Distrib Parallel Databases (2017) 35:177–196
DOI 10.1007/s10619-017-7192-2

An I/O-efficient and adaptive fault-tolerant framework for distributed graph computations

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

Published online: 9 March 2017

Abstract In recent year, many large-scale iterative graph computation systems such as Pregel have been developed. To ensure that these systems are fault-tolerant, checkpointing, which archives graph states onto distributed file systems periodically, has been proposed. However, fault-tolerance remains to be challenging because the whole data set is archived with a static interval, rendering underlying graph computations to entail I/O-costs in terms of disk and network communication. Motivated by this, we first propose to dynamically adjust checkpoint intervals based on a carefully designed cost-analysis model, by taking the underlying computing workload into account. Furthermore, for algorithms that can be restarted from any point during computations, we prioritize graph states and then checkpointing can be performed with selected data, instead of the entire dataset, to reduce archiving overhead while simultaneously guaranteeing the failure recovery efficiency. Finally, we conduct extensive performance

✉ Ge Yu
yuge@cse.neu.edu.cn

Zhigang Wang
wangzhiganglab@gmail.com

Yu Gu
guyu@cse.neu.edu.cn

Yubin Bao
baoyubin@cse.neu.edu.cn

Lixin Gao
lgao@ecs.umass.edu

1 School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China

2 Department of Electrical and Computer Engineering, University of Massachusetts Amherst, Amherst, MA 01003, USA

studies to confirm the effectiveness of our approaches over existing up-to-date solutions using a broad spectrum of real-world graphs.

Keywords Iterative graph computations · Fault-tolerance · Checkpointing

1 Introduction

The big data era is coming with strong and ever-growing demands on iterative graph computations in the cyber world, and many distributed systems have been developed for efficiency, mainly including Pregel [17] and its variants [10–12,16,19,21,26], such as Hama, Giraph, GPS, GraphLab, and GraphX (on top of Spark [3]). Because of the frequent machine failures in cloud environments, fault-tolerance is crucial to these Pregel-like systems [17], especially for algorithms with a large number of iterations. For example, the connected components algorithm over a web graph requires up to 744 iterations to fully converge [9].

Generally, a prominent fault-tolerance solution enables the system to efficiently recover from failures by optimizing the re-computation cost, while simultaneously reducing the impact on *failure-free* (i.e., no failure occurs) performance as much as possible. This is typically achieved by a checkpointing method [17], which is widely used in Pregel-like systems. The basic idea is that data are archived onto a distributed file system underneath (e.g., HDFS) periodically, and hence any failure can be recovered by rolling back the whole computing states (including failed ones and surviving ones) to the most recent available point, instead of recomputing from scratch. Unfortunately Pregel-like systems usually suffer high performance degradation during *failure-free* execution, which is up to 8–31x if the checkpointing interval is set as 1 [18]. This is because that the traditional checkpointing implementation requires to archive a large volume of data, including vertex states, messages and edges. The goal of our work is thereby designing new solutions to reduce the expensive overhead.

Challenges Recently, there are several techniques to deal with the overhead issue, all of which are far from ideal. First, the range of checkpoint data can be reduced to vertex states only by separating the logic of vertex update and message generation [29]. But the volume of vertices is still considerable based on our tests. Second, data can be archived in an unblock way to partially overlap underlying graph computations and I/O-accesses [28]. However, the experiment results in [28] show that it does not work well when remote I/O-requests cost more time than local computations. In another word, in this scenario, the system is still blocked to wait for expensive writes. Finally, some efforts are devoted into designing reactive solutions without checkpointing [7,18,20,25]. Upon failures, lost data can be recovered based on surviving replications [7,18] or special functions [20,25]. These solutions are not always feasible because replicating data increases the memory footprint [34] and designing special functions is a non-trivial task for users [28].

We are also aware that some advanced methods focus on improving the recovery efficiency resorting to confining re-computations to failed data only and parallelizing these workloads [17,23,25]. All of them are complementary to our work.

Our contributions Different from existing work, this paper analyzes features of iterative graph algorithms and then designs two improved checkpointing solutions tailored for different scenarios, in order to reduce the archiving overhead.

First, a dynamic checkpointing is proposed to archive data with a tunable interval. In many cases, the underlying computing workload is always varying, such as dynamic PageRank [10], connected components, and shortest paths. This is because some vertices have converged and will not be involved in subsequent computations, which dominates the actual runtime per iteration. On the other hand, as far as we know, all of existing checkpoint-based methods are performed using a static checkpointing interval. That means the recovery cost, i.e., the cost of recomputing from the last available checkpoint to the failed iteration, is also changeable, but the archiving overhead is constant. Obviously, it is not cost-effective when recovery costs less time than archiving data. In order to balance the tradeoff relationship between them, we dynamically adjust the interval parameter. Besides, the parameter value in existing static solutions is empirically given by users. But in our scenario, it is non-trivial since we need to take the variable workload into account. Fortunately, the historical running statistics offer sufficient information to build a cost-analysis model. And hence, the interval can be optimized dynamically.

Second, we design a prioritized checkpointing to further reduce the archiving overhead. This paper derives insights from two facts. (1) Vertices play different roles in dominating the convergence progress, i.e., they exhibit different importance [32]. (2) For some algorithms, such as connected components and shortest paths, the computation interrupted by failures can be restarted from any point [20,25]. The two properties open up a new field in fault-tolerance because we can save partial important vertices only in checkpoint, which potentially reduces the volume of archived data. Motivated by this, our newly designed solution first automatically prioritizes vertices using static information such as outdegree in conjunction with dynamically collected data such as the number of performing updates. Accordingly, we distinguish vertices and archive selected ones based on a user-specified threshold. Different from existing reactive solutions [20,25], our prioritized method removes the requirement of special functions, which largely eases the burden of users.

In summary, this paper makes the following contributions:

- We present a novel adaptive checkpointing method to archive data using a tunable interval. It strikes a nice balance between checkpointing costs and failure recovery performance by dynamically adjusting the key interval parameter.
- A new prioritized checkpointing method is devised to distinguish the importance of different vertices for the algorithm convergence speed and then only selected data are archived to reduce the volume of archived data for efficiency. A priority computation model is used to make the smart decision when selecting data, which guarantees the recovery efficiency.
- Both of our two solutions work automatically based on the carefully designed models, which is feasible and facilitates fault-tolerance for various Pregel-like systems and different graph algorithms.

Paper organization The remainder of this paper is organized as follows. Section 2 introduces preliminaries about iterative graph algorithms, the fault-tolerance prob-

lem and the state-of-the-art technique. Section 3 presents our adaptive checkpointing method and the cost-analysis model, based on analyzing the runtime features of algorithms with dynamic workload. Section 4 describes the details of our prioritized checkpointing solution and the priority computation model. Section 5 reports experimental studies over real graphs. Section 6 highlights the related work. Finally, we conclude this paper in Sect. 7.

2 Preliminaries

2.1 Iterative graph computations

We restrict our discussion to a directed graph $G = (V, E, w)$, where V is the vertex set with $|V|$ vertices and E is the edge set with $|E|$ edges. w is a weight function: $\forall e \in E, e \rightarrow w(e)$. An iterative graph algorithm repeatedly refines each vertex $v_i \in V$ by a user-defined update function f, until an explicitly specified condition is satisfied. This process consists of multiple iterations separated by synchronous barriers. At the $(t+1)$th iteration, the workload mainly includes updating every vertex v_i by consuming messages received from its in-neighbors at the previous iteration, i.e., M_{in}^t, and sending new messages along outgoing edges (denoted by $\Gamma(v_i)$) to its out-neighbors, i.e., M_{out}^{t+1}. M_{out}^{t+1} will be used at the next iteration. Eq. (1) shows it mathematically.

$$(v_i^{t+1}, M_{out}^{t+1}) \leftarrow f(v_i^t, M_{in}^t, \Gamma(v_i)) \tag{1}$$

Many algorithms follow the expression in Eq. (1), such as PageRank [4] and its variants, Katz metric [14], simulating advertisements [15], connected components, and shortest paths. In the following, we give more details about simulating advertisements and shortest paths since they are used as example algorithms in our experiments (Sect. 5). Both of them are terminated when no messages are exchanged.

Simulating advertisements [15] Each vertex represents a person with a list of favorite advertisements as its value. A user-given vertex is identified as a source and broadcasts its value to out-neighbors. Subsequently, a received advertisement is either forwarded or ignored, based on the receiver's interests. In particular, we use the same computing logic with [26]. That is, a person only forwards the advertisement that a maximum number of his/her neighbors have.

Shortest paths By "shortest paths", we refer to the single source shortest distance computation [17]. It finds the shortest distance between a given source vertex to any other one. The source vertex has a distance 0 and then in every iteration, a vertex value is updated to be the minimum value received from its in-neighbors, if the received value is lower than its current distance. After updating a vertex, it will immediately broadcast messages to its out-neighbors by adding edge weights to its new value.

2.2 Fault-tolerance

In distributed environments, some machines may fail during iterations. To tolerate failures, the most widely used method is checkpointing. That is, the system archives

data periodically and then the iterative computations can be rolled back to the most recent available point upon failures. From the perspective of data volume, the up-to-date technique is to decompose the computing function in Eq. (1) into two parts: vertex update and message generation, as shown in Eqs. (2) and (3), respectively. In this way, only vertices are required to be archived [29]. When recovering failures, we can load archived vertices and then re-execute Eq. (3) to get required messages to restart computations. In Sect. 5, we compare our proposals with this technique.

$$v_i^{t+1} \leftarrow update(v_i^t, M_{in}^t) \tag{2}$$

$$M_{out}^{t+1} \leftarrow f(v_i^{t+1}, \Gamma(v_i)). \tag{3}$$

2.3 A Pregel-like system: HybridGraph

Finally, we introduce a Pregel-like system, HybridGraph [26], since our proposals are implemented on top of it. Different from memory-resident systems, such as Pregel, HybridGraph is designed to perform graph computations I/O-efficiently. To this end, it also uses Eqs. (2) and (3) to handle graph algorithms. Then messages can be generated on demand of target vertices in a block-centric way, instead of being prepared beforehand, and consumed immediately. That largely reduces the memory footprint and hence expensive disk I/Os are avoided. For more details, please refer to [26]. We use HybridGraph as the underlying framework because it naturally supports the up-to-date fault-tolerance technique [29] as mentioned in Sect. 2.2 and either its memory-based engine or disk-based engine has prominent performance when compared with other existing systems.

3 Dynamic checkpointing

This section describes our adaptive checkpointing solution by discussing three key issues. First, we analyze the runtime features of iterative algorithms, which is the motivation of our proposal. Second, we give the details on how to archive data using a tunable interval. Third, we present the cost-analysis model to decide a reasonable interval automatically and prove that our solution performs better than existing work.

3.1 Runtime features of iterative algorithms with dynamic workload

As mentioned in Sect. 2.1, vertex values are refined again and again in iterative algorithms. In this process, vertices will converge to their fixed points individually at different speeds. Once a vertex converges, it will not be involved in remaining iterations, to save limited computing resources and network bandwidth. As an indication of how the workload changes with iterations, Fig. 1 plots the number of vertices involved in computations and the runtime per iteration, by running Simulating Advertisements on a web graph Wiki ($|V| = 5.7$ million, $|E| = 130$ million, 5 Amazon

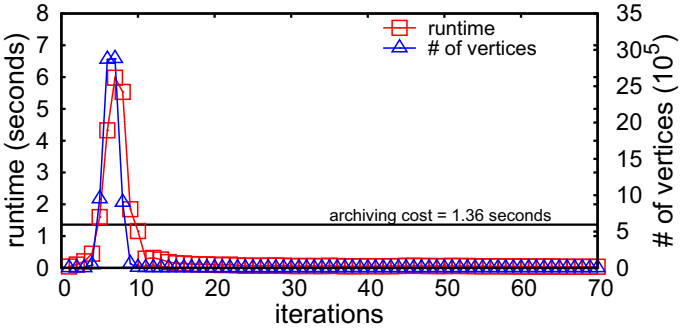

Fig. 1 Analysis of dynamic workload for iterative algorithm (Simulating Advertisements on Wiki)

EC2 c3.2xlarge instances). We only report the statistics of the first 70 iterations since the trend of the remaining computations is the same with that after the 50th iteration.

We can easily find that after the 10th iteration, only 10% of vertices are involved in computations, which drops the runtime so much that it is greatly less than the average archiving time. In this case, frequently saving checkpoints after the 10th iteration is not cost-effective because recomputing lost iterations is very fast. Most algorithms exhibit the similar features, including dynamic PageRank [10], Katz metric [14], connected components and shortest paths.

3.2 Archiving data with tunable checkpointing interval

The basic idea of our adaptive checkpointing is to adjust the checkpointing interval dynamically. Figure 2 illustrates the difference between traditional static solution and our solution. Assume that a graph processing job consists of three distributed tasks and the static interval $\tau = 5$. When *task_2* fails at the 18th iteration ($t = 18$), three checkpoints have been archived and hence the system can roll back data in all tasks to the most recent available point, i.e., ck_3. That means 3 iterations need to be re-executed in recovery. Different from that, in our adaptive solution, perhaps only one checkpoint is saved, i.e., ck_1 at $t = 7$, before failures. Of course we save the overhead of two checkpoints when compared with the static solution, but the recovery requires to repeat 11 iterations from $t = 7$ to $t = 18$. Obviously, increasing the interval reduces the I/O-costs, but trades off the time of writing checkpoints with the time required in recovery. This key issue is solved by our cost-analysis model by collecting historical information. We will discuss it in Sect. 3.3.

Now we introduce how to integrate the dynamic solution into an existing Pregel-like system. As shown in Algorithm 1, the key modification is to replace the static checkpointing judgement logic, i.e., "$t\ MOD\ \tau\ is\ 0$?", with our function $isCheckpoint()$ (Line 7). $isCheckpoint()$ runs our cost-analysis model described in Sect. 3.3 to decide whether a checkpoint is supposed to be saved. In order to run this function, we need to collect some runtime information including the runtime of underlying computations (C^t_{ite}, lines 5–6) and overhead of archiving data (C^t_{arc}, lines 8–9). They are maintained in two queues, *histIteQ* and *histArcQ*, respectively.

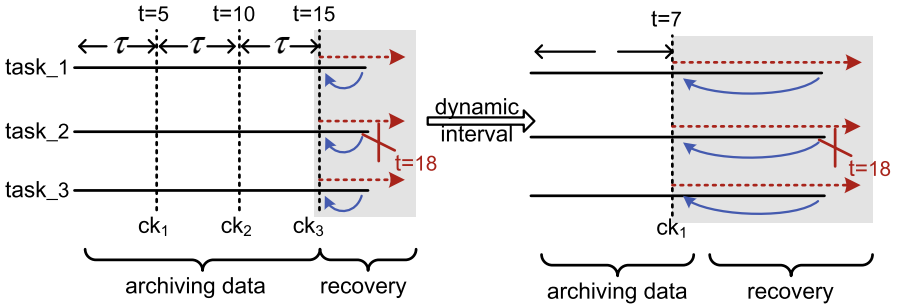

Fig. 2 Illustration of dynamic checkpointing

Algorithm 1: Dynamic Checkpointing

Input : static checkpointing interval τ

```
1  initialize histIteQ to record historical runtime of underlying computation
2  initialize histArcQ to record historical runtime of arching data
3  t ← 1
4  while convergent condition is not satisfied do
5      C^t_ite ← runIteration()
6      add C^t_ite into histIteQ
7      if isCheckpoint(histIteQ, histArcQ, τ, t) then
8          C^t_arc ← archiveData()
9          add C^t_arc into histArcQ
10     t ← (t + 1)
11 return
```

3.3 Cost-analysis model

The main idea behind the cost-analysis model is that the overhead of archiving data is supposed to be no more than that of recovery. The latter, C_{rec}, is estimated by summing up the actual iteration runtime C^y_{ite} since the last checkpoint (i.e., the ith one) location ck_i, where $(ck_i + 1) \leq y \leq t$. On the other hand, we calculate the average cost of completed checkpoints, $\bar{C}_{arc}$, as the final value used in judgement, to improve the accuracy. Eq. (4) shows it mathematically.

$$\bar{C}_{arc} = \frac{1}{i}\sum_{x=1}^{i} C^x_{arc} \leq C_{rec} = \sum_{y=ck_i+1}^{t} C^y_{ite} \tag{4}$$

At the very beginning of iterations, $\bar{C}_{arc}$ is initialized to zero and hence Eq. (4) always returns "TRUE". On the other hand, when the workload is high, such as at the 7th iteration in Fig. 1, C^y_{ite} may be much more than $\bar{C}_{arc}$. Equation (4) thereby returns "TRUE". In both of the two cases, perhaps our solution archives checkpoints in a higher frequency compared with the static method. To avoid this scenario, we still need to consider the static interval τ. As the dynamic checkpointing location destroys

the "MOD" regular expression, we directly compare τ with completed iterations since the last checkpoint, as shown in Eq. (5)

$$\tau \leq (t - ck_i) \tag{5}$$

Thus, the main body of *isCheckpoint()* used in Algorithm 1 is performing the logical "AND" operation, i.e., Eqs. (4) and (5), to guarantee that the number of checkpoints is no more than that in the static method, which is proved in the following. Let I denote the total number of checkpoints archived in our dynamic solution. In contrast, for the static solution, it is $\lfloor T/\tau \rfloor$, where T is the number of completed iterations. Lemma 1 then describes that the former is always no more than the latter.

Lemma 1 $I \leq \lfloor \frac{T}{\tau} \rfloor$.

Proof For the static checkpointing, obviously, $T = \lfloor \frac{T}{\tau} \rfloor \cdot \tau + y_s$, $0 \leq y_s \leq \tau$. Now, we analyze the value of I for our dynamic solution. We discuss it in two parts. First, assume that the dynamic solution is performed with tunable intervals in $(m \cdot \tau + y_d)$ iterations, $0 \leq y_d \leq \tau$. During these iterations, the actual interval is always no less than τ. Thus, the actual number of checkpoints is no more than m. Second, the number of remaining iterations with static interval τ is $(\lfloor \frac{T}{\tau} \rfloor - m) \cdot \tau + (y_s - y_d)$. The actual number of checkpoints is thereby $(\lfloor \frac{T}{\tau} \rfloor - m) + \frac{(y_s - y_d)}{\tau}$. Then we can infer that $I \leq m + (\lfloor \frac{T}{\tau} \rfloor - m) + \frac{(y_s - y_d)}{\tau} = \lfloor \frac{T}{\tau} \rfloor + \frac{(y_s - y_d)}{\tau}$. Because $-1 < \frac{(y_s - y_d)}{\tau} < 1$, $I \leq \lfloor \frac{T}{\tau} \rfloor$ is proved. □

After that, Lemma 2 gives the saved archiving overhead due to adjusting interval dynamically. Finally, Theorem 1 compares the overall performance of the two solutions by considering the archiving cost and recovery cost, where θ_s/θ_d is the recovery cost for the static/dynamic solution.

Lemma 2 *The dynamic checkpointing reduces the archiving overhead by* $(\lfloor \frac{T}{\tau} \rfloor - I) \cdot \bar{C}_{arc}$ *when compared with the static solution.*

Theorem 1 $(\lfloor \frac{T}{\tau} \rfloor - I) \cdot \bar{C}_{arc} - (\theta_d - \theta_s) \geq 0$.

Proof We prove this theorem by analyzing the performance of our dynamic solution in the worst case, that is, a failure occurs during iterations where the dynamic interval is used. Based on Eqs. (4) and (5), the upper bound of θ_d is $\bar{C}_{arc}$. Meanwhile, the lower bound of θ_s is 0. Thus, for the dynamic solution, the extra recovery cost compared with the static one is $(\theta_d - \theta_s) \leq \bar{C}_{arc}$. Based on Lemmas 1 and 2, when $I = \lfloor \frac{T}{\tau} \rfloor$, the dynamic checkpointing is equivalent to the static solution. That is, $\theta_d = \theta_s$, and then both the saved archiving cost and the extra recovery cost are 0. Otherwise, $(\lfloor \frac{T}{\tau} \rfloor - I) \geq 1$, which can offset the upper bound of the extra recovery cost $\bar{C}_{arc}$. Then we have the claim. □

4 Prioritized checkpointing

In this section, we present our prioritized checkpointing method, which is developed on top of dynamic checkpointing to further reduce I/O costs of archiving data, but only

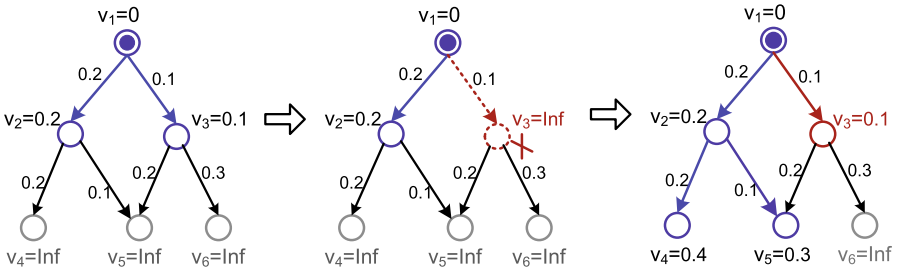

Fig. 3 Failure recovery of Shortest Paths

suitable for some algorithms mentioned in [20], like Shortest Paths and connected components.

4.1 Direct failure recovery for Shortest Paths

An interesting phenomenon for some algorithms is that the computation can be restarted from any point upon failures. We demonstrate it in Fig. 3, taking Shortest Paths as an example. Given a weighted directed graph and a source vertex v_1, at the 1st iteration, v_1's direct neighbors, v_2 and v_3, are updated. Suppose that a failure occurs at the 2nd iteration and v_3 is lost. To recover failures, we can directly load lost data only and then initialize them again to repeat computations for them. Meanwhile, data on surviving machines will be preserved (e.g., v_2) and their updates will keep going when recovering lost data (e.g., $v_2 \rightarrow (v_4, v_5)$). Even so, we can still get the same computing results with that in *failure-free* execution. Some other algorithms, such as connected components, also fall into this category.

Schelter et al. [20] utilize this property to tolerate failures without checkpoints. But in this paper, we use it to design a prioritized checkpointing with nearly zero archiving cost because only selected vertices are saved. Meanwhile, our solution is supposed to recover failures more efficiently since some lost data are replaced with checkpoints, instead of initial values.

4.2 Archiving data with priority

The only difference between dynamic checkpointing and our prioritized solution is that when *isCheckpoint()* returns "TRUE", only selected vertices, not all of them, are archived as a checkpoint. That reduces $\bar{C}_{arc}$ and then potentially increases the checkpointing frequency, as shown in Fig. 4. However, based on Theorem 1, the overall I/O-efficiency is still better than the static solution. Upon failures, only replacements for failed machines (i.e., tasks) will roll back to the last available checkpoint. Lost vertices are initialized to checkpoint values if they have been selected and archived, or initial states. After that, all machines continue the computations.

Even though not all data are archived, compared with the dynamic solution, the recovery is potentially accelerated due to two reasons. First, surviving machines will

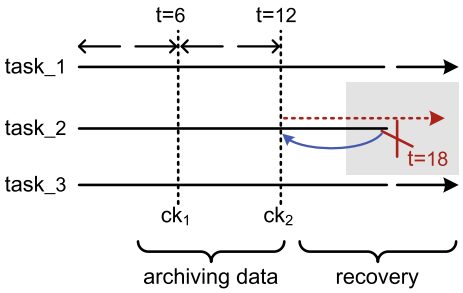

Fig. 4 Illustration of prioritized checkpointing

not load checkpoints, which saves I/O accesses and network bandwidth. Second, data on surviving machines will not be rolled back, that avoids re-computation cost and can speed up the recovery of lost data by broadcasting their values. In Sect. 4.3, we will discuss how to select important vertices when performing checkpointing, to further improve the recovery efficiency.

4.3 Priority metric

This paper computes vertex priority by considering its number of performing updates and degree (i.e., $|\Gamma(v_i)|$). This is because once a vertex is updated, it will broadcast its new value to out-neighbors based on Eq. (3). Thus, the recovery workload is roughly proportional to the product of the two parameters. Equation (6) shows the priority of v_i at the tth iteration. Here, $upd(v_i, x)$ is 1 if v_i is updated at the xth iteration. Otherwise, it's zero. Before launching a new checkpoint, all tasks in the current graph processing job will sort their local vertex priorities and then select top-K ones as checkpoint. The priority queue size K is given by users. With its increase, the prioritized checkpointing eventually degrades into the dynamic solution. We will experimentally discuss its impact in Sect. 5.6.

$$pri(v_i, t) = |\Gamma(v_i)| \cdot \sum_{x=1}^{t} upd(v_i, x). \tag{6}$$

5 Evaluation

This section evaluates our proposals and analyzes the performance on many real-world graphs as listed in Table 1.

5.1 Experimental setup

Solutions tested in experiments For simplicity, our proposals described in Sects. 3 and 4 are indicated by Dyn-CK (checkpointing with dynamic interval) and Pri-CK (checkpointing with priority), respectively. We use the state-of-the-art technique [29] as a Baseline. That is, vertices are archived periodically with a static interval. All of the three solutions are implemented on top of a recently published system called

Table 1 Real graph datasets

Graph	Vertices	Edges	Degree	Type
LiveJ[a]	4,847,571	68,028,541	14.24	Social Network in LiveJournal
Wiki[b]	5,716,808	130,158,845	22.77	Web Graph of Wikipedia
Road[c]	23,947,132	58,332,307	2.44	Full USA Road Network
Holly[d]	2,180,759	229,181,085	105.09	Social Network of movie actors in Hollywood
UK[e]	18,520,486	298,113,762	16.10	Web Graph from a crawl of the .uk domain

[a] https://snap.stanford.edu/data/soc-LiveJournal1.html
[b] http://haselgrove.id.au/wikipedia.htm
[c] http://www.dis.uniroma1.it/challenge9/download.shtml
[d] http://law.di.unimi.it/webdata/hollywood-2011/
[e] http://law.di.unimi.it/webdata/uk-2002/

HybridGraph [26], in order to present an end-to-end performance comparison. Experiments are run using HybridGraph's memory-resident engine, but our ideas can also work for its disk-resident engine. Note that existing parallel methods [23,28] are not involved in our study as they are complementary to our Dyn-CK and Pri-CK.

Experimental cluster We run all of our experiments on the Amazon EC2 cluster using c3.2xlarge instances (8 virtual cores, 15 GB of RAM, and 30 GB HDDs) running Ubuntu Server 14.04. The cluster consists of 5 instances with an additional one as Master because HybridGraph employs a typical master-slave framework, like Pregel.

Algorithms and datasets Dyn-CK and Pri-CK can work for many algorithms, like Baseline. However, here we conduct testing using Simulating Advertisements and Shortest Paths as representatives (described in Sect. 2.1). For some well-known algorithms, like PageRank, all vertices are typically involved in updates at every iteration. The underlying computation cost is usually bigger than the archiving overhead and the computation workload will not change with iterations. In this case, Dyn-CK is equivalent to Baseline based on the cost-analysis model in Sect. 2.3 and hence we ignore its experiment results. Connected components, as another widely used algorithm, exhibits the varied computation workload because the number of involved vertices gradually decreases with iterations. But the workload of Shortest Paths will increase first and then decrease, which is more complicated than connected components. We thereby use Shortest Paths instead of connected components. Simulating Advertisements is chosen as an example which cannot be supported by Pri-CK.

All tests are done over real graphs in numerous applications, including social networks, road networks, and web graphs, in order to evaluate the effectiveness of our proposals in different real-world scenarios. Detailed dataset descriptions are given in Table 1. For Shortest Paths, each edge is assigned a random number between 0 and 1 as its weight.

Failure simulation Failure is simulated by setting one task failed manually at some iteration. Because we expect Dyn-CK and Pri-CK to reduce the volume of archived data as much as possible, this may increase the recovery cost to recompute lost data, especially when the failure occurs at the end of computations. Thus, we evaluate our

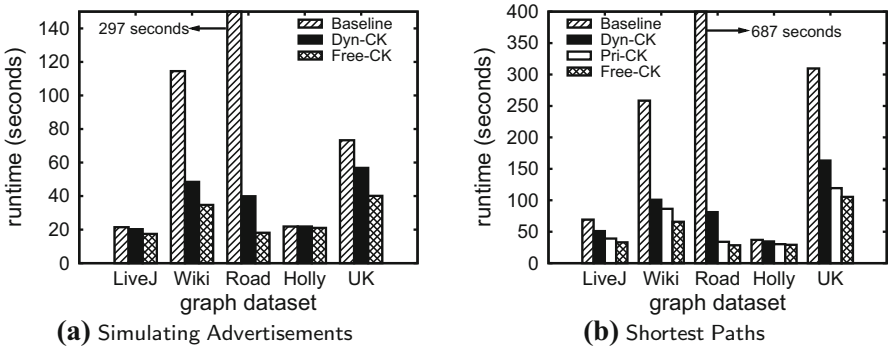

Fig. 5 The *failure-free* performance of different solutions

proposals in the worst case. That is, without loss of generality, the failure location is given as $0.9 \cdot T$, where T is the total number of iterations in *failure-free* execution and we can get its value by running algorithms beforehand.

Other settings Unless otherwise specified, the checkpointing interval is set as 5 and the priority queue size used in Pri-CK is initialized as $0.2|V|$, based on the experience of [8,29,32]. HDFS in hadoop-0.20.2 [2] is used as the distributed file system to store archived data (64 MB per block, 3 replications).

5.2 Data archiving overheads

We run Simulating Advertisements and Shortest Paths on all graphs in *failure-free* execution to evaluate the I/O-efficiency of our proposals in comparison with Baseline. In particular, we only show the statistics of the first 200 iterations for the Road graph for simplicity, even though it requires thousands of iterations to fully converge due to its extremely large diameter.

Figure 5 plots the runtime under different checkpointing policies. "Free-CK" means no data is archived during computations. As the sub-figures show, Dyn-CK always performs better than Baseline. Prioritized archiving mechanism creates an additional gap between Dyn-CK and Pri-CK but the impact is less significant on runtime.

Generally, the speedup of Dyn-CK/Pri-CK compared with Baseline is up to 8.5x/20.3x (Shortest Paths on Road), but Simulating Advertisements on Holly has the least improvement. Holly is a social network with high fan-out. Expensive message processing dominates the overall runtime and hence the overhead of archiving vertices can be negligible. Also, the perfect reachability leads to a fast convergence, especially for Simulating Advertisements. That means only a few checkpoints will be saved even for Baseline, which largely limits the impact of our proposals on reducing the volume of archived data. In contrast, the road network Road exhibits a large diameter, which usually requires a lot of iterations to converge. In particular, at each iteration, the underlying workload is tiny due to the low degree. In this case, archiving vertices is the dominant cost and then changing interval and prioritized selection are curial.

We now present the performance analysis in more detail. Figs. 6 and 7 show the numbers of archived checkpoints and vertices. Of course Dyn-CK significantly drops

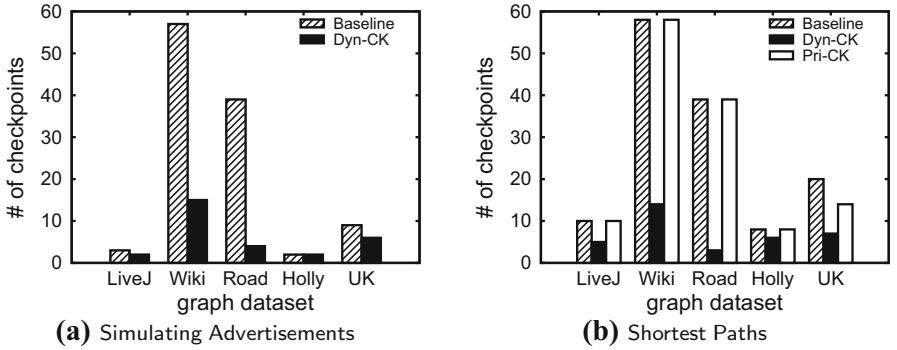

Fig. 6 The number of checkpoints archived in *failure-free* execution. (a) Simulating Advertisements. (b) Shortest Paths

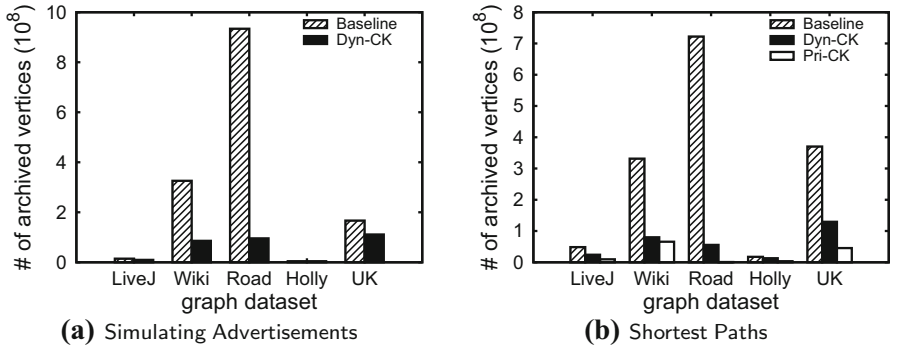

Fig. 7 The number of vertices archived in *failure-free* execution. (a) Simulating Advertisements. (b) Shortest Paths

the frequency of performing checkpoints. To explain that, we demonstrate the runtime per iteration on the Wiki graph in Fig. 8. For our both example algorithms, the underlying computation cost of every 5 iterations is much smaller than that caused by archiving all vertices. Our cost-analysis model thereby can change the interval dynamically to archive data using a large interval. Accordingly, the number of archived vertices can be reduced, which improves the overall performance.

An interesting observation we can find is that in all cases excluding Shortest Paths on the UK graph, Pri-CK has the same number of checkpoints with Baseline. The reason is that the sum of five iterations' costs is usually more than that of prioritized checkpointing. Even so, the total number of archived vertices is still smaller than that in Dyn-CK benefitting from the prioritized selection.

5.3 Recovery runtime and overall runtime

To show the impact of changing interval (Dyn-CK) and prioritized selection (Pri-CK), Figs. 9 and 10 plot the recovery runtime and overall runtime, respectively, when encountering a machine failure at the end of computations.

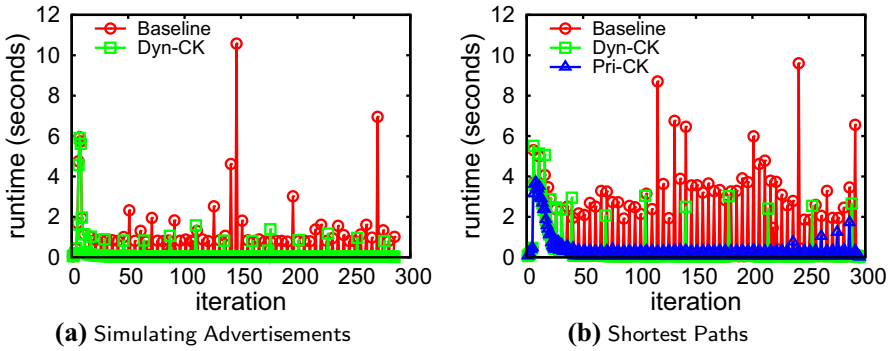

Fig. 8 Runtime per iteration in *failure-free* execution (on the Wiki graph)

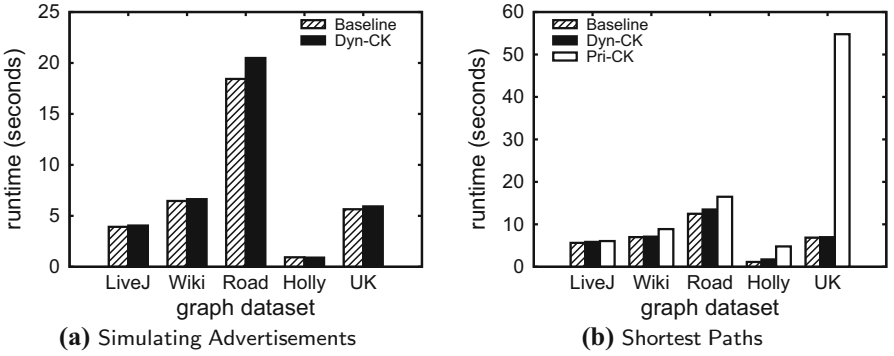

Fig. 9 Recovery runtime when one machine fails at the end of computations

Generally, Dyn-CK takes more time (shown in Fig. 9) than Baseline to recover lost workloads since the former archives fewer vertices. However, from the perspective of overall runtime (shown in Fig. 10), Dyn-CK beats Baseline in all cases. This can be explained that archiving vertices costs more time than re-executing some lightweight underlying iterative computations, and Dyn-CK can make a smart decision to initiate a checkpoint based on its cost-analysis model.

Pri-CK can further reduce the volume of archived data, but its recovery runtime is slightly more than that of Dyn-CK in many cases (shown in Fig. 9). As a result, Pri-CK usually beats Dyn-CK in terms of overall runtime (shown in Fig. 10). We explain it from three perspectives. First, our priority computing model can smartly archive the most important vertices based on their re-computation costs, which reduces the performance degradation in recovery. Second, when performing recovery, data on surviving machines will not be rolled back as Baseline and Dyn-CK. That confines re-computations to data on failed machines only, which is efficient. Also, these surviving vertices can accelerate the recovery processing of lost data by broadcasting their values as reported in [25]. The exception is Shortest Paths on UK. Fig. 10 shows that Dyn-CK beats Pri-CK. This reason is that although Pri-CK archives fewer vertices (I/O costs) than Dyn-CK (Fig. 7), it largely increases the recovery cost as shown in Fig. 9.

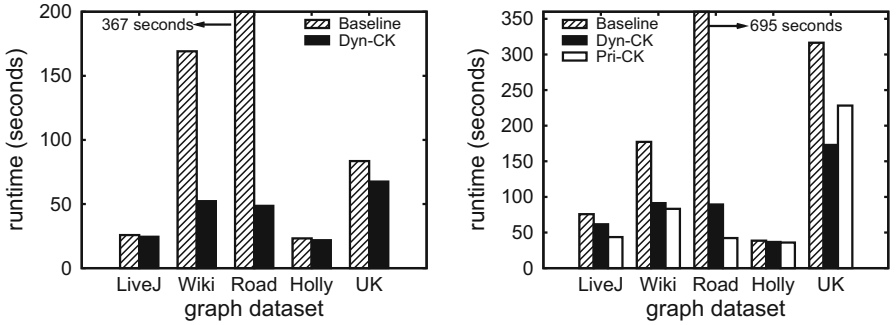

Fig. 10 Overall runtime when a machine fails at the end of computations. (a) Simulating Advertisements. (b) Shortest Paths

5.4 Impact of checkpointing interval

Now we explore the impact of checkpointing interval on overall runtime when failure occurs at different locations. Before showing results, we introduce some symbols. "F = 12" indicates that one machine fails at the 12th iteration. In addition, the "+∞" interval means we disable the checkpointing function, i.e., no data is archived. This suite of experiments is performed using a social network LiveJ and a web graph Wiki.

The performance of Baseline largely depends on the combination of specific algorithms, failure locations, and datasets. Specifically, in Fig. 11, the optimal checkpointing interval varies from 5 to 6, when changing the failure location from 12 to 13. On the other hand, both Figs. 11(b) and 12(b) show that on the Wiki graph, users are supposed to disable checkpointing since recomputing from scratch performs best. While, on the LiveJ graph, especially for Shortest Paths, disabling checkpointing leads to suboptimal performance. Obviously, it is a great challenge for users to choose a reasonable checkpointing interval empirically.

Different from Baseline, a quite large range of interval values, such as 1–5, can allow our Dyn-CK to work well. Furthermore, Dyn-CK always runs faster than recomputing from scratch (i.e., the interval is +∞).

5.5 Impact of priority queue size

Table 2 reports the overall runtime when encountering a failure under different priority queue sizes in Pri-CK. The runtime slightly increases because archiving more data offsets the gain achieved by fast recovery. Zhang et al. draw the similar conclusion in prioritized computations [32]. Thus, we use $0.2|V|$ in experiments.

5.6 Impact of the number of failed machines

We finally explore the features of all solutions when varying the number of failed machines. A new compared solution called "Optimistic" [20] is added since it utilizes

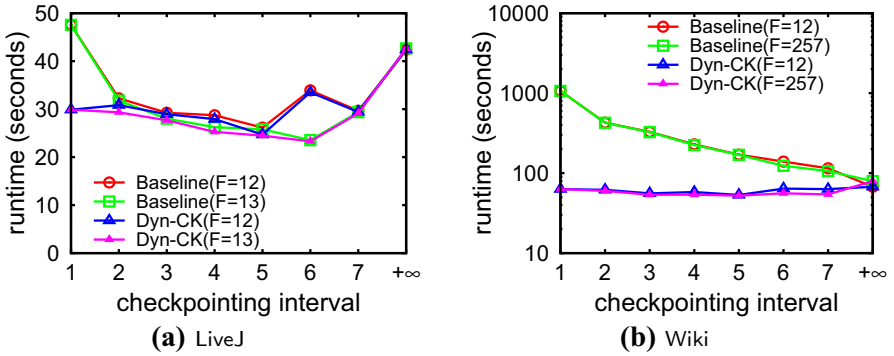

Fig. 11 The impact of checkpointing intervals (Simulating Advertisements)

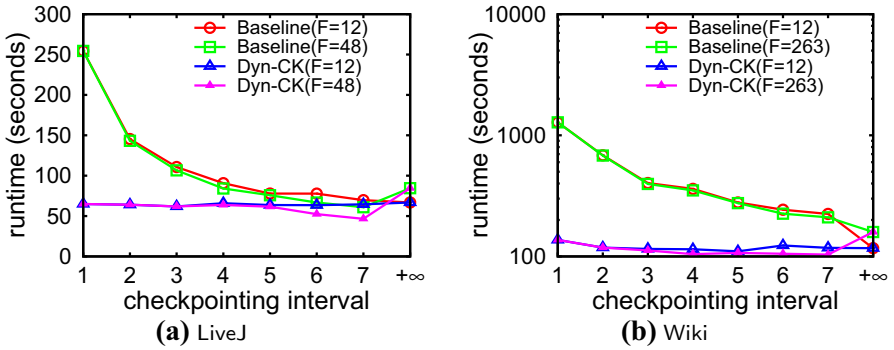

Fig. 12 The impact of checkpointing intervals (Shortest Paths)

Table 2 The impact of priority queue size (seconds)

Graph	$0.2\|V\|$	$0.4\|V\|$	$0.6\|V\|$	$0.8\|V\|$	$\|V\|$(Dyn-CK)
LiveJ	43.56	48.84	49.32	49.85	61.58
Wiki	91.25	99.27	102.39	104.30	106.94

the same property (Sect. 4.1) for failure recovery as our Pri-CK, but no checkpoint is archived. We test Optimistic here because its performance depends on how many workloads are preserved. All tests are performed using Shortest Paths since Pri-CK and Optimistic cannot work for Simulating Advertisements. Using the same setting in Fig. 10, Fig. 13 reports the overall runtime.

Not surprisingly, the performance of Baseline and Dyn-CK is not sensitive to the number of failed machines because both of them replace vertices on every machine with the most recent available checkpoint. By contrast, with the increase of the number of failed machines, the performance of Pri-CK and Optimistic degrades. The reason is that the two methods try to utilize data on surviving machines to recover lost workloads. Apparently, more surviving machines means more data are preserved and then the recovery is faster.

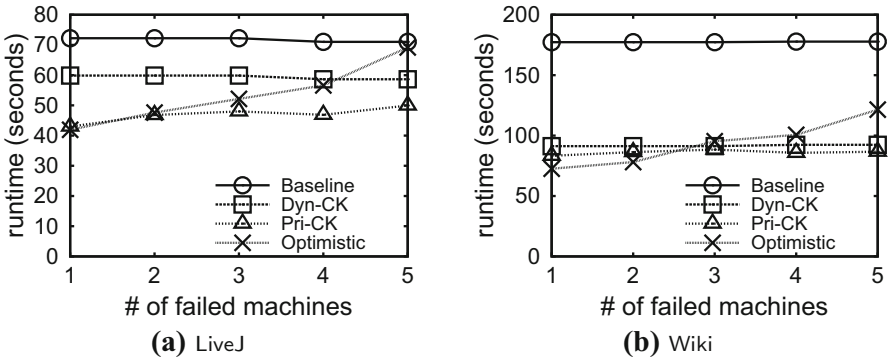

Fig. 13 The impact of the number of failed machines (Shortest Paths)

In particular, Optimistic is slightly better than our Pri-CK if a small number of machines are failed because the former does not archive any checkpoint data. However, when increasing the number of failed machines, the performance of Optimistic degrades seriously since most of data are lost. By contrast, benefitting from the priority checkpointing, the overall runtime of Pri-CK increases slightly.

6 Related work

Fault-tolerance is a core component for iterative graph processing systems. This section first lists some representative graph systems and then discusses the applicability of our proposals. Afterwards, we summarize fault-tolerance techniques used in today's systems to distinguish our work from them.

6.1 Large-scale graph processing systems

The widely used iterative graph algorithms and the rapid increase of data volume have attracted both industry and academia to develop highly scalable graph systems. Since Pregel [17] by Google is proposed as an early framework, a lot of variants have been developed to enhance performance from perspectives of graph partitioning [6,10], dynamic load balance [15,19,22], asynchronous computation [10,27,33], block-centric vertex update [24,30], and I/O-efficiency [5,26,34]. Besides, some general-purpose dataflow frameworks [1,3], can also support iterative graph mining.

Our proposals can be easily integrated into most of systems above, excluding asynchronous variants since they remove the synchronous barrier. On the other hand, the priority idea has been used to accelerate underlying graph computations in [32]. But in this paper, we employ it to speed up archiving data. Also, our priority is computed automatically, rather than using a user-specified function [32].

6.2 Fault-tolerance techniques

In systems mentioned above, efforts on fault-tolerance basically fall into three categories, and we introduce them as follows.

Checkpoint-based solutions As an early fault-tolerance technique, checkpointing proposed in Pregel has been widely used in many systems due to its simplicity. There exist two important branches to improve its performance in terms of archiving and recovering operations, respectively.

From the perspective of archiving data, Spark [3] first separates static data (i.e., edges) and dynamic data (i.e., vertices and messages), to reduce archived data to the latter only. Xue et al. further reduce it to vertices only by decomposing the computation logic into vertex update and message generation [29]. In this way, messages can be re-generated based on archived vertex states. However, it only works for graphs with extremely large degree. This is because in this scenario, the time of archiving vertices occupies only a small proportion of the overall runtime which is mainly dominated by sending messages along a large number of edges. On the other hand, Xu et al. focus on writing checkpoint along with the generation or computation of vertices [28]. This unblocking approach can partially overlap CPU and I/O operations to reduce the idle time. But it does not work well as expected based on reported experimental results in many cases. This is largely because that the data generating/consuming rate is usually greater than the checkpoint writing rate, which limits the effectiveness of the overlapping mechanism. Different from these techniques above, our proposals can significantly drop the volume of archived data by smartly deciding the checkpointing interval and selecting vertices in priority.

Another research line is to improve the recovery efficiency. To this end, recomputation workload is confined to lost data on failed machines, and data on surviving machines will not be rolled back [17,23,28]. Furthermore, the recovery workload can be re-assigned onto multiple machines to be performed in parallel [23,28]. Since our work only focuses on optimizing the archiving overhead, these techniques are complementary to our proposals.

Besides, some checkpointing variants can be performed asynchronously by removing the global barrier [16,25], which are tailored for asynchronous engines. It is beyond the scope of this paper since most systems work in synchronous manner and the gains due to asynchronous engines are not so much in many cases [13].

Lineage-based solutions Spark [3] employs a lineage method to track the dependency of data [31], instead of recording data themselves. Failures can be recovered by recomputing based on lineage. Since lineage as metadata is much smaller than graph states, it can significantly save storage space and network bandwidth. Nevertheless, for iterative algorithms, checkpointing is still required to cut a long lineage, in order to accelerate recovery.

Reactive solutions Reactive recovery solutions can directly recover failures without checkpointing. That reduces the impact on *failure-free* execution to be zero, and hence attracts a lot of attention. This can be achieved by replicating data [7,18], but memory resources may be quickly exhausted [34]. Another implementation is to carefully

design an algorithm-specified compensation function [20], which is usually a nontrivial task [28].

7 Conclusion

This paper proposes two new adaptive and I/O-efficient checkpointing solutions to tolerate failures in Pregel-like systems. Different from existing work, our solution can dynamically change the checkpointing interval based a cost-analysis model and/or archive selected vertices by computing their priorities. In experiments, we show that our proposals obviously outperform the up-to-date techniques.

Acknowledgements This work was supported by the National Natural Science Foundation of China (61472071, 61433008, 61528203, 61272179, and 61602103), and the U.S. NSF Grant CNS-1217284. Zhigang Wang was a visiting student at UMass Amherst, supported by China Scholarship Council, when this work was performed. Authors are also grateful to anonymous reviewers for their constructive comments.

References

1. Apache flink. https://flink.apache.org/
2. Apache hadoop. http://hadoop.apache.org/
3. Apache spark. http://spark.apache.org/
4. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Proceedings of the 7th International Conference on the World Wide Web, pp. 107–117. Elsevier, Amsterdam (1998)
5. Bu, Y., Borkar, V., Jia, J., Carey, M.J., Condie, T.: Pregelix: Big (ger) graph analytics on a dataflow engine. Proc. VLDB Endow. **8**(2), 161–172 (2014)
6. Chen, R., Shi, J., Chen, Y., Chen, H.: Powerlyra: differentiated graph computation and partitioning on skewed graphs. In: Proceedings of EuroSys, p. 1. ACM, New York (2015)
7. Chen, Z.: Algorithm-based recovery for iterative methods without checkpointing. In: Proceedings of HPDC, pp. 73–84. ACM, New York (2011)
8. Daly, J.T.: A higher order estimate of the optimum checkpoint interval for restart dumps. Future Gen. Comput. Syst. **22**(3), 303–312 (2006)
9. Ewen, S., Tzoumas, K., Kaufmann, M., Markl, V.: Spinning fast iterative data flows. Proc. VLDB Endow. **5**(11), 1268–1279 (2012)
10. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: distributed graph-parallel computation on natural graphs. In: Proceedings of OSDI, vol. 12, p. 2 (2012)
11. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: graph processing in a distributed dataflow framework. In: Proceedings of OSDI, pp. 599–613 (2014)
12. Giraph. http://giraph.apache.org/
13. Han, M., Daudjee, K., Ammar, K., Özsu, M.T., Wang, X., Jin, T.: An experimental comparison of pregel-like graph processing systems. Proc. VLDB Endow. **7**(12), 1047–1058 (2014)
14. Katz, L.: A new status index derived from sociometric analysis. Psychometrika **18**(1), 39–43 (1953)
15. Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., Kalnis, P.: Mizan: a system for dynamic load balancing in large-scale graph processing. In: Proceedings of Eurosys, pp. 169–182. ACM, New York (2013)
16. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
17. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of SIGMOD, pp. 135–146. ACM, New York (2010)
18. Pundir, M., Leslie, L.M., Gupta, I., Campbell, R.H.: Zorro: zero-cost reactive failure recovery in distributed graph processing. In: Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC), pp. 195–208. ACM, New York (2015)

19. Salihoglu, S., Widom, J.: GPS: a graph processing system. In: Proceedings of SSDBM, p. 22. ACM, New York (2013)
20. Schelter, S., Ewen, S., Tzoumas, K., Markl, V.: 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, pp. 1919–1928. ACM, New York (2013)
21. Seo, S., Yoon, E.J., Kim, J., Jin, S., Kim, J.S., Maeng, S.: Hama: an efficient matrix computation with the mapreduce framework. In: CloudCom, pp. 721–726. IEEE, Washington (2010)
22. Shang, Z., Yu, J.X.: Catch the wind: graph workload balancing on cloud. In: Proceedings of ICDE, pp. 553–564. IEEE, New York (2013)
23. Shen, Y., Chen, G., Jagadish, H., Lu, W., Ooi, B.C., Tudor, B.M.: Fast failure recovery in distributed graph processing systems. Proc. VLDB Endow. **8**(4), 437–448 (2014)
24. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From think like a vertex to think like a graph. Proc. VLDB Endow. **7**(3), 193–204 (2013)
25. Wang, Z., Gao, L., Gu, Y., Bao, Y., Yu, G.: A fault-tolerant framework for asynchronous iterative computations in cloud environments. In: Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC), pp. 71–83. ACM, New York (2016)
26. Wang, Z., Gu, Y., Bao, Y., Yu, G., Yu, J.X.: Hybrid pulling/pushing for i/o-efficient distributed and iterative graph computing. In: Proceedings of SIGMOD, pp. 479–494. ACM, New York (2016)
27. Xie, C., Chen, R., Guan, H., Zang, B., Chen, H.: Sync or async: time to fuse for distributed graph-parallel computation. In: Proceedings of PPoPP, pp. 194–204. ACM, New York (2015)
28. Xu, C., Holzemer, M., Kaul, M., Markl, V.: Efficient fault-tolerance for iterative graph processing on distributed dataflow systems. In: Proceedings of ICDE, pp. 613–624. IEEE, New York (2016)
29. Xue, J., Yang, Z., Qu, Z., Hou, S., Dai, Y.: Seraph: an efficient, low-cost system for concurrent graph processing. In: Proceedings of HPDC, pp. 227–238. ACM, New York (2014)
30. Yan, D., Cheng, J., Lu, Y., Ng, W.: Blogel: a block-centric framework for distributed computation on real-world graphs. Proc. VLDB Endow. **7**(14), 1981–1992 (2014)
31. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of NSDI, pp. 2–2. USENIX Association, Berkeley (2012)
32. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Priter: a distributed framework for prioritizing iterative computations. IEEE Trans. Parallel Distrib. Syst. **24**(9), 1884–1893 (2013)
33. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation. TPDS **25**(8), 2091–2100 (2014)
34. Zhou, C., Gao, J., Sun, B., Yu, J.X.: Mocgraph: scalable distributed graph processing using message online computing. Proc. VLDB Endow. **8**(4), 377–388 (2014)