A Fault-Tolerant Distributed Framework for Asynchronous Iterative Computations

Tian Zhou, Lixin Gao, *Fellow, IEEE,* Xiaohong Guan, *Fellow, IEEE*

Abstract—Asynchronous iterative computations (AIC) are common in machine learning and data mining systems. However, the lack of synchronization barriers in asynchronous processing brings challenges for continuous processing while workers might fail. There is no global synchronization point that all workers can roll back to. In this paper, we propose a fault-tolerant framework for asynchronous iterative computations (FAIC). Our framework takes a virtual snapshot of the AIC system without halting the computation of any worker. We prove that the virtual snapshot capture by FAIC can recover the AIC system correctly. We evaluate our FAIC framework on two existing AIC systems, Maiter and NOMAD. Our experiment result shows that the checkpoint overhead of FAIC is more than 50% shorter than the synchronous checkpoint method. FAIC is around 10% faster than other asynchronous snapshot algorithms, such as the Chandy-Lamport algorithm.Our experiments on a large cluster demonstrate that FAIC scales with the number of workers.

Index Terms—Fault-tolerance, cloud computing, asynchronous iterative computation, asynchronous snapshot

1 Introduction

Many machine learning and data mining algorithms perform iterative computations through iterative refinement. As the size of raw data gets larger and larger, distributed computation frameworks such as MapReduce [1] and Dryad [2] have been deployed to process the data. Traditionally, distributed iterative computations are implemented in a synchronous manner. Many distributed synchronous frameworks such as HaLoop [3], iMapReduce [4] use the Bulk Synchronous Parallel (BSP) model [5]. While distributed synchronous frameworks are a natural choice for iterative computations due to their simplicity, a synchronization barrier between two consecutive iterations is essential to synchronize the progress of all workers. That is, each worker has to pause and wait until all the workers reach the synchronization point.

Recently, myriad asynchronous computation models have been proposed to accelerate iterative computations. One common theme of these proposed asynchronous computation models is the removal of the synchronization barriers. As a result of this, each worker is able to perform iterative updates without waiting for other workers to complete the iteration. Therefore, asynchronous computations can accelerate iterative computations. On the other hand, the lack of synchronization barriers brings challenges for continuous processing in the event of server failures. The synchronization barriers provide a global synchronization point where all workers can safely roll back to in the case of server failures. The asynchronous iterative computation, however, does not naturally contain such rollback points.

In addition to providing on-demand servers, many cloud service providers offer transient resources such as spare servers at a fraction of the cost of on-demand servers. These transient resources may be revoked and tasks can be preempted at any time [6]. Iterative machine learning jobs on large-scale datasets typically require a large amount of resources and usually are not urgent tasks. So they are ideally suited to run on such transient resources. Since the computation resources may be revoked at any time, it is better to checkpoint more frequently. Most popular distributed frameworks, such as Hadoop [7], Spark [8], and Maiter [9] are designed to work on on-demand servers where the mean time between errors is in the order of hours or even days. They might suffer huge cost of re-computations in case of frequent revocations on transient resources.

However, it is hard to perform recovery from failures or revocations on asynchronous iterative systems. A distributed system supports fault tolerance by setting checkpoints and recovering to them after a failure happens. Different from synchronous iterative systems, states of servers are usually not sufficient to describe the state of an asynchronous iterative computation system. Inflight messages are not part of the local state but contains necessary information about how to update the state of a worker.

In this paper, we propose a Fault-tolerant framework for Asynchronous Iterative Computations (FAIC framework). We capture the messages by leading the asynchronous iterative computation system into a state where inflight messages are absorbed by workers. And workers keep on computing during that period. Instead of forcing all workers to reach such a state at a certain moment, we design a method that captures the messages and local state on each worker independently. We make a checkpoint of the system by constructing these local states captured at different moment into a *virtual snapshot*. Therefore, FAIC brings only small check-

- *T. Zhou was with Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China, 710049 and Department of Electrical and Computer Engineering, University of Massachusetts Amhers, Amherst, USA, 01003.*
  E-mail: tianzhou@stu.xjtu.edu.cn, tzhou@umass.edu
- *L. Gao was with Department of Electrical and Computer Engineering, University of Massachusetts Amhers, Amherst, USA, 01003.*
  E-mail: lgao@ecs.umass.edu
- *X. Guan was with Systems Engineering Instituteg, Xi'an Jiaotong University, Xi'an, China, 710049.*
  E-mail: xhguan@sei.xjtu.edu.cn

point overhead because it does not halt the computation of any worker during checkpointing. We prove that a virtual snapshot can recover the system correctly in Section 4. Meanwhile, FAIC can be more efficient than asynchronous snapshot methods such as the Chandy-Lamport snapshot algorithm [10]. While the Chandy-Lamport algorithm has to capture messages one-by-one, FAIC can aggregate messages before archiving them. As a result, FAIC archives less amount of data and takes less time in checkpointing than the Chandy-Lamport algorithm does.

We evaluate our FAIC framework on two existing asynchronous iterative frameworks, Maiter and NOMAD. Our evaluation shows that the checkpoint overhead of FAIC is around half of that in the synchronous checkpoint method on a homogeneous cluster and about 1/3 of that on a heterogeneous cluster. FAIC is about 10% faster than the Chandy-Lamport algorithm, because it archives less messages. Even on NOMAD where messages are not accumulative, FAIC has the same running time as the Chandy-Lamport algorithm. Our experiments on a large cluster also demonstrate that FAIC scales well with the cluster size.

2 Asynchronous Iterative Computation

In this section, starting with two existing frameworks, we introduce the asynchronous iterative computation model for distributed computation. Then, we formulate a distributed framework supporting asynchronous iterative computation models.

2.1 Asynchronous Iterative Computation Model

Many distributed frameworks that use asynchronous iterative computation model. We first show two example frameworks: Maiter and NOMAD. Then, we introduce the mathematical model of asynchronous iterative computations.

2.1.1 Maiter

Maiter [9] is an asynchronous distributed framework for graph processing. We use PageRank as an example to show how the Maiter framework can be used for a large class of graph processing algorithms. PageRank iteratively updates the PageRank score v_i of each node i as follows.

$$v_i = \frac{1-d}{N} + \sum_{j \in IN(i)} \frac{d \cdot v_j}{|ON(j)|}$$

where N is the total number of nodes, $IN(i)$ and $ON(i)$ is the in-neighbor and out-neighbor set of node i respectively, and d is a damping factor. Note that the PageRank score of node i depends on the PageRank score of its in-neighbor nodes. If each node's PageRank score is initialized with

$$v_i^0 = \frac{1-d}{N},$$

and Maiter maintains a change variable for each node i with Δv_i, then node j, an in-neighbor of node i, will contribute to the change to node i with

$$\frac{d \cdot \Delta v_j}{|ON(j)|}.$$

As a result, each node i needs to perform the following operations.

$$\Delta v_i = \sum_{j \in IN(i)} \frac{d \cdot \Delta v_j}{|ON(j)|}$$

$$v_i \leftarrow v_i + \Delta v_i$$

Note that computation $\frac{d \cdot \Delta v_j}{|ON(j)|}$ can be done at node j. The result is sent to all out-neighbors of node j and a node i accumulates the result into Δv_i upon receiving such a result. As a result, Pagerank can be performed asynchronously at each node where node i will perform the following operations:

- For each message m, accumulate the message to Δv_i. That is, $\Delta v_i = \Delta v_i + m$
- $v_i \leftarrow v_i + \Delta v_i$.
- Compute $\frac{d \cdot \Delta v_i}{|ON(i)|}$ and send the result to all out-neighbors.

2.1.2 NOMAD

NOMAD [11] is an asynchronous distributed framework for big matrix completion problems. Let $M \in \mathbb{R}^{n \times m}$ be a matrix where only some of its entries is observed, denoted with $\Omega \subseteq \{1, \ldots, m\} \times \{1, \ldots, n\}$, the matrix completion problem is to predict the values of unobserved entries. A popular model is to find two small matrix $H \in \mathbb{R}^{n \times m}$ and $W \in \mathbb{R}^{n \times h}$ where $h \ll \min\{n, m\}$ such that $M \approx HW$. H and W can be viewed as a set of row blocks and column blocks respectively. That is, $H = [H_1, H_2, \cdots, H_n]^T$ and $W = [W_1, W_2, \cdots, W_m]$. The goal is to minimize the loss function

$$\begin{aligned} J(H, W; M) &= \frac{1}{2}||M - HW||_2^2 \\ &= \frac{1}{2} \sum_{(i,j) \in \Omega} (M_{i,j} - H_i W_j)^2 \\ &= \sum_{(i,j) \in \Omega} J(H_i, W_j, M_{i,j}) \end{aligned}$$

It can be solved by iteratively updating H and W using gradient descent as follows where η is a positive learning rate.

$$H_i \leftarrow H_i - \eta \sum_j \frac{\partial J(H_i, W_j, M_{i,j})}{\partial H_i} = H_i - \eta \sum_j W_j^T (H_i W_j - M_{i,j})$$

$$W_j \leftarrow W_j - \eta \sum_i \frac{\partial J(H_i, W_j, M_{i,j})}{\partial W_j} = W_j - \eta \sum_i H_i^T (H_i W_j - M_{i,j})$$

To perform the computation asynchronously, NOMAD guarantees that there is only one copy of each H_i and W_i. Thus, we can independently perform the computation $W_j^T (H_i W_j - M_{i,j})$ and $H_i^T (H_i W_j - M_{i,j})$ for each (i, j) pair $\in \Omega$. We assign each H_i to a computation node i and let W_j blocks traverse all nodes through messages. When the message of W_j reaches node i, node i performs the following operations.

- $H_i \leftarrow H_i - \eta \sum_j W_j^T (H_i W_j - M_{i,j})$.
- $W_j \leftarrow W_j - \eta \sum_i H_i^T (H_i W_j - M_{i,j})$.
- Send new W_j to node $i + 1$.

2.1.3 Other Distributed Frameworks

Many other distributed frameworks can be described with this asynchronous iterative computation model. For example, the GraphLab [12] is a distributed framework designed for graph processing tasks, especially for iterative graph algorithms. It models a graph with state data on each node and edge. A node or edge updates its state value according to its neighbors' current state values. GraphLab supports both the synchronous execution and asynchronous execution. In the asynchronous execution, when a node updates its states, that change is pushed by its neighbors via data messages. When the message is received by another node, the corresponding change becomes visible to the receiving node. Each node updates its own state asynchronously. When it is going to update its own state, it uses the neighbor states which is currently visible to it. In this model, the receiving thread keeps on receiving messages and make them visible to local nodes. The computing thread updates local state using the locally visible data and generates messages about the new state. The sending thread sends out these messages to workers holding their neighbor nodes.

2.1.4 AIC Model

We summarize the asynchronous iterative computation (AIC) model in this section. There are two types of data in the model, the static data and mutable variables. The static data is usually the input of the problem which is not changed during the computation process, like the graph topology for the PageRank algorithm. While the mutable variables describe the current state of all units, like the PageRank scores. We describe a basic operation unit in an iterative computation model as a computation *node*. Each node holds a part of the static data and the corresponding part of the mutable data.

Each node iteratively updates its mutable variable as Equation (1). In general, a node i accumulates the mutable variables v_j of all other nodes and use them to update its own mutable variable v_i.

$$v_i = \bigoplus_j f(v_j, v_i; D_j, D_i) \tag{1}$$

The term $f_{j,i}(v_j, v_i; D_j, D_i)$ in the equation quantifies the impact of node j's mutable variable to node i's mutable variable. Note that the update function f involves both the mutable variables and static data of node i and j. Here, we use an abstract accumulating operator $\oplus$ instead of a typical addition operator $+$ to cover a larger variety of algorithms. For example, in the shortest path algorithm, the accumulating operator $\oplus$ should be $\min$ so that the node with the shortest distance v_j is selected. Mathematically, we require the accumulating operator to be $\oplus$ commutative *i.e.,* $a \oplus b = b \oplus a$.

To make the iterative computation asynchronous, the computation should be able to be transformed into a form with independent sub-computations. The key idea is to decompose the f into two parts, and each part only involves the data of one node. So that each can be done independently on different workers. We can decompose it with g and h as shown in Equation (2).

$$f(v_j, v_i; D_j, D_i) = g(h(v_j, i; D_j), v_i; D_i) \tag{2}$$

The h function is invoked on source nodes. The result of $h(v_j, i; D_j)$ is a *message* from node j to node i. And the g function works on node i. $g(m, v_i; D_i)$ transforms the message $m = h(v_j, i; D_j)$ to the form ready for accumulation using the data of node i. Thus, we can perform $h(v_j, i; D_j)$ for different j independently on different nodes. Since the accumulating operator is commutative, we can asynchronously accumulate them on node i. So in an AIC system, when the message $m_{j,i}$ reaches node i, node i performs the following operations.

- Update mutable variable v_i by $v_i \leftarrow v_i \oplus g(m_{j,i})$
- Compute message $m_{i,k} = h(v_i; D_i)$
- Send message $m_{i,k}$ to node k if $m_{i,k} \neq \mathbf{0}$

Note that $\mathbf{0}$ is the identity element of $\oplus$, which is 0 for addition and $+\infty$ for minimization.

We can use this model to express the Maiter and NOMAD framework as follows. We use the PageRank algorithm as an example for Maiter. For PageRank, each graph vertex is a computation node. The mutable data of a node i is two variables p_i and u_i where p_i is the PageRank score and u_i is the delta term of the PageRank score. The static data of a node i contains its out-neighbor list $ON(i)$. Therefore, we have the following h and g functions.

- $h(\langle p_i, u_i \rangle, k; D_i) = \begin{cases} \dfrac{du_i}{|ON(i)|}, & k \in ON(i) \\ 0, & \text{otherwise} \end{cases}$
- $g(m, v_i; D_i) = m$
- $a \oplus a' = a + a'$

The h function sends $\frac{du_j}{|ON(j)|}$ to the out-neighbors of node j. While the g does no modification to the received message. And the messages are accumulated with addition. For the matrix completion example of NOMAD, the mutable data of a node i consists of two parts, H_i and a dummy W_*. The dummy W_* part is used to forward the W_j block from the input message to the output message. Correspondingly, the W_* part is not accumulated. It is replaced with a new W_j each time a new W_j is received. And the static data D_i is the i-th row of the matrix M.

- $h(\langle H_i, W_* \rangle, k; D_i) = \begin{cases} W_*, & k = i + 1 \\ 0, & \text{otherwise} \end{cases}$
- $g(W_j, \langle H_i, W_* \rangle; D_i) =$ $\langle -\eta W_j^T (H_i W_j - M_{i,j}), W_* - \eta H_i^T (H_i W_j - M_{i,j}) \rangle$
- $\langle h, w \rangle \oplus \langle h', w' \rangle = \langle h + h', w' \rangle$

Here, the message generation function h just forward the updated W_*. Upon the receiving of a message, the g function calculates the gradient of the local H_i variable and the message W_j.

2.2 AIC Distributed Frameworks

In this subsection, we show the framework supporting the AIC model in distributed environments. This distributed framework consists of a set of workers and a coordinator. Workers hold computation nodes and perform the computation task. The coordinator has an overview of worker progress and determines when to terminate the computation.

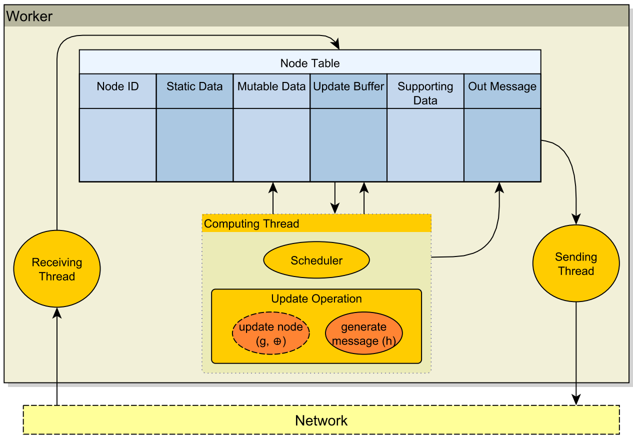


Fig. 1. Worker Model of an AIC system

2.2.1 Coordinator

A coordinator has a global view of the system and handles tasks like state monitoring, progress tracking, and termination control. A worker can hardly handle these tasks since they require information for all workers. Before a computation task starts, the coordinator checks whether all workers are available. The coordinator then broadcasts a starting signal to let all workers start the computation. The most important task of the coordinator is to keep track of the availability and progress of all workers. The coordinator periodically gathers information from all workers. Meanwhile, it also estimates the global progress of the whole computation. If the computation finishes, the coordinator needs to gather the result and terminate workers.

2.2.2 Worker

A worker is an entity used to hold the data of nodes and perform their corresponding computation. Nodes together with their static and mutable data are partitioned among workers. A typical partition method is to use a hash function on the node index. A more sophisticated method to partition is to assign nodes with tight connections to the same worker. So as to make most messages get transmitted within each worker.

Fig. 1 illustrates the basic structure of a worker. Nodes assigned to a worker are organized in a table. In addition to the static data and mutable data of a node, the table also buffers the messages sent to each node, referred to as *update buffer*, and the message generated by each node, referred to as *out-message buffer*. Each out-message buffer can be organized as a FIFO message queue. Together with them, the table also contains some supporting data to fulfill the system execution, like priority values, state flags, and some data caches for computation if necessary.

In a worker, three threads cooperate via the node table to perform the distributed asynchronous iterative computation as follows. A *receiving thread* receives messages from the network and put them into the update buffer of their corresponding nodes. Note that the messages are not processed yet. A *computing thread* keeps on picking nodes from the table and then updates their mutable data using the messages in their update buffers. When the mutable data is updated, the computing thread generates corresponding messages and puts them input the out-message buffer of that node. We will introduce the computing thread in detail later. A *sending thread* keeps on checking the out-message buffers and sends found messages to their destination worker via the network.

The computing thread consists of a scheduler and an update operator. The scheduler selects nodes to be updated. The scheduling algorithm can be a round-robin selection or a prioritized scheduling with a given priority generation method. For example, Maiter uses the absolute value of Δv_i as the priority value of node i, so as to perform larger changes earlier. The update operator of the computing thread consists of two parts, update the mutable data and generate new messages, shown with the orange cycles in Fig. 1. For a selected node, the updating step invokes the g function to preprocess the buffered messages and use them to update the mutable data via the $\oplus$ operator. Then the computing thread invokes the h function to generate new messages. When a message is generated, the computing thread checks which worker contains the destination node of that message, using the partition hash function. If the destination worker is the current worker, the computing thread directly puts the new message into the destination node's update buffer. If the destination is another worker, the computing thread pushes the new message into the out-message buffer of the source node.

In many algorithms, messages in the update buffer and the out-message buffer can be aggregated. For example, messages in the Pagerank algorithm are delta values of the Pagerank scores. Adding two delta values consecutively onto a Pagerank score results in the same score as adding the summation of the two delta values on the original score. So the receiving thread can just maintain one copy of received delta value for each node in the update buffer instead of keeping everyone. And once it receives a new message, it accumulates the message with the one in the update buffer using $+$ operation. Similarly, the computing thread can also accumulate out-going messages in the out-message buffer before the sending thread sends them out.

The three threads within a worker also work asynchronously. The receiving thread puts received messages into update buffers. The computing thread is not triggered by the receiving thread. It asynchronously picks and updates nodes with non-empty update buffers. Similarly, the computing thread and the sending thread work asynchronously via the out-message buffer.

3 Fault-Tolerant Framework for AIC (FAIC)

In this section, we introduce our fault-tolerance mechanism for a distributed asynchronous iterative computation systems. We provide an overview of the system first. Then, we elaborate on how it works in a distributed AIC framework and how it recovers a system when a failure happens.

3.1 Overview

The key idea of our mechanism is to lead the AIC system into a state where there is no inflight message. Then, we can

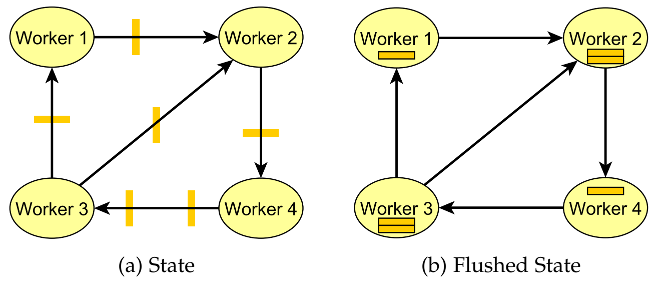


Fig. 2. Snapshot Example

take a snapshot of that state as a checkpoint. Fig. 2a shows the snapshot of an AIC system, which is the state of the whole system at some time point. There are usually some messages being transmitted in the network. We want to flush the network so that there is no inflight message across workers in the system as Fig. 2b. Then, workers capture their local states asynchronously among different workers. We call the local states captured from all workers collectively as a *flushed snapshot* of the system.

We can capture the local state of each worker in a flushed snapshot asynchronously. In a distributed system, the time points when each worker gets all messages flushed to it can be quite different. So we capture the local state of a worker at the time when the worker receives the last inflight messages sent to it. Assuming workers of Fig. 2b receive their last messages in the order of Worker-1, Worker-2, Worker-3, Worker-4, we capture the local state of each worker as Fig. 3. We denote the time points when workers receive their last inflight messages as $T1$, $T2$, $T3$, and $T4$.

Since the computations on workers keep change their local state, we denote the local state of a worker i at time point T as $S_i(T)$. Obviously, $S_1(T1) \neq S_1(T2)$. So what we capture is $\langle S_1(T1), S_2(T2), S_3(T3), S_4(T4)\rangle$. The captured data seems to be a snapshot but it does not exist at any time point, so we call this "snapshot" as a *virtual snapshot*.

During the process of taking a virtual snapshot, the computing thread of each worker continues. Computing threads can continue processing as long as there are data messages in the update buffer. There are two sources to the update buffer. First, the receiving thread buffers some inflight data messages in the local update buffers. Second, as mentioned in Section 2.2.2, when the destination node of a locally-generated message is local, the message is directly put into the corresponding update buffer. Note that, it is not guaranteed that the computing always continues. If the buffered data messages are used up and destination nodes of all newly generated messages are held by other workers, the computing thread of a worker stops. But that scenario does not happen usually.

A virtual snapshot can be used to recover an AIC system. We give a formal proof in later sections. The general idea is to show that by controlling the working pace of each thread, it is possible to construct a snapshot which is identical to the virtual snapshot. Intuitively, halting a thread in an AIC system can be viewed as a special case of asynchronous execution where zero working time is scheduled to that thread. Therefore, we can construct a snapshot by halting the computing thread of a worker at the moment when the local state is captured. As a result, the local state of that worker remains unchanged since that moment. When the computing threads of all workers get halted, the current snapshot is exactly the virtual snapshot we captured.

3.2 Interactions among Coordinator and Workers

Fig. 4 shows the interaction among workers and the coordinator for taking a virtual snapshot. Our method can be viewed as three steps. The first step is a flushing step. It starts when the coordinator broadcasts a flush message to all workers, as illustrated with "1-flush" in the figure. When a worker receives the flush message, it pauses its further sending operation to flush its out-going inflight data messages. Once the flushing is done, a worker notifies others by broadcasting a clear message to every worker including itself as denoted with "2-clear". This step is performed by each worker individually because channels between workers are independent. The second step is to archive the local states of workers. A worker is not going to receive any more messages once all inflight messages to it are flushed out. Then, the worker archives its local state including the mutable data, update buffer, and out-message buffer of all its local nodes. The last step is to resume the whole system back to the normal computation mode. After archiving local states, a worker notifies the coordinator about its progress via a finish message, as shown with "3-finish" from a worker to the coordination in the figure. After all workers finish their archiving operations, the coordinator broadcasts a resume message, shown with "4-resume" in the figure, to all workers to make them resume the network communication.

3.3 Interactions among Threads within a Worker

Each worker experiences three phases corresponding to the three steps while taking a virtual snapshot. Now we introduce how does the receiving thread, computing thread and sending thread of a worker interact.

3.3.1 *Flushing Phase*

When a worker receives the flush message from the coordinator, it moves into the flushing phase. The worker broadcasts a *clear message* to every worker, including itself, and halts its sending thread. More precisely, the receiving thread sends an *internal halting signal* to the sending thread, as shown with the dashed line denoted with "1-halt" in Fig. 5. When the sending thread receives this internal halting signal, it stops its normal sending task and immediately broadcasts the clear message. Therefore, the clear message is the last message that another worker is able to receive from this worker. Meanwhile, the receiving thread keeps receiving messages and the computing thread keeps updating local nodes. The generated messages are temporarily buffered in the out-message buffers.

A worker finishes its flushing phase as follows. The flushing phase of a worker finishes when the worker receives clear messages from all workers. When a worker receives a clear message from another worker, there is no inflight message between these two workers. Because a worker only sends one clear message to another worker,

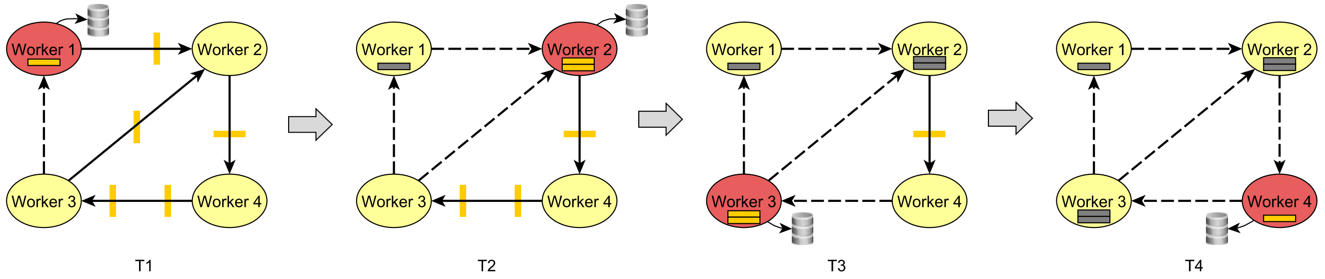


Fig. 3. Virtual Snapshot Example $\langle S_1(T1), S_2(T2), S_3(T3), S_4(T4)\rangle$

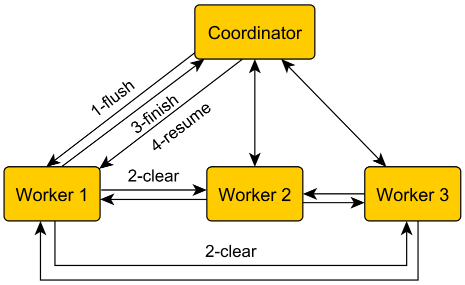


Fig. 4. Interactions among Coordinator and Workers

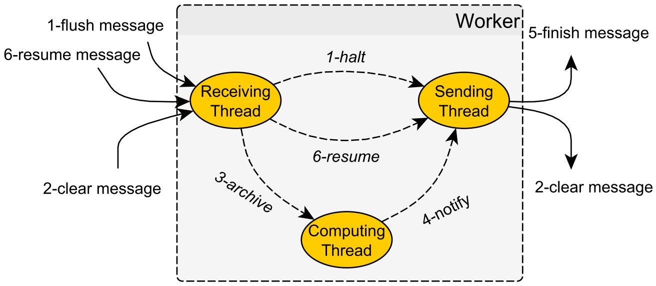


Fig. 5. Interaction among Receiving, Computing and Sending Thread of a Worker

the flushing to a worker finishes when it receives n clear messages where n is the number of total workers. So we set up a counter for the number of received clear messages. The counter starts with 0. Note that, due to the network delay, a worker may receive a clear message before it receives the flush message. So the initializing and resetting of the counter is performed at the resuming phase instead of the beginning of the flushing phase. Whenever the receiving thread receives a clear message from another worker, it increases the counter. When the value of the counter reaches n, the worker finishes its flushing phase and moves to the archiving phase.

3.3.2 *Archiving Phase*

In the archiving phase, a worker archives its local state and notifies the coordinator. When the counter reaches n, the receiving thread sends an internal archiving signal to the computing thread, as shown with the dashed line denoted with “2-archive” in Fig. 5. We use the computing thread to handle the archive task to prevent the update operations from changing values during archiving them. When the computing thread receives the archiving signal, it inserts an archive operation after the current update operation. The archive operation archive states of all local nodes including the mutable data, update buffer and out-message buffer.

After the archiving operation is done, the computing thread sends an internal finish signal to the sending thread, as shown with the dashed line denoted with “3-finish” in Fig. 5. When the sending thread receives that internal signal, it sends a *finish message* to the coordinator to notify the coordinator about this local finish. Thus, a worker finishes its archiving phase and moves to the resuming phase.

Note that some messages are aggregated before gets archived in AIC systems, like Maiter, whose messages are accumulative. The computing thread archives messages after all inflight messages sent to this worker are received. During this period, messages keep being accumulated in the update buffer and out-message buffer. Our method archives the accumulated messages. In AIC systems whose messages are accumulative, several messages are aggregated into one.

3.3.3 *Resuming Phase*

After all workers finish their own archive tasks, the system goes back to the normal working mode by resuming all sending threads. The coordinator monitors the checkpointing progress by counting the received finish messages. When the coordinator receives n finish messages, it broadcasts a *resume message* to all workers. As illustrated in Fig. 5, when a worker receives the resume message, the receiving thread resets the counter about clear messages and sends an internal resume message to the sending thread. When the sending thread gets that internal signal, it resumes its normal sending functionality of the data messages. Then a checkpoint procedure finishes.

3.4 Recovery from a Checkpoint

When a failure happens, the system is restored back to the latest checkpoint. The restoring task is done by just loading the archived local state of each worker. Each worker loads its own part of the archived local state from the checkpoint including the mutable data, update buffers, and out-message buffer into its node table. Then, the worker starts its three threads.

4 Correctness Proof of FAIC

In this section, we prove that the virtual snapshot captured by FAIC can be used to correctly recover an AIC system.

First, we formally describe how an AIC system works and the relationship between actions and snapshots. Then, we describe the actions while taking a snapshot. After that, we show that we can always construct an action sequence to reach a snapshot that contains identical data to the virtual snapshot captured by FAIC.

4.1 Formal Model for AIC

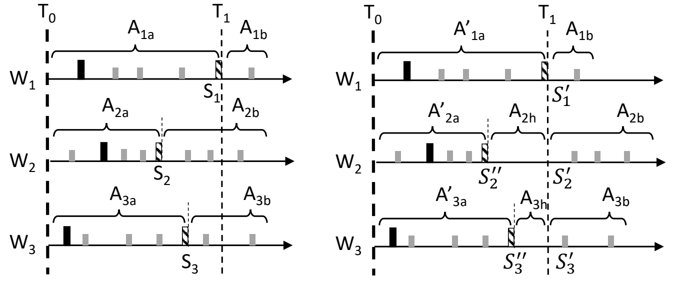
We can use a sequence of actions to represent how a worker executes its computation. To simplify the discussion, we can treat time as a sequence of small time slots and a worker only takes one action in one time slot. An action may take several time slots. We assume that an action changes the state of the system at the moment when it finishes. In addition to the receiving, computing and sending action introduced in previous sections, a worker may do nothing during one time slot. We use a halting action to represent such a case. So that the execution of a worker can be expressed with an *action sequence* consists of such 4 types of actions. It can be understood as how computing resources are assigned to the receiving, computing, and sending thread. If a worker takes a receiving action while there is nothing on the wire, this action has no effect on the state of the worker. Similarly, the computing action and sending action do nothing if there is no message to be processed or sent. We refer to there actions that do nothing including halting actions as *invalid actions* and the rests are *valid actions*.

We can use the action sequences of all workers to express the execution of the AIC system. We refer to these action sequences as an *action sequence group*. One concrete execution of the AIC system maps to one specific action sequence group. Note that given an initial snapshot and an action sequences group, we can get a sequence of snapshots of the AIC system at the end of each time slot.

The state of an AIC system at a certain time point is expressed as a snapshot and actions update the snapshot. A snapshot consists of the *local state* I_i, V_i, O_i of each worker W_i, and the inflight message set M, where I_i, V_i, O_i are the update buffer, the mutable date and the out-message buffer of worker W_i respectively. The inflight message set M consists of messages sent to each worker M_i. Therefore, a snapshot can be expressed with a list of *augmented local states* $\langle S_i \rangle_n = \langle I_i, V_i, O_i, M_i \rangle_n$. A receiving action on worker W_i updates M_i and I_i by picking one message from M_i and put it into I_i. A computing action on worker W_i changes I_i, V_i, and O_i. It picks a message from I_i and uses it to update V_i using the f function meanwhile it may also generate some new messages into O_i. A sending action on worker W_i picks a message from O_i and put it into M_k where k is the destination worker of the message. Except for the halting action, each action moves a snapshot S to another snapshot S'. Since an action changes the state of the system when it finishes, a snapshot in the middle of an action is the same as the one when this action starts.

Start from an initial snapshot, an action sequence group moves the system to another snapshot. A *potential snapshot* is a snapshot that can be derived by an action sequence group from the initial snapshot.

We define a *recoverable state* of an AIC system as a state starting from which the AIC system can continue computing. Apparently, a snapshot is a recoverable state because it contains all data of an AIC system. Because a potential snapshot is a snapshot, we have Lemma 1.



(a) Virtual Snapshot $\langle S_1, S_2, S_3 \rangle$ (b) Constructed Snapshot $\langle S_1', S_2', S_3' \rangle$

Fig. 6. Constructing an Equivalent Snapshot from a Virtual Snapshot. (Black bars are the actions that start local checkpointing procedures. There is no valid sending action after a black bar of a worker. Dashed bars denote the receiving actions of each worker's last clear messages. There is no further valid receiving action after it on a worker.)

Lemma 1. *A potential snapshot is a recoverable state for the AIC system.*

4.2 Correctness of FAIC

We first review the procedure of taking a virtual snapshot using the action sequence concept. Fig. 6a shows the procedure using a three-worker example. We use bars to represent valid actions of each worker. When a worker receives the flush message from the coordinator, denoted with a black bar, it broadcasts a clear message to other workers and halts the sending thread. There is no valid sending action on a worker after the black bar. Note that, as discussed in Section 3.3.1, the action of the black bar can either be the receiving action of the flush message from the coordinator or the receiving action of the first clear message received from a worker. On the receiving action of the last clear messages, denoted with a dashed bar, a worker W_i archives its current local state S_i and continues to perform computing actions. Note that, the receiving of the last clear messages means there is no more inflight message sent to this worker. From the perspective of the action sequence, there is no more valid receiving action after a gray bar of a worker. After all workers finish the archiving, they resume the sending threads. From then on, there are valid sending and receiving actions again.

We are going to prove that a virtual snapshot captured by FAIC is a potential snapshot. FAIC captures the local state of each worker at different time points and constructs them as a virtual snapshot which does not show up at any time slot of the execution. In Lemma 2, we are going to show that there is a potential snapshot that captures the same local states as the virtual snapshot. The general idea of proving this lemma is to construct an action sequence group which leads the AIC system to such a potential snapshot.

Lemma 2. *A virtual snapshot captures in FAIC is a potential snapshot of the AIC system.*

Proof. The statement is true as long as for any virtual snapshot we can find an action sequence group that leads the system to a potential snapshot and the content of this potential snapshot is identical to the virtual snapshot. We prove it by giving a method of constructing such an action sequence group for any virtual snapshot.

We construct the action sequence group for the potential snapshot by modifying the action sequence group in the execution of taking the virtual snapshot. We cut the action sequence of a worker W_i into two parts at the receiving action of the last clear message, *i.e.,* the gray bars in Fig. 6a, as A_{ia} and A_{ib}. We denote the moment when the last worker takes its local snapshot at T_1. For each worker, we construct its new action sequence in two steps. First, we copy the A_{ia} sequence *i.e.,* the actions until the receiving last clear message, to the constructed action sequence. Second, we pad an action sequence A_{ih} which only contains halting actions to each worker to make the action sequence of every worker reaches time T_1. Then, at time T_1, we get a snapshot $\langle S'_1, S'_2, S'_3 \rangle$ from the constructed action sequence group.

Now, we are going to prove that the potential snapshot $\langle S'_1, S'_2, S'_3 \rangle$ of constructed action sequences group is identical to the virtual snapshot $\langle S_1, S_2, S_3 \rangle$. We prove it in two steps. We will show that the state S'_i at T_1 is the same as the state S''_i at the gray bar for each worker in the constructed execution. Then, we will prove that for each worker the state S''_i at the gray bar of the constructed execution is the same as the state S_i at the gray bar of the original execution.

First, we will prove that in the constructed execution the augmented local state S'_i at T_1 is the same as the one at the gray bar S''_i. Since no worker sends messages to worker W_i after black bars, there is no valid sending actions on any worker after the gray bar of worker W_i. In addition, all inflight messages to worker W_i are received at the gray bar. Therefore, the inflight message part M_i of S''_i keeps empty during the period of A_{ih}. And A_{ih} only contains halting actions that do not change anything. So that, for each worker the state S''_i at the gray bar keeps the same until the state S'_i at T_1.

Second, we will prove the augmented local state S''_i at the gray bar of the constructed execution is the same as the augmented local state S_i archived in the virtual snapshot. Apparently, for the first worker that archives its local state $S''_i = S_i$, which is the W_2 in Fig. 6. Since the halting sequence A_{2h} does not send anything to any worker, the inflight message part of the second worker archiving its local state, which is W_3 in the example, is updated by the same action sequences as the original execution used to take the virtual snapshot. In addition, the constructed action sequence until the gray bar of W_3 is the same as that in the original action sequence group. So does the action sequence of other workers until that moment, which is W_1 in the example, especially the sending actions to W_3. Therefore, all actions which may update the local state of W_3 in the constructed action sequence group is the same as those in the original action sequence group. As a result, the augmented local state S''_3 in the constructed snapshot equals the state S_3 archived in the virtual snapshot. Similarly, the actions which update the augmented local state of a worker in the constructed action sequence group are exactly the same as those in the original action sequence group. Therefore, for all workers, we have $S''_i = S_i$. □

Theorem 3. *The virtual snapshot captured in FAIC is able to recover the AIC system correctly.*

Proof. By putting Lemma 1 and Lemma 2 together, we know that the virtual snapshot captured in FAIC is a recoverable state. Therefore, we get Theorem 3. □

5 Evaluation

In this section, we evaluate the empirical performance of our fault-tolerant framework with two adoptions of existing iterative asynchronous computation frameworks. First, we introduce the setup of our experiments in Section 5.1. Then, we evaluate the overhead of making checkpoints and compare it with some typical methods in Section 5.3. In Section 5.4, we test the overall running time of our FAIC framework when there are some failures. At the end, we test the scalability of FAIC in Section 5.5.

5.1 Experimental Setup

We implement our FAIC framework for two existing AIC frameworks, Maiter [9] and NOMAD [11]. We add the fault-tolerance related code on top of their implementation and make our code public accessible on github [1] [2].

We use Maiter to test how FAIC performs when message aggregation before archiving is available. Maiter is a delta-base accumulative iterative computation model. Its messages are accumulative. We run the PageRank algorithm on top of power-law graphs. We synthesized test graphs by setting the distribution parameter $\alpha = 2.3$, to simulate the topology of an online social network. The size of the graphs ranges from 100 thousand nodes to 5 million nodes. We apply graphs with 1 million nodes by default.

We use NOMAD to test how FAIC performs when messages are not accumulative. We run matrix completion algorithm using synthesized matrices of size 10,000 $\times$ 200 by default. To make NOMAD works with FAIC, we make the following changes to it. First, we add a virtual computing thread for each worker. The NOMAD model employs multiple computing threads in each worker while they share the same receiving and sending thread. The virtual computing thread holds all original computing threads and takes over their interaction with the receiving and sending thread. Second, we add a error-based termination mechanism to measure the time of reaching identical precision level. The coordinator keeps checking whether the training error becomes smaller than a predefined threshold and terminates workers if it happens. Since computing the error is a part of the training procedure, this additional checking does not bring noticeable performance impact.

By default, we perform our experiments using 6 workers of a local cluster equipped with Xeon(R) E5-2620 CPU and 32 GB memory. In the experiments of a heterogeneous cluster, we replace 2 workers with Xeon E5607 CPU, which is about 40% slower than the other workers. For the scalability experiments, we set up a cluster utilizing Amazon Web Services (AWS) EC2 platform. We set up a cluster with 100 t3a.small instances, which has 2 vCPU and 2 GB memory.

[1] Maiter: https://github.com/yxtj/maiter/tree/checkpoint2
[2] NOMAD: https://github.com/yxtj/nomad

By default, we set up the checkpoint interval as 30 seconds. We choose this value to emulate the frequent server preemption in transient resources offered in clouds. To reduce the progress-loss after a worker-loss, it is necessary to make checkpoints frequently. We evaluate how checkpoint intervals affects the total running time in Section 5.4.

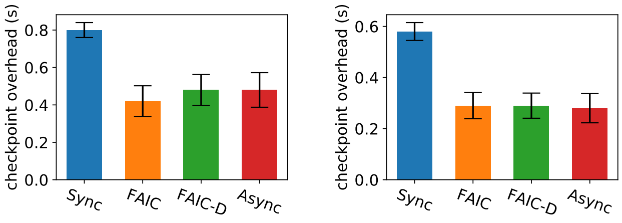
5.2 Checkpoint Methods to Compare with

We compare our virtual-snapshot-based FAIC checkpoint method with two checkpoint methods, a synchronous method and an asynchronous method. The synchronous checkpoint method, referred to as *Sync*, pauses the computation and communication of all workers to flush inflight messages. When a worker receives a starting message from the coordinator, it pauses its computing and sending threads. After the receiving thread finds that there is no more inflight message sent to that worker, it notifies the coordinator. When the coordinator is notified by all workers, it broadcasts another special message to make all workers to take a snapshot and then resume the computing and sending threads.

We also compare our FAIC framework with an asynchronous checkpoint method, referred to as *Async*. Async adopts the idea of the Chandy-Lamport snapshot algorithm [10] to capture states of workers and messages between workers. Instead of taking a snapshot, Async makes a checkpoint with a process, during which the inflight messages are captured one by one. Meanwhile, neither the computing or the sending and receiving are halted. Async starts when the coordinator broadcasts a starting message to all workers. When a worker receives the starting messages, it immediately archives its local mutable data and broadcasts a special token to other workers. Meanwhile, the worker starts archiving messages. This token can be views as the flush message. It means all messages received before this token are part of the snapshot. A worker archives every data message from worker W_i until before it receives the token from worker W_i. Async finishes when all workers receive all tokens from other workers. In comparison, FAIC archives the local mutable data and message buffers together. And FAIC archives data at the end of the checkpointing procedure on each worker instead of the beginning.

5.3 Checkpoint Overheads

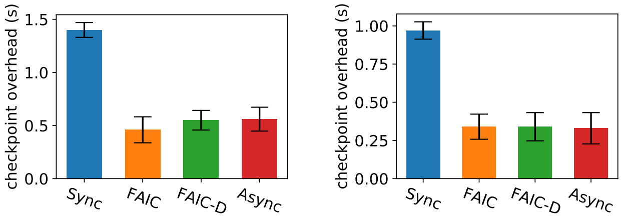
We measure the overhead caused by making checkpoints to demonstrate the impact of our fault-tolerant framework onto the normal computation. The overhead of a checkpoint cannot be directly measured by simply summing the archiving time and flushing time. For example, in FAIC, the communication is halted during the checkpointing, which may slow down the global computation. Therefore, we measure the *overhead* as the running time difference as follows. By setting identical data set, termination condition and checkpoint interval, we measure the running time of Sync, FAIC, Async and also the case without checkpoints. Thus, we get the total overhead of a checkpoint method. Since there are multiple checkpoints in each run, we compute the *overhead* of each checkpoint by dividing the total overhead with the number of checkpoints in that experiment.



(a) Maiter

(b) NOMAD

Fig. 7. Comparing Checkpoint Overhead (Homogeneous Cluster)



(a) Maiter

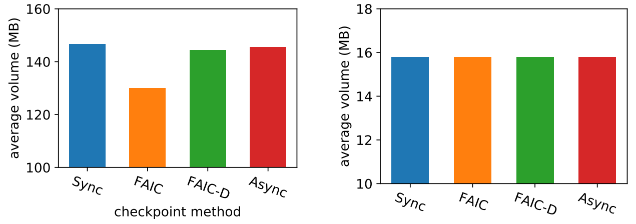
(b) NOMAD

Fig. 8. Comparing Checkpoint Overhead (Heterogeneous Cluster)

We compare the overhead of Sync, FAIC and Async on both Maiter and NOMAD. In order to demonstrate the average performance, we run each method multiple times. We also set up different checkpoint intervals for each run. So that the system captures snapshots at different progress levels in each run. We show the averaged overhead of a checkpoint in Fig. 7 and Fig. 8. The standard derivation of the overhead is shown with an error bar. To show the influence of the message aggregation technique, we also include a FAIC variation without aggregation denoted with "FAIC-D" in the figures. Fig. 7 shows the overhead comparison under a homogeneous cluster. On both Maiter and NOMAD, FAIC reduces about half of the overhead of Sync and performs roughly the same as Async. For specifically, Fig. 7a shows that the overhead of FAIC on Maiter is about 51% of Sync and 90% of Async. And Fig. 7b shows the overhead of FAIC on NOMAD is about 54% of Sync and 103% of Async.

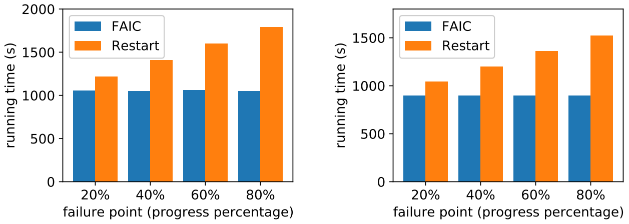
We also compare the overhead in a heterogeneous environment. Two workers in the heterogeneous cluster is about 40% slower than others. Shown in Fig. 8, overhead of FAIC is only about 31% of Sync. It because that Sync pauses the computation until every worker finishes its local tasks while FAIC and Async keeps running the computation. The overhead of Sync is about 40% longer than that in the homogeneous cluster while FAIC and Async only slow down about 8%-15% than the homogeneous case. This ratio is consistent with the global slow-down ratio which is $2/6 \times 40\% \approx 13.3\%$.

Fig. 7a and 8a show that the overhead of FAIC is obviously smaller than Async on Maiter. FAIC pauses the cross worker communication during making a checkpoint, while Async does not pause any computation or communication. So intuitively the overhead of FAIC should not be smaller than Async. But FAIC still performs local computation and



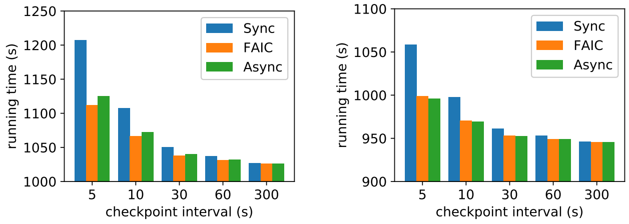
(a) Maiter (b) NOMAD

Fig. 9. Comparing Volume of Archived Message for each Checkpoint



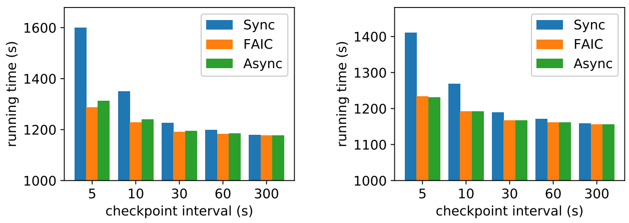
(a) Maiter (b) NOMAD

Fig. 10. Running Time with one Failure



(a) Maiter (b) NOMAD

Fig. 11. Running Time of Checkpoint-based Methods with one Failure (Homogeneous Cluster)



(a) Maiter (b) NOMAD

Fig. 12. Running Time of Checkpoint-based Methods with one Failure (Heterogeneous Cluster)

usually a checkpointing period is not long enough to make a worker finish all local computation tasks. In addition, a local computation task may also generate messages to local data nodes and trigger new local computation.

FAIC outperforms Async on Maiter because FAIC archives less amount of messages. As introduced in Section 2.2.2, on systems like Maiter, multiple messages targeted at the same data node can be aggregated into one. In comparison, Async must capture each received message individually. It is because that Async archives the local state, including the mutable data and update buffers, at the beginning of a checkpoint procedure. If we archive aggregated messages later, the beginning state of update buffer is actually captured twice. But FAIC archives aggregated local states after all inflight messages got flushed. This state contains the beginning state and all inflight messages. So FAIC does not capture anything twice. Because of the message aggregation, there is usually less amount of message to be archived in FAIC than Async. To verify it, we disable the aggregation function of FAIC and measure its average message volume in each checkpoint. Fig. 9 compares the average message volume of Sync, FAIC with aggregation, FAIC without aggregation (denoted with FAIC-D), and Async. We can see that the average message volume of Sync, Async and FAIC-D are roughly the same while FAIC is about 12% smaller than others on Maiter. On systems like NOMAD where messages cannot be accumulated, as shown in Fig. 9b, FAIC captures the same amount of messages as others. Therefore, the overhead of FAIC and Async are roughly the same.

5.4 Running Time with Failures

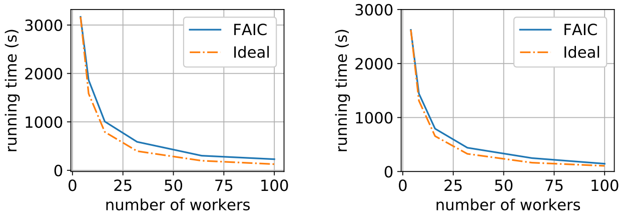
We demonstrate the efficiency of FAIC's failure recovery in Fig. 10. We artificially insert a failure at a certain moment and recover the system. We measure the total running time of FAIC and compare it with a baseline method, denoted with *Restart*. The baseline method does not make checkpoints during computation and restarts the computation from stretch after a failure. Fig. 10 compares the total running time of FAIC and Restart with a failure at some different computation progress points. We can see that the total running time of FAIC is roughly the same no matter when the failure happens. But the restart suffers a linear increase in the time. When a failure happens at 20% of progress, FAIC is about 12% faster than Restart. When a failure happens at 80% of progress, FAIC saves about 70% of computation time.

FAIC and the other two checkpoint-based methods lose the progress between the failure and the last checkpoint. Making checkpoints takes about 8% of the computation time. Therefore, ideally, when failure happens at 80% of progress, FAIC should be 72% faster than Restart. FAIC loses an additional 2% of running time, *i.e.*, about 22 seconds, because the progress between the latest checkpoint and the failure loses. It also explains why FAIC is not always better than Async on Maiter. So making checkpoints more frequently is more likely to recover more computation progress after a failure happens. But since taking checkpoint cost some time itself, we should not make checkpoints too frequently. We need to find the best trade-off between the checkpointing cost and the recovery efficiency depending on the failure frequency.

We also compare the total running time of FAIC with other checkpoint-based methods. Fig. 11 and Fig. 12 show the running of Sync, FAIC and Async with a failure at 30% of the progress. Fig. 11 shows the running time under a homogeneous cluster and Fig. 12 shows that under a heterogeneous cluster. In both experiments, Sync is the slowest. FAIC outperforms Async about 2% in Maiter and performs roughly the same as Async on NOMAD. The total time depends on the checkpoint frequency. As shown in Fig. 11, in a homogeneous cluster, the total running time of FAIC is about 2% and 0.4% shorter than Sync and Async respectively on Maiter with the checkpoint frequency of 10 seconds. In a heterogeneous cluster where the overhead of each checkpoint is larger, as shown with Fig. 12, FAIC outperforms Sync and Async about 8% and 0.4% respectively under the same condition. When larger checkpoint intervals are adopted, fewer checkpoints are taken and the difference between different methods decreases.

5.5 Scalability

In this subsection, we test the scalability of FAIC.We use a power-law graph containing 5 million nodes and roughly 20 million edges for Maiter and a 50,000 × 200 matrix for NOMAD. We compare the running time of reaching the identical error level. Shown in Fig. 13, the running time decreases with the increase in the number of workers. We use the running time of the 4-worker case as a reference point and compute the ideal scaling curve with it. They are shown with dash lines. As we can see, up to 100 workers, the running time of FAIC sticks to the ideal case.



(a) Maiter

(b) NOMAD

Fig. 13. Scalability of FAIC

6 Related Work

Many people use synchronous barriers to make checkpoints in distributed iterative computations. Some models [3], [4], [8], [13], [14], [15] use the Bulk Synchronous Parallel (BSP) [5] model where the synchronous barriers naturally exist. Some other asynchronous systems [9], [16] set up additional barriers to force the whole system to reach a global consistent point.

Zaharia et al. proposed a linkage based recovery mechanism for the synchronous distributed model called Resilient Distributed Dataset (RDD) [17]. Each piece of data maintains a sequence of its previous modifications. If a worker is down, the data on it is re-acquired by redoing several latest modifications since the last reliable point. But RDD requires the data modifications to be synchronous, so that there is a global consistent point at some moment. In addition, when a modification involves some cross-worker cooperation, the RDD model downgrades to a global rollback-redo model.

There are also some prior works providing asynchronous checkpoint to specific AIC systems. GraphLab [12] and PowerGraph [18] are asynchronous iterative computation models focusing on graph processing. They implemented a node-wise checkpointing method based on the typical Chandy-Lamport algorithm [10]. Instead of working on a fine-grained graph node level, our framework works on the level of workers which can handle various tasks in various granularity. FAIC provides fault-tolerance for any asynchronous iterative framework which can be expressed with the message-passing model. In addition, our mechanism of signal-and-handler provides additional flexibility to designers who want to customize the checkpointing behaviors like adopting a customized message compressing function or adding a statistics function.

Chandy-Lamport snapshot algorithm [10] is a classical idea of finding a global consistent snapshot. It employs a delicate protocol to capture the states of communication channels which are the inflight messages in AIC systems. In our FAIC framework, we try to find a snapshot where there is no inflight message. In addition, we propose a method to compose such a snapshot in AIC systems without forcing it to appear.

Different from checkpoint based recovery, some works aim to provide algorithm-based recovery. By using correct algorithmic compensations, some works [19], [20], can reach a consistent state even after failures. However, defining the compensation function is non-trivial and such functions only exist for specific algorithms. Recently, some researchers proposed Zorro [21] which exploits vertex replication to quickly rebuild the state of failed servers for graph processing. It reduces the overhead during failure-free execution to zero but sacrifices the accuracy of the result. Similarly, some other works [22], [23] reduce the overhead during failure-free executions for iterative solvers. For certain iterative approximating algorithms like PageRank, some researchers proposed a fault-tolerant algorithm without a checkpoint or a compensation function [24].

In the high performance computing community, some researchers proposed "partial snapshots" for complex tasks. These works [25], [26] identify the independent task branches int the data flow graph of a complex task. Then, they can take partial snapshots on each branch independently. In our work, we focus on AIC systems whose data flow is usually a set of contact cycles. Each cycle is the data flow of one data point. And different cycles run at different speeds.

7 Conclusion

We propose a distributed fault-tolerant framework for asynchronous iterative computations. In an AIC system, messages across workers are hard to capture and are crucial for the computation and recovery. Our key idea is to archive a state of the AIC system where there is no inflight message. We propose a FAIC framework which leads each worker into such a state without halting the computation of any worker. These local states are captured on each worker at different time and they form a virtual snapshot. We prove that the virtual snapshot can recover the system correctly. After that we evaluate the FAIC framework with two existing AIC system, Maiter and NOMAD. Our evaluation results shows the overhead of FAIC is about 50% smaller than the synchronous snapshot method, like BSP, and roughly the same as the asynchronous snapshot method, like the

Chandy-Lamport snapshot algorithm. For AIC systems like Maiter whose messages are accumulative, FAIC is about 10% faster than the Chandy-Lamport snapshot algorithm. Our experiments on large cluster also shows that FAIC scales with the number of workers.

Acknowledgments

This work was supported by National Science Foundation (grant number: CNS-1815412, CNS-1908536).

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[3] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[4] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," *Journal of Grid Computing*, vol. 10, no. 1, pp. 47–68, 2012.

[5] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[6] R. Singh, P. Sharma, D. Irwin, P. Shenoy, and K. Ramakrishnan, "Here today, gone tomorrow: Exploiting transient servers in datacenters," *IEEE Internet Computing*, vol. 18, no. 4, pp. 22–29, 2014.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 2010, pp. 1–10.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[9] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 8, pp. 2091–2100, 2014.

[10] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[11] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. Dhillon, "Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 975–986, 2014.

[12] 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," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[13] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[14] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables." in *OSDI*, vol. 10, 2010, pp. 1–14.

[15] C. Xu, M. Holzemer, M. Kaul, and V. Markl, "Efficient fault-tolerance for iterative graph processing on distributed dataflow systems," in *Proc. of ICDE*. IEEE, 2016.

[16] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: a distributed framework for prioritized iterative computations," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 13.

[17] 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 *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 17–30.

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

[20] S. Dudoladov, C. Xu, S. Schelter, A. Katsifodimos, S. Ewen, K. Tzoumas, and V. Markl, "Optimistic recovery for iterative dataflows in action," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1439–1443.

[21] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell, "Zorro: Zero-cost reactive failure recovery in distributed graph processing," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2015, pp. 195–208.

[22] L. Jaulmes, M. Casas, M. Moretó, E. Ayguadé, J. Labarta, and M. Valero, "Exploiting asynchrony from exact forward recovery for due in iterative solvers," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[23] L. Jaulmes, M. Moreto, E. Ayguade, J. Labarta, M. Valero, and M. Casas, "Asynchronous and exact forward recovery for detected errors in iterative solvers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 1961–1974, 2018.

[24] Z. Wang, L. Gao, Y. Gu, Y. Bao, and G. Yu, "A fault-tolerant framework for asynchronous iterative computations in cloud environments," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 71–83.

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

[26] T. Martsinkevich, O. Subasi, O. Unsal, F. Cappello, and J. Labarta, "Fault-tolerant protocol for hybrid task-parallel message-passing applications," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 563–570.

Tian Zhou received the B.E. degree in automation from Xi'an Jiaotong University, Xi'an, China, in 2012. He is currently pursuing the Ph.D. degree with the School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an and the Department of Electrical and Computer Engineering, University of Massachusetts at Amherst. His current research interests include big graph mining, data mining and distributed systems.

Lixin Gao (Fellow, IEEE) received the PhD degree in computer science from the University of Massachusetts at Amherst in 1996. She is a professor of electrical and computer engineering with the University of Massachusetts at Amherst. Her research interests include social networks, and Internet routing, network virtualization, and cloud computing. She is a fellow of the ACM and the IEEE.

Xiaohong Guan (Fellow, IEEE) received the B.S. and M.S. degrees in control engineering from Tsinghua University, Beijing, China, in 1982 and 1985, respectively, and the Ph.D. degree in electrical engineering from the University of Connecticut, Storrs, CT, USA, in 1993.

He was a Senior Consulting Engineer with PG&E from 1993 to 1995. He was with the Division of Engineering and Applied Science, Harvard University, from 1999 to 2000. Since 1995, he has been with the Systems Engineering Institute, Xi'an Jiaotong University, Xi'an, China, and has been the Cheung Kong Professor of Systems Engineering since 1999, the Dean of the Faculty of Electronic and Information Engineering since 2008. His research interests include allocation and scheduling of complex networked resources, network security, and sensor networks.