Exploiting Unblocking Checkpoint for Fault-Tolerance in Pregel-Like Systems

Yi Yang[1], Zhenhua Yang[1], and Chen Xu[1,2](✉)

[1] East China Normal University, Shanghai, China
{yiyang,zhyang}@stu.ecnu.edu.cn, cxu@dase.ecnu.edu.cn
[2] Science and Technology on Parallel and Distributed Processing Laboratory (PDL), Changsha, China

Abstract. With the explosive growth of graph size, a series of Pregel-like systems have emerged. Typically, these systems employ checkpointing and rollback mechanisms to achieve fault-tolerance in either blocking or unblocking manner. The blocking checkpointing pauses the iterative processing while checkpointing, whereas the unblocking checkpointing writes the checkpoints in parallel with the iterative processing. The unblocking checkpointing decreases the checkpointing overhead, but incurs resource contention due to checkpointing concurrently. Hence, it may prolong the time on execution and checkpointing. In this work, we propose a *queuing strategy* to alleviate the contention. This strategy employs a checkpoint queue to store all the pending checkpoints, which allows to concurrently write a certain number of checkpoints at most from the queue following a First-In-First-Out (FIFO) policy. To further utilize the characteristics of checkpoint in Pregel-like systems, we define checkpoint staleness and checkpoint tardiness, and then propose *staleness/tardiness-aware skipping* policy to replace the FIFO policy. Extensive experiments verified that the queuing strategy with the skipping policy outperforms blocking and unblocking checkpointing in Pregel-like systems.

Keywords: Graph processing · Fault tolerance · Checkpoint

1 Introduction

Graphs are widely employed in various application areas including social network analysis and online recommendation. With the explosive growth of graph size, the big data represented by graphs might exceed the capacity of a single machine in terms of computation and storage, etc. To effectively process large graph data, Pregel [10] as well as many Pregel-like systems Giraph [1] and Sedge [17] typically scale out in a distributed way by increasing the number of compute nodes. However, failure is common in distributed environment especially with a large number of computational nodes [6]. Meanwhile, graph processing often costs a long execution time, because they require iterative computations. Hence, it is important for Pregel-like systems to effectively handle failures.

© Springer Nature Switzerland AG 2021
W. Zhang et al. (Eds.): WISE 2021, LNCS 13080, pp. 71–86, 2021.
https://doi.org/10.1007/978-3-030-90888-1_6

A typical fault tolerance mechanism proposed in [12] employs a reactive approach to tolerate failure. This approach does not perform any operations during normal execution. Once failure happens, this approach reloads and repartitions the input data to recover the lost vertices as well as edges. Then, it recovers the value of vertices by utilizing the replicas of vertices and a user-defined compensation function. This approach sacrifices the accuracy of computed result [13], even though it achieves fault-tolerance. To keep the result accuracy, most of Pregel-like systems adopt a proactive checkpoint-based approach to tolerate failure [16]. This approach requires the systems to periodically write checkpoints into stable storage during normal execution. Once failure happens, the systems recover from the latest checkpoint. Usually, these Pregel-like systems (e.g., Giraph and Hama [2]) write checkpoints in a blocking manner, which saves the state of the system while pausing computation. Clearly, it incurs an additional overhead on execution time. Different from the blocking checkpointing, the unblocking checkpointing proposed in the literature writes the checkpoints in parallel with the computation [5]. However, its adoption in Pregel-like systems remains unexplored. While applying the unblocking checkpointing to these systems, there may be a resource contention problem due to concurrently checkpointing. Further, the contention brings a negative impact on the time of overall execution and checkpointing. In particular, resource contention may intensify the delay on the execution when multiple checkpoints are issued simultaneously.

The goal of this work is to alleviate the resource contention incurred by writing multiple unblocking checkpoints concurrently, so as to reduce the total execution time. Instead of issuing checkpoints without a limitation, we propose a *queuing strategy* to limit the number of concurrent unblocking checkpoints. Our queuing strategy inserts all the pending checkpoints into a checkpoint queue, and employs a First-In-First-Out (FIFO) policy by default. At a certain time, this strategy limits the number of checkpoints taken from the queue, so as to ensure that the system writes up to k checkpoints concurrently. Our experimental results show that the unblocking checkpointing with queuing strategy decreases the total execution time by 48.1% and 41.5% compared to the blocking and unblocking checkpointing in the failure-free cases, respectively.

Moreover, in Pregel-like systems, the checkpoint in a later superstep is more useful for failure recovery than the one in an earlier superstep, since recovering from a later superstep enables the system to recompute less supersteps once failure happens. In addition, we find that the checkpoint with a shorter writing time is more helpful for failure recovery than the one with a longer writing time, as the one with a shorter writing time becomes available sooner. In Pregel-like systems, this writing time can be estimated by the checkpoint size. Clearly, the FIFO policy is oblivious to the characteristics of checkpointing, which may result in more recovery time when failure happens. Instead of FIFO policy, we first define checkpoint staleness and checkpoint tardiness, and then propose *staleness/tardiness-aware skipping* policy, in order to reduce the recovery time in case of failure. Our experiments show that the staleness/tardiness-aware skipping policy saves 52.3% of recovery time as against the FIFO policy.

In the rest of this paper, we introduce the background of Pregel-like systems in Sect. 2. Then, we make the following contributions:

– We propose *queuing strategy* in Sect. 3 to alleviate the resource contention incurred by multiple concurrent unblocking checkpointing, so as to reduce the overall execution time.
– We propose *staleness/tardiness-aware skipping* policy in Sect. 4 to replace the FIFO policy in queuing strategy, in order to reduce the recovery time.
– We implement a prototype system on Giraph and our experimental studies in Sect. 5 demonstrate that the queuing strategy with staleness/tardiness-aware skipping policy significantly improves the performance of the existing unblocking checkpointing and blocking checkpointing.

In addition, we review the related work in Sect. 6 and conclude our work in Sect. 7.

2 Background of Pregel-Like Systems

This section introduces the workflow of Pregel-like systems, and illustrates two typical checkpointing approaches, i.e., blocking and unblocking checkpointing.

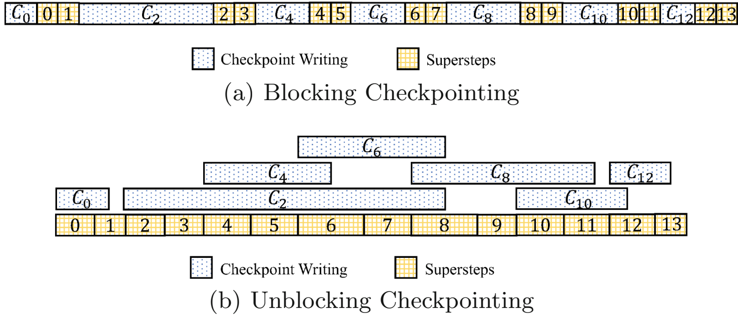
2.1 Execution Workflow

Pregel-like systems (e.g., Giraph, Seraph) adopt a vertex-centric programming model for iterative graph processing. The communication among the vertices in these systems is usually achieved by the message passing technique. Generally, message passing is implemented via synchronized execution [11]. The execution includes a sequence of iterations, called *supersteps*. Each superstep consists of three phases, including computation, communication and synchronization.

In the computation phase, the state of each vertex is either active or inactive. These active vertices handle the messages received from the last superstep and execute the same user-defined function to update the value of vertices in parallel. In the communication phase, each vertex sends messages to their neighbor vertices. The neighbor vertices temporarily store the messages and will handle them in the next superstep. In addition, the vertices update their states accordingly. In the synchronization phase, the vertices that complete the communication phase in advance have to wait for other vertices. Only when all vertices complete the communication phase, they will enter the next superstep. The three phases are executed iteratively till all vertices are inactive.

2.2 Checkpointing

Blocking Checkpointing. Many Pregel-like systems achieve fault tolerance in the way of blocking checkpointing. For example, in Giraph, users specify a checkpoint interval and the system writes checkpoints to a distributed file system by pausing the superstep where the checkpoint is needed. This checkpoint



(a) Blocking Checkpointing

(b) Unblocking Checkpointing

Fig. 1. The process of checkpointing

consists of all vertex states, edges and messages. Once failure occurs, the system reads the latest checkpoint for rollback. In this work, we employ C_i to indicate the checkpoint at superstep i. Figure 1(a) shows the blocking checkpointing that writes a checkpoint every two supersteps. At superstep 2, the system writes the checkpoint C_2 and then starts the execution of this superstep. If failure happens at superstep 3, the system rolls back to superstep 2 and reloads C_2. Clearly, blocking checkpointing incurs a significant overhead cost on execution time.

Unblocking Checkpointing. Different from the blocking checkpointing, some studies in high performance computing propose the unblocking checkpointing, which writes the checkpoints in parallel with the computation [5]. By adopting the unblocking checkpointing, the Pregel-like systems decrease the overhead of blocking checkpointing. As shown in Fig. 1(b), the system writes C_2 along with the execution of supersteps 2, 3 and 4, which avoids the execution time overhead incurred by writing C_2.

3 Queuing Strategy

In this section, we discuss the resource contention in unblocking checkpointing, and then propose a queuing strategy to alleviate the resource contention.

3.1 Resource Contention

Unblocking checkpointing may incur resource contention, although it decreases the execution time compared to blocking checkpointing. The resource contention prolongs the time of superstep execution and checkpointing, since checkpointing is in parallel to the execution of supersteps. As shown in Fig. 1(b), the unblocking checkpointing requires more time to write C_6 and execute superstep 6, compared to the blocking checkpointing in Fig. 1(a). Moreover, the system may write multiple checkpoints concurrently if it takes more time to write a single checkpoint than to execute a single superstep. Writing multiple checkpoints concurrently intensifies the resource contention, which further increases the time of superstep execution and checkpointing.

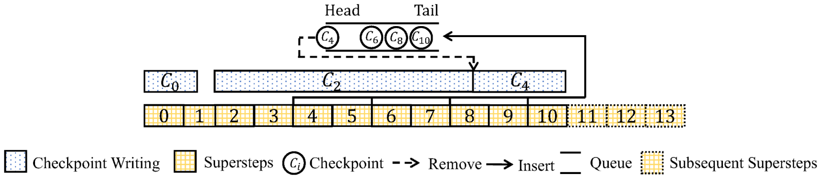


Fig. 2. Checkpoint queue

The overall execution time of unblocking checkpointing with resource contention may still be shorter than the one of blocking checkpointing. However, the delay of checkpointing has a negative impact on the execution time in case of failures. In addition, since the resource contention prolongs checkpointing, the checkpoint is more likely to be unavailable when failure happens, leading to longer recovery time.

3.2 Checkpoint Queuing

Instead of issuing the unblocking checkpoints without a limitation, we propose a queuing strategy to limit the number of concurrent checkpoints as a user-defined parameter k, so as to balance the trade-off between maximizing resource usage and alleviating resource contention [18]. The optimal choice of k is determined by the system configuration. In our queuing strategy, all the pending checkpoints are inserted into a checkpoint queue. When the number of checkpoints n that the system is writing is less than k allowed by our strategy, our queuing strategy follows a First-In-First-Out (FIFO) policy and takes the first $k - n$ checkpoints in the queue to write concurrently.

Figure 2 depicts how the queuing strategy works during execution when $k = 1$. In Fig. 2, the queuing strategy inserts the pending checkpoints C_4, C_6 and C_8 into the queue while writing checkpoint C_2. Once the system writes the checkpoint C_2 completely, as shown in Fig. 2, the system takes the C_4 in the queue and starts writing this checkpoint. Similarly, the queuing strategy inserts the checkpoint C_{10} into the queue when the system reaches superstep 10.

Algorithm 1 illustrates the implementation details of the queuing strategy. Before executing the first superstep, the system registers a handler for the event which is triggered when a checkpoint completes (line 2). During the execution of supersteps, when the superstep i meets the checkpoint interval τ, the system inserts the checkpoint C_i into the queue Q and calls the function CHECKPOINTING (line 4 to line 5). This function gets the number of checkpoints being written, i.e., n (line 11). When $n < k$, the queuing strategy removes the first $k - n$ checkpoints in the queue and temporarily saves these checkpoints in list L (line 12 to 14). Finally, the system writes the checkpoints stored in L (line 15). Moreover, to avoid the circumstance that there exists checkpoints in the queue Q while less than k checkpoints are being written, the system triggers the forementioned event and calls the function CHECKPOINTING when it writes a checkpoint completely.

Algorithm 1. Queuing Strategy

```
1: initial queue Q, list L, counter c;
2: register event handler CHECKPOINTING;
3: while there are active vertices do
4:     if superstep i mod τ = 0 then
5:         insert C_i into Q;
6:         CHECKPOINTING(Q);
7:     end if
8: end while
9:
10: function CHECKPOINTING(queue Q)
11:     n ← calculate the number of checkpoints being written;
12:     if n < k then
13:         Q ← Skipping(Q);
14:         L ← first k − n checkpoints from Q;
15:         write the checkpoints in L;
16:     end if
17: end function
18:
19: function SKIPPING(queue Q)
20:     return Q;
21: end function
```

4 Skipping Policy

In this section, we illustrate the staleness and tardiness of checkpoints in the queue. Then, we design the staleness/tardiness-aware skipping policy to improve the queuing strategy.

4.1 Checkpoint Staleness

The checkpoints in Pregel-like systems store the vertices, edges as well as messages at a certain superstep into external storage. Interestingly, the checkpoint in a later superstep is more useful for failure recovery than the one in an earlier superstep, since recovering from a later superstep enables the system to recompute less supersteps once failure happens. For example, in Fig. 2, compared to the checkpoint C_4 at superstep 4, the checkpoint C_8 at superstep 8 avoids the recomputation from superstep 4 to superstep 7 once the failure happens. To evaluate this property, we illustrate checkpoint staleness in Definition 1.

Definition 1. *(Checkpoint Staleness). For a checkpoint C_i at superstep i, the staleness $S(C_i)$ of this checkpoint is the reciprocal of i, i.e., $S(C_i) = 1/i$.*

The smaller the value of $S(C_i)$ is, the later the superstep i is. Consequently, the smaller staleness value indicates a more useful checkpoint for failure recovery. The queuing strategy in Sect. 3 takes the checkpoints from the head of the queue. Once failure happens, the system has to roll back to an earlier superstep which

requires more recovery time, since these checkpoints at the head of the queue have larger staleness values. Hence, the system should take out k checkpoints with smaller staleness values, so as to take less time to recover in case of failure. In Example 1, the system may avoid the recomputation from superstep 4 to superstep 7 if it takes out C_8 instead of C_4 and C_6.

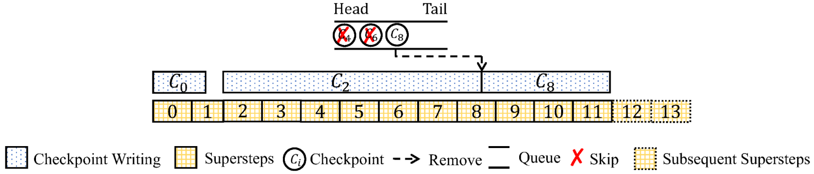


Fig. 3. Checkpoint staleness

Example 1. Figure 3 illustrates the process of checkpointing considering the checkpoint staleness. For simplicity, we assume $k = 1$. Clearly, we have $S(C_8) < S(C_6) < S(C_4)$. Hence, after finishing writing C_2, the system will skip both C_4 and C_6, and then start writing C_8. In case of failure happens at superstep 13, the system recomputes from superstep 8 rather than superstep 4.

4.2 Checkpoint Tardiness

In addition to the staleness, we observe that the checkpoint with a shorter writing time is more useful for failure recovery than the one with a longer writing time. As shown in Fig. 2 and 3, the checkpoint C_4 has a shorter writing time than C_8. Clearly, the checkpoint C_4 is more useful than C_8 when failure happens at the superstep 11. In this case, C_8 cannot be used for recovery, since the system has not completed C_8 yet. Instead, the system rolls back to superstep 4, since C_4 has been finished at superstep 10. To evaluate this property, we define checkpoint tardiness in Definition 2.

Definition 2. *(Checkpoint Tardiness). For a checkpoint C_i at superstep i, the tardiness $T(C_i)$ of this checkpoint is denoted by $T(C_i) = t_i$, where t_i is the time consumed on writing C_i.*

The larger the value of $T(C_i)$ is, the more time to write C_i the system takes. Hence, it is possible that the checkpoint C_i is unavailable when failure occurs, so that the system has to roll back to an earlier superstep. Considering the checkpoint tardiness, the system should pick up checkpoints with a smaller tardiness values, in order to spend less time to recover once failure happens. As described in Example 2, the system owns a complete checkpoint earlier if it picks up C_4 rather than C_8.

Example 2. Following Example 1, we take Fig. 4 as an example to illustrate the checkpoint tardiness. Here, we suppose $T(C_4) = T(C_6) < T(C_8)$. Different from Example 1, the system should write either C_4 or C_6 when considering the checkpoint tardiness. As shown in Fig. 4, the system picks up checkpoint C_4 and finishes

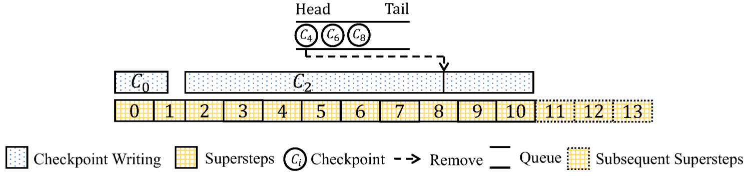


Fig. 4. Checkpoint tardiness

writing this checkpoint at superstep 10. In case of failure happens at superstep 11, the system would recompute from superstep 4, since it has a complete checkpoint C_4. However, if the system chooses to write C_8, this checkpoint has not been completed writing yet. Hence, the system has to recompute from superstep 2.

In general, the tardiness value $T(C_i)$ of the checkpoint C_i depends on the checkpoint size. The tardiness value increases, along with the increasing of the checkpoint size. Hence, we employ the checkpoint size to indicate the length of $T(C_i)$. In Pregel-like systems, the checkpoint consists of the vertices, edges and messages. Hence, we calculate the size of the checkpoint C_i by summing the size of vertices V_i, edges E_i and messages M_i at superstep i. Consequently, the tardiness value $T(C_i)$ is estimated by the checkpoint size, i.e., $T(C_i) \approx Size(V_i) + Size(E_i) + Size(M_i)$.

4.3 Staleness/Tardiness-Aware Skipping

Clearly, the queuing strategy with a FIFO policy is oblivious to the checkpoint staleness and tardiness. In particular, it is unnecessary to write the checkpoint at the head of the queue with a large staleness and tardiness value. Alternatively, the system should skip such kind of checkpoints. Here, we propose a staleness/tardiness-aware skipping policy to replace the FIFO policy.

Ideally, the checkpoint with both smaller staleness and tardiness values enables the system to reduce the recovery time. However, it is not always true that checkpoint with a small staleness value also has a small tardiness value. As in Example 2, compared to the checkpoint C_4 with a staleness value of 1/4, the checkpoint C_8 with a staleness value of 1/8 has a larger tardiness value. Intuitively, once the system completes a checkpoint, it can be utilized for failure recovery. Hence, we prefer to write a checkpoint with a small tardiness value, even though its staleness value is large. Instead of FIFO policy, our staleness/tardiness-aware skipping policy first sorts the checkpoints in the queue by tardiness value and then applies a secondary sort on the staleness value. After that, it issues writing the first $k - n$ checkpoints in the queue. Example 3 illustrates how the skipping policy works during execution.

Example 3. Following Example 1 and 2, the system should pick up checkpoint C_6 when both checkpoint staleness and tardiness are considered, since $T(C_4) = T(C_6) < T(C_8)$ and $S(C_8) < S(C_6) < S(C_4)$. As shown in Fig. 5, the system

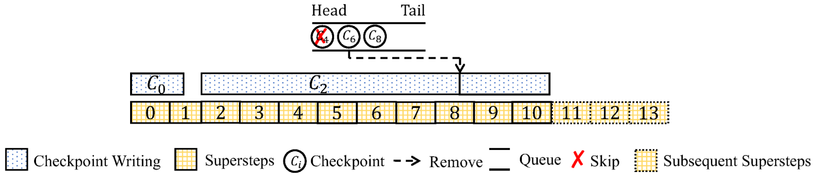


Fig. 5. Staleness/tardiness-aware skipping policy

skips C_4 and picks up C_6 to write. Compared to the system picking up C_4 or C_8, the system picking up C_6 takes the least time to recover when failure happens at superstep 11.

Algorithm 2 describes the implementation details of the skipping policy, which replaces the SKIPPING function in Algorithm 1. The skipping policy estimates the tardiness value $T(C_i)$ (line 2) and calculates the staleness value $S(C_i)$ of checkpoint C_i (line 3). The checkpoint queue is sorted by a composite key with two attributes $T(C_i)$ and $S(C_i)$. The sorting employs $T(C_i)$ as the primary attribute for comparing checkpoints and $S(C_i)$ as the secondary attribute for comparing checkpoints with the same primary attribute (line 4). Finally, this policy returns the queue Q (line 5).

Algorithm 2. Skipping Policy

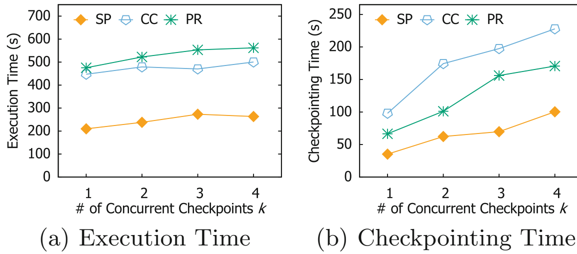
1: **function** SKIPPING(queue Q)
2: $T(C_i) \leftarrow Size(V_i) + Size(E_i) + Size(M_i)$;
3: $S(C_i) \leftarrow 1/i$;
4: sort Q by a composite key $(T(C_i),\ S(C_i))$;
5: return Q;
6: **end function**

5 Experimental Studies

We implemented unblocking checkpointing [7,9] as well as our proposed queuing strategy and skipping policy in Giraph. This section reports the efficiency of our two contributions under failure-free and failure cases, respectively.

5.1 Experimental Setting

Cluster Setup. In order to run Giraph, we deploy Hadoop 2.5.1 on a seven-node cluster. Each node has 2×4-core Intel Xeon E5606 CPUs, 100GB memory, a 2TB HDD and 1 Gbps Ethernet. In addition, we deploy Zookeeper 3.5.5 on this cluster for Giraph's master election and barrier synchronization. By default, we limit the number of map tasks executing on each node to one and allocate 76GB memory for each map task. Among these map tasks, one is the master of Giraph and the others are the workers of Giraph.



(a) Execution Time (b) Checkpointing Time

Fig. 6. The impact of # of concurrent checkpoints (i.e., k) on queuing strategy

Workloads. We choose the Single Source *Shortest Path*, *Connected Components* and *PageRank* algorithms for our experiments, since they are widely adopted to evaluate the performance of graph processing systems [4,16,17]. For simplicity, in the rest of this paper, we refer to the Single Source *Shortest Path*, *Connected Components* and *PageRank* algorithms as *SP*, *CC* and *PR*, respectively. Moreover, we conducted these algorithms over two real-life social network graphs, Twitter[1] and Friendster[2], of million-scale and billion-scale edges respectively.

Baselines. In our experiments, we employ the default blocking checkpointing approach in Giraph and the unblocking checkpointing implemented in Giraph as two baselines. In comparison to these baselines, we evaluate the efficiency of the queuing strategy and staleness/tardiness-aware skipping policy. In the following, to simplify the presentation, we denote the unblocking checkpointing using queuing strategy with FIFO policy as *queuing strategy*. Similarly, we denote the unblocking checkpointing with both the queuing strategy and staleness/tardiness-aware skipping policy as *skipping policy*.

5.2 Efficiency of Queuing Strategy

In this group of experiments, we evaluate the impact of the number of concurrent checkpoints (i.e., k) on queuing strategy, and then compare the performance of queuing strategy against blocking and unblocking checkpointing.

Impact of k**.** To evaluate the impact of k on the performance of queuing strategy, we run three algorithms on the Twitter dataset. Figure 6 illustrates the execution time of the system and the checkpointing time. Here, we set the checkpoint interval to one and vary the value of k from one to four. As shown in Fig. 6(a), when the value of k increases, the execution time of the system using the queuing strategy increases. Moreover, in Fig. 6(b), the checkpointing time follows the trend of execution time. Clearly, the queuing strategy achieves the best performance when the $k = 1$. Hence, we set $k = 1$ in subsequent experiments.

[1] https://networkrepository.com/soc-twitter.php.
[2] https://snap.stanford.edu/data/com-Friendster.html.

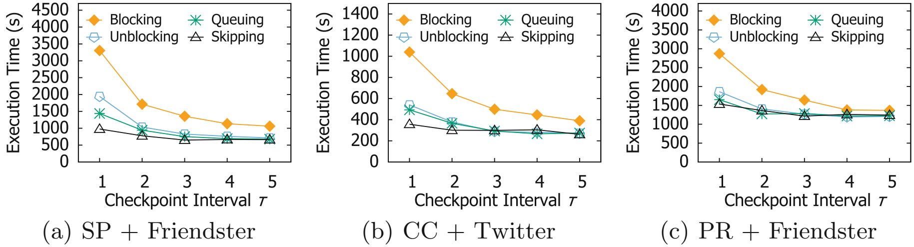


Fig. 7. Execution time without failure

Failure-free Cases. The change of the checkpoint interval τ has an impact on the number of concurrent checkpointing, and therefore decides the performance of the queuing strategy. To evaluate the performance of queuing strategy, Fig. 7 shows the total execution time in different τ. We vary τ from one to five. As shown in Fig. 7, the queuing strategy outperforms the blocking and the unblocking checkpointing. For example, in Fig. 7(a), when $\tau = 1$, the queuing strategy decreases the overhead by 56.4% on execution time compared to the blocking checkpointing. This is because the blocking checkpointing pauses the execution of supersteps to write the checkpoints, which incurs significant overhead of execution time. Likewise, compared to the unblocking checkpointing, the queuing strategy reduces the overhead by 25.9%, since it alleviates the resource contention by setting $k = 1$.

As τ increases, benefits of the queuing strategy on execution time decrease as against the blocking checkpointing and unblocking checkpointing. However, the queuing strategy performs at least as well as these two checkpointing approaches. As shown in Fig. 7(b), when τ increases from one to five, the overhead reduced by queuing strategy decreases from 52% to 29.5% compared to the blocking checkpointing. The reason is that the increasing of τ reduces the number of checkpoints, and the overhead of a blocking checkpoint on execution time is higher than that of an unblocking checkpoint. Compared to the unblocking checkpointing, the execution time of queuing strategy is shorter when $\tau < 3$. However, when $\tau \geq 3$, the queuing strategy has a similar execution time as the unblocking checkpointing. This is because the system with large τ does not write multiple checkpoints concurrently, so that the queuing strategy is ineffective. In other words, as τ increases, the queuing strategy eventually degenerates to unblocking checkpointing.

Failure Cases. Next, we study the performance of queuing strategy under failure cases. Figure 8 depicts the overall execution time. Here, we set $\tau = 1$ and vary the failed superstep from 8 to 12. As shown in Fig. 8, no matter in which superstep failure happens, the queuing strategy outperforms the blocking checkpointing and the unblocking checkpointing. As an example in Fig. 8(a), when failure happens at superstep 8, the queuing strategy decreases the overhead by 48.1% as against the blocking checkpointing. Similarly, in comparison to the unblocking checkpointing, the queuing strategy decreases the execution time by 41.5%. In addition, the unblocking checkpointing in Fig. 8(b) fails to recover

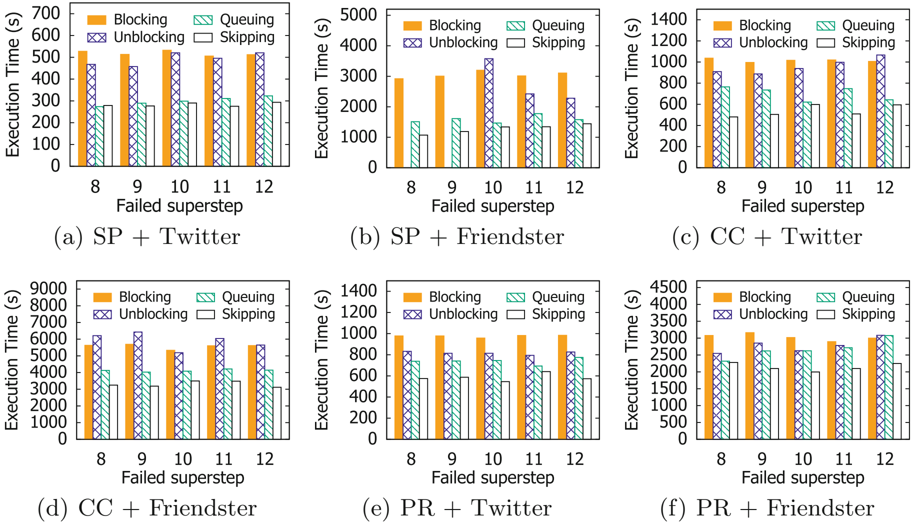


Fig. 8. Execution time with failure ($\tau = 1$)

when failure happens at superstep 8 or superstep 9, since the system has not finished writing any checkpoints.

In summary, by alleviating the resource contention, the queuing strategy effectively reduces the overall execution time and the checkpointing time, so as to achieve better performance than the blocking and unblocking checkpointing. Moreover, the queuing strategy has at least the same performance as the blocking and unblocking checkpointing even if the queuing strategy loses effect.

5.3 Efficiency of Skipping Policy

In this section, we evaluate the performance of the skipping policy against the other checkpointing approaches under failure-free and failure cases. To further illustrate the efficiency of skipping policy under failure cases, we discuss the recovery time of these checkpointing approaches. Moreover, we also analyze the impact of the checkpoint interval on the performance of these checkpointing approaches under failure cases.

Failure-free Cases. As shown in Fig. 7, the skipping policy outperforms other checkpointing approaches. As an example in Fig. 7(a), the skipping policy decreases the execution time by 32% compared to the queuing strategy when $\tau = 1$, since it picks up the checkpoints with smaller tardiness values instead of the checkpoints with larger tardiness values, which reduces the time of resource contention.

Similar to queuing strategy, as τ increases, the benefits of skipping policy on execution time decreases as against the blocking checkpointing and unblocking checkpointing. Moreover, both of them eventually achieve the same performance. For example, in Fig. 7(b), when $\tau \leq 3$, the skipping policy decreases the execution

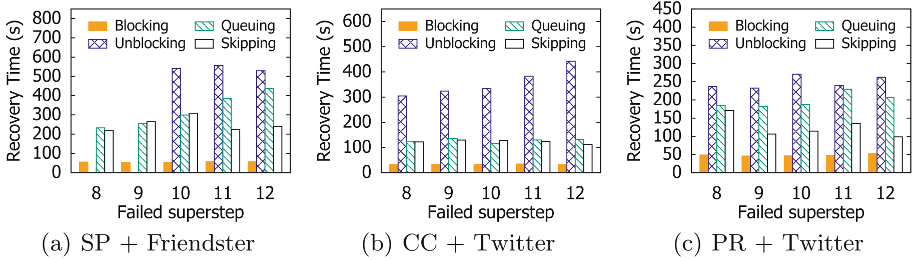


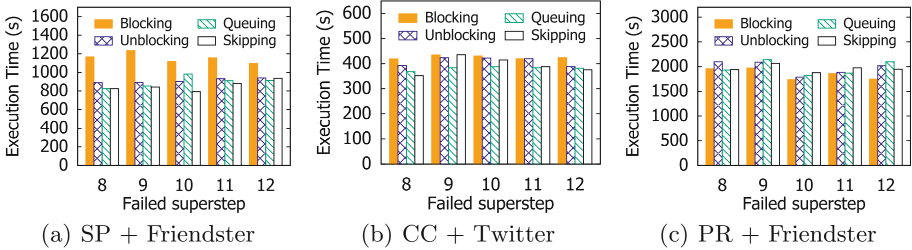
Fig. 9. Recovery time ($\tau = 1$)

time by up to 27.6% as against queuing strategy. Then, the execution time of skipping policy is almost the same as the queuing strategy when $\tau > 3$. The reason is that both the skipping policy and the queuing strategy eventually degenerate to the unblocking checkpointing.

Failure Cases. As shown in Fig. 8, in most failure cases, the skipping policy always outperforms other checkpointing approaches, and achieves the best performance. For example, in Fig. 8(c), when failure happens at superstep 8, the skipping policy decreases the overhead by 37.1% compared to the queuing strategy. The skipping policy allows writing checkpoints with smaller tardiness values which have an impact on the recovery time. Next, in order to further evaluate the performance of these checkpointing approaches, we analyze recovery time of them in failure cases. Moreover, we study the impact of checkpoint interval τ on the performance of the skipping policy under failure cases, since the experiments in Fig. 8 only consider $\tau = 1$.

Recovery Time. Figure 9 provides the recovery time across checkpointing approaches when $\tau = 1$. Due to space limitation, we take *SP* on Friendster dataset, *CC* on Twitter dataset and *PR* on Friendster dataset as examples. However, similar results hold in other cases. In addition, Fig. 9(a) does not include the results of unblocking checkpointing, because there are no available checkpoints to recover for unblocking checkpointing. Clearly, the blocking checkpointing achieves the lowest overhead as against other checkpointing approaches, since it rolls back to the latest superstep. However, since the blocking checkpointing incurs an additional execution time overhead, the performance of queuing strategy and skipping policy is still better than that of the blocking checkpointing.

In addition, the queuing strategy obtains a lower recovery overhead than the unblocking checkpointing. For example, in Fig. 9(c), when failure happens at superstep 12, the queuing strategy decreases the overhead by 21.3% as against the unblocking checkpointing. The reason is because the queuing strategy speeds up the writing of checkpoints so as to obtain a complete checkpoint at a later superstep. Besides, the skipping policy recovers faster than the queuing strategy. Compared to the queuing strategy, the skipping policy decreases the overhead by up to 52.3%. The reason is that the skipping policy avoids writing the checkpoints with larger staleness and tardiness values, which allows the system to have an



(a) SP + Friendster (b) CC + Twitter (c) PR + Friendster

Fig. 10. Execution time with failure ($\tau = 5$)

available checkpoint at a later superstep. Therefore, the skipping policy achieves better performance than the unblocking checkpointing and the queuing strategy.

Checkpoint Interval. Similarly, Fig. 10 takes $\tau = 5$ as an example to illustrate the execution time of these checkpointing approaches. Clearly, the skipping policy achieves similar performance to the unblocking and queuing strategy. Moreover, the skipping policy outperforms the blocking checkpointing. As an example in Fig. 10(a), no matter in which superstep failure happens, the execution time of skipping policy is similar to that of the unblocking checkpointing and the queuing strategy. The reason is that the system does not write multiple checkpoints concurrently when $\tau = 5$, making the queuing strategy and the skipping policy ineffective. In other words, the skipping policy and queuing strategy degenerates to the unblocking checkpointing when $\tau = 5$. In addition, when failure happens at superstep 8, the skipping policy saves up to 29.2% of execution time as against the blocking checkpointing, since the blocking checkpointing pauses the execution of supersteps while writing the checkpoints.

Generally, as a complement to the queuing strategy, the skipping policy effectively avoids writing the checkpoint with a larger staleness value and tardiness value, so as to decrease the execution time and minimize the rollback under failure cases. Moreover, even if the skipping policy loses effect, it is still no worse than the blocking and the unblocking checkpointing as well as the queuing strategy. Consequently, the skipping policy achieves the best performance.

6 Related Work

There are many studies of fault tolerance in distributed graph processing systems. The survey in [8] summarizes the techniques as checkpointing and rollback mechanism, and message logging as well as replication.

Many Pregel-like systems such as Giraph [1] and Hama [2] employ checkpointing and rollback mechanism to achieve fault tolerance. These systems usually adopt a blocking manner to write the checkpointing, while our work foucus on unblocking checkpointing. To decrease the overhead of blocking checkpointing, the work in [15] proposes unblocking checkpointing for graph processing on Flink [3]. This work focuses on the dataflow systems, whereas our work aims

to address the issues of unblocking checkpointing on the Pregel-like systems. Lightweight checkpointing [16] removes messages from the checkpoint and generates messages from checkpointed vertex states during recovery. Moreover, it avoids to restore the same edges as in the previous checkpoint. Nevertheless, it focus on decreasing the checkpointing overhead by reducing the data volume of checkpoint and supplements our work.

By logging messages, Pregel [10] confines the recovery to the failed workers, so as to accelerate the recovery. This technique is complementary to our proposal. The replication-based technique [14] maintains the replicas of each vertex during normal execution and recovers the lost vertex states from the replica once failure. To reduce the overhead cost on checkpointing, the replication-based technique employs the replicas instead of the checkpoints, whereas our work improves the unblocking checkpointing.

7 Conclusions

In this paper, we present a queuing strategy with a FIFO policy to alleviate the resource contention incurred by unblocking checkpointing, so as to reduce the overall execution time and the writing time of checkpoints. Moreover, we exploit the checkpoint characteristics in the Pregel-like system to improve the queuing strategy. To utilize the characteristics, we first define checkpoint staleness and checkpoint tardiness, and then propose staleness/tardiness-aware skipping policy to replace FIFO policy. Our experiments illustrate that the unblocking checkpointing with queuing strategy and staleness/tardiness-aware skipping policy achieves better performance than the blocking and unblocking checkpointing. Presently, we have implemented the queuing strategy and skipping policy by extending Giraph. However, the queuing strategy and skipping policy are also fit for other Pregel-like systems such as Pregel+ and Sedge.

Acknowledgment. This work was supported by the National Natural Science Foundation of China (No. 61902128), Shanghai Sailing Program (No. 19YF1414200).

References

1. Apache giraph. https://giraph.apache.org/
2. Apache hama. https://hama.apache.org/
3. Carbone, P., et al.: Apache flink™: stream and batch processing in a single engine. IEEE Data Eng. Bull. **36**, 28–38 (2015)
4. Cheng, Y., et al.: Which category is better: benchmarking relational and graph database management systems. Data Sci. Eng. **4**(4), 309–322 (2019)
5. Coti, C., et al.: Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In: SC, p. 127 (2006)
6. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
7. Gonzalez, J.E., et al.: Powergraph: distributed graph-parallel computation on natural graphs. In: OSDI, pp. 17–30 (2012)

8. Heidari, S., et al.: Scalable graph processing frameworks: a taxonomy and open challenges. ACM Comput. Surv. **51**(3), 60:1-60:53 (2018)
9. Low, Y., et al.: Distributed graphlab: a framework for machine learning in the cloud. PVLDB **5**(8), 716–727 (2012)
10. Malewicz, G., et al.: Pregel: a system for large-scale graph processing. In: SIGMOD, pp. 135–146 (2010)
11. McCune, R.R., et al.: Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Comput. Surv. **48**(2), 25:1-25:39 (2015)
12. Pundir, M., et al.: Zorro: zero-cost reactive failure recovery in distributed graph processing. In: SoCC, pp. 195–208 (2015)
13. Vora, K., et al.: Coral: confined recovery in distributed asynchronous graph processing. In: ASPLOS, pp. 223–236 (2017)
14. Wang, P., et al.: Replication-based fault-tolerance for large-scale graph processing. In: DSN, pp. 562–573 (2014)
15. Xu, C., et al.: Efficient fault-tolerance for iterative graph processing on distributed dataflow systems. In: ICDE, pp. 613–624 (2016)
16. Yan, D., et al.: Lightweight fault tolerance in pregel-like systems. In: ICPP, pp. 69:1–69:10 (2019)
17. Yang, S., et al.: Towards effective partition management for large graphs. In: SIGMOD, pp. 517–528. ACM (2012)
18. Yildirim, E., et al.: Prediction of optimal parallelism level in wide area data transfers. IEEE Trans. Parallel Distrib. Syst. **22**(12), 2033–2045 (2011)