Replication-based Fault-tolerance for Large-scale Graph Processing

Peng Wang† Kaiyuan Zhang† Rong Chen† Haibo Chen† Haibing Guan§

Shanghai Key Laboratory of Scalable Computing and Systems
†*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*
§*Department of Computer Science, Shanghai Jiao Tong University*

Abstract

The increasing algorithm complexity and dataset sizes necessitate the use of networked machines for many graph-parallel algorithms, which also makes fault tolerance a must due to the increasing scale of machines. Unfortunately, existing large-scale graph-parallel systems usually adopt a distributed checkpoint mechanism for fault tolerance, which incurs not only notable performance overhead but also lengthy recovery time.

This paper observes that the vertex replicas created for distributed graph computation can be naturally extended for fast in-memory recovery of graph states. This paper proposes Imitator, a new fault tolerance mechanism, which supports cheap maintenance of vertex states by replicating them to their replicas during normal message exchanges, and provides fast in-memory reconstruction of failed vertices from replicas in other machines. Imitator has been implemented by extending Hama, a popular open-source clone of Pregel. Evaluation shows that Imitator incurs negligible performance overhead (less than 5% for all cases) and can recover from failures of more than one million of vertices with less than 3.4 seconds.

Keywords—graph-parallel system; fault-tolerance;

1 Introduction

Graph-parallel abstraction has been widely used to express many machine learning and data mining (MLDM) algorithms, such as topic modeling, recommendation, medical diagnosis and natural language processing [1], [2], [3], [4]. With the algorithm complexity and dataset sizes continuously increase, it is now a common practice to run many MLDM algorithms on a cluster of machines. For example, Google has used hundreds to thousands of machines to run some MLDM algorithms [5], [6], [7].

Many graph algorithms can be programmed by following the "think as a vertex" philosophy [5], by coding graph computation as a vertex-centric program that processes vertices in parallel and communicates along edges. Typically, many MLDM algorithms are essentially iterative computation that iteratively refines input data until a convergence condition is reached. Such iterative and convergence-oriented computation has driven the development of many graph-parallel systems, including Pregel [5] and its open-source clones [8], [9], Trinity [10], GraphLab [11] and PowerGraph [12].

Running graph-parallel algorithms on a cluster of machines essentially faces a fundamental problem in distributed systems: fault tolerance. With the increase of problem sizes (and thus execution time) and machine scales, the failure probability of machines would increase as well. Currently, most graph-parallel systems use a checkpoint-based approach. During computation, the runtime system will periodically save the runtime states into a checkpoint on some reliable global storage, e.g., a distributed file system. When some machines crash, the runtime system will reload the previous computational states from the last checkpoint and then restart the computation. Example approaches include synchronous checkpoint (e.g., Pregel and PowerGraph) and asynchronous checkpoint using the Chandy-Lamport algorithm [13] (e.g., Distributed GraphLab [11]). However, as the processes of checkpoint and recovery require saving and reloading from slow persistent storage, such approaches incur notable performance overhead during normal execution as well as lengthy recovery time from a failure. Consequently, though most existing systems have been designed with fault tolerance support, they are disabled during production run by default.

This paper observes that many distributed graph parallel systems require creating replicas of vertices to provide local access semantics such that graph computation can be programmed as accessing local memory [14], [11], [12]. Such replicas can be easily extended to ensure that there are always at least *K+1* replicas

(including master) for a vertex across machines, in order to tolerate K machine failures.

Based on this observation, Imitator proposes a new approach that leverages existing vertex replication to tolerate machine failures, by extending existing graph-parallel systems in three ways. First, Imitator extends existing graph loading phase with fault tolerance support, by explicitly creating replicas for vertices without replication. Second, Imitator maintains the freshness of replicas by synchronizing the *full states* of a master vertex to its replicas through extending normal messages. Third, Imitator extends the graph-computation engine with fast detection of machine failures through monitoring vertex states and seamlessly recovers the crashed tasks from replicas in multiple machines in a parallel way, inspired by the RAMCloud approach [15].

Imitator uses a randomized approach to locating replicas for fault tolerance in a distributed and scalable fashion. To balance load, a master vertex selects several candidates at random and then chooses among them using more detailed information, which provides near-optimal results with small cost. Imitator currently supports two failure recovery approaches. The first approach, which is called Rebirth based recovery, recovers graph states on a new backup machine when a hot-standby machine for fault tolerance is available. The second one, the Migration based recovery, distributes graph states of the failed machines to multiple surviving machines.

We have implemented Imitator by extending Hama [9], a popular open-source clone of Pregel [5]. To demonstrate the effectiveness and efficiency of Imitator, we have conducted a set of experiments using four popular MLDM algorithms on a 50-node EC-2 like cluster (200 CPU cores in total). Experiments show that Imitator can recover from one machine failure in around 2 seconds. Performance evaluation shows that Imitator incurs an average of 1.2% (ranging from -0.6% to 3.7%) performance overhead for all algorithms and datasets. The memory overhead from additional replicas is also modest.

This paper makes the following contributions:

- A comprehensive analysis of current checkpoint-based fault tolerance mechanisms for graph-parallel computation model (Section 2).
- A new replication-based fault tolerance approach for graph computation (Section 3, Section 4 and Section 5).
- A detailed evaluation that demonstrates the effectiveness and efficiency of Imitator (Section 6).

2 Background and Motivation

This section first briefly introduces checkpoint-based fault tolerance in typical graph-parallel systems. Then, we examine performance issues during both normal computation and recovery.

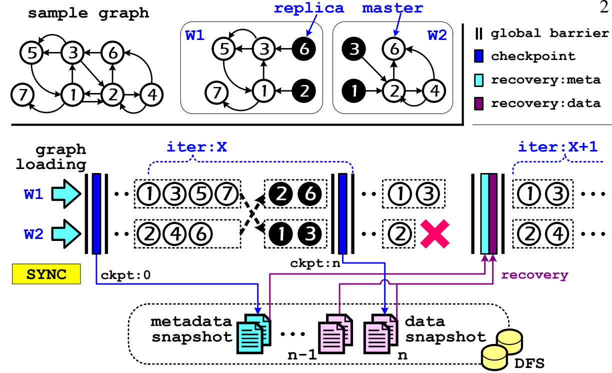

Fig. 1: *The sample of checkpoint-based fault tolerance.*

2.1 Graph-parallel Execution

Many existing graph-parallel systems usually provide a shared memory abstraction[1] to a graph program. To achieve this, replicated vertices (vertex 1, 2, 3 and 6 in Fig. 1) are created in machines where there are edges connecting to the original master vertex. To enable such an abstraction, a master vertex synchronizes its states to its replicas either synchronously or asynchronously through messages.

The scheduling of computation on vertices can be synchronous (SYNC) or asynchronous (ASYNC). Fig. 1 illustrates the execution flow of synchronous mode on a sample graph, which is divided into two nodes (i.e., machines). Vertices are evenly assigned to two nodes with ingoing edges, and replicas are created for edges spanning nodes. In the synchronous mode, all vertices are iteratively executed in a fixed order within each iteration. A global barrier between consecutive iterations ensures that all vertex updates in current iteration are simultaneously visible in next iteration for all nodes through batched messages. The computation on vertex in the asynchronous mode is scheduled on the fly, and uses the new states of neighboring vertices immediately without a global barrier.

Some graph-parallel systems such as Trinity [10], PowerGraph [12], GRACE [16] and Giraph++ [17], provide both execution modes, but usually use synchronous computation as the default mode. Hence, this paper only considers synchronous mode. How to extend Imitator to asynchronous execution will be our future work.

2.2 Checkpoint-based Fault Tolerance

Existing graph-parallel systems implement fault tolerance through distributed **checkpointing** for both synchronous and asynchronous modes. After loading a graph, each node stores an immutable graph topology of

1. Note that this is a restricted form of shared memory such that a vertex can only access its neighbors using shared memory abstraction.

its own graph partition to a metadata snapshot on the persistent storage. Such information includes adjacent edges and the location of replicas. During execution, each node periodically logs updated data of its own partition to incremental snapshots on the persistent storage, such as new values and states of vertices and edges. For synchronous mode, all nodes will simultaneously do logging for all vertices in the global barrier to generate a consistent snapshot. While for asynchronous mode, all nodes initiate logging at fixed intervals, and perform a consistent asynchronous snapshot based on the Chandy-Lamport algorithm [13]. The checkpoint frequency can be selected based on the mean time to failure model [18] to balance the checkpoint cost against the expected recovery cost. Upon detecting any node failures, the graph states will be recovered from the last completed checkpoint. During recovery, all the nodes first reload the graph topology from the metadata snapshot in parallel and then update states of vertices and edges through data snapshots. Fig. 1 illustrates an example of checkpoint-based fault tolerance for synchronous mode.

2.3 Issues of Checkpoint-based Approach

Though many graph-parallel systems provide checkpoint-based fault tolerance support, it is disabled by default due to notable overhead during normal computation and lengthy recovery time. To estimate checkpoint and recovery cost, we evaluate the overhead of checkpoint (Imitator-CKPT) based on Apache Hama [9] [2], an open-source clone of Pregel. Note that Imitator-CKPT is several times faster than Hama's default checkpoint mechanism (up to 6.5X for Wiki dataset [19]), as it further improves Hama through vertex replication to *incrementally* launch checkpoint and avoids storing *messages* in snapshot. Further, Imitator-CKPT only records the necessary states according to the behavior of graph algorithms. For example, Imitator-CKPT skips edge data for PageRank. Hence, Imitator-CKPT can be viewed as an optimal case of prior checkpoint-based approaches.

In the rest of this section, we will use Imitator-CKPT to illustrate the issues with checkpoint-based fault tolerance on a 50-node EC-2 like cluster[3].

2.3.1 Checkpointing

Checkpointing requires time-consuming I/O operations to create snapshots of updated data on a globally visible persistent storage (we use HDFS [20] here). Fig. 2(a) illustrates the performance cost of one checkpoint for different algorithms and datasets. The average execution time of one iteration without checkpointing is also provided as a reference. The relative performance overhead of checkpointing for LJournal and Wiki is relatively small, since HDFS is more friendly to writing large data. Even for the best case (i.e., Wiki), creating one checkpoint still incurs more than 55% overhead.

Fig. 2(b) illustrates an overall performance comparison between turning on and off checkpointing on Imitator-CKPT for PageRank with the LJournal dataset [21] by 20 iterations. We configure Imitator-CKPT using HDFS to store snapshots and using different intervals from 1 to 4 iterations. Checkpointing snapshots to HDFS is not the only cause of overhead. The imbalance of global barrier also contributes a notable fraction of performance overhead, since checkpoint operation must execute in the global barrier. In addition, though decreasing the frequency of intervals can reduce overhead, it may result in snapshots much earlier than the latest iteration completed before the failure, and increase the recovery time due to replaying a large amount of missing computation. The overall performance overhead for intervals 1, 2, and 4 iteration(s) reaches 89%, 51% and 26% accordingly. Hence, such a significant overhead becomes the main reason to the limited usage of checkpoint-based fault tolerance for graph-parallel models in practice.

2.3.2 Recovery

Though most fault tolerance mechanisms focus on minimizing overhead in logging, the time for recovery is also an important metric of fault tolerance. The poor performance and scalability in recovery is another issue of checkpoint-based fault tolerance. In checkpoint-based recovery, a new node for recovery needs to reload states of the crashed node in last snapshot from the persistent storage or even through network. The time for recovery are mainly limited by the I/O performance of the newbie node. Even worse, optimizations in checkpointing, such as incremental snapshot and lower frequency of intervals, may further increase the recovery time.

Fig. 2(c) illustrates a comparison between the average execution time of one iteration and recovery on Imitator-CKPT for PageRank with the LJournal dataset [21]. The recovery consists of three steps, including *(reload)*ing (meta)data, *(reconstruct)*ing in-memory graph states, and *(replay)*ing the missing computation. The reloading from snapshots on persistent storage incurs the major overhead, since it only utilizes the I/O resources in single node.

In addition, a *standby* node for recovery may not always be available, especially in a resource-scarce in-house cluster. It is also impractical to wait for rebooting of the crashed node. This constrains the usage scenario of such an approach. Further, as it only enables to

2. We extended and refined Hama's checkpoint and recovery mechanism as it currently does not support completed recovery and without optimizations.

3. Detailed experimental setup can be found in section 6.1

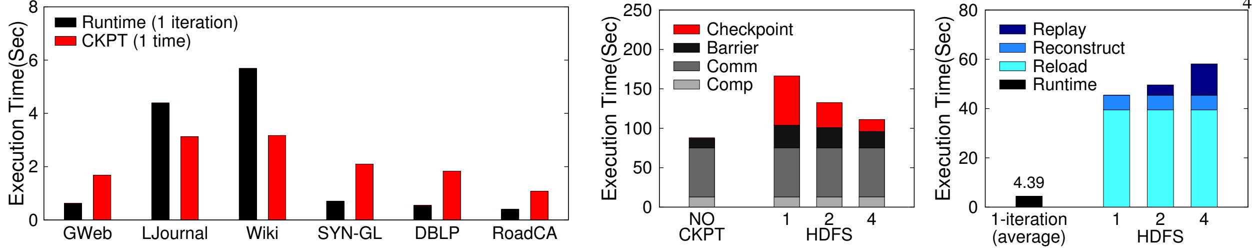

Fig. 2: *(a) The performance cost of once checkpointing in Imitator-CKPT for different algorithms and datasets. (b) The breakdown of overall performance overhead of checkpoint-based fault tolerance for PageRank algorithm with LJournal dataset using different configuration. (c) The breakdown of recovery time for PageRank algorithm with LJournal dataset using different configuration.*

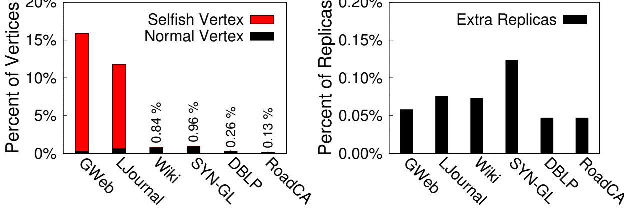

Fig. 3: *(a) The percent of vertices without replicas, including normal and selfish vertex. (b) The percent of extra replicas for fault tolerance.*

migrate the workload on crashed node to a single surviving node, it may result in significant load imbalance and performance degradation of normal execution after recovery.

3 Replication-based Fault Tolerance

This section first identifies challenges and opportunities in providing efficient fault tolerance, and then describes the design of Imitator.

3.1 Challenges and Opportunities

Low Overhead in Normal Execution: Compared to data-parallel computing models, the dependencies between vertices in graph-parallel models demand a fine-grained fault tolerance mechanism. Low overhead re-execution [22] and coarse-grained transformation [23] can hardly satisfy such requirement. In contrast, checkpoint-based fault tolerance in existing graph-parallel systems sacrifices the performance of normal execution for fine-grained logging.

Fortunately, existing replicas for vertex computation in a distributed graph-parallel system open an opportunity for efficient fine-grained fault tolerance. Inspired from fault tolerance in distributed file system (e.g., GFS [24]), the replicas originally used for local access in vertex computation can be reused to backup data of vertices and edges, while the synchronization messages between a master vertex and its replicas can be reused to maintain the freshness of replicas.

To leverage vertex replicas for fault tolerance, it is necessary that each vertex has at least one replica; otherwise extra replicas for these vertices have to be created, which incurs additional overhead for communication during normal execution. Fig. 3(a) shows the percentage of vertices without replicas on a 50-node cluster for a set of graphs [21] using the default hash-based (random) partitioning. Only GWeb and LJournal contain more than 10% of such vertices, while others only contain less than 1% vertices. The primary source of vertices without replicas is selfish vertices, which have no outgoing edges (e.g., vertex 7 in Fig. 1). For most graph algorithms, the value of a selfish vertex has no consumer and only depends on ingoing neighbors. Consequently, there is no need to create extra replica for selfish vertices. In addition, the performance cost in communication depends on the number of replicas, which is several times the number of vertices. Fig. 3(b) illustrates the percentage of extra replicas required for fault tolerance regardless of selfish vertices, which is less than 0.15% for all dataset.

Fast Recovery from Failure: For checkpoint-based fault tolerance, recovery from a snapshot on the persistent storage cannot harness all resources in the cluster. The I/O performance of a single node becomes the bottleneck of recovery, which does not scale with the increase of nodes. Further, a checkpoint-based fault tolerance mechanism also depends on standby nodes to take over the workload on crashed nodes.

Fortunately, the replicas of a vertex scattered across the entire cluster provide a new opportunity to recover a node failure in parallel, which is inspired from the fast recovery in DRAM-based storage system (e.g., RAMCloud [15]). Actually, the time for recovery may be less with more nodes if the replicas selected to recovery can be evenly assigned to all nodes.

In addition, an even distribution of replicas for vertices on crashed node further provides the possibility to support migrating the workload on crashed nodes to all surviving nodes without using additional standby nodes for recovery. This may also reserve the load balance of execution after recovery.

3.2 Overall Design of Imitator

Based on the above observation, we propose Imitator, a replication-based fault tolerance scheme for graph-

Algorithm 1: Imitator Execution Model

```
   Input: Date Graph G = (V, E, D)
   Input: Initial active vertex set 𝕍
1  if is_newbie() then                      // newbie node
2  |   newbie_enter_leave_barrier()
3  |   iter = newbie_rebirth_recovery()
4  while iter < max_iter do
5  |   compute()
6  |   send_msgs()
7  |   state = enter_barrier()
8  |   if state.is_fail() then              // node failure
9  |   |   rollback()
10 |   |   if is_rebirth() then rebirth_recovery(state)
11 |   |   else migration_recovery(state)
12 |   |   continue
13 |   else                                 // normal execution
14 |   |   commit_state()
15 |   |   iter ++
16 |   state = leave_barrier()
17 |   if state.is_fail() then              // node failure
18 |   |   if is_rebirth() then rebirth_recovery(state)
19 |   |   else migration_recovery(state)
```

parallel systems. Unlike prior systems, Imitator employs replicas of a vertex to provide fault tolerance rather than periodically checkpointing graph states. The replicas of a vertex inherently provide a remote consistent backup, which are synchronized during each global barrier. When a node crashes, its workload (vertices and edges) will be reconstructed on a standby node or evenly migrated to all surviving nodes.

Note that Imitator assumes a fail-stop model where a machine crash won't cause wild or malicious changes to other machines. How to extend Imitator to support more complicated faults like byzantine faults [25], [26] will be our future work.

Execution Flow: Imitator extends existing synchronous computation with detection of potential node failures and seamless recovery. As shown in Algorithm 1, each iteration consists of three steps. First, all vertices are updated using the messages from neighboring vertices in the computation phase (line 5). Secondly, an update of vertex states is synchronized from a master vertex to its replicas in the communication phase through message passing (line 6). Note that all messages have been received before entering the global barrier. Finally, all new vertex states are consistently committed within a global barrier (line 14 and 15).

Imitator employs a highly available and persistent distributed coordination service (e.g., Zookeeper [27]) to provide barrier-based synchronization and distributed shared states among workers. Node failures will be

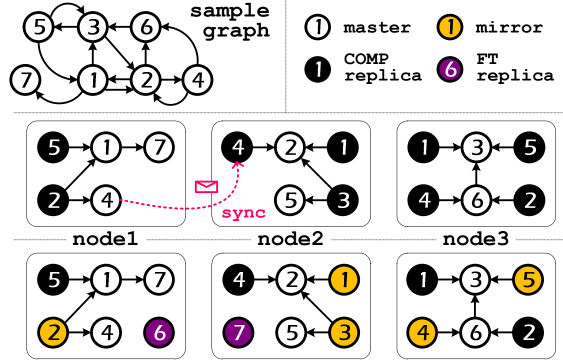

Fig. 4: *A sample of replicas in Imitator.*

detected before (line 7) and after (line 16) the global barrier. Before recovery, each worker must enforce the consistency of its local graph states. If a failure occurs before the global barrier, each surviving node should roll back its states (line 9) and execution flows (line 12) to the beginning of the current iteration, since messages from crashed nodes may be lost. Imitator provides two alternative recovery mechanisms: *Rebirth* and *Migration*. For Rebirth, a standby node will join the global synchronization (line 2), and reconstruct the graph states of the crashed nodes from all surviving nodes (line 3, 10 and 18). For Migration mode, the vertices on crashed nodes will be reconstructed from all surviving nodes (line 11 and 19).

4 Managing Replicas

Many graph-parallel systems construct a local graph on each node by *replicating* vertices to avoid remote accesses during vertex computation. As shown in the middle of Fig. 4, vertices are evenly assigned to three machines with their ingoing edges, and replicas are created to provide local vertex accesses. These replicas will be synchronized with their master vertices to maintain consistency. Imitator reuses these *replicas* as consistent backups of vertices for recovery from failures. However, replication-based fault tolerance requires that every vertex has at least a replica with exactly the same states with the master vertex, while currently replicas are only with partial states. Further, not all vertices have replicas.

This section describes extensions for fault-tolerance oriented replication, creating full state replicas and an optimization for selfish vertices. Here, we only focus on creating at least one replica to tolerate one machine failure; creating more replicas can be done similarly.

4.1 Fault Tolerant (FT) Replica

Original replication for local accesses may cause some vertices to have no replicas. For example, the internal vertex (e.g., vertex 6 in Fig. 4) has no replica as all its edges are stored at the same node. A failure of node 3 may cause an irrecoverable state for vertex 6.

For such an internal vertex, Imitator creates an additional *fault tolerant (FT) replica* on another machine when loading the graph. There is no constraint on the location of these replicas, which provides an opportunity to balance the workload among nodes to hide the performance overhead. Before assignment, the number of replicas and internal vertices are exchanged among nodes. Each node proportionally assigns FT replicas to the rest nodes. For example, vertex 6 has no computation replicas and its additional FT replica is assigned to node 1, which has fewer replicas, as shown in Fig. 4.

4.2 Full-state Replica (Mirror)

The replica to provide local access does not have full states to recover the master vertex, such as the location of replicas. Some algorithms (e.g. SSSP) also need to associate data to edges, such as the weight of edge. However, it is not efficient to upgrade all replicas to be homogeneous with their masters, which will cause excessive memory and communication overhead. Imitator selects one replica to be the homogeneous replica with the master vertex, namely *mirror*. All data of ingoing edges will be synchronized from master to mirror.

Most additional states in mirrors are static, such as the location of replicas and the weight of ingoing edges, which are replicated during graph loading. The rest states are dynamic, such as whether a vertex is active or inactive in next iteration, and should be transferred with synchronization message from master to mirror in each iteration.

As mirrors are responsible to recover their masters on a crashed node, the distribution of mirrors may affect the scalability of recovery. Since the locations of mirrors are restricted by the locations of all candidate replicas, each machine adopts a greedy algorithm to evenly assign mirrors on other machines independently: each machine maintains a counter of existing mirrors, and the master always assigns mirrors to replicas whose hosting machine has least mirrors so far.

Note that the FT replica is always the mirror of vertex. As shown in the bottom part of Fig. 4, the mirrors of vertex 1 and 4 on node 1 are assigned to node 2 and 3 accordingly, and the mirror of vertex 7 has to be assigned to node 2.

4.3 Optimizing Selfish Vertices

The main overhead of Imitator during normal execution is from the synchronization of additional FT replicas. According to our analysis, many vertices requiring FT replicas have no neighboring vertices consuming their vertex data (selfish vertices). For example, vertex 7 has no outgoing edges in Fig. 4. Further, for some algorithms (e.g., PageRank), the new vertex data is computed only according to its neighboring vertices.

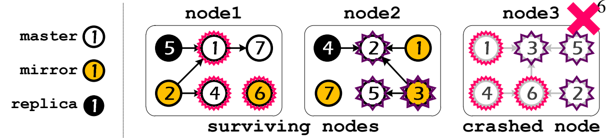

Fig. 5: *A sample of Rebirth recovery approach in Imitator.*

For such vertices, namely *selfish vertices*, Imitator only creates an FT replica for recovery, and never synchronizes them with masters in normal execution. During recovery, the static states of selfish vertices can be obtained from its FT replicas, and dynamic states can be re-computed using neighboring vertices.

5 Recovery

The main issue of recovery is knowing which vertices, either master vertices or other replicas, are assigned to the crashed node. A simple approach is adding a layer to store the location of each vertex. This, however, may become a new bottleneck during the recovery. Fortunately, when a master vertex creates its replicas, it knows its replicas' locations. Thus, by storing its replicas' locations, a master vertex knows if its replicas are assigned to the crashed node. As its mirrors are responsible for recovery when a master vertex crashes, a master needs to replicate this information to its mirrors as well.

During recovery, each surviving node will check in parallel whether master vertices or replica vertices related to the failed nodes have been lost and reconstruct the states accordingly. As each remaining node has the complete information of its related graph states, such checking and reconstruction can be done in a decentralized way and in parallel.

Imitator provides two strategies for recovery: Rebirth based recovery, which recovers graph states in crashed nodes to standby ones; Migration based recovery, which scatters vertices on the crashed nodes to surviving ones.

5.1 Rebirth Based Recovery

During recovery, the location information of vertices will be used by master vertices or mirrors to check whether there are some vertices to recover. Rebirth based recovery comprises three steps: *Reloading*, where the surviving nodes send the recovery messages to the standby nodes to help it recover states; *Reconstruction*, which reconstructs the states (mainly the graph topology) necessary for computation; and *Replay*, which redoes some operations to get the latest states of some vertices.

5.1.1 Reloading

Through checking the location of its replicas, a master vertex will know whether there are any of its replicas located in the crashed nodes. If so, the master vertex

will generate messages to recover such replicas. If a master vertex is on the crashed nodes, its mirror will be responsible to recover this crashed master. Based on this rule, each surviving node can just use the information from its local vertices to decide whether it needs to participate in the recovery process.

For the sample graph in Fig. 5, node 3 crashed during computation. After a new standby node (i.e., machine) awakes node 1 and node 2 from the barrier operation, these two nodes will check whether they have some vertices to recover. Node 1 will check its master vertices (master vertex 1, 4, and 7), and find that there are some replicas (replica 1 and mirror 4) on the crashed node. Hence, it needs to generate two recovery messages to recover replica 1 and mirror 4 on the new node. Node 1 will also check its mirrors to find whether there is any mirror whose master was lost. It then finds that the master of vertex 6 was lost, and thus generates a message to recover master vertex 6. Node 2 will act the same as node 1.

The surviving nodes also need to send some global states to the newbie, such as the number of iterations so far. All the recovery messages are sent in a batched way when its number has exceeded a threshold.

5.1.2 Reconstruction

For the new machine, there are three types of states to reconstruct: the graph topology, runtime states of vertices, and global states (e.g., number of iterations so far). The last two types of states can be retrieved directly from recovery messages. The graph topology is a complex data structure, which is non-trivial to recover.

A simple way to recover the graph topology is using the raw edge information (the "point to" relationship between vertices) and redoing operations of building topology in the graph loading phase. In this way, after receiving all the recovery messages, the new machine will create vertices based on the messages (which contain the vertex types, the edges and the detailed states of a vertex). After creating all vertices, the new machines will use the raw edge information on each vertex to build the graph topology. One issue with this approach is that building graph topology can only start after creating all vertices. Further, due to the complex "point to" relationship between vertices, it is not easy to parallelize the topology building process.

To expose more parallelism, Imitator instead uses enhanced edge information for recovery. Since all vertices are stored in an array in each machine, the topology of a graph is represented by the array index. This means that if there is an edge from vertex A to vertex B, vertex B will have a field to store the index of vertex A in the array. Hence, if Imitator can ensure a vertex is placed at the same position of the vertex array in the new machine, reconstruction of graph topology can be done in parallel on all the surviving nodes.

To ensure this, Imitator also replicates the master's position to its mirror with other states in the graph loading phase. When a mirror recovers its master, it will create the master vertex and its edges, and then encode the vertex and the master position into the recovery message. On receiving the message, the new machine just needs to retrieve the vertex from the message and put it at the given position. Recovering replicas can be done in the same way.

Since every crashed vertex only needs one vertex to do the recovery, there is only one recovery message for one position. Thus, there is no contention on the array (which is thus lock-free) and can be done immediately when receiving the message. Hence, it is completely decentralized and can be done in parallel. Note that there is no explicit reconstruction phase for this approach because the reconstruction can be done during the reloading phase when receiving recovery messages.

5.1.3 Replay

Imitator can recover most states of a vertex directly from the recovery message, except the activation state, which cannot be timely synchronized between masters and mirrors. The reason is that a master vertex may be activated by some neighboring vertices that are not located on the same node. When a master replicates its states to its mirrors, the master may still not be activated by its remote neighbors. Hence, the activation state can only be recovered by replaying the activation operations. However, the neighboring vertex of a master vertex might also locate at the crashed node. As a result, a master vertex needs to replicate its activation information (which vertices it should activate) to its mirrors. A vertex (either master or mirror) doing recovery will attach the corresponding activation information to the recovery message. The new node will re-execute the activation operations according to these messages on all the vertices.

5.2 Migration Approach

When there are no standby nodes for recovery, Migration based recovery will scatter graph computation from the crashed nodes to surviving ones. Fortunately, the mirrors, which are isomorphic with their masters, provide a convenient way to migrate a master vertex from one node to another. Other data to be used by the new master in future computation can be retrieved from its neighboring vertices.

The Migration approach also consists of three steps: *Reloading*, *Reconstruction*, and *Replay*, of which the processes are only slightly different from the Rebirth approach. In the followings, we will use the example in

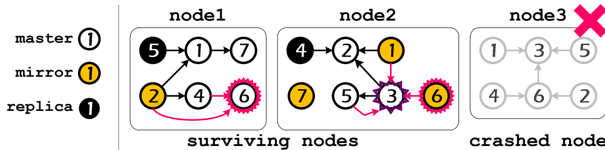

Fig. 6: *A sample of Migration recovery approach in Imitator.*

Fig. 6 as a running example to illustrate how Migration based recovery works. Here, we only describe recovering graph states to continue execution for simplicity. Creating additional replicas to retain the original fault tolerance level can be done similarly by creating an additional mirror when tolerating recurrent failures.

5.2.1 Reloading

The main differences between Rebirth and Migration for *reloading* is that mirror vertices will be "promoted" to masters and take over the computation tasks for future execution.

On detecting a failure, all surviving nodes will get the information about the crashed ones from the master node of a cluster. Surviving nodes will scan through all of their mirrors to find whose masters were on the crashed nodes. In Fig. 6, they are mirror 6 on node 1 and mirror 3 on node 2. These mirrors will be "promoted" as new masters.

After "promotion", all surviving nodes will broadcast the information of the "promoted" mirrors. The broadcast information will be used by other nodes as a hint to send the necessary data to the new masters. Such data includes:

- **New replicas:** Since the new master will be on a different node, new replicas of its neighbors on different nodes whose out-edge points to it are necessary to make the states of those neighbors accessible during computation. Replica 6 on node 2 in Fig. 6 illustrates this case.
- **Edges from old replicas to the new master:** If there is already a replica on the machine where the new master resides, a new replica is not necessary. Instead, the new edges will be sent to that node and will then be added to existing replicas. Edge from mirror 2 to master 6 on node 1 in Fig. 6 is one example of such case.
- **Edges between masters:** With the new masters, there may be some new edges between the masters, either between two new masters or between a new master and an old one. Edge from master 5 to master 3 on node 2 in Fig. 6 shows the case.

All surviving nodes will scan its masters, including both old masters and the newly "promoted" ones, check if some data should be reconstructed, and prepare the necessary information for the *reconstruction* phase.

5.2.2 Reconstruction

During *reconstruction*, all surviving nodes will assemble new graph states from the recovery messages sent in the *reloading* phase. After the *reconstruction* phase, the topology of the graph and most of the states of the vertices (both masters and replicas), are migrated to the surviving nodes.

5.2.3 Replay

The Migration approach also needs to fix the activation states for new masters. However, the Rebirth approach needs to fix such states for all recovered masters, while the Migration approach only needs to fix the states of newly promoted masters, which are only a small portion of all master vertices on one machine. Hence, we choose to treat these new masters specially instead of redoing the activation operation on all the vertices. Imitator checks whether a new master is activated by some of its neighbors or not. If so, Imitator will correct the activation states of the new master. When finishing the *Replay* phase, the surviving nodes can now resume the normal execution.

5.3 Additional Failure Models

5.3.1 Multiple Machine Failures

To tolerate multiple nodes failure at the same time, Imitator just needs to ensure that the number of mirrors for each vertex in Imitator is equal or larger than the expected number of machines to fail. When a single machine failure happens, if all mirrors participate the recovery, it is a waste of network bandwidth. Hence, during graph loading, each mirror is assigned with an ID; only the surviving mirrors with the lowest ID will do the recovery work. Since a mirror has the location information of other mirrors and the new coming node's logic ID of this job, mirrors need not communicate with each other to elect a mirror to do recovery.

5.3.2 Other Types of Failures

When a failure happens during the system is loading graph, since the computation has not started, we just restart the job. If a failure happens during recovery, such a failure is almost the same as the failure happening during the normal execution. Hence, we just restart the recovery procedure.

There is a single master for a cluster, and it is only in charge of job dispatching and failure handling. It has nothing to do with the job execution. Since the possibility of master failure is very small, and there are a lot of prior work to address the single master failure, we do not try to address the master failure in this paper.

TABLE 1: *A collection of input graphs, and the execution time on Hama and Imitator without fault tolerance on 50 nodes.*

Algorithm	Graph	$\|V\|$	$\|E\|$	Hama	w/o FT
PageRank	GWeb	0.87M	5.11M	17.0	12.2
	LJournal	4.85M	70.0M	280.5	86.7
	Wiki	5.72M	130.1M	482.6	120.7
ALS	SYN-GL	0.11M	2.7M	42.5	13.7
CD	DBLP	0.32M	1.05M	17.2	14.8
SSSP	RoadCA	1.97M	5.53M	295.3	341.4

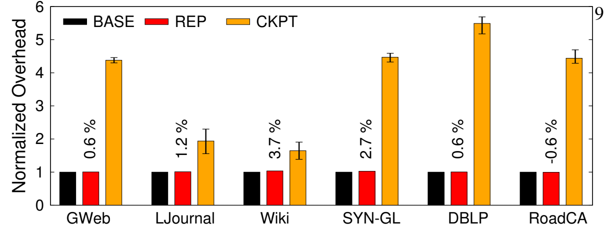

Fig. 7: *A comparison of runtime overhead between replication (REP) and checkpoint (CKPT) based fault tolerance over Imitator w/o fault tolerance (BASE).*

6 Evaluation

We have implemented Imitator based on Hama [9], an open source clone of Pregel implemented in Java, but extended Hama by vertex replication instead of pure message passing as the communication mechanism. The support of fault tolerance requires no source code changes to graph algorithms. To measure the efficiency of Imitator, we use four typical graph algorithms (PageRank, Alternating Least Squares (ALS), Community Detection (CD) and Single Source Shortest Path (SSSP)) to compare the performance and scalability of different systems and configuration. We also provide a case study to illustrate the effectiveness of Imitator by illustrating the execution of different recovery approaches under injected node failures.

6.1 Experimental Setup

All experiments are performed on a 50-node EC2-like cluster. Each node has four AMD Opteron cores, 10GB of RAM, and connected via a 1 GigE network. We use HDFS on the same cluster as the distributed storage layer to store input files and checkpoints.

Table 1 lists a collection of large graphs for our experiments. Most of them are from Stanford Large Network Dataset Collection [21]. The Wiki dataset is from [19]. The dataset for the ALS algorithm is synthetically generated by tools that used in the Gonzalez et al. [12]. The SSSP algorithm requires the input graph to be weighted. Since the RoadCA graph is not originally weighted, we synthetically assign a weight value to each edge, where the weight is generated based on a log-normal distribution ($\mu = 0.4, \sigma = 1.2$) from the Facebook user interaction graph [28].

6.2 Hama vs. Imitator's Baseline

As the baseline of Imitator is extended from Hama, we first compare the baseline performance of Imitator with that of Hama. As shown in Table 1, Imitator actually has better performance compared to Hama in all the applications except SSSP. For the three applications, Imitator always outperforms Hama and the speedup can reach up to 4X in the largest dataset Wiki.

Hama adopts the pure message passing to vertex communication, in which the message is simpler than the synchronization message between the master vertex and replica vertex in Imitator. However, supporting dynamic computation [11] is hard for a message passing system, but is quite natural for a replica-based one. With the support of dynamic computation, the total number of messages to send and vertex computations in Imitator is notably less than that in Hama for PageRank, ALS, and CD, which mainly contributes to the speedup.

For SSSP, a vertex only needs to activate its neighbors when its distance to the source vertex changes, so there is little chance for dynamic computation. As the message in Hama is simpler, Imitator is a little bit slower than the original Hama in SSSP.

Unless specified, we will use Imitator without fault tolerance as the baseline and Imitator-CKPT as the checkpoint-based fault tolerance system for comparison.

6.3 Runtime Overhead

Fig. 7 shows the runtime overhead due to applying different fault tolerance mechanisms on the baseline system. The overhead of Imitator is less than 3.7% for all algorithms with all datasets, while the overhead of the checkpoint-based fault tolerance system is very large, varying from 65% for PageRank on Wiki to 449% for CD on DBLP. Even using in-memory HDFS, checkpoint-based approach still incurs performance overhead from 33% to 163% partly due to the cross machine triple replication in HDFS. In addition, writing to memory also causes significant pressure on memory capacity and bandwidth to the runtime, occupying up to 42.1GB extra memory for SSSP on RoadCA.

The time of checkpointing once is from 1.08 to 3.17 seconds for different size of graphs, since the write operations to HDFS can be batched and are insensitive to the data size. The overhead of each iteration in Imitator is lower than 0.05 seconds, except 0.22 seconds for Wiki, which is still several tens of times faster than checkpointing.

6.4 Overhead Breakdown

Fig. 8(a) shows the extra replicas among all the replicas used for fault tolerance. The rates of extra replicas are all very small without selfish vertices, even the largest rate is only 0.12%. Fig. 8(b) shows the redundant messages among the total messages during the normal execution. Since the rate of extra replicas is very small,

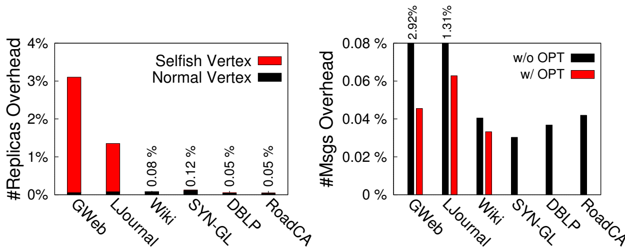

Fig. 8: *The overhead of (a) #replicas and (b) #msgs for Imitator w/ and w/o selfish optimization.*

TABLE 2: *The recovery time (seconds) of different approaches (Checkpoint, Rebirth and Migration).*

Algorithm	Graph	CKPT	Rebirth	Migration
PageRank	GWeb	8.17	2.08	1.20
	LJournal	41.00	8.85	2.32
	Wiki	55.67	14.12	3.40
ALS	SYN-GL	6.86	1.00	1.28
CD	DBLP	3.88	0.67	1.09
SSSP	RoadCA	12.06	2.27	1.57

the additional messages rate is very small, with only 2.92% for the worst case. When enabling the optimization for selfish vertices, the messages overhead is lower than 0.1%.

6.5 Efficiency of Recovery

Replication-based fault tolerance provides a good opportunity to fully utilize the entire resources of the cluster for recovery. As shown in Table 2, the replication-based recovery outperforms checkpoint-based recovery by up to 6.86X (from 3.93X) and 17.67X (from 3.55X) for Rebirth and Migration approaches accordingly. Overall, Imitator can recover 0.95 million and 1.43 million vertices (including replicas) from one failed node in just 2.32 and 3.4 seconds for LJournal and Wiki dataset accordingly.

For large graphs (e.g., LJournal and Wiki), the performance of Migration is relatively better than that of Rebirth, since it avoids data movement (e.g., vertex and edge values) in the reloading phase and distributes replaying operations to all surviving nodes rather than on the single new node. On the other hand, for small graphs (e.g., SYN-GL and DBLP), the performance of Rebirth is relative better than that of Migration, since there are multiple rounds of message exchanges in Migration. This causes slowdown to recovery, ranging from 28% to 63%, compared with Rebirth.

6.6 Scalability of Recovery

We evaluate the recovery scalability of Imitator for PageRank with the Wiki dataset using different numbers of nodes that participate in recovery. As shown in Fig. 9, both recovery modes scale with the increase of recovery machines, since all machines can participate the workload in the reloading phase. Because the local graph has been constructed in the reloading phase, there

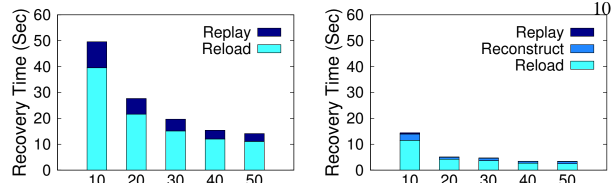

Fig. 9: *The recovery time of Rebirth (a) and Migrate (b) on Imitator for PageRank with the increase of nodes.*

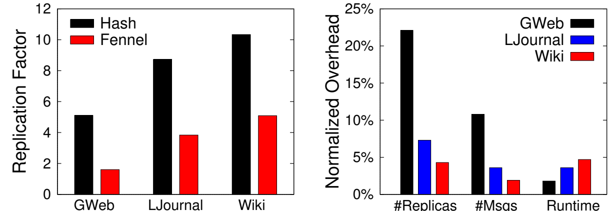

Fig. 10: *(a) The replication factor of different partitioning schemes. (b) The overhead of Imitator using Fennel algorithm.*

is no explicit reconstruction phase for the Rebirth mode. Further, the replay operations are only executed in new node for the Rebirth mode, while are distributed to all surviving nodes for the Migration mode.

6.7 Impact of Graph Partitioning

To analyze the impact of graph partitioning algorithms, we implement Fennel [29] on Imitator, which is a heuristic graph partitioning algorithm. As shown in Fig. 10(a), compared to the default Hash-based partitioning, Fennel significantly decreases the replication factor for all datasets, reaching 1.61, 3.84 and 5.09 for GWeb, LJournal and Wiki respectively.

Fig. 10(b) illustrates the overhead of Imitator under Fennel partitioning. Due to lower replication factor, Imitator requires more additional replicas for fault tolerance, which also result in the increase of message overhead. However, the runtime overhead is still small, ranging from 1.8% to 4.7%.

6.8 Handling Multiple Failures

When Imitator is configured to tolerate multiple node failures, there will be more extra replicas to add. The overhead tends to be larger. Fig. 11 shows the overall overhead when Imitator is configured to tolerate 1, 2 and 3 node failure(s). As shown in Fig. 11(a), the overhead of Imitator is less than 10% even when it is configured to tolerate 3 nodes failures simultaneously.

Fig. 11(b) shows the recovery time of the largest dataset, Wiki, when different numbers of nodes crashed. In Rebirth mode, since the surviving nodes need more messages to exchange when the crashed nodes increase, the time to send and receive recovery messages increases. However, the time to rebuild graph states and replay some pending operations is almost the same as that of a single node failure. Since Migration strategy harnesses the cluster resource for recovery, the time of

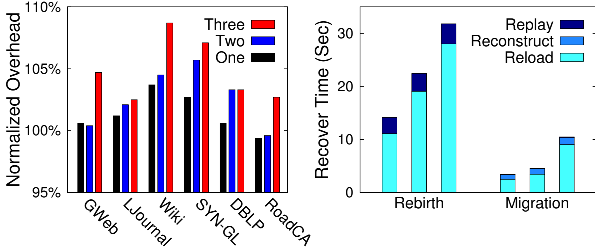

Fig. 11: *(a) The runtime overhead, and (b) The recovery time for tolerating 1, 2, and 3 machine(s) failure.*

TABLE 3: *Memory and GC behavior of Imitator with different fault tolerance setting for PageRank on Wiki*

Config	Max Cap(GB)	Max Usage(GB)	Young/Full GC Number	Young/Full GC Time (Sec))
w/o FT	3.85	2.76	40/15	13.7/13.4
FT/1	5.05	3.70	50/29	19.9/21.7
FT/2	6.24	4.51	55/29	23.6/26.1
FT/3	6.99	4.91	58/30	25.7/29.7

every operation in Migration is relatively small.

6.9 Memory Consumption

As Imitator needs to add extra replicas to tolerate faults, we also measure the memory overhead. We use *jstat*, a memory tool in JDK, to monitor the memory behavior of the baseline system and Imitator. Table 3 illustrates the result of one node of the baseline system and Imitator on our largest dataset Wiki. If Imitator is configured to tolerate one node failure during computation, the memory overhead is modest, the memory usages of the baseline system and Imitator is comparable.

6.10 Case Study

Fig. 12 presents a case study for running PageRank using LJournal dataset with none or one machine failure during the execution of 20 iterations. Different recovery strategies are applied to illustrate their performance. The symbols, BASE, REP, and CKPT/4, denote the execution of the baseline, replication and checkpoint-based fault tolerance systems without failure accordingly, where others illustrate the cases with a failure between the 6th and 7th iterations. Note that the interval of checkpointing is 4 iterations.

The scheme of failure detection is the same for all strategies, of which the time span is about 7 seconds. For the recovery speed, the Migration strategy, of which recovery time is about 2.6 seconds, is the fastest due to the fact that it harnesses all resources and minimizes data movements. The Rebirth strategy has a time span of 8.8 seconds. This still outperforms the 45 seconds recovery time of CKPT/4, which does the incremental checkpoint with an interval of four iterations, due to fully exploiting network resources and without accessing distributed file system.

After the recovery has finished, REP with Rebirth

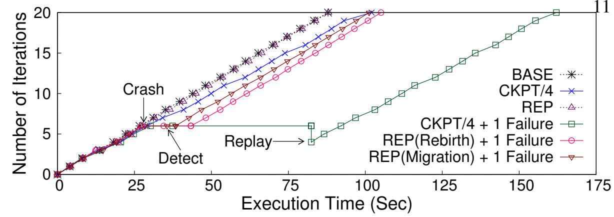

Fig. 12: *An execution of PageRank of LJournal with different fault tolerance settings. One failure occurs between the 6th iteration and the 7th iteration.*

can still execute at full speed, since the execution environment before and after the failure is the same in this approach. On the other hand, the REP with Migration is slower since the available computing resource has decreased, but only slightly. For the CKPT/4, it still has to replay 2 lost iterations after a long time recovery.

7 Related Work

Checkpoint-based fault tolerance is widely used in graph-parallel computation systems. Pregel [5] and its open-source clones [8], [9] adopt synchronous checkpoint to save the graph state to the persistent storage, including vertex and edge values, and incoming messages. GraphLab [11] designs an asynchronous alternative based on the Chandy-Lamport [13] snapshot to achieve fault tolerance. Trinity [10] and PowerGraph [12] provide both synchronous and asynchronous checkpointing for different modes.

Piccolo [30] is a data-centric distributed computation system, which provides user-assisted checkpoint mechanism to reduce runtime overhead. However, user needs to save additional information for recovery. MapReduce [22] and other data-parallel models [31] adopt simple re-execution to recover tasks on crashed machines, since they suppose all tasks are deterministic and independent. Graph-parallel models do not satisfy such assumptions. Spark [23] and Discretized Streams [32] propose a fault tolerant abstraction, namely Resilient Distributed Datasets (RDD), for coarse-grained operations on datasets, which only logs the transformation used to build a dataset (lineages) rather than the actual data. It is hard to apply RDD to graph-parallel models, since the computation on vertex is a fine-grained update.

Replication is widely used in large-scale distributed file systems [24], [20] and streaming systems [33], [34] to provide high availability and fault tolerance. In these systems, all replicas are full-time for fault tolerance, which may introduce high performance cost. RAMCloud [15] is a DRAM-based storage system, it achieves a fast recovery from crashes by scattering its backup data across the entire cluster and harnessing all resources of cluster to recover the crashed nodes. Distributed storage only provides simple abstraction (e.g., a key-value) and does not consider data dependency and

computation on data.

SPAR [14] is a graph-structured middleware to store social data for key-value stores. It also briefly mentions of storing more ghost vertices for fault tolerance. However, it does not consider the interaction among vertices, and only provides background synchronization and eventual consistency between master and replicas, which does not fit for graph-parallel systems.

8 Conclusion

This paper presented a replication-based approach called Imitator to provide low-overhead fault tolerance and fast crash recovery. The key idea of Imitator is leveraging and extending existing replication mechanism with additional mirrors and complete states as the master vertices, such that vertices in a failed machine can be reconstructed using states from its mirrors. Evaluation showed that Imitator incurs very small normal execution overhead, and provides fast crash recovery from failures.

9 Acknowledgment

We thank the anonymous reviewers for their insightful comments. This work is supported in part by Doctoral Fund of Ministry of Education of China (Grant No. 20130073120040), the Program for New Century Excellent Talents in University of Ministry of Education of China, Shanghai Science and Technology Development Funds (No. 12QA1401700), a foundation for the Author of National Excellent Doctoral Dissertation of PR China, China National Natural Science Foundation (No. 61303011) and Singapore NRF (CREATE E2S2).

References

[1] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” in *WWW*, 1998, pp. 107–117.
[2] J. Ye, J. Chow, J. Chen, and Z. Zheng, “Stochastic gradient boosted distributed decision trees,” in *ACM CIKM*, 2009, pp. 2061–2064.
[3] J. E. Gonzalez, Y. Low, C. Guestrin, and D. O’Hallaron, “Distributed parallel inference on large factor graphs,” in *Proc. Conference on Uncertainty in Artificial Intelligence*, 2009, pp. 203–212.
[4] A. Smola and S. Narayanamurthy, “An architecture for parallel topic models,” *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 703–710, 2010.
[5] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *SIGMOD*, 2010.
[6] Q. V. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng, “Building high-level features using large scale unsupervised learning,” in *Proc. ICML*, 2012.
[7] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng, “Large scale distributed deep networks,” in *Proc. NIPS*, 2012, pp. 1232–1240.
[8] “Apache Giraph,” http://giraph.apache.org/.
[9] “Apache Hama,” http://hama.apache.org/.
[10] B. Shao, H. Wang, and Y. Li, “Trinity: A distributed graph engine on a memory cloud,” in *Proc.SIGMOD*, 2013.
[11] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed GraphLab: a framework for machine learning and data mining in the cloud,” *VLDB Endow.*, vol. 5, no. 8, pp. 716–727, 2012.
[12] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “PowerGraph: Distributed graph-parallel computation on natural graphs,” in *OSDI*, 2012.
[13] K. Chandy and L. Lamport, “Distributed snapshots: determining global states of distributed systems,” *ACM TOCS*, vol. 3, no. 1, pp. 63–75, 1985.
[14] J. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, “The little engine (s) that could: scaling online social networks,” in *ACM SIGCOMM*, 2010, pp. 375–386.
[15] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, “Fast crash recovery in RAMCloud,” in *Proc. SOSP*, 2011, pp. 29–41.
[16] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, “Asynchronous large-scale graph processing made easy.” in *CIDR*, 2013.
[17] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, “From “think like a vertex” to “think like a graph”,” in *Proc. VLDB*, 2013.
[18] J. W. Young, “A first order approximation to the optimum checkpoint interval,” *Comm. of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.
[19] H. Haselgrove, “Wikipedia page-to-page link database,” http://haselgrove.id.au/wikipedia.htm, 2010.
[20] “HDFS (Hadoop Distributed File System),” http://hadoop.apache.org/common/docs/current/hdfs_design.html.
[21] S. N. A. Project, “Stanford large network dataset collection,” http://snap.stanford.edu/data/.
[22] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Commun. of the ACM*, vol. 51, no. 1, pp. 107–113, 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492
[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proc. NSDI*, 2012.
[24] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in *Proc. SOSP*, 2003, pp. 29–43.
[25] M. Castro and B. Liskov, “Practical byzantine fault tolerance,” in *Proc. OSDI*, 1999.
[26] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: speculative byzantine fault tolerance,” in *Proc. SOSP*, 2007.
[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: wait-free coordination for internet-scale systems,” in *Proc. Usenix ATC*, 2010.
[28] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao, “User interactions in social networks and their implications,” in *EuroSys*, 2009, pp. 205–218.
[29] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, “Fennel: Streaming graph partitioning for massive scale graphs,” Microsoft, Tech. Rep. 175918, 2012.
[30] R. Power and J. Li, “Piccolo: building fast, distributed programs with partitioned tables,” in *OSDI*, 2010, pp. 1–14.
[31] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” in *EuroSys*, 2007, pp. 59–72.
[32] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *Proc. SOSP*, 2013.
[33] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, “Fault-tolerance in the borealis distributed stream processing system,” *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 1, p. 3, 2008.
[34] M. A. Shah, J. M. Hellerstein, and E. Brewer, “Highly available, fault-tolerant, parallel dataflows,” in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 827–838.