Lightweight Fault Tolerance in Pregel-Like Systems

Da Yan
University of Alabama at Birmingham
yanda@uab.edu

James Cheng
The Chinese University of Hong Kong
jcheng@cse.cuhk.edu.hk

Hongzhi Chen
The Chinese University of Hong Kong
hzchen@cse.cuhk.edu.hk

Cheng Long
Nanyang Technological University
c.long@ntu.edu.sg

Purushotham Bangalore
University of Alabama at Birmingham
puri@uab.edu

ABSTRACT

Pregel-like systems are popular for iterative graph processing thanks to their user-friendly vertex-centric programming model. However, existing Pregel-like systems only adopt a naïve checkpointing approach for fault tolerance, which saves a large amount of data about the state of computation and significantly degrades the failure-free execution performance. Advanced fault tolerance/recovery techniques are left unexplored in the context of Pregel-like systems. This paper proposes a non-invasive lightweight checkpointing (LWCP) scheme which minimizes the data saved to each checkpoint, and additional data required for recovery are generated online from the saved data. This improvement results in 10x speedup in checkpointing, and an integration of it with a recently proposed log-based recovery approach can further speed up recovery when failure occurs. Extensive experiments verified that our proposed LWCP techniques are able to significantly improve the performance of both checkpointing and recovery in a Pregel-like system.

1 INTRODUCTION

Google's Pregel [5] pioneered a think-like-a-vertex programming model intuitive for writing distributed programs for iterative graph computations (e.g., random walks and graph traversals), where vertices communicate by message passing. This vertex-centric model has been followed by many systems such as Giraph [2], GraphX [4] and Pregel+ [11], and are generally called Pregel-like systems.

As a distributed framework, Google's Pregel [5] supports fault tolerance to combat machine failures: a checkpointing-based approach which periodically saves the current state of computation to a failure-resilient storage (which survives machine failures), so that the latest saved state can be loaded in case failure happens to avoid re-computation from scratch. We assume Hadoop Distributed File System (HDFS) is used as the failure-resilient storage hereafter.

However, a checkpoint in Pregel contains all vertex states, edges, and messages sent by vertices in an iteration, and writing them to HDFS hurts the failure-free performance and is often disabled [5].

Faster fault tolerance techniques such as incremental checkpointing and log-based methods have been proposed [3], but surprisingly, their adoption in Pregel-like systems remains unexplored. This is likely because vertex-centric model is very general, and there are always some applications where only naïve checkpointing is applicable. As Sec. 2 shall discuss, more efficient fault tolerance techniques have been applied to only to more restricted models than Pregel, other than [10] which explored the use of message logs to speed up failure recovery; but we find that [10] ignores the need and cost of garbage collecting outdated message logs, and when counting that cost, the failure-free execution time is increased.

In this paper, we study advanced fault tolerance techniques for Pregel-like systems, and our contributions are as follows:

- We apply faster fault tolerance techniques to a Pregel-like system while keeping the impact to users' programming minimal, which no change or at most some minor changes needed depending on specific applications, for which previously the only choice is expensive naïve checkpointing.
- A lightweight checkpointing (LWCP) scheme is proposed to significantly reduces the data volume to be saved in a checkpoint, which reduces checkpointing time by 10x.
- Built on LWCP, a vertex-state log based approach is proposed for faster failure recovery. This approach is also the first to truly support faster failure-free execution by avoiding the high cost of deleting outdated message logs which would otherwise be required using [10]'s message logging scheme.
- We implement all methods under a unified framework built with ULFM [1] which enjoys MPI's efficiency and portability.

The rest of this paper is organized as follows. We review the related work on fault tolerance in Sec. 2, and review Pregel and our abstraction of its workflow in Sec. 3. Then, Sec. 4 introduces our LWCP solution that significantly reduces the checkpointing time, and Sec. 5 describes our vertex-state log based approach that supports faster recovery. Finally, experimental results are reported in Sec. 6 and the paper is concluded in Sec. 7.

2 RELATED WORK

Studies of fault tolerance in distributed message-passing systems date back to the 80s–90s, and [3] surveys these techniques such as coordinated checkpointing and incremental checkpointing. Almost all existing Pregel-like systems adopt coordinated checkpointing which writes a checkpoint after a synchronization barrier at the end of an iteration, leaving other techniques unexplored.

The other technique, incremental checkpointing, avoids rewriting portions of states that do not change between consecutive checkpoints, but has not been considered in Pregel-like systems.

However, fault-tolerance protocols surveyed in [3] are mainly designed for a general message-passing system rather than a concrete

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337823

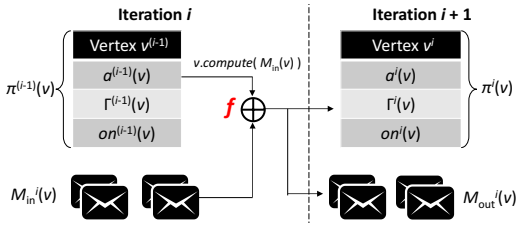

Figure 1: Vertex State & "Compute" Function

computation model like Pregel. The general-purpose protocols incur additional overheads like piggybacked information and dependency tracking, and their non-invasive integration with the vertex-centric programming model of Pregel remains an open problem.

In the context of a general Pregel-like model, [10] proposes a message logging method which is able to reduce the amount of communication during recovery. However, the work ignores the cost of deleting outdated message logs (which is needed to avoid using up disk space), and failure-free execution hurts when considering that cost. This weakness outweighs the benefit of faster failure recovery since machine failure does not occur frequently.

3 A FAULT-RESILIENT PREGEL WORKFLOW

This section first reviews Pregel's model in Sec. 3.1, and then introduces our fault-tolerant Pregel workflow abstraction in Sec. 3.2.

3.1 Pregel Review

Notations. We consider an input graph $G = (V, E)$ stored on HDFS, where each vertex $v \in V$ has a unique ID (also denoted as v for simplicity) and an adjacency list $\Gamma(v)$. If G is undirected (resp. directed), $\Gamma(v)$ contains all v's neighbors (resp. out-neighbors). In Pregel, each vertex v also maintains (1) a value $a(v)$ which gets updated during computation, and (2) a boolean label $on(v)$ indicating whether v is active ("on") or halted ("off"). A Pregel program is run on a cluster of worker machines (or simply workers), denoted by $\mathbb{W}$.

Fig. 1 shows the contents of a vertex v, where we use superscript (i) to indicate that a value is updated by Iteration i, or equivalently, is at the beginning of Iteration $(i+1)$. We also define the state of a vertex v as a triplet $\pi(v) = \langle a(v), \Gamma(v), on(v)\rangle$.

Pregel. A Pregel program starts by loading a graph from HDFS, where each vertex v is distributed to a worker $W_i \in \mathbb{W}$ (along with $\Gamma(v)$) according to a partitioning function $hash(.)$, i.e., $i = hash(v)$. We define V_W as the set of all vertices assigned to worker W.

To write a Pregel program, one needs to specify the behavior of a vertex v in a user-defined function (UDF) *compute*(*msgs*), where *msgs* is the set of messages received by v (sent in the previous iteration). In v.*compute*(.), v may update its value $a(v)$ and neighbor list $\Gamma(v)$, generate and send new messages to other vertices, and vote to halt (i.e., set $on(v)$ as *false*). Only active vertices will call *compute*(.) in an iteration, but a halted vertex will be reactivated if it receives a message. The program terminates when all vertices are halted and there is no pending message for the next iteration.

Let us define $M_{in}^{(i)}(v)$ as the set of messages received by v when Iteration i begins, and $M_{out}^{(i)}(v)$ as the set of messages generated and sent by v in Iteration i. As Fig. 1 shows, UDF v.*compute*(*msg*) essentially defines a function f below (recall that v denotes v's ID):

$$\pi^{(i)}(v),\ M_{out}^{(i)}(v) \leftarrow f(v,\ \pi^{(i-1)}(v),\ M_{in}^{(i)}(v)), \tag{1}$$

We explain how to write UDF *compute*(.) using two examples.

Example 1: PageRanks. Given a directed web graph $G = (V, E)$, where each vertex (page) v links to a list of pages $\Gamma(v)$, the first problem computes the PageRank of every vertex $v \in V$, stored in $a(v)$. In v.*compute*(.), if the current iteration is Iteration 1, v initializes $a(v) \leftarrow 1/|V|$ and distributes the value $a(v)/|\Gamma(v)|$ to each out-neighbor in $\Gamma(v)$. In Iteration $i > 1$, each vertex v sums up the received values sent from its in-neighbors, denoted by *sum*, and computes $a(v) \leftarrow 0.15/|V| + 0.85 \times sum$. It then distributes $a(v)/|\Gamma(v)|$ to each out-neighbor. This process is repeated in iterations until $a(v)$ of every vertex v converges to its PageRank.

Example 2: Hash-Min. This algorithm computes connected components (CCs) of an undirected graph. The idea is to let each vertex v remember and broadcast the smallest vertex ID it has ever seen, which is kept in $a(v)$. When the process converges, for every vertex v, $a(v)$ is the smallest vertex ID in the CC that v locates in.

Consider v.*compute*(.). In Iteration 1, v initializes $a(v)$ as $id(v)$, broadcasts it to its neighbors, and votes to halt (i.e., $on(v) \leftarrow$ *false*). In Iteration $i > 1$, v receives messages from its neighbors; let *min* be the smallest ID received, if $min < a(v)$, v sets $a(v) \leftarrow min$ and broadcasts it to neighbors. All vertices vote to halt at the end of an iteration. The job finishes when no vertex receives a smaller ID.

Combiners. Users may implement a message combiner to specify how to combine messages that are directed to the same vertex u, so that on a worker W, outgoing messages targeting u will be combined into a single message and then sent by W to u. This effectively reduces message number. For example, in PageRank (resp. Hash-Min), the combiner logic can be taking the sum (resp. minimum), as *compute*(.) only needs the sum (resp. minimum).

Aggregators. Pregel also supports an aggregator for global communication. Each vertex can provide a value to an aggregator in *compute*(.) in an iteration. The system aggregates those values and makes the aggregated result available to all vertices in the next iteration. In implementation, each worker first aggregates values provided by its vertices locally, which are then globally aggregated.

3.2 Our Resilient Pregel Workflow

We first abstract the key operations of a fault-resilient Pregel framework, and explain how existing solutions fit in the framework.

The Three Key Operations. Our framework takes 3 operations to specify when implementing a fault-resilient Pregel system:

- OP1: actions of a vertex during normal execution;
- OP2: when failure occurs, actions of a surviving vertex;
- OP3: when failure occurs, actions of a revived vertex.

In naïve checkpointing, (OP1) a vertex v receives all its incoming messages and call UDF *compute*(.) to advance its state and to generate messages for the next iteration. When failure occurs, (OP2) a vertex on a surviving machine loads everything about its state from the latest checkpoint (let it be i), and then advances its computation. Starting from Iteration $(i+1)$, the execution becomes normal like in OP1. Finally, (OP3) for a vertex on a crashed machine, it will be revived on another machine, after which it will load the latest checkpoint and advance its computation exactly like in OP2.

Consider the example of PageRank computation, and assume that a checkpoint is saved for every $\delta = 10$ iterations. If a machine

crashes at Iteration 17, then the latest checkpoint saved at Iteration 10 will be loaded to roll the state of every vertex back to the end of Iteration 10, and the computation then reruns from Iteration 11.

However, since the state of a surviving vertex is already at Iteration 17, re-computing from Iteration 11 wastes prior computation. The message logging approach of [10] aims to avoid this waste. It does not roll back the states of surviving vertices, and only revives those vertices in crashed machine(s), and reruns their computation.

A gap remains here: (1) when a revived vertex v reruns its computation, say, at Iteration 12, v needs to receive its messages from all vertices, including those sent from surviving vertices at Iteration 11; (2) surviving vertices do not rerun from Iteration 11 to Iteration 17, and thus will not generate any message during the recovery.

To close this gap, [10] lets every machine $W \in \mathbb{W}$ log the messages generated by vertices in W to local disk(s). For the previous example, for computing Iteration 12, surviving vertices may simply re-send their messages logged at Iteration 11 to the revived vertices. The recovery is much faster since only those messages targeting revived vertices need to be transmitted (rather than all messages).

In this case, (OP2) a surviving vertex does not read a checkpoint, but needs to read logged messages for sending in each recovery iteration; and (OP3) a revived vertex loads its state from the latest checkpoint, and re-computes during the recovery iterations.

Besides the above 2 approaches, we will also introduce our proposed approaches in Sec. 4 and 5 under the three key operations. We remark that our description of [10]'s algorithm here is kept simple (e.g., cascaded failure is not considered, will discuss in Sec. 5).

MPI & ULFM. While our proposed fault tolerance approaches are general to a Pregel-like system, w.l.o.g., we implement them in our resilient framework built upon Pregel+ [11] using User-Level Failure Mitigation (ULFM) [1] to enjoy the efficiency and portability of MPI. ULFM is a recently proposed resilience extension to MPI that includes new communication primitives for failure notification and processing, and is supported by OpenMPI (c.f., http://fault-tolerance.org/) and MPICH (c.f., http://www.mpich.org/static/docs/v3.2/).

We run multiple workers on each machine, and MPI automatically tracks worker-to-machine mapping. MPI provides a set of communication primitives, each takes a "communicator" object which specifies a set of workers among which the communication happens. Our framework deals with 3 worker sets/communicators: (1) $\mathbb{W}_{\textbf{all}}$: the set of all workers; (2) $\mathbb{W}_{\textbf{alive}}$: the set of workers surviving a failure; (3) $\mathbb{W}_{\textbf{new}}$: the set of new workers, which revives those workers on crashed machine(s) after a failure occurs.

When a failure occurs, the set of surviving workers $\mathbb{W}_{\textbf{alive}}$ may generate $\mathbb{W}_{\textbf{new}}$ using an MPI primitive MPI_Comm_spawn. As we shall see, $\mathbb{W}_{\textbf{alive}}$ can be obtained using the new ULFM primitve MPIX_Comm_shrink when a failure is detected.

Recall that Pregel tracks the vertex-to-worker mapping using a hash function $W_i = hash(v)$. To allow $hash(.)$ to still be valid after machine failures, we decouple it from the worker-to-machine mapping tracked by MPI. If a machine W crashes, for each worker on W we respawn a new worker on a surviving/standby machine to replace it. This is achieved by assigning it the same worker ID (or "rank") as of W, while IDs of surviving workers remain unchanged.

The respawned workers are evenly assigned to the alive machines so that the workload is still balanced after recovery (recall that each machine runs multiple workers). Since the vertex-to-worker mapping is fixed, we can simply **extend the concept *surviving vertex* (resp. *revived vertex*) to *surviving worker* (resp. *revived worker*)**. If a worker failed and then revived (on another machine), then all its vertices are revived; if a worker is surviving, so are its vertices. We thus use surviving/revived workers rather than surviving/revived vertices in the presentation hereafter.

Worker State Commits. We first define some important concepts that are used in our resilient Pregel framework design.

Pregel+ runs 3 phases of processing in each iteration:

- **P1:** vertex-centric computation is performed on all active vertices, which generates all out-going messages;
- **P2:** messages are combined and then sent to target machines;
- **P3:** all workers synchronize their partially aggregated data and control information, to obtain overall aggregated value and job status (e.g., to determine whether job terminates).

Since machine failures can only be detected by communication (i.e., in P2 and P3), it is guaranteed that when a worker W detects a failure in Iteration i (after P1), all vertex states and partially aggregated data and control information of W have been fully updated. Here, we say that the **state** of W, denoted by $s(W)$, is at Iteration i, or simply, $s(W) = i$. We also say that W **partially commits** Iteration i. In other words, *when a failure occurs at Iteration i, every worker must have partially committed Iteration i.*

If all workers also finish P2 and P3, we say that the iteration is **fully committed** since (1) all messages reach the receiver side and (2) the global aggregator value and control information are obtained.

We only checkpoint an iteration after it is fully committed, so that the checkpoint of Iteration i can save (1) $M_{in}^{i+1}(v)$ of every vertex v (input to *compute*(.)) and (2) the aggregator value, both of which can be later loaded to advance computation to Iteration $(i + 1)$.

Master Election. We eliminate single-point-of-failure risks by allowing any surviving worker to be elected as a **master**. Specifically, we let the master be a worker W with the largest state $s(W)$, i.e., the longest-living worker (denoted by W_{max}), with ties broken by worker ID. Master W_{max} is important for log-based recovery where surviving workers do not compute to generate partial aggregate and control information: all workers can directly obtain global information from W_{max}, which logs information till Iteration $s(W_{max})$.

The Resilient Framework. Recall our key operations OP1–OP3 for specifying a resilient Pregel system. They are positioned in our framework as shown in Fig. 2, highlighted in bold red font.

Fig. 2(a) shows the **main execution flow** of a worker, where we omit low-level details such as initializing iteration number and registering $\mathbb{W}_{all}$ with an error handling function (i.e., function *error_handling*() in Fig. 2(c)) using MPI_Comm_set_errhandler. Line 1 refers to the recovery process detailed in Fig. 2(b), and is only run by a respawned worker. A worker that starts with a job normally goes directly to Line 2, where it backs up the execution environment before the iterative computation in Line 5. Here, we use the *setjmp* and *longjmp* functions of the C library. If a worker calls *setjmp*(*env*) to back up its environment to *env*, it can later call *longjmp*(*env*) to return to the backup position. Line 3 checks whether the worker is a survivor of a failure, who just jumped back from error handling. If so, it enters Line 4 to recover its data. Finally, Line 5 performs

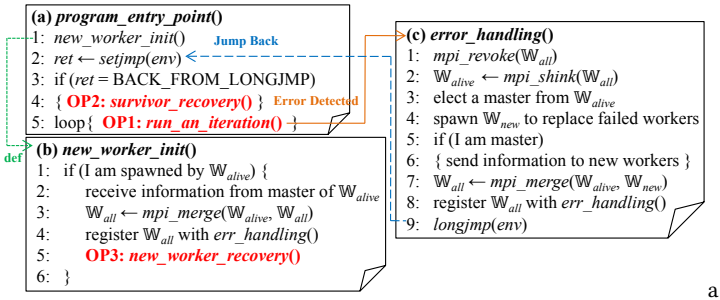

Figure 2: The Resilient Pregel Framework

iterative computation, and this is where communication error may occur to jump execution flow to *error_handling*().

We now consider the **execution flow of a surviving worker**. Suppose a worker W_f fails, then any worker W_d communicating with W_f will detect the failure and call *error_handling*(). In Fig. 2(c), W_d will then call ULFM primitive MPIX_Comm_revoke at Line 1 to notify other workers in $\mathbb{W}_{all}$ about the failure, and blocks on another ULFM primitive MPIX_Comm_shrink at Line 2. Upon receiving revoke notification, any other worker will immediately abort its on-going MPI communication primitive, and report an error; this directs its execution flow into *error_handling*() in Fig. 2(c), and reaching MPIX_Comm_shrink at Line 2.

MPIX_Comm_shrink ignores revoke notifications and blocks until every surviving worker calls it, upon which time it returns the set of surviving workers $\mathbb{W}_{alive}$ (see Line 2 in Fig. 2(c)). Then, the surviving workers elect master W_{max} at Line 3, and spawn a set of $(|\mathbb{W}_{all}| - |\mathbb{W}_{alive}|)$ new workers, $\mathbb{W}_{new}$, to replace the failed ones (Line 4) by calling MPI_Comm_spawn. The elected master then sends information to each new worker, such as the assigned worker ID and the latest checkpoint to load (Lines 5–6). Afterwards, a surviving worker recovers $\mathbb{W}_{all}$ by merging $\mathbb{W}_{alive}$ and $\mathbb{W}_{new}$ (Line 7) by calling MPI_Intercomm_merge, and then registers *error_handling*() to it (Line 8). Finally, *longjmp* is called at Line 9 to jump back to Line 2 of Fig. 2(a), after which Line 4 is called where a surviving worker recovers its data.

Finally, we consider the **execution flow of a respawned worker**. When $\mathbb{W}_{new}$ is created by Line 4 of Fig. 2(c), we have $\mathbb{W}_{all} = \mathbb{W}_{new}$ for every respawned worker. A respawned worker enters Line 1 of Fig. 2(a) to initialize its state, which is detailed in Fig. 2(b). Specifically, the worker first obtains information like its assigned worker ID and the latest checkpoint (Line 2), and then incorporates $\mathbb{W}_{alive}$ into $\mathbb{W}_{all}$ (Line 3) by calling MPI_Intercomm_merge and registers *error_handling*() to it (Line 4). Finally, the worker restores the pre-failure state of a failed worker in Line 5 (e.g., by loading a checkpoint), before returning to the main execution flow for iterative computation (i.e., Line 5 of Fig. 2(a)).

4 LIGHTWEIGHT CHECKPOINTING

Motivation. A conventional checkpoint is *heavyweight*, since it saves the following data for every vertex v: (1) value $a(v)$, (2) adjacency list $\Gamma(v)$, and (3) the set of received messages $M_{in}(v)$ in the next iteration (i.e., after message shuffling). Here, $M_{in}(v)$ is needed to update $a(v)$ and compute new messages in the next iteration, while $\Gamma(v)$ is needed since Pregel supports topology mutation.

However, a heavyweight checkpoint is often an overkill. Consider PageRank computation again. Recall that in Iteration i, once

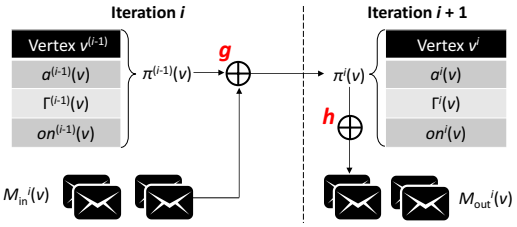

Figure 3: "Compute" Function for LWCP

a vertex v computes $a^{(i)}(v)$ using incoming messages, its outgoing messages can be directly derived from $a^{(i)}(v)$ and $\Gamma(v)$. Specifically, message value is $a^{(i)}(v)/|\Gamma(v)|$ and message targets are $\Gamma(v)$. As a result, it suffices to save a *lightweight* checkpoint (LWCP) that includes only the PageRank $a(v)$ of every vertex v: during recovery, $\Gamma(v)$ is static and can be directly loaded from the input, and outgoing messages of v can be computed from $a(v)$ and $\Gamma(v)$ for sending. Messages and edges do not need to be checkpointed.

This is a huge save in checkpointing cost, since $|E| \gg |V|$ usually holds in a real graph G, and messages are usually sent along edges in each iteration with an amount comparable to $O(|E|)$. Sec. 6 will show that when computing PageRanks on a web graph, it takes 60s to write a conventional checkpoint, but only 2s to write an LWCP.

Formally, LWCP is applicable if UDF *compute*(.) can be formulated as two functions below running in order:

$$\pi^{(i)}(v) \leftarrow g(v, \pi^{(i-1)}(v), M_{in}^{(i)}(v)), \quad (2)$$

$$M_{out}^{(i)}(v) \leftarrow h(v, \pi^{(i)}(v)). \quad (3)$$

Put simply, $g(.)$ first computes a new state for v from its old state and the messages received by v, and then $h(.)$ generates messages solely from the new state of v (rather than $M_{in}^{(i)}(v)$). Fig. 3 illustrates $g(.)$ and $h(.)$. Note that it is a special case of Fig. 1 and Eq (1).

Challenges. While LWCP is straightforward for computing PageRanks, some problems remain when generalizing it to an arbitrary Pregel program since: (1) some vertices may be inactive and they should not generate outgoing messages (e.g., in Hash-Min); (2) for Pregel algorithms with topology mutation, we should correctly recover adjacency list $\Gamma(v)$ of every vertex v from the checkpointed data; (3) we should tolerate the case where a vertex v needs to examine $M_{in}(v)$ to update $a(v)$ and generate new messages.

To illustrate Point (1), consider the Hash-Min algorithm where a vertex updates its state and sends messages only when it receives a smaller vertex ID. In this case, we can expand vertex value $a(v)$ with a boolean tag indicating whether the value is updated, so that $h(.)$ can be properly formulated without checking $M_{in}(v)$: $h(.)$ generates messages targeting neighbors $\Gamma(v)$ only if the tag is set.

Point (3) is common for Pregel algorithms that use the pointer jumping technique to bound the number of iterations, such as the S-V algorithm of [12] for computing connected components, and the minimum spanning forest algorithm of [8]. In these algorithms, a vertex v needs to respond to more and more vertices as the computation goes on, and we thus cannot expand $a(v)$ to include all their IDs. Fortunately, such cases only happen in a small fraction of iterations, and we can avoid writing an LWCP in those iterations.

Solution Overview. Interestingly, every Pregel algorithm we have seen so far falls into one of 3 categories: (1) *always-active algorithms* where all vertices are active and compute in all iterations, such as

computing PageRanks; (2) *traversal algorithms* where a vertex only sends messages in an iteration if its value is updated by the incoming messages, such as Hash-Min; (3) *request-responding algorithms* where in some iterations, a vertex needs to respond its state to many requesting vertices (recall the previous paragraph); we call such an iteration as a ***responding iteration***, where LWCP is inapplicable.

We aim to apply LWCP to an arbitrary Pregel algorithm as non-invasive as possible. For example, a Pregel program needs no change for always-active algorithms; but minor changes are needed in other cases to formulate UDF *compute*(.) according to the 2-phase functions f and g as shown in Eq. (2) and (3). This is a reasonable tradeoff between programming simplicity and checkpointing performance, and provides an option beyond naïve checkpointing.

Our LWCP solution consists of 2 techniques: (1) message recovery from vertex states; and (2) incremental edge checkpointing:

Message Recovery from Vertex States. We keep the familiar *compute*(.) function for users to implement, rather than ask users to explicitly implement two UDFs corresponding to Eq. (2) and (3). This is because some Pregel algorithms have responding iterations that cannot be thus formulated, while other algorithms like PageRank computation need no reformulation. However, we do require users to keep the 2-phase operations of Eq. (2) and (3) in mind when writing UDF *compute*(.) to utilize LWCP, as explained below.

Firstly, we require users to mask those iterations where LWCP is inapplicable (i.e., responding iterations) via another UDF *LWCPable*() called right before each iteration. If such an iteration is scheduled for checkpointing, our framework will postpone checkpointing till the first subsequent iteration where LWCP is applicable.

Secondly, for an iteration where LWCP is applicable, users may need to include additional fields into the vertex value (e.g., for Hash-Min), and they need to formulate the logic in two steps, (i) updating vertex state using incoming messages (i.e., Eq. (2)), followed by (ii) sending messages according to the updated vertex state (i.e., Eq. (3)). This is because when failure occurs, a revived worker only needs to run Step (ii) after loading the latest checkpointed vertex state (already updated before being checkpointed); as we shall see soon, our framework reuses UDF *compute*(.) for regenerating messages, but it disables vertex state updates in Step (i) to ensure correct message generation from checkpointed vertex states.

Due to the space limitation, we provided an example on how to write *compute*(.) of LWCP for the triangle finding algorithm of [7] to account for above considerations in the appendix of our earlier technical report of this paper on arXiv [13] for interested readers.

Incremental Edge Checkpointing. A conventional checkpoint stores $\Gamma(v)$ of every vertex v which costs $O(|E|)$ space in each checkpoint. We reduce the amount of saved data through incremental checkpointing: each worker W logs its requests of topology mutation to the local disk(s), and when W writes a new checkpoint, these logged requests are appended (i.e., committed) to a log file E_W on HDFS. The locally logged requests are then deleted from W's local disk(s). To recover the adjacency lists of vertices on W, W simply loads the initial edge data, and then replays the logged mutation requests (loaded from HDFS file E_W) till the latest checkpoint.

As an example, consider the k-core finding algorithm of [7] which only performs edge deletions during iterative computation. Using incremental checkpointing, at most $O(|E|)$ edge mutation data are written to HDFS throughout an entire job! Incremental edge checkpointing also applies to log-based recovery to be described in Sec. 5, as a surviving worker may forward the necessary edge mutation requests (loaded from its local log) to revived workers.

Implementation. Let us call the conventional heavyweight checkpointing approach as HWCP, and our new proposal as LWCP. We now explain how both approaches are implemented using our framework shown in Fig. 2, especially the behavior of OP1–OP3.

Since checkpointing is performed during normal execution, it is implemented inside OP1: *run_an_iteration*(), where a worker W processes each iteration in the following four steps:

- S1: UDF *compute*(.) is called on every active vertex in W;
- S2: messages are shuffled to the receiver side, global aggregator value and control information are synchronized;

and if the current iteration needs to be checkpointed:

- S3: the data of vertices in W are written to HDFS along with the synchronized aggregator value;
- S4: the previous checkpoint on HDFS is deleted.

A barrier is needed before "S3" to ensure that all workers have globally committed the iteration before checkpointing begins. A barrier is also needed before "S4" to ensure that all data of the current checkpoint has been saved (or old checkpoint is still valid).

We denote the checkpoint for Iteration i by $\Theta^{(i)}$, which consists of a file $\Theta^{(i)}_W$ on HDFS for each worker $W \in \mathbb{W}_{all}$. Specifically, each worker W contributes to $\Theta^{(i)}$ by writing the data of its vertices. To roll back later to Iteration i, W may simply load $\Theta^{(i)}_W$.

Our LWCP approach implements both recovery functions OP2 and OP3 in Fig. 2 with the same logic where each worker W:

- loads its vertices' states from the latest LWCP $\Theta^{(i)}_W$;
- generates messages from the loaded states (c.f. Eq. (3));
- shuffles the generated messages to the receiver side for use by vertices for running UDF *compute*(.) in Iteration $(i+1)$.

Moreover, adjacency lists are loaded from the input graph followed by replaying the logged topology mutations. If there is no topology mutation, surviving workers need not load adjacency lists.

Note that after loading an LWCP, we need to generate messages and then shuffle them. This is in contrast to HWCP which directly loads the shuffled messages at the receiver side. However, this one-off message generation cost for failure recovery pays off, as LWCP's faster checkpointing provides faster failure-free performance.

One problem remains: if an inactive vertex v does not receive any message in Iteration i, it will not call *compute*(.) and generate messages; as a result, if an LWCP $\Theta^{(i)}_W$ is loaded for recovery, v also should not generate any message, but this condition cannot be derived from v's loaded state which is after update by Iteration i. For example, $on^{(i)}(v) = false$ does not indicate whether v is inactive at the beginning of Iteration i, or v is active but then votes to halt.

To address this issue, we let each LWCP $\Theta^{(i)}$ additionally keep a boolean tag $\tau^{(i)}(v)$ for each vertex v indicating whether v.*compute*(.) is called in Iteration i. After $\Theta^{(i)}$ is loaded for recovery, our LWCP algorithm generates messages for a vertex v only if $\tau^{(i)}(v) = true$.

Another problem is how to generate messages from loaded vertex states according to Eq. (3) after failure happens. We would like to keep message generation as a transparent process without letting users implement another UDF for Eq. (3), i.e., by reusing UDF *compute*(.) which is formulated according to Eq. (2) and (3).

To avoid the part of $v.compute(.)$ that corresponds to Eq. (2) from changing (again) v's state loaded from $\Theta^{(i)}$, we condition the behavior of functions like *set_value*(.) and *vote_to_halt*(.) that are called in *compute*(.) on the context: if they are called right after loading $\Theta^{(i)}$ for recovery, they return without updating v's state.

5 LOG-BASED FASTER RECOVERY

Motivation. The LWCP approach described in Sec. 4 only reduces the failure-free execution cost paid for a job to be fault-tolerant. Even when only one machine fails, every machine still needs to rerun from the latest checkpointed iteration. Sec. 3.2 have discussed [10]'s message logging approach where surviving workers do not recompute during recovery, and we denote this algorithm as HWLog.

HWLog achieves faster recovery since messages are only transmitted to revived machines. Although a machine needs to log every message sent, [10] observed that streaming messages to local disk(s) is faster than sending messages over network, and hence message logging incurs negligible overhead during failure-free execution.

However, [10] never garbage-collects logged messages, which leads to problems in a realistic environment. Specifically, consider PageRank computation on a graph $G = (V, E)$ again, where each iteration sends a message along every edge. If the computation runs for 100 iterations, then $100 \cdot |E|$ messages are logged in total, whose volumn is about 100× that of G. This is yet one job, and message logs will soon use up disk space if multiple jobs are executed.

As a practical implementation, all previously logged messages should be garbage-collected right after a checkpoint is written, since only messages logged after the latest checkpoint are needed for recovery. Applying this approach to PageRank computation using checkpointing frequency $\delta = 10$ iterations, the log data volume never exceeds 10× that of G. However, as Sec. 6 will show, deleting the messages logged for the previous 10 iterations is quite time-consuming (e.g., OS needs to traverse inodes that keep the log file data), and this increases the failure-free job execution time.

We propose a novel solution called LWLog to address the above problem, and meanwhile, since LWLog is built on top of LWCP, it also enjoys faster checkpointing. Instead of logging messages like in [10], LWLog logs vertex states. When a surviving vertex needs to send messages to revived vertices, these messages are re-generated from the logged vertex states. Since the data volume of vertex states is much smaller than that of messages, deleting them is much faster and incurs negligible overhead during the failure-free execution.

Challenge. In Sec. 3.2 we have seen how HWLog works for PageRank computation, where we assume that a checkpoint is written every 10 iterations, and a failure occurs at Iteration 17. However, things become more complicated if cascading failures are considered, as the states of vertices may be at more than 2 different iterations. For example, assume that the first failure happens at Iteration 17 on worker W_1, and then during recovery, another failure happens at Iteration 15 on W_2. In this case, the states of revived vertices since the first (resp. second) failure are at Iteration 15 (resp. 10), while the states of all other vertices are at Iteration 17.

Our log-based approaches (HWLog and LWLog) aim to recover from any numbers of cascading failures, and the key design idea is that ***a vertex whose state is at Iteration i should perform vertex-centric computation only after Iteration i is recovered.***

Our approaches also aim to provide a wholesome solution to log-based recovery, including how to recover aggregator values and control information, which are not touched upon in [10].

Assumptions & Idea Overview. In both HWLog and LWLog, we assume that all current local logs are garbage-collected by the respective workers after a new checkpoint is written.

Recall from Sec. 3.2 that $s(W)$ is the state of W, i.e., $s(W) = i$ means that W partially commits Iteration i and thus, all vertex states and partially aggregated data and control information of W have been fully updated by Iteration i.

During log-based recovery, a worker W may have a state $s(W) > i$ in Iteration i, since the states of surviving workers are not rolled back, and these workers simply forward messages loaded (or generated) from local logs to those workers that perform computation.

We let each worker W keep track of the states of every other worker in $W' \in \mathbb{W}_{all}$ (i.e., $s(W')$), so that W knows whether W' will need messages for computing the next iteration, and only when this is the case will W send messages to W'.

To get updated worker states, when $\mathbb{W}_{all}$ is recovered as $\mathbb{W}_{alive} \cup \mathbb{W}_{new}$ after a failure, the workers would synchronize their states with each other. The synchronization is necessary since surviving workers can be at different iterations due to cascading failures, and a respawned worker has to get the states of all surviving workers for inferring the new master W_{max} (which has the largest state).

HWLog Implementation. We first consider OP1: *run_an_iteration*() in Fig. 2. In Iteration i, if a worker W performs vertex-centric computation (which updates $s(W)$ to i), the generated messages towards each $W' \in \mathbb{W}_{all}$ are processed in 2 steps as follows:

- messages are appended to a queue and combined at last;
- the combined messages are sent to W', and meanwhile, concurrently written to a file $F^{(i)}_{W \to W'}$ on W's local disk.

Since local disk write is typically faster than network transmission [10], log writing finishes earlier than message transmission.

We regard Iteration i as partially committed by W only if $F^{(i)}_{W \to W'}$ is fully written for every $W' \in \mathbb{W}_{all}$, since any worker W' may fail and need message retransmission during recovery.

Thus, when failure happens, the execution of *error_handling*() by a surviving worker in Fig. 2(b) needs to block until all its concurrent log-writes are complete. A worker also needs to guarantee that all log-writes are complete before fully committing an iteration, but this adds no overheads as message transmission is slower.

Let the current iteration number be i. In OP1: *run_an_iteration*(), a worker W's behavior has 3 cases depending on the value of $s(W)$:

- **Case 1: $\mathbf{s(W) \geq i}$.** In this case, W is a surviving worker who has partially committed Iteration i before, and thus it does not need to perform vertex-centric computation. Instead, it loads messages from $F^{(i)}_{W \to W'}$ for each target worker W' such that $s(W') \leq i$, and sends them to W'. This is because such a worker W' will perform computation at the next iteration (i.e., Iteration $(i + 1)$), which requires these messages.
- **Case 2: $\mathbf{s(W) = i - 1}$.** In this case, W needs to perform vertex-centric computation and updates its state $s(W)$ from $(i - 1)$ to i. All generated messages need to be logged, since any worker may fail later and request messages from W for recomputation. However, like in Case 1, only those messages towards a worker W' with $s(W') \leq i$ are actually sent.

- **Case 3: $s(W) < i - 1$.** This case is impossible, which can be proved by induction on i using the fact that in Case 2, if the state of a worker is less than the current Iteration i, it will perform computation and update its state to i.

As a wholesome solution, we now also consider the recovery of aggregator and control information. Recall that W_{max} is the elected master with the largest state, i.e., $s(W_{max})$. Then, we have 3 cases depending on how current iteration number i compares with $s(W_{max})$:

- **Case 1: $i < s(W_{max})$.** In this case, no worker synchronization is performed since recovery has not reached the previous failed Iteration $s(W_{max})$ yet, and thus every worker may simply obtain the global aggregator value and control information from W_{max} who logged them before.
- **Case 2: $i = s(W_{max})$.** In this case, W_{max} has not globally committed Iteration i yet and thus worker synchronization is necessary for recovering Iteration i; note that W_{max} is partially committed and already has logged the partially aggregated value and control information to be synchronized.
- **Case 3: $i > s(W_{max})$,** which is impossible since W_{max} is the longest-living worker, and thus must live through the current iteration (i.e., $s(W_{max}) \geq i$).

We next consider the logic of OP2: *survivor_recovery*() in Fig. 2(a) and OP3: *new_worker_recovery*() in Fig. 2(b), when failure happens.

- In OP2: *survivor_recovery*(), a surviving worker W retains its state $s(W)$ but sets the iteration number back to the latest checkpointed one; the message queues of W are cleared to remove on-the-fly messages (being transmitted in the failed iteration), so that these queues can be used to accommodate messages read from local logs during later recovery.
- In OP3: *new_worker_recovery*(), a revived worker W sets both its state $s(W)$ and the iteration number to the latest checkpointed iteration; it also loads the latest checkpoint, which contains incoming messages for the next iteration.

Once the states of surviving and revived workers are restored by OP2 and OP3, respectively, subsequent recovery proceeds in iterations by running OP1: *run_an_iteration*() as described before where only necessary messages are sent depending on how worker states compares with the current iteration number. Once Iteration $s(W_{max})$ is recovered, computation returns to normal.

LWLog Implementation. The algorithm of LWLog is similar to that of HWLog, and we focus on presenting their differences.

Note that LWLog is built on top of LWCP, and thus like in LWCP, LWLog requires users to formulate *compute*(.) according to Eq. (2) and (3). In order words, if one uses LWCP by properly formulating *compute*(.), LWLog is a free ride and can be enabled.

Unlike HWLog which logs messages generated by vertices, LWLog directly logs vertex states. As a result, the amount of data written to a local log is much smaller and hence faster to garbage-collect.

For each vertex v, LWLog logs only $\tau^{(i)}(v)$ and $a^{(i)}(v)$. Recall that $\tau^{(i)}(v)$ indicates whether v calls *compute*(.) in Iteration i. If a worker needs to generate messages of Iteration i for forwarding, it generates messages for a vertex v only if $\tau^{(i)}(v) = true$.

While an LWCP saves $on^{(i)}(v)$ of a vertex v, $on^{(i)}(v)$ does not need to be stored in a local vertex-state log since the logged states are just for message generation and do not overwrite the current vertex states. In other words, it suffices to store $\tau^{(i)}(v)$ and $a^{(i)}(v)$ to a local log, for generating messages in potential future recovery.

Like in our LWCP approach, LWLog also uses *compute*(.) for generating messages during recovery rather than asks users to specify $h(.)$ of Eq. (3). When recovering messages, *compute*(.) temporarily ignores updates to vertex states as in LWCP.

Let us denote the latest checkpointed iteration by ℓ. There are 2 places that require message generation:

- When a failure occurs, a *respawned worker* W loads checkpoint $\Theta_W^{(\ell)}$ and uses the loaded vertex states to generate messages for sending, which is the same as in LWCP. In contrast, a *surviving worker* W directly loads the proper local vertex-state log file(s) and generates messages from the loaded vertex states for sending. This is possible because LWLog adopts a slightly different garbage collection strategy from HWLog: when a new LWCP $\Theta^{(i)}$ is written, all local logs written before Iteration i are deleted, but the logs written in Iteration i is retained for later recovery.
- During a recovery iteration $i > \ell$, a surviving worker that needs to forward messages simply loads the proper local vertex-state log file(s) and generates messages from the loaded vertex states for sending.

One problem remains: how LWLog handles a masked iteration where LWCP is inapplicable? Our solution is simple: since the outgoing messages need to be computed using the incoming messages in such an iteration (rather than recovered only from the vertex states), LWLog switches temporarily to message logging instead of vertex-state logging if an iteration is masked, so that these messages can be directly loaded for forwarding in potential future recovery.

Table 1: Graph Datasets

Data	Type	|V|	|E|	AVG Deg	Max Deg
WebUK	directed	133,633,040	5,507,679,822	41.21	22,429
WebBase		118,142,155	1,019,903,190	8.63	3,841
Friendster	undirected	36,869,292	3,612,134,270	55.06	5,214
Orkut×12		164,732,473	2,812,437,576	4.69	1,637,619

6 EXPERIMENTS

We now report the performance of checkpointing-based methods (1) HWCP and (2) LWCP and log-based methods (3) HWLog and (4) LWLog. Our focus is on checkpointing time and recovery time.

All experiments were conducted on a cluster of 15 machines connected by Gigabit Ethernet, each with two Intel Xeon E5-2620 CPUs and 48GB RAM. We ran 8 workers on each machine, and thus 120 workers in total. We repeat each experiment for 10 times and the reported time are averaged over the 10 runs. All our codes are released at http://www.cse.cuhk.edu.hk/pregelplus/ft.html.

Datasets. Table 1 shows the datasets used in our experiments, including two directed web graphs *WebUK*[1] and *WebBase*[2], and two undirected social networks *Friendster*[3] and *Orkut*[4]. Since *Orkut* is a small graph, we replicate *Orkut* for 12 times to get an undirected graph whose size is comparable to *Friendster*.

[1] http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05
[2] http://law.di.unimi.it/webdata/webbase-2001
[3] http://snap.stanford.edu/data/com-Friendster.html
[4] http://konect.uni-koblenz.de/networks/orkut-links

Algorithms. We consider 3 Pregel algorithms with different workloads: (1) PageRank, (2) triangle counting, and (3) Hash-Min.

PageRank computation is time-consuming for a big graph since in each iteration, every vertex v needs to receive values from its in-neighbors and distributes its new PageRank among v's out-neighbors. However, the time of an iteration is stable throughout the computation, since the workload in every iteration is the same.

Triangle finding generates huge amounts of intermediate messages during the computation. For example, in the algorithm of [7], to find a triangle, $\triangle v_1v_2v_3$ (assuming $v_1 < v_2 < v_3$ ordered by vertex ID), vertex v_1 needs to send a message to v_2 asking it whether $v_3 \in \Gamma(v_2)$. Since a graph can have $O(|E|^{1.5})$ triangles [9], the message volume is at least $O(|E|^{1.5})$, which is superlinear to the graph size. Finding all triangles in one round leads to long-running iterations that are susceptible to machine failures and total recomputation. Also, the aggregated memory in a cluster may not be sufficient to buffer all the messages. We adopt a multi-round solution like in [6] where each round only computes a fraction of triangles, and our algorithm is detailed in our technical report [13]. Its variant for *triangle counting* was used in our experiments.

Hash-Min [12] computes the connected components of an undirected graph by letting every vertex maintain and propagate the smallest vertex ID it has seen. For a large graph, Hash-Min is time-consuming in the first few iterations where most vertices call *compute*(.) and send messages, but as the computation goes on, most vertices are converged and thus each iteration takes a short time.

Since PageRank is designed for (directed) web graphs, we ran it on the two directed graphs, *WebUK* and *WebBase*. In contrast, we ran experiments of triangle counting and Hash-Min on the two undirected graphs *Friendster* and *Orkut* ×12, since these problems are defined for undirected graphs.

For all the 3 Pregel algorithms, we write a checkpoint every δ iterations. However, in triangle counting and Hash-Min, the running time of an iteration decreases with the iteration number, and the short-running iterations towards the end are so fast that checkpointing every δ iterations becomes the major overhead rather than the actual computation. Therefore, for triangle counting and Hash-Min, we mask those later iterations to avoid checkpointing.

6.1 Experiments on PageRank Computation

In this set of experiments, we ran the PageRank algorithm of [5], and wrote a checkpoint for every 10 iterations. We killed a worker at Iteration 17 to simulate a worker failure. For PageRank computation, during normal execution (or recovery), every iteration generates the same number of messages and thus has a stable runtime. We thus report the average running time of an iteration.

Performance Metrics. There are 4 stages in this set of experiments, which give rise to four time metrics as follows:

- **Stage 1:** the job first executes normally from Iteration 1 to Iteration 16, and we define $\boldsymbol{T_{norm}}$ as the running time of an iteration averaged over these 16 iterations.
- **Stage 2:** after failure occurs at Iteration 17, the latest checkpointed iteration (i.e., 10) is recovered in time $\boldsymbol{T_{cpstep}}$. This time is dependent on the fault tolerance approach used: in HWCP & LWCP, every worker W loads checkpoint $\Theta_W^{(10)}$; also, in LWCP & LWLog, messages generated from vertex states need to be shuffled to the receiver side.
- **Stage 3:** after Iteration 10 is recovered, the job reruns from Iteration 11 to Iteration 16. We define $\boldsymbol{T_{recov}}$ as the running time of an iteration averaged over these 6 iterations, which is expected to be much shorter than T_{norm} in HWLog & LWLog since messages are not sent to surviving workers.
- **Stage 4:** finally, the recovery reaches Iteration 17, and we denote the time of recovering it by $\boldsymbol{T_{last}}$. This metric represents the time of recovering the iteration where the failure occured. We separate T_{last} from T_{recov} since all messages need to be transmitted in Iteration 17, even for HWLog and LWLog, as normal computation resumes at Iteration 18, which need all the messages generated in Iteration 17.

While our algorithms support cascading failures, considering them leads to more stages and thus more time-metrics to report. We avoid this complexity to keep the experiment presentation succinct.

Among the metrics, T_{norm} is averaged over 16 iterations and T_{recov} is averaged over 6 iterations, which is good enough since the time of an iteration is stable in each stage. All of T_{cpstep}, T_{recov} and T_{last} reflect the performance of recovery, but T_{recov} is the most important since it is averaged over multiple recovery iterations while T_{cpstep} and T_{last} each just reflects the time of one recovery iteration. Also, T_{norm} is the time of running one normal iteration, and is reported only for reference rather than for improving.

We also report metrics on the cost of checkpointing and logging:

- $\boldsymbol{T_{cp}}$: the time of writing a checkpoint, which is $\Theta_W^{(10)}$ in this set of experiments. It includes the time of any garbage collection operations following the checkpoint writing.
- $\boldsymbol{T_{cpload}}$: the time of loading a checkpoint (i.e., $\Theta_W^{(10)}$ here). This time is averaged over every worker W that loads $\Theta_W^{(10)}$ from HDFS, and it is actually part of T_{cpstep}. Note that survivors do not load checkpoint in HWLog and LWLog.
- $\boldsymbol{T_{log}}$: the time of writing a local log. This time is averaged over all workers that write a log and over all iterations (both in normal execution and during recovery).
- $\boldsymbol{T_{logload}}$: the time of loading a local log. Like T_{log}, this time is averaged over all workers that load a log, and over all iterations during recovery.

Performance Highlights. Fig. 4 (resp. Fig. 5) highlights the most important performance metrics including T_{norm}, T_{recov} and T_{cp}, for computing PageRanks over *WebUK* (resp. *WebBase*). We can see that during normal execution, an iteration takes around 32 s on *WebUK*, and around 17 s on *WebBase*.

In HWCP and LWCP, T_{recov} is similar to T_{norm} since they simply rerun the computation during recovery. In contrast, in HWLog and LWLog, T_{recov} is many times shorter than T_{norm} since they only transmit those messages towards the respawned worker during recovery. For example, T_{recov} is around 4× (resp. 8× times) shorter than T_{norm} on *WebUK* (resp. *WebBase*).

However, recall that we only kill one of the 120 workers and thus the message volume to be transmitted is reduced to approximately 1/120 of that during normal execution. But T_{recov} is not reduced to 1/120 of T_{norm}, which is because of 2 reasons:

- Vertex-centric computation and message combining are performed in parallel by all workers during normal execution, and this time cannot be reduced since the revived worker still needs to perform these operations in every iteration;

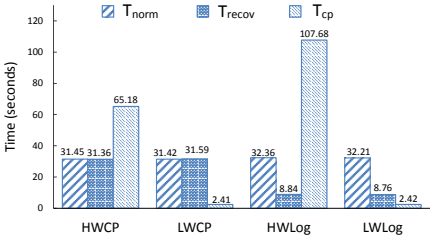

Figure 4: PageRank over *WebUK*

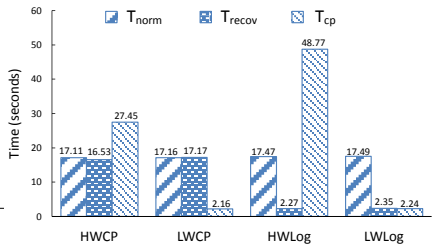

Figure 5: PageRank over *WebBase*

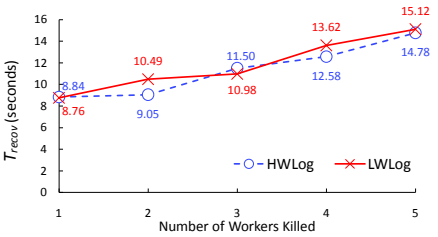

Figure 6: Effect of # of Failed Workers

- Only the respawned worker receives messages, which results in a communication bottleneck on the receiver side.

We also examined the performance of HWLog and LWLog when more workers were killed at Iteration 17. The most important observation is the increase in T_{recov}, which is because more messages need to be transmitted to the respawned workers during recovery.

Fig. 6 reports the value of T_{recov} as the number of killed workers increases in the same previous experiments on *WebUK*. We can see that T_{recov} increases linearly with the number of workers killed.

From Fig. 4 and 5, we also see that T_{cp} is sensitive to the fault tolerance approach adopted. In LWCP and LWLog, T_{cp} is less than 2.5 s on both datasets, which demonstrates that LWCPs are efficient. Compared with the corresponding T_{norm}, the checkpointing time is negligible. In contrast, in HWCP and HWLog, T_{cp} is a few times that of the corresponding T_{norm} since checkpoints are heavyweight.

Also note that HWLog has a much longer T_{cp} than HWCP. For example, T_{cp} is 65.18 s in HWCP but 107.68 s in HWLog. The additional time consumed by HWLog is for deleting the logged messages of the previous $\delta = 10$ iterations. We can see that if garbage collection is performed, HWLog even degrades the failure-free performance compared with HWCP, though recovery becomes faster. In contrast, we can see that LWLog has similar T_{cp} to LWCP, since the additional cost for deleting vertex-state logs is negligible.

Other Metrics. Table 2 reports the other time metrics related to the cost of checkpointing and recovery, though T_{norm} is also included for reference. The entry value '–' in Table 2 means not applicable.

Table 2: Other Time Metrics for PageRank Computation

(a) Other Time Metrics on *WebUK*

	T_{norm}	T_{cpstep}	T_{last}	T_{cpload}	T_{log}	$T_{logload}$
HWCP	31.45 s	15.43 s	31.51 s	5.95 s	–	–
LWCP	31.42 s	40.84 s	30.34 s	3.28 s	–	–
HWLog	32.36 s	16.83 s	29.61 s	3.69 s	1.31 s	0.84 s
LWLog	32.21 s	18.00 s	30.62 s	3.14 s	0.19 s	0.11 s

(b) Other Time Metrics on *WebBase*

	T_{norm}	T_{cpstep}	T_{last}	T_{cpload}	T_{log}	$T_{logload}$
HWCP	17.11 s	6.58 s	17.74 s	2.83 s	–	–
LWCP	17.16 s	21.64 s	17.01 s	1.96 s	–	–
HWLog	17.47 s	4.79 s	15.99 s	2.23 s	0.81 s	0.56 s
LWLog	17.49 s	7.59 s	16.33 s	2.10 s	0.08 s	0.02 s

First, look at the metrics related to recovery: T_{cpstep} of LWCP and LWLog is longer than that of HWCP and HWLog, since in order to recover Iteration 10 after rolling back, a worker in LWCP and LWLog needs to generate messages from vertex states and shuffle them to the receiver side, while a worker in HWCP and HWLog directly loads incoming messages for Iteration 11 from $\Theta^{(10)}$.

Also, T_{cpstep} is much shorter than T_{norm} in HWCP and HWLog, since when recovering Iteration 10, incoming messages are directly loaded from $\Theta^{(10)}$ whose time cost is much less than that of vertex-centric computation plus message combining and transmission as required in normal execution.

In contrast, T_{cpstep} is even longer than T_{norm} in LWCP, since LWCP transmits the same amount of messages during recovery as in normal execution, but these messages are generated from vertex states which are first loaded from $\Theta^{(10)}$ on HDFS. However, this does not mean that LWCP is inferior to HWCP, since T_{cpstep} is just a one-off cost for recovering a failure (which happens infrequently), while LWCP significantly reduces the checkpointing time and thus improves the failure-free performance of any job.

Although we have seen that T_{cp} is much more expensive in HWCP and HWLog than in LWCP and LWLog, T_{cpload} is relatively efficient (around 2–5 s) for all the fault tolerance approaches. This is because HDFS favors reading over writing: HDFS writing needs to replicate data to multiple machines for fault tolerance, but HDFS reading may just read the nearest data replica which is efficient.

Table 2 also shows that the cost of log loading/writing is negligible. Specifically, T_{log} is only around 1 second for HWLog, and even much shorter for LWLog. Similarly, $T_{logload}$ is also very short. Such efficiency is contributed by the fact that OS memory cache provides locality for sequential local reads/writes.

Since a worker in our log-based approaches transmits and logs outgoing messages in parallel and T_{log} is much shorter than T_{norm}, logging incurs negligible overhead to normal execution.

6.2 Experiments on Triangle Counting

We now report our experiments on triangle counting, running on *Friendster* and *Orkut*. Unlike in PageRank, the time of each round decreases as the algorithm runs on, since more and more vertices exhaust their neighbor-pairs for triangle probing.

The time of an iteration has not dropped significantly for the first 20 (resp. 8) iterations on *Friendster* (resp. *Orkut*), and thus we write a checkpoint every 10 (resp. 4) iterations and kill a worker at Iteration 20 (resp. 8). Since the average time of an iteration is no longer representative, we redefine the metrics for *Friendster* as follows (*Orkut*'s metrics are similarly defined): (1) T_{norm}: the total time taken by running Iterations 11–19 normally before worker failure occurs; (2) T_{recov}: the total time taken by recovering Iterations 11–19 after worker failure is detected; (3) T_{cp}: the time for checkpointing an iteration. We focus only on iterations between 10 and 20 in order to compare T_{recov} with T_{norm}.

Fig. 7 and 8 report the performance results where we obtain similar observations as in the PageRank experiments: HWLog and LWLog have much smaller T_{recov} than T_{norm}, and LWCP and LWLog have much smaller T_{cp} than HWCP and HWLog.

6.3 Experiments on Hash-Min

We now report our experiments on Hash-Min, running on *Friendster* and *Orkut*. The time of each iteration decreases as the algorithm

runs on, but it has not dropped significantly for the first 4 iterations on both datasets. For example, on *Friendster*, the first 4 iterations each takes around 18-19 s, but Iterations 5 and 6 take 4.1 s and 0.22 s, respectively. We thus write a checkpoint every 2 iterations and kill a worker at Iteration 4. We define the metrics similarly as in triangle counting, and the results are reported in Fig. 9 and 10.

We again obtain similar observations as in previous experiments: HWLog and LWLog have much smaller T_{recov} than T_{norm}, and LWCP and LWLog have much smaller T_{cp} than HWCP and HWLog.

6.4 System Comparison

To show fairness of our comparison, we now demonstrate that our baseline algorithm, HWCP, is already faster than existing systems including Giraph 1.0.0, GraphLab 2.2 and GraphX (Spark 1.1.0), which only support the HWCP approach. Thus, the performance improvement by our LWCP and LWLog is not due to unfavorable implementation of our baseline. We repeated the PageRank experiments of Sec. 6.1 on these systems and report the major time metrics T_{norm} and T_{cp} in Fig. 11 and 12. We can see that our HWCP implementation has a much shorter T_{norm} than the others, and that our T_{cp} is comparable to Giraph's, and much shorter than that of GraphLab and GraphX.

Since [10] implements HWLog in Giraph, we also repeated our PageRank experiments using [10]'s system. Their system does not work properly with the multithreading option of Giraph 1.0.0, and we were only able to run one worker on each machine. Fig. 13 (resp. Fig. 14) reports the major performance metrics of their system for running HWCP and HWLog, where we also compared with our implementation (to be fair, we only run one worker per machine in this comparison). We see that [10]'s implementation is much more expensive than ours.

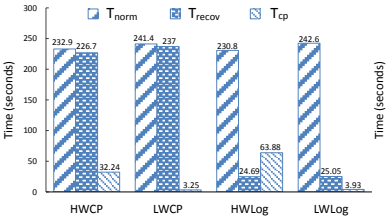

Figure 7: Triangle-Count over *Friendster*

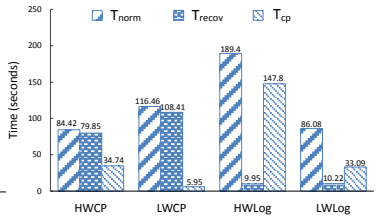

Figure 8: Triangle-Count over *Orkut*

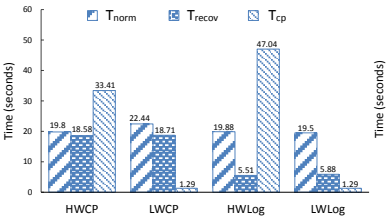

Figure 9: Hash-Min over *Friendster*

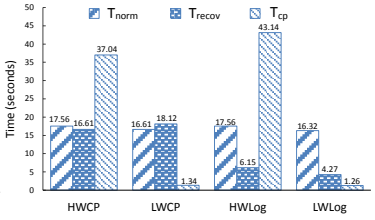

Figure 10: Hash-Min over *Orkut*

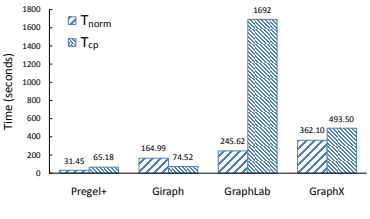

Figure 11: HWCP on *WebUK*

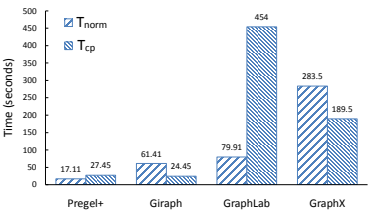

Figure 12: HWCP on *WebBase*

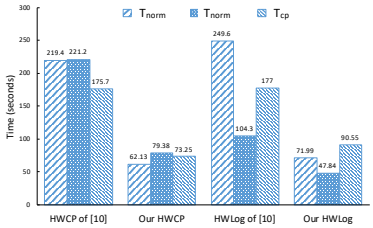

Figure 13: HWLog on *WebUK*

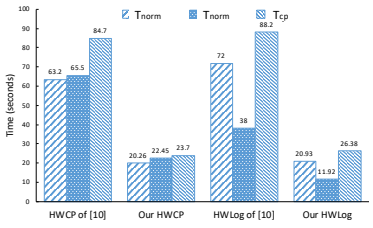

Figure 14: HWLog on *WebBase*

7 CONCLUSIONS

This paper proposed a lightweight checkpointing method that significantly reduces the checkpointing time, and handles challenges like graph mutation and iterations where LWCP is inapplicable.

The idea is further combined with vertex-state log based recovery to reduce recovery time, without sacrificing the benefit of faster checkpointing provided by LWCP.

Acknowledgments. This work was partially supported by NSF OAC-1755464, DGE-1723250 and CCF-1562306; ITF 6904945 and GRF 14222816; NTU SUG (Singapore) and SPARC (MHRD, India).

REFERENCES

[1] Wesley Bland, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. 2012. A proposal for User-Level Failure Mitigation in the MPI-3 standard. *Dept. of EECS, University of Tennessee* (2012).

[2] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *PVLDB* 8, 12 (2015), 1804–1815.

[3] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.* 34, 3 (Sept. 2002), 375–408.

[4] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*. 599–613.

[5] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*. 135–146.

[6] Ha-Myung Park, Francesco Silvestri, U. Kang, and Rasmus Pagh. 2014. MapReduce Triangle Enumeration With Guarantees. In *CIKM*. 1739–1748.

[7] Louise Quick, Paul Wilkinson, and David Hardcastle. 2012. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. In *ASONAM*. 457–463.

[8] Semih Salihoglu and Jennifer Widom. 2014. Optimizing Graph Algorithms on Pregel-like Systems. *PVLDB* 7, 7 (2014), 577–588.

[9] Thomas Schank. 2007. Algorithmic aspects of triangle-based network analysis. *Phd in computer science, University Karlsruhe* (2007).

[10] Yanyan Shen, Gang Chen, H. V. Jagadish, Wei Lu, Beng Chin Ooi, and Bogdan Marius Tudor. 2014. Fast Failure Recovery in Distributed Graph Processing Systems. *PVLDB* 8, 4 (2014), 437–448.

[11] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation. In *WWW*. 1307–1317.

[12] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. *PVLDB* 7, 14 (2014), 1821–1832.

[13] Da Yan, James Cheng, and Fan Yang. 2016. Lightweight Fault Tolerance in Large-Scale Distributed Graph Processing. *CoRR* abs/1601.06496 (2016).