c_{i_0})$. As a result, $\prod_p(c_{i_0})$ is removed from the result of $(\sum_{i \in \Gamma_j^{in}} \oplus\, g_{\{i,j\}}(\tilde{v}_i^0) \ominus \tilde{v}_j^0)$. Otherwise, $\tilde{v}_i^0 = \mathbf{0}$. Then $\prod_p(c_{i_0})$ as the lost contribution from $i$ exists in the comparison result. We denote by $\bigcup_{N \in \mathbb{N}_F} V_N$ the set of vertices on failed nodes, where $\mathbb{N}_F$ is the set of failed nodes. Eq. (11) shows the total lost contributions for a surviving vertex $j$.

$$Lost = c_j \oplus \sum_{l=1}^{f} \oplus \left( \sum_{\substack{p \in P(j,l) \\ \wedge i \in p}} \oplus \left( \prod_p (c_{i_0}) \right) \right), \quad s.t.\ i \in \Gamma_j^{in} \wedge i \in \bigcup_{N \in \mathbb{N}_F} V_N \tag{11}$$

We then discuss how and when we can compensate such lost contributions. First, $c_j$ is locally compensated because it is one element in $\Delta\tilde{v}_j^0$ given by Eq. (5) at the restarting point. Second, $\prod_p(c_{i_0})$ is compensated by $g_{\{i,j\}}(\prod_{p'}(c_{i_0}))$ if $\prod_{p'}(c_{i_0})$ is contained in $\tilde{v}_i$. Recursively, the condition is satisfied if $\prod_{p''}(c_{i_0})$ is available in $i$'s direct in-neighbor $i_{l-2}$, where $p'' = p/\{i,j\}$. Finally, because $c_{i_0}$ as the compensation part in $\Delta\tilde{v}_{i_0}^0$ is always accumulated into $\tilde{v}_{i_0}$, $\prod_p(c_{i_0})$ is compensated in a recursive way. When $c_j$ and all $\prod_p(c_{i_0})$ in Eq. (11) are collected, compensation is done, i.e., the impact of the local purge message is offset. Then subsequent messages along any path can be processed as normal.

Note that if $j$ is kept by a failed node, the compensation is immediately accomplished once the local $c_j$ or any message value is received, because now $\tilde{v}_j^0$ is $\mathbf{0}$. □

As shown in Fig. 2, the re-constructed delta value $\Delta\tilde{v}_{i_3}^0 = -0.07808 < 0$. This is because the message values along paths $p_1 \cup \{i_3\}$ $(0.04 \times 0.8^3 = 0.02048)$, $p_2 \cup \{i_3\}$ $(0.04 \times 0.8^2 = 0.0256)$ and $\{i_2, i_3\}$ $(0.04 \times 0.8 = 0.032)$, are not available. However, $c = 0.04$ is included in $\Delta\tilde{v}^0$ for every vertex. When the three paths are available again for $i_3$, $\Delta\tilde{v}_{i_3}$ becomes positive, i.e., the lost contributions have been compensated.

Similarly, the correctness of FR-WAC with pruning can also be guaranteed by replacing the value of $\tilde{v}_i^0$ with checkpoint data.

*Implementation*: Our experiments in Section 5 show that the pruning technique works well. However, it requires users to provide more algorithm-level details. In the original FR-WORB/FR-WAC, we do not need to judge when the lost contributions of a vertex have been compensated, because definitely the compensation will be accomplished at the fixed point. However, in the variants with pruning, that is a key issue as it decides when vertices can update themselves and forward received messages. The logic of the judging function $f$ varies with algorithms, which must be specified by users. Further, different from the theoretical analysis in Eq. (11), in practice, message values accumulated into the vertex state do not exist separately. Instead, they might be accumulated when being transferred and then processed as a whole. Hence, we cannot judge the compensation progress by analyzing each element in Eq. (11). In the following, we show how to implement our pruning technique for reversible and irreversible algorithms, respectively.

- Reversible algorithms: FR-WORB and FR-WAC are designed for reversible algorithms where '$\oplus$' and '$\ominus$' are reversible, because of frequent purge and compensation operations on vertex states. Both purge messages and compensation messages are first accumulated into $\Delta\tilde{v}_j$ and then $\tilde{v}_j$. Clearly, $\Delta\tilde{v}_j$ identifies how many contributions are still lost at any time. The judging function, $f_{\{j\}}(\cdot)$, thereby takes $\Delta\tilde{v}_j$ as input. Then messages can be received and accumulated as shown in Eq. (2). However, vertex update and message propagation in Eq. (3) cannot be triggered until $f_{\{j\}}(\cdot)$ returns "true" and $\Delta\tilde{v}_j$ is not $\mathbf{0}$. We demonstrate the pruning process exemplified by PR. In the *failure-free* execution, $\Delta v_j$ is always greater than zero because the pagerank score monotonously increases from 0. However, upon failures, it is possible that $\Delta\tilde{v}_j < 0$ because scores from some in-neighbors are not available. In FR-WORB and FR-WAC with pruning, the negative $\Delta\tilde{v}_j$ is held by $j$. We do not accumulate it into $\tilde{v}_j$ or send it, until it becomes positive after accumulating enough messages. The $f$ function is defined as: true if $\Delta\tilde{v}_j > 0$; false otherwise.
- Irreversible algorithms: Many algorithms cannot satisfy the reversibility requirement. For example, SSSP and CC in Table 1 have the irreversible "min" and "max" operators. Intuitively, we can also judge whether the compensation is done by directly comparing $\tilde{v}_j^1$ with $\tilde{v}_j^0$. If the former has accumulated more messages than the latter, then the lost contribution of $j$ is compensated. In this way, we can completely discard the purge operation and hence remove the reversibility constraint. Because comparing $\tilde{v}_j^1$ with $\tilde{v}_j^0$ is always feasible, our pruning strategy can extend FR-WORB and FR-WAC to any irreversible algorithm supported by Maiter. Besides, we can store $\tilde{v}_j^1$ as $\Delta\tilde{v}_j^0$ to save memory resources. To do this, we define '$\ominus$' as $\tilde{v}_j^1 \ominus \tilde{v}_j^0 = \tilde{v}_j^1$, to initialize $\Delta\tilde{v}_j^0$ with $\tilde{v}_j^1$. $\Delta\tilde{v}_j$ then computes the real-time vertex state based on received messages. When $f$ returns true, we update the local $\tilde{v}_j$ based on $\Delta\tilde{v}_j$ and then re-initialize the latter to normally accumulate messages. Taking SSSP as an example, lost contributions are compensated if a distance smaller than $\tilde{v}_j^0$ is found. That is, $f$ returns true if $\Delta\tilde{v}_j < \tilde{v}_j^0$; false otherwise.

*Performance Analysis*: Because of the non-deterministic update behaviors, we use PR as an example to show that the pruning technique improves the recovery efficiency. For PR, based on Eq. (5), the constructed delta value $\Delta\tilde{v}_j^0 < 0$ if $\left((1-d)/|V| + d\sum_{i\in\Gamma_j^{in}}(\tilde{v}_i^0/|\Gamma_i^{out}|)\right) < \tilde{v}_j^0$. Then the judging function $f$ returns false to avoid sending $\Delta\tilde{v}_j^0$ as well as updating $\tilde{v}_j$, that saves the network bandwidth and CPU time. The condition is often satisfied when the failure happens at the late phase of computations and $j$ resides on a surviving node. This is because $v_j$ can accumulate many message values and be reserved as $\tilde{v}_j^0$. Experiments in Section 5 validate that the pruning technique works well especially when a single node fails at the late phase of computations.

### 4.2 Non-Blocking Recovery without Standby Nodes

*Using Standby Nodes or Not?* Today's distributed systems typically require spare capacity (technically called standby nodes). Upon failures, computations on failed nodes can be quickly restarted on standbys. Although such standby nodes keep empty during the *failure-free* execution, we still need to pay for them on public cloud platforms such as Amazon EC2. Otherwise, we must acquire new resources when failures occur, which takes up to 90 seconds as reported [23]. Worse, configuring systems requires additional efforts. Clearly, before new nodes are available, the graph algorithm cannot proceed any more.

As analyzed above, there exists a tradeoff between the expense and the recovery efficiency. Therefore, a naturally desirable goal of tolerating failures is to pursue a solution that (1) avoids provisioning spare capacity to save the budget; and (2) can fully utilize the restarting time (resource acquiring time and configuring time) to perform the recovery progressively.

*Non-Blocking Recovery without Standby Nodes.* We achieve our goal by maximally unleash the computational power of surviving nodes. That is, data on failed nodes are temporarily managed by surviving nodes. Surviving nodes spend time doing more useful computations instead of waiting for preparing replacements of failed nodes. Such computations include performing the preprocessing work and re-constructing input for recovery (see Section 3.1.1), and then updating vertices by Eqs. (2) and (3).

One challenge is that perhaps a surviving node will handle more data than its pre-assigned capacity, if it loads any partition originally assigned to a failed node. Keeping the new partition in memory inevitably consumes additional resources, which is prohibitively expensive especially in multi-tenant environments. Our solution is to utilize the local disk. Specifically, for any node with two or more partitions, one partition is computed in memory while others are stored on local disk. These partitions are scheduled between memory and disk in a round-robin scheme with a scheduling interval $\lambda$.

*Discussion on Scheduling Interval*: $\lambda$ is a key parameter which decides how long a partition is kept in memory. We first discuss its range. Let $L$ denote the initialization overhead of loading a partition into memory. Then $\lambda > L$ so that we can perform useful updates in the remaining time $(\lambda - L)$. On the other hand, it is necessary to broadcast vertex states, in order to re-construct delta values, as shown in Eqs. (4) and (5). We need to schedule partitions between memory and disk multiple times to guarantee that all partitions can receive messages. Specifically, if $|\mathbb{N}_F|$ nodes fail, then the $|\mathbb{N}_F|$ lost partitions are evenly loaded by $|\mathbb{N}_S|$ surviving nodes, and each of the latter totally manages $(1+\lceil\frac{\mathbb{N}_F}{\mathbb{N}_S}\rceil)$ partitions. Hence, $(2+\lceil\frac{\mathbb{N}_F}{\mathbb{N}_S}\rceil)$ times of scheduling operations are required. Let $\mathbb{R}$ stand for the restarting cost. We have $\mathbb{R} > (2+\lceil\frac{\mathbb{N}_F}{\mathbb{N}_S}\rceil)\cdot\lambda$ to ensure that we can finish reconstructing delta values. Together, Eq. (12) gives the range of $\lambda$. $\mathbb{R}$ can be estimated by history, while $L$, $\mathbb{N}_F$ and $\mathbb{N}_S$ are online available. Eq. (12) also tells the system that whether it can find a valid $\lambda$ to start the round-robin scheduling.

$$L < \lambda < \frac{\mathbb{R}}{2+\lceil\frac{\mathbb{N}_F}{\mathbb{N}_S}\rceil} \tag{12}$$

Further, even though a valid $\lambda$ exists, it is difficult to find an optimal value in theory. A small $\lambda$ increases I/O costs because of frequent scheduling operations. However, a large value may slow down the progress per unit of time, due to the absence of fresh messages produced and/or forwarded by disk-resident partitions. In this paper we empirically pick up a proper $\lambda$ (explored in Section 5.7) to balance the tradeoff between I/O costs and the message freshness.

### 4.3 Recovery with Load Balancing

After re-constructing delta values, vertices on failed nodes are recomputed from scratch (FR-WORB) or the most recent checkpoint (FR-WAC), instead of the point right before failures like surviving vertices. That leads to a heavy load imbalance problem. If there exist sufficient standby nodes, we can re-assign lost data on $m$ failed nodes to $n$ standby nodes, where $n = m + x$, and $x \geq 1$. The additional $x$ nodes can balance the recovery load. When the failure recovery is done, data on the additional $x$ nodes will be sent back to $m$ replacements to release resources.

Intuitively, data can be re-assigned onto $n$ nodes 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], [2], [5], 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. (13). By using $(|\mathbb{N}|\cdot n)$ instead of $n$, we can evenly partition data. Here, *mapTable* is a lookup table with only $n$ elements, which maps $idx$ into a new unique node id *node_id*.

$$node\_id = mapTable[idx], idx = j\ mod\ (|\mathbb{N}|\cdot n) \tag{13}$$

We are aware that there are advanced partitioning techniques, but none of them is suitable for our re-assignment scenario. For example, multi-level partitioning [24] is not cost-effective as its expensive runtime will be counted in our online processing time. By contrast, streaming partitioning [25] requires to be run on a single machine. The scalability is poor.

## 5 Evaluation

Now we evaluate our fault-tolerance methods. The details of the general experiment setting are given below.

*Solutions for Comparison*: We analyze the performance of our FR-WORB (termed WORB) and FR-WAC (termed

TABLE 2
Real Graph Datasets (M: Million)

| Graph | # Vertices | # Directed or undirected edges | Type |
|---|---|---|---|
| LiveJ [1] | 4.8M | 68/86M | Social networks |
| Wiki [2] | 5.7M | 130/209M | Web graphs |
| Orkut [3] | 3.1M | 234/234M | Social networks |

[1]*http://snap.stanford.edu/data/soc-LiveJournal1.html*
[2]*http://haselgrove.id.au/wikipedia.htm*
[3]*http://socialnetworks.mpi-sws.org/data-imc2007.html*

WAC) solutions in comparison with the baseline method FR-Scratch (called Scratch). The improved variants with message pruning (in Section 4.1) are denoted by pruWORB and pruWAC, respectively. Note that existing checkpoint-based methods are not involved in our study because of expensive blocking or rollback costs. Specifically, as reported in Ref. [13], the synchronous checkpointing solution suffers from blocking costs at global barriers. On the other hand, to our knowledge, only GraphLab provides an asynchronous checkpointing solution for fault-tolerance as claimed in Ref. [4]. It indeed avoids blocking underlying computations since global barriers are removed. However, vertices are archived only when receiving checkpointing flags broadcasted in the input graph. Propagating flags significantly increases the elapsed time of performing a complete checkpoint, and hence the rollback cost. For example, Fig. 4 in Ref. [4] shows that a checkpoint of a three-dimensional mesh is completed in 104s. However, it takes less than 2s for WAC with the same setting. Hence, authors of GraphLab have dropped the asynchronous checkpointing solution from the publicly available source codes. Instead of these proactive checkpointing methods, we use an up-to-date reactive solution Zorro [11] as a competitor to demonstrate the advantages of our proposals. Zorro is designed based on the fact that some systems replicate vertices over multiple nodes to optimize communication costs [26]. That means we can use replicas on surviving nodes to directly recover lost vertices. Although Zorro is implemented on top of synchronous systems, it also works on the asynchronous system Maiter by replicating vertices.

*Experiment Cluster*: We conduct experiments on Amazon EC2 using 33 t2.micro instances/nodes. Each node running Ubuntu Server 14.04 is equipped with 1 virtual core, 1 GB of RAM, and 16 GB SSD storage. The distributed file system, HDFS [27] (version: 1.0.1), is used to persist checkpoint data in WORB.

*Graph Algorithms and Datasets*: Limited by the manuscript length, we give the detailed evaluation results of our solutions on two representative algorithms PR and PHP. We also simply report the performance over other examples in Table 1, SSSP and CC, in a subsection (Section 5.9).

All tests are done over real graphs listed in Table 2. We convert these graphs into undirected ones as inputs when running CC. Besides, the web graph Wiki has a larger diameter than social networks LiveJ and Orkut.

*Evaluation Metrics*: Two metrics are evaluated in experiments, *Runtime of Recovery* (ROR) and *Runtime after Failure* (RAF).

Assume that failures occur at $t_f$. ROR is defined as the time elapsed from $t_f$ to $t_r$ where all lost vertex states have been recovered. In theory, we can continuously compare the state value achieved right before $t_f$ and the recomputed value at $t_k$ in failure recovery for each failed vertex. Then we can compute an accurate $t_r$ and hence ROR. However, that means we need to continuously archive a vertex as the possible recovery target once it is updated. The cost is prohibitively expensive. In this paper, we approximately infer ROR by the summation over every vertex state $v_j$ on failed nodes. That is, at $t_k$, a progress metric is given by $PM_k = \sum_{j \in V_N} \oplus v_j^k$, where $N \in \mathbb{N}_F$ and $\mathbb{N}_F$ is the set of failed nodes. Because $v_j$ is monotonously increased/decreased to a fixed point, the recovery is done when $PM_r = PM_f$. Clearly, $PM$ can be updated incrementally with negligible overheads.

On the other hand, we use a global summation progress metric (*GPM*) over all vertices to estimate RAF. $GPM_k = \sum_{j \in V} \oplus v_j^k$. Let $GPM_\infty$ be the summation at the fixed point. Then RAF is the time elapsed from $t_f$ to $t_k$ where $GPM_\infty = GPM_k$. $GPM_\infty$ is available by offline running algorithms without failures.

*Experiment Design*: Different from synchronous systems, 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 four scenarios, $T_1$, $T_2$, $T_3$ and $T_4$. That is, failures occur at times $T_1$, $T_2$, $T_3$, and $T_4$, respectively. Specifically, we run an algorithm without failures beforehand to record the total runtime $t$ and then set the specific values of $T_1$, $T_2$, $T_3$ and $T_4$ as $0.1t$, $0.5t$, $0.7t$ and $0.9t$, respectively. We inject a failure by manually killing the daemon process on the selected node, that simulates a network failure. The node selection and the impact on performance are given in Section 5.11. Besides, Maiter enables priority computations. The priority queue size is set to $q = 0.2|V|$ to achieve prominent performance based on priori tests [5], [16].

Note that there exist two complex failure settings: multiple failures and cascading failures. We introduce them respectively in the following. We say a job encounters *multiple failures* if multiple nodes fail at the same time. Multiple failures often happen when using transiently available cloud resources, like Amazon EC2 spot instances and Google pre-emptible instances. These instances can be used with low price but may be revoked whenever necessary [28]. Usually, instances in a region can be revoked at the same time. *Cascading failures* happen if a node fails before data on another node that fails earlier are recovered. That is, the two failures happen at different times. In this paper, we consider the scenario where a new failure is encountered before the restarting point for the old failure is constructed. Our solutions will abandon the uncompleted restarting point and then construct a new one to recover the two failures together. Note that for cascading failures and multiple failures, we do not make any assumption about the relationship among partitions where failures occur.

Below, we first illustrate the benefits brought by re-constructing delta values, by reporting the values of ROR (Section 5.1) and RAF (Section 5.2). Important factors that affect the fault-tolerance performance are also explored, such as multiple failures (Section 5.3), cascading failures (Section 5.4) and load balancing (Section 5.5). We further evaluate the effectiveness of our non-blocking technique in the absence of standby nodes (Section 5.6). Then we

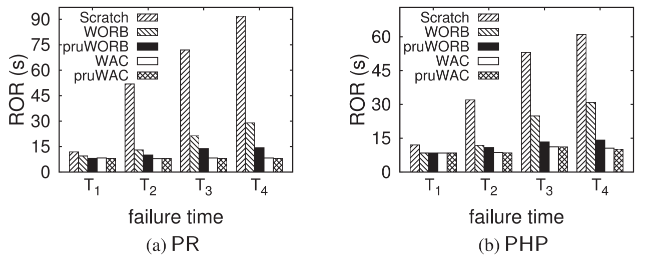


Fig. 3. ROR on LiveJ.

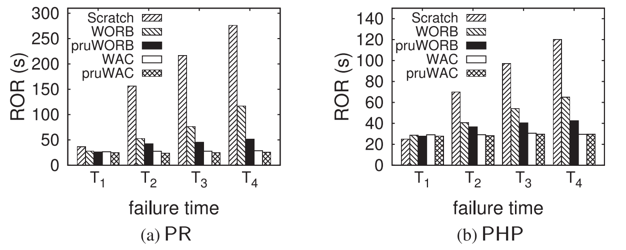


Fig. 5. ROR on Orkut.

experimentally give optimal parameters used in our methods (Section 5.7), including checkpointing intervals for WAC and scheduling intervals for the non-blocking technique. Section 5.8 validates the correctness and Section 5.9 reports the recovery efficiency on more algorithms. We finally provide a comparison between our proposals and Zorro (Section 5.10), and test the impact when different partitions are lost (Section 5.11).

## 5.1 Runtime of Recovery (ROR)

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 ROR values of Scratch, WORB, WAC, pruWORB, and pruWAC. PR and PHP are tested over all datasets. In particular, the checkpointing interval $\tau$ used in WAC and pruWAC is set to 8 seconds for PR on the Orkut graph, and 4 seconds for other combinations of algorithms and datasets. A detailed discussion about $\tau$ is given in Section 5.7.

All tests are performed in the four scenarios: $T_1$ where the five compared methods exhibit similar performance, and $T_2 - T_4$ where our proposals are supposed to be better. Specifically, as plotted in Figs. 3, 4, and 5, the speedup of WORB compared with Scratch is 4x (PR over LiveJ, $T_2$) at most. For WAC, it is even up to 11x (PR over LiveJ, $T_4$). Our pruning technique can further boost the recovery efficiency, especially for WORB in $T_3$ and $T_4$ (up to 62 percent improvement for PR on Wiki, $T_4$).

In $T_1$, the benefit brought by our proposals is not significant. In some cases, Scratch is even a preferred solution. The reasons are twofold. First, in the early phase of computations, accumulated workloads on surviving nodes are not so many that the cost of recomputing them from scratch can be negligible. Second, there exist synchronization barriers in our methods when flushing messages, incurring additional overheads.

By contrast, in $T_2 - T_4$, Scratch takes more time to recompute workloads on surviving nodes. However, our WORB and WAC avoid this problem and can leverage such workloads to accelerate the recovery speed of lost data. They thereby have superior performance to Scratch. WAC usually outperforms WORB, especially in $T_4$. This is because at the late phase of computations, it is still time-consuming to recompute lost data on failed nodes from scratch, even with the help of surviving vertices.

Our pruning technique creates an additional performance gap between pruWORB/pruWAC and other competitors, particularly when failures occur in $T_3$ and $T_4$. The reason is that in such scenarios, more workloads are lost and then more purge messages (e.g., negative delta values for PR and PHP) are propagated. Such messages can be effectively pruned by pruWORB and pruWAC. That can also explain the phenomenon that the pruning impact is less significant for WAC, since checkpointing has largely reduced the number of lost workloads.

An interesting observation we can make is that in Fig. 4b, WORB and WAC perform similarly. This is because PHP traverses a graph starting from a given source vertex. For the large diameter graph Wiki, there exists a long convergent stage where only a few vertices are updated and others have already converged. Clearly, convergent vertices on surviving nodes can greatly accelerate the recovery upon failures, —which largely narrows the performance gap between WORB and WAC.

## 5.2 Runtime After Failure (RAF)

We test RAF using the same setting described in Section 5.1. As shown in Figs. 6, 7, and 8, WORB/WAC is at most 2.6/5.7 times faster than Scratch (PR over LiveJ, $T_4$), while pruWORB further offers up to 57.6 percent performance improvement in comparison with WORB (PHP over LiveJ, $T_4$). Generally, the gain is not so large as that in ROR, because the time of running remaining underlying workloads after recovery may occupy a large proportion of the overall runtime, especially in $T_1$ and $T_2$. Different from ROR, RAF of our proposals is inversely proportional to the failure time, because our proposals can avoid rolling back completed computations. However, for Scratch, it is a constant since Scratch always recomputes from scratch. When failures occur in $T_1$ and $T_2$, compared against WORB,

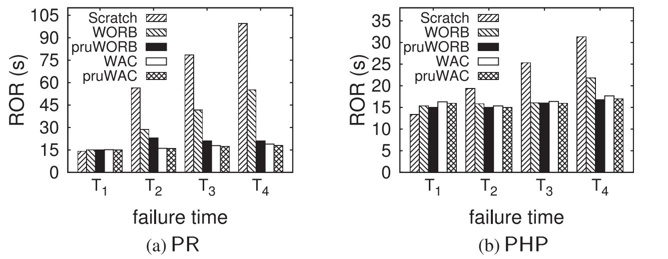


Fig. 4. ROR on Wiki.

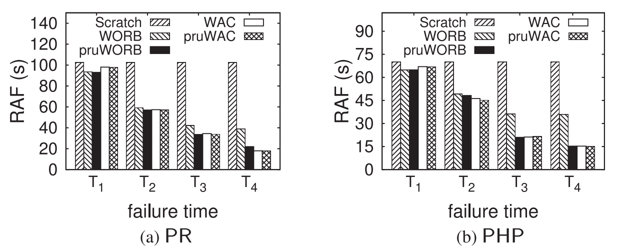


Fig. 6. RAF on LiveJ.

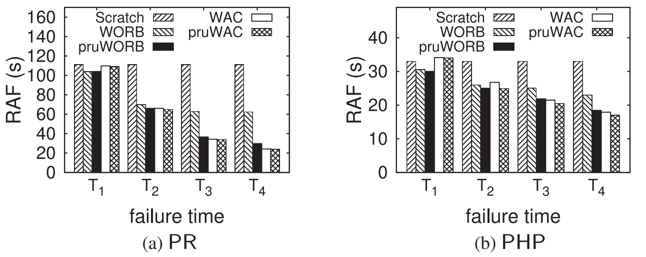


Fig. 7. RAF on Wiki.

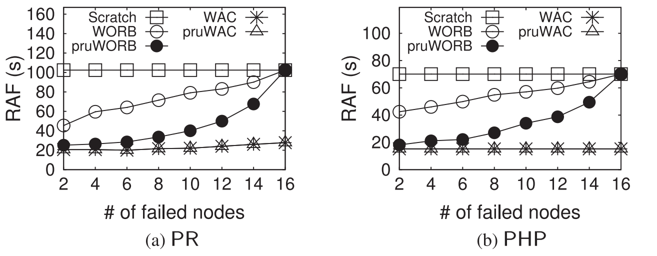


Fig. 9. RAF with multiple failures (LiveJ).

checkpointing in WAC brings marginal benefit, and even slightly performance degradation. The reason is that RAF counts the cost of archiving data for WAC.

Now we show another interesting observation. Let $t_{remain}$ denote the time spent by a recovery solution on $W_{remain}$—the remaining underlying workloads after failure recovery. Further, the superscripts "b" and "o" respectively indicate the baseline solution Scratch and one of our proposals. By analyzing the relationship between ROR (Figs. 3, 4, and 5) and RAF (Figs. 6, 7, and 8), we can find that in many cases $t^o_{remain} = (RAF^o - ROR^o) \leq t^b_{remain}$. The reasons are twofold. First, $W^o_{remain} \leq W^b_{remain}$. In fact, upon failures, there is the same number of underlying workloads left to do in our proposals as in Scratch. However, some workloads in the former have already been run along with the recovery of failures. This can be explained by the fact that surviving vertices are continuously updated without pausing. As a result, for the remaining underlying workloads after failure recovery, we have $W^o_{remain} \leq W^b_{remain}$. Second, our proposals may possibly speed up the computation of $W^o_{remain}$. As mentioned in Section 2.1, Maiter prioritizes the update order of vertices for fast convergence. More specifically, computational nodes in Maiter independently perform iterative updates over local vertices. In one local iteration, a node always selects a fixed number of vertices with high priorities (i.e., large delta values) for updates and skips others. If there exists a heavy skewed distribution of priorities, great progress can be made with little effort. However, after several local iterations, many large delta values have been consumed. The degree of skew becomes less significant. In this scenario, priority scheduling is not cost effective since sorting priorities and performing selection are not free. Note that selected vertices will propagate their delta values to neighbors as shown in Eq. (3), that dynamically updates the distribution of priorities. However, the impact is limited because only a subset of vertices are processed per local iteration. Clearly, shuffling all non-zero delta values will enhance the impact and then increase the degree of skew. This can be achieved by running a non-prioritized global synchronous iteration. We have studied the performance variation of PR and PHP over all graphs when running the non-prioritized synchronous iteration at different times ($T_1 - T_4$) in their *failure-free* executions. We find that the increased degree of skew can reduce the runtime by 8.8 percent at most (from 1.2 percent). Such synchronous iteration is also performed when building the restarting point in our recovery mechanism. Thus, the computation of $W^o_{remain}$ can be accelerated. Together with the fact that $W^o_{remain} \leq W^b_{remain}$, for WORB and pruWORB, we have $t^o_{remain} \leq t^b_{remain}$. On the other hand, WAC and pruWAC must write checkpoint data when running $W^o_{remain}$, incurring additional but acceptable costs. Hence, in many cases, we can still have $t^o_{remain} \leq t^b_{remain}$.

In a word, from the RAF perspective, WORB is a preferred solution when failures occur during the early computations, while WAC is more suitable for the scenario where the process is interrupted at the late phase of computations. By specifying pruning functions, pruWORB provides comparable performance to WAC.

## 5.3 RAF with Multiple Failures

We explore the features of compared solutions when multiple failures are encountered. Using the same setting in Section 5.1, Fig. 9 reports RAF of PR and PHP in $T_4$, over the LiveJ graph.

Scratch keeps RAF stable since it always forgets about completed workloads on all nodes. Benefitting from checkpoint data, a similar trend can also be observed for WAC/pruWAC, but the performance is more prominent than Scratch. On the contrary, RAF of WORB gradually increases with the increase of failed nodes, because more lost data are recomputed from scratch and the benefits brought by surviving vertices become less significant. Nevertheless, even when 50 percent of nodes fail, WORB still runs 1.4 times faster than Scratch. pruWORB alleviates the performance degradation but the effectiveness is limited in extreme cases such as more than 75 percent (12 out of 16) of nodes fail.

## 5.4 RAF with Cascading Failures

Fig. 10 reports RAF when cascading failures happen. We remove the results of WORB and WAC to reduce clutter,

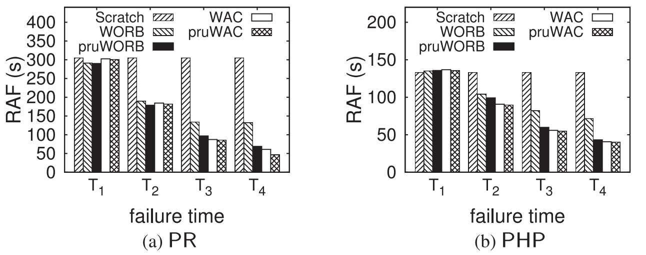


Fig. 8. RAF on Orkut.

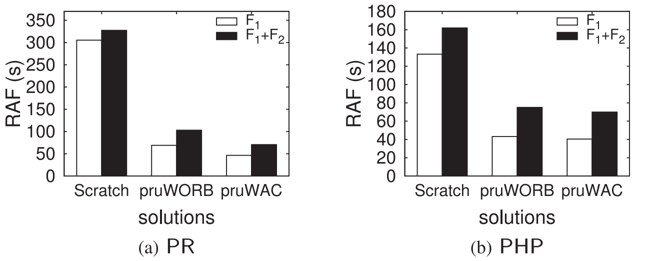


Fig. 10. RAF with cascading failures (Orkut).

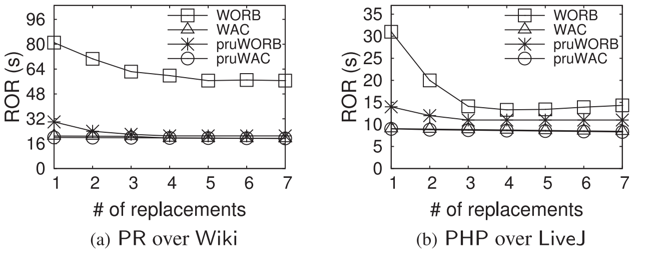


Fig. 11. ROR with load balancing.

since pruWORB and pruWAC have been shown to give better performance. All tests are run in $T_4$ and the other setting is the same as that used in Section 5.1. Here, $F_1$ indicates a single failed node, and $(F_1 + F_2)$ means that another node fails at time $t'_f$ when constructing the restarting point for $F_1$. For Scratch, we also set another node failed at $t'_f$. We can find that both pruWORB and pruWAC outperform Scratch.

## 5.5 ROR with Load Balancing

This group of experiments validates that balancing the load of recovery (Section 4.3) can drop ROR. Without loss of generality, PR over Wiki and PHP over LiveJ as two cases are tested. Specifically, we run them using 10 nodes but when one fails in $T_4$, the lost data are evenly assigned to multiple standby nodes, i.e., replacements (the # of replacements ranges from 1 to 7).

As shown in Fig. 11, ROR of WORB is reduced by up to 30.2 and 53.8 percent for PR and PHP, respectively. This is because recomputing from scratch is extremely time-consuming and then leads to a heavy load imbalance. However, pruWORB, WAC and pruWAC have already optimized the majority of recovery workloads. Thus, using more replacements brings marginal benefit.

## 5.6 Non-Blocking Fault-Tolerance

We next show the effectiveness of the non-blocking optimization technique. Based on Ref. [23] and our experience, it takes about 90 seconds to acquire new Amazon EC2 instances. Worse, we need to spend additional 10 seconds on configuring the system. The restarting cost $\mathbb{R}$ is thereby 100 seconds. An important parameter, scheduling interval $\lambda$, is set by $\lambda = 4L$. More discussions regarding $\lambda$ are given in Section 5.7. Because of the effectiveness of pruning messages, here we only study the performance of pruWORB and pruWAC. Let nb-pruWORB and nb-pruWAC stand for their non-blocking variants. Using PR over LiveJ as an example and the same setting in Section 5.1, Fig. 12 depicts RAF values.

Sub-figure (a) shows that non-blocking variants consistently outperform pruWORB and pruWAC. The performance gap is particularly large in $T_3$ and $T_4$ because PR under nb-pruWORB/nb-pruWAC has converged before the newly applied instance is ready. Sub-figure (b) shows that both nb-pruWORB and nb-pruWAC scale well with failed nodes.

## 5.7 Determining Fault-Tolerance Parameters

In previous experiments, we know WAC usually exhibits prominent performance but requires to archive checkpoint data periodically. Also, the non-blocking technique is important for efficiency but requires to schedule data between memory and disk. Both checkpointing interval ($\tau$) and scheduling interval ($\lambda$) are user-specified and now we empirically give the roughly optimal values.

*Checkpointing Interval*: We analyze the runtime of the *failure-free* execution using different $\tau$ values. $\tau = +inf$ means that no checkpoint is archived. Fig. 13 shows the performance variation when archiving data onto local file systems. Clearly, a quite large range of $\tau$ can guarantee that the overhead of archiving data is nearly zero because of the asynchronous design. Our further study reveals that even archiving data onto distributed file systems such as HDFS with multiple data replicas, the additional overhead is still negligible (Fig. 14). Thus, we use HDFS to store checkpoint data with 3 replicas, to provide high availability. Because a smaller $\tau$ can provide a more recent checkpoint to reduce the number of re-computations, we then set $\tau = 8$ seconds for PR over Orkut and $\tau = 4$ seconds for other cases.

*Scheduling Interval*: To find a real optimal interval, we repeatedly run PR and manually set the $\lambda$ value every time. Fig. 15 shows RAF of nb-pruWORB on different graphs with $\mathbb{R} = 100$ seconds. With $\lambda$ being increased, we find that RAF first decreases and then increases. This can be explained by the tradeoff between I/O costs and the message freshness. Let $L$ indicate the initialization overhead before computing a partition, which is available by online statistics. We find $\lambda = 4L$ can roughly hit the "sweet spot". Automatically finding this sweet spot is a subject for future work.

## 5.8 Empirical Validation of Correctness

Sections 3.2 and 4.1 prove the correctness of WORB, WAC and their pruning variants. Now we empirically validate it

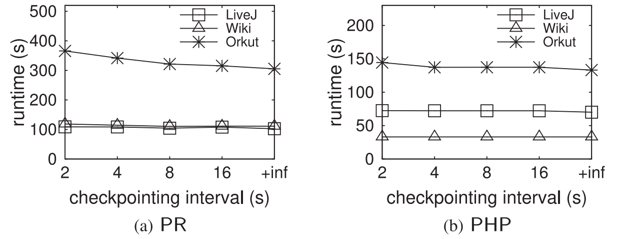


Fig. 13. Impact of archiving data onto local file systems.

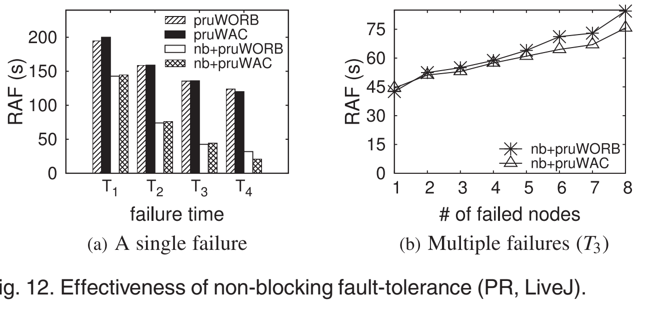


Fig. 12. Effectiveness of non-blocking fault-tolerance (PR, LiveJ).

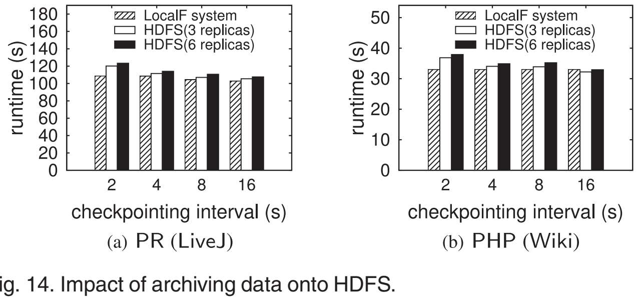


Fig. 14. Impact of archiving data onto HDFS.

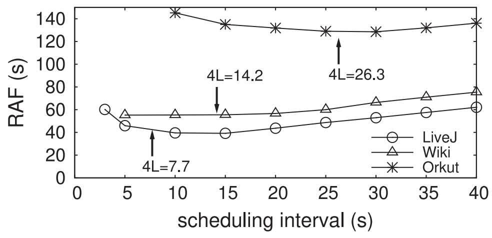


Fig. 15. Impact of scheduling intervals (PR, a single failure in $T_4$).

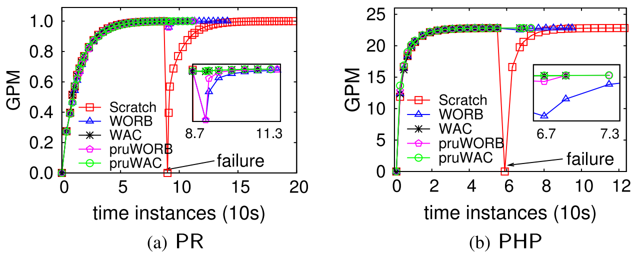


Fig. 16. Correctness (LiveJ, a single failure in $T_4$).

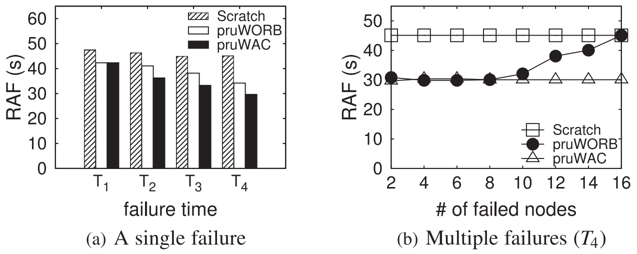


Fig. 17. RAF of SSSP on Orkut.

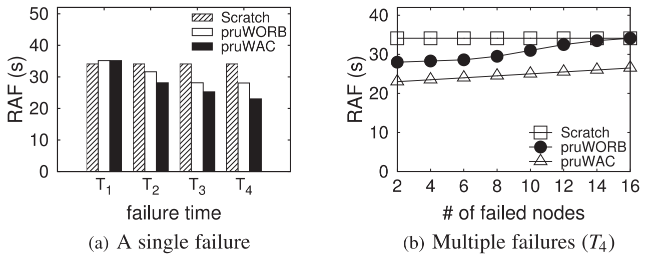


Fig. 18. RAF of CC on Orkut.

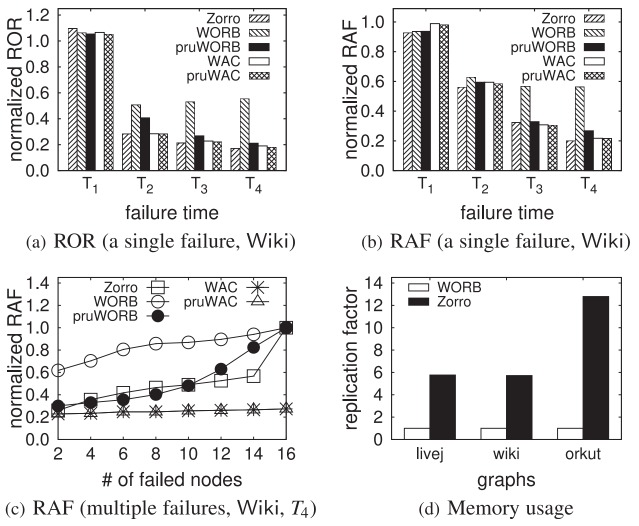


Fig. 19. Runtime analysis and memory usage of Zorro (PR).

by continuously monitoring the global progress metric $GPM_k$ at sampled time instances. Without loss of generality, in Fig. 16, we report the results of PR and PHP over LiveJ, using the same setting in Fig. 6. More specifically, the failure time $T_4 = t_f = 90/57$ second for PR/PHP. By running PR/PHP until they converge, we can observe that algorithms under our proposals converge to the same fixed point as achieved in Scratch.

## 5.9 Performance on More Graph Algorithms

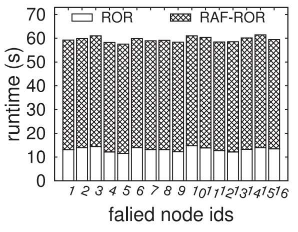
Because of the removal of the reversibility requirement, pruWORB and pruWAC can be applied to irreversible algorithms like SSSP and CC tested in this section. The experiments are run using the same setting in Section 5.1.

Figs. 17 and 18 present RAF on the Orkut graph. We find that both pruWORB and pruWAC outperform Scratch. However, different from the results of PR and PHP, the performance gap on SSSP and CC is less significant. This is because SSSP/CC is a traversal algorithm [29] where the shortest distances or the component ids are propagated based on the graph topology. Many vertices can quickly converge and then do not participate in computations, leading to a very short overall runtime $t$. Upon failures, Scratch with $RAF = t$ becomes a strong competitive solution. On the other hand, in our solutions, constructing the restarting point as an atomic operation requires to broadcast states of all vertices. The broadcasting cost $t_b$ is fixed no matter when failures occur. Given a short $t$, $\frac{t_b}{t}$ can be large (74 percent for SSSP and 47 percent for CC). Then the benefits of our solutions are limited when compared with Scratch. Many other irreversible algorithms are also traversal, like Breadth-First-Search [29], Betweenness Centrality and Closeness Centrality [30]. We are then motivated to optimize $t_b$ for better running these algorithms. One possible method is to prioritize the importance of vertices and then only broadcast states of some vertices with high priorities. We will investigate it as future work.

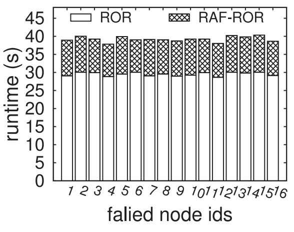
## 5.10 Comparison with Zorro

We then conduct experiments against Zorro in Fig. 19. Zorro recovers failures based on replicas of vertices. However, replicating vertices will improve the performance of an algorithm [26]. To make a fair comparison, we first run recovery solutions respectively on Maiter and its variant with vertex replications, and then normalize ROR and RAF to Scratch.

Using PR as an example, we first evaluate normalized ROR and RAF in sub-figures (a)-(c). When a single failure happens, Zorro generally performs best. This is because almost all of lost vertices can be recovered by replicas and there is no extra cost incurred by archiving data. However, our asynchronous checkpointing methods WAC and pruWAC outperform Zorro if multiple nodes fail. In this case, many replicas are also lost and hence a lot of vertices on failed nodes will be reset to **0**. RAF thereby increases as shown in sub-figure (c). Sub-figure (d) then compares the vertex replication factors of our proposals and Zorro on varies of graph datasets. Generally, we can see that Zorro achieves prominent ROR and RAF at the price of huge memory consumption (i.e., a large replication factor).



(a) $T_2$



(b) $T_4$

Fig. 20. Runtime analysis when selecting different nodes to simulate a single failure at $T_2$ and $T_4$ (PR, LiveJ, WORB).

### 5.11 Impact of Losing Different Partitions

We finally explore the impact on ROR and RAF when losing different partitions. By selecting different nodes to simulate a single failure at $T_2$ and $T_4$, Fig. 20 shows the runtime analysis of WORB. We notice that the runtime variation is negligible. This is mainly due to the Hash-based graph partitioning policy which is widely used in distributed systems. Hash provides a uniform distribution of vertices among partitions. Then different partitions evenly contribute to the algorithm convergence. Accordingly, in all experiments through this paper, we can randomly select a node and then manually kill its daemon process to simulate a network failure. Without loss of generality, we always select the node with $id = 1$ as the failed node. When simulating multiple failures and cascading failures, nodes are selected in an ascending order of ids.

## 6 Related Work

Many representative techniques have been designed for tolerating failures in graph processing systems. All of them basically fall into three categories, checkpoint-based, lineage-based, and reactive paradigms. We summarize them as follows.

*Checkpoint-Based Solutions*: The synchronous checkpointing solution as an early technique presented in Ref. [1] has been extensively used in Pregel-like systems [1], [2], [31]. Researchers have proposed many variants to improve its performance, such as partially overlapping CPU computations and I/O operations [18], reducing the checkpoint data to vertex values [19], confining re-computations to lost data and/or parallelizing such re-computations [17], [18], [32]. These techniques work well in synchronous systems, but result in suboptimal performance in asynchronous systems. This is because they require expensive global barriers. Then the benefit brought by asynchronous computations can be offset.

GraphLab [4] employs a variant of the Chandy-Lamport method [9] for its asynchronous engine. This variant can archive data without global barriers, reducing the checkpointing overhead. However, the elapsed time of performing a complete checkpoint is large. Then the recovery cost is expensive since workloads on surviving nodes are still rolled back to the most recent checkpoint.

Differently, our proposals remove the requirement of barriers and allow surviving vertices to keep performing updates in failure recovery without rollback. Also, we design a new data re-assignment function with a small lookup table

*Lineage-Based Solutions*: Some systems like Spark employ a lineage method to track the dependency of coarse-grained data structures [8], [33], [34]. Lost data can be recomputed by analyzing the lineage upon failures. Unlike checkpointing, the lineage solution saves storage space and network bandwidth since the volume of dependency information can be much smaller than that of algorithm-specified data. However, asynchronous computations generate fine-grained updates, which makes the dependency relationship prohibitively complicated and hence largely increases the overhead of maintaining the lineage.

*Reactive Solutions*: Recently, reactive recovery solutions without checkpointing have attracted a lot of attention. Some techniques focus on utilizing redundant data to recover lost data [10], [11], [12]. However, redundant information may quickly exhaust memory resources [35]. These techniques are not always feasible to scale to massive datasets. Schelter et al. [13] design a restarting point to resume computations for synchronous systems. However, it requires users to carefully design an algorithm-specified compensation function, which is usually a nontrivial task [18].

Our proposals replace vertex values automatically, which eases the burden of users. But more importantly, they do not store multiple replicas of vertices, largely reducing the memory consumption.

## 7 Conclusion

This paper proposes two fault-tolerance methods for asynchronous graph processing systems. Unlike the most widely used checkpointing techniques, our methods remove expensive synchronous barriers and hence are naturally suitable for asynchronous engines. Many optimization techniques, like pruning messages, non-blocking recovery, and load balancing, are also designed to further boost the efficiency in different scenarios. Performance studies on real-world graphs validate the effectiveness of our proposals.

## Acknowledgments

This work was supported by the National Natural Science Foundation of China (61433008, U143520006, 61472071, and 61528203). Lixin Gao was supported by the U.S. National Natural Science Foundation grant CNS-1217284. Zhigang Wang was a visiting student at UMass Amherst, supported by China Scholarship Council.

## References

[1] 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. SIGMOD*, 2010, pp. 135–146.

[2] "Giraph." (2016, Oct.). [Online]. Available: http://giraph.apache.org/, Accessed on: Nov. 2017.

[3] "Apache spark." (2017, Dec.). [Online]. Available: http://spark.apache.org/, Accessed on: Nov. 2017.

[4] 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," in *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[5] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *IEEE Trans. Parallel Distrib Syst.*, vol. 25, no. 8, pp. 2091–2100, Aug. 2014.

[6] 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. SIGMOD*, 2016, pp. 479–494.

[7] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," *Proc. VLDB Endowment*, vol. 7, no. 12, pp. 1047–1058, 2014.

[8] 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. NSDI*, 2012, pp. 2–2.

[9] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.

[10] Z. Chen, "Algorithm-based recovery for iterative methods without checkpointing," in *Proc. HPDC*, 2011, pp. 73–84.

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

[12] J. Wang, M. Balazinska, and D. Halperin, "Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1542–1553, 2015.

[13] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl, "All roads lead to rome: optimistic recovery for distributed iterative data processing," in *Proc. CIKM*, 2013, pp. 1919–1928.

[14] Z. Wang, L. Gao, Y. Gu, Y. Bao, and G. Yu, "A fault-tolerant framework for asynchronous iterative computations in cloud environments," in *Proc. SoCC*, 2016, pp. 71–83.

[15] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proc. WWW*, 1998, pp. 107–117.

[16] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A distributed framework for prioritizing iterative computations," *TPDS*, vol. 24, no. 9, pp. 1884–1893, 2013.

[17] Y. Shen, G. Chen, H. Jagadish, W. Lu, B. C. Ooi, and B. M. Tudor, "Fast failure recovery in distributed graph processing systems," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 437–448, 2014.

[18] C. Xu, M. Holzemer, M. Kaul, and V. Markl, "Efficient fault-tolerance for iterative graph processing on distributed dataflow systems," in *Proc. ICDE*, 2016, pp. 613–624.

[19] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: An efficient, low-cost system for concurrent graph processing," in *Proc. HPDC*, 2014, pp. 227–238.

[20] Z. Guan, J. Wu, Q. Zhang, A. Singh, and X. Yan, "Assessing and ranking structural correlations in graphs," in *Proc. SIGMOD*, 2011, pp. 937–948.

[21] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.

[22] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proc. ICDM*, 2009, pp. 229–238.

[23] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *Proc. Cloud Comput.*, 2012, pp. 423–430.

[24] D. LaSalle and G. Karypis, "Multi-threaded graph partitioning," in *Proc. IPDPS*, 2013, pp. 225–236.

[25] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. SIGKDD*, 2012, pp. 1222–1230.

[26] I. Hoque and I. Gupta, "Lfgraph: Simple and fast distributed graph analytics," in *Proc. 1st ACM SIGOPS Conf. Timely Results Operating Syst.*, 2013, Art. no. 9.

[27] "Apache hadoop." (2017, Dec.). [Online]. Available: http://hadoop.apache.org/, Accessed on: Nov. 2017.

[28] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy, "Spoton: A batch computing service for the spot market," in *Proc. SoCC*, 2015, pp. 329–341.

[29] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proc. SC*, 2010, pp. 1–11.

[30] "Traversal algorithms." [Online]. Available: https://github.com/jegonzal/PowerGraph/tree/master/toolkits/graph_algorithms

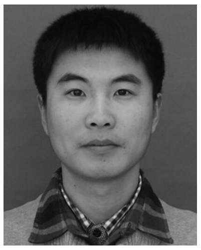
[31] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proc. SSDBM*, 2013, Art. no. 22.

[32] D. Yan, J. Cheng, and F. Yang, "Lightweight fault tolerance in large-scale distributed graph processing," *CoRR*, vol. abs/1601.06496, 2016, http://arxiv.org/abs/1601.06496

[33] T. Martsinkevich, O. Subasi, O. Unsal, F. Cappello, and J. Labarta, "Fault-tolerant protocol for hybrid task-parallel message-passing applications," in *Proc. CLUSTER*, 2015, pp. 563–570.

[34] C. Cao, T. Herault, G. Bosilca, and J. Dongarra, "Design for a soft error resilient dynamic task-based runtime," in *Proc. IPDPS*, 2015, pp. 765–774.

[35] C. Zhou, J. Gao, B. Sun, and J. X. Yu, "Mocgraph: Scalable distributed graph processing using message online computing," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 377–388, 2014.



**Zhigang Wang** received the BSc and MSc degrees in computer science from Northeastern University, China, in 2011 and 2013, respectively. He is currently working towards the PhD degree in computer science at Northeastern University. He has been a visiting PhD student in the University of Massachusetts Amherst during December 2014 to December 2016. His research interests include cloud computing, distributed graph processing, and machine learning.

**Lixin Gao** received the PhD degree in computer science from the University of Massachusetts at Amherst, in 1996. Now she is a professor of electrical and computer engineering with the University of Massachusetts at Amherst. Her research interests include social networks, Internet routing, network virtualization and cloud computing. Between May 1999 and January 2000, she was a visiting researcher in AT&T Research Labs and DIMACS. She was an Alfred P. Sloan fellow between 2003-2005 and received an NSF CAREER Award in 1999. She won the best paper award from IEEE INFOCOM 2010 and ACM SoCC 2011, and the test-of-time award in ACM SIGMETRICS 2010. She received the Chancellors Award for Outstanding Accomplishment in Research and Creative Activity in 2010. She is a fellow of the IEEE and ACM.

**Yu Gu** received the PhD degree in computer software and theory from Northeastern University, China, in 2010. Currently, he is a professor and the PhD supervisor with Northeastern University, China. His current research interests include big data analysis, spatial data management and graph data management. He is a senior member of the China Computer Federation (CCF).

**Yubin Bao** received the PhD degree in computer software and theory from Northeastern University, China, in 2003. Currently, he is a professor with Northeastern University, China. His current research interests include data warehouse and OLAP, graph data management, and cloud computing. He is a senior member of the China Computer Federation (CCF).

**Ge Yu** received the PhD degree in computer science from Kyushu University of Japan, in 1996. He is currently a professor and the PhD supervisor with Northeastern University of China. His research interests include distributed and parallel database, OLAP and data warehousing, data integration, graph data management, etc. He is a member of the IEEE Computer Society, IEEE, ACM, and a fellow of the China Computer Federation (CCF).