Processing Concurrent Graph Analytics with Decoupled Computation Model

Jilong Xue, Zhi Yang, Shian Hou, and Yafei Dai, *Member, IEEE*

Abstract—Graph processing systems have been widely used in enterprises like online social networks to process their daily jobs. With the fast growing of social applications, they have to efficiently handle massive concurrent jobs. However, due to the inherent design for a single job, existing systems incur great inefficiency in terms of memory usage, execution and fault tolerance. Motivated by this issue, in this paper we introduce Seraph, a graph processing system that enables efficient job-level parallelism. Seraph is designed based on a decoupled computation model, which decouples both the runtime job data and the computation logic. Decoupling the runtime data allows multiple concurrent jobs to share graph structure data in memory, which fundamentally increases job-level concurrency and reduces fault tolerance overhead. Decoupling computation logic could extend scheduling space, which benefits both execution performance and memory consumption. Seraph adopts a copy-on-write semantic to isolate the graph mutation of concurrent jobs, and a lazy snapshot protocol to generate consistent graph snapshots for jobs submitted at different time. Based on the decoupled model, it provides unified programming interfaces for both synchronous and asynchronous graph applications. Moreover, Seraph implements a lightweight checkpoint mechanism which can tremendously reduce the fault tolerance overhead. The evaluation results show that Seraph significantly outperforms popular systems (such as Giraph, Spark, GraphX and PowerLyra) in both memory usage and job completion time, when executing concurrent graph jobs.

Index Terms—Graph processing, concurrent jobs, graph sharing, fault tolerance, asynchronous computation, scheduling

1 Introduction

Due to the increasing need to process and analyze large volumes of graph-structured data (e.g., social networks and web graphs), there has been a significant recent interest in parallel frameworks for processing graphs, such as Pregel [1], GraphLab [2], powerGraph [3], GPS [4], Giraph [5] and Grace [6]. These systems allow users to easily process large-scale graph data based on certain computation models (e.g., BSP [7] model).

Recently, enterprises like online social networks begin to leverage these graph systems to process their daily analysis over large graph data [8]. For example, Facebook [9] uses Apache Giraph [5] to compute various graph algorithms, such as label propagation, variants of Pagerank, k-means clustering, etc. With a large number of emerging applications running on the same graph platform, it is very often to observe the case of multiple jobs running concurrently on the same graph. Our measurements on a large social network's computing cluster confirm that more than 83.4 percent of time has at least two graph jobs executing concurrently.

- *J. Xue, Z. Yang, and S. Hou are with the Department of Computer Science, School of Electronics and Computer Engineering, Peking University, Beijing 100871, China. E-mail: {xjl, yangzhi, hsa}@pku.edu.cn.*
- *Y. Dai is with the Institute of Big Data Technologies Shenzhen Key Lab for Cloud Computing Technology & Applications, School of Electronics and Computer Engineering, Peking University, Beijing 100871, China. E-mail: dyf@pku.edu.cn.*

Manuscript received 5 Apr. 2016; revised 6 Aug. 2016; accepted 2 Sept. 2016. Date of publication 18 Oct. 2016; date of current version 13 Apr. 2017. Recommended for acceptance by J. Cao. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2016.2618923

However, existing graph systems are inherently designed for a single job, which could involve significant overhead when processing multiple concurrent jobs. First, current systems can only support a limited number of jobs running concurrently due to high memory cost. Specifically, these systems do not allow multiple jobs to share graph data in memory, which in turn leads that each of them has to maintain separate graph data. This huge memory consumption significantly limits the number of running jobs. Moreover, the current checkpointing-based fault tolerance mechanism is quite time-consuming for graph computation, as the previous works demonstrated [10]. This issue is especially getting worse in the situation where jobs are running concurrently. Because each job has to periodically store the large volume of memory data into disk, making the I/O quickly become a bottleneck. Finally, a sophisticated scheduling mechanism in multi-job system is much more important for optimizing the execution performance and controlling the resource consumption. However, such scheduling mechanism is less considered in single-job systems. For example, an asynchronous graph job without controlling can easily saturate system memory, which can lead to crash of all other concurrent jobs.

We address the above issues in a real scenario: computing platform from a social network company, where a lot of analytic jobs are processed on the platform every day. Our measurements have demonstrated all the aforementioned inefficiency of using existing graph computing system. To address these issues, we propose a decoupled graph computation model (denoted as GES model), which decouples the job-related runtime data into graph, states and exchanging data, and correspondingly decouples the computation logic into state computing and message generating. We argue that, guided by the GES model, all the above issues for concurrent jobs can be naturally addressed.

On the one side, through decoupling the runtime data, we can separately manage the graph and job-specific data. The system only needs to maintain graph structure, and each running job can use the graph in a shared manner with only little job-specific data. This decoupling can fundamentally save both the memory consumption and the cost of fault tolerance for supporting multiple jobs. To enable the graph sharing, we adopts a "copy-on-write" semantic to isolate the graph mutations from concurrent jobs. Furthermore, we use a *lazy snapshot protocol* to maintain the continuous graph updates, in order to provide consistent snapshot for each new submitted job. Finally, we further design a *delta-graph checkpointing* to reduce the cost of fault tolerance for graph data.

On the other side, decoupling the existing computing logic into two computing and generating phases can bring much flexibility for both fault tolerance mechanism and scheduling policy. First, GES model explicitly defines the data dependency and transformations logics among the three parts of runtime data. This allows us to easily make the computational data reliable/recoverable through only checkpointing the tiny states data (e.g., vertex's value), which significantly reduces fault tolerance cost. Based on this idea, we design a *light-weight fault tolerance mechanism* for both synchronous and asynchronous computing engines. Second, decoupling the computing logic provides opportunities to schedule the execution and control the resource usage (e.g., memory). Driven by the GES model, we propose a *Double-Queue based framework* for asynchronous engine, and apply a message controlling mechanism on it.

In this paper, we integrate the above ideas into Seraph, a new graph processing system based on the GES model, which is particularly designed for efficiently processing multiple concurrent graph jobs. Seraph is composed of three major parts: *graph manager*, *synchronous engine* and *asynchronous engine*. The graph manager is responsible for maintaining the continuously changed and shared graph data. It can also transparently provides the isolation for multiple jobs using the same graph data. Seraph supports both synchronous and asynchronous graph application with the two execution engines, which both adopt a unified `compute-generate` programming interface derived from GES model.

We evaluate the overall performance of Seraph through comparing it with popular systems, including Giraph (graph-parallel system) and Spark (shared memory data-parallel system), as well as its graph extension of GraphX. Our experiments show that Seraph could reduce 67.1 percent memory usage and 58.1 percent average job execution time, as compared with Giraph. When failure occurs, the recovery of Seraph is more than $4\times$ faster than Giraph. Compared with Spark and GraphX, we show that Seraph can save memory usage than both Spark and GraphX by $14\times$, with a constantly faster speed than both of them.

This paper significantly extends an earlier version [11] in the following ways. First, we added a new Section 4.2 to prove the generality of our proposed computation model. Second, we added a new full Section 5 to introduce our recent work of asynchronous computation on this system, where we design an efficient asynchronous scheduling mechanism with our decoupled model. Third, we added the performance comparison with GraphX in Section 7.1. Forth, we added a new Section 7.3 to evaluate the performance of asynchronous graph computing component.

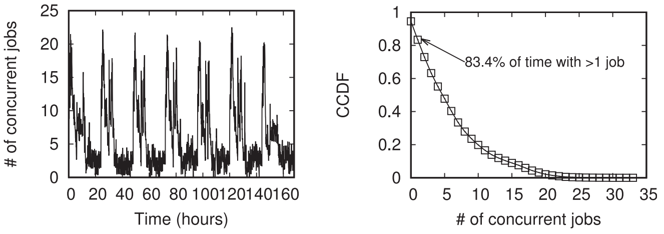

Fig. 1. One week's graph computation workload from a real cluster. The left figure is the concurrency distribution and the right one is the CCDF of concurrency.

This paper is organized as follows: Section 2 introduces the practical issue of existing graph system and motivation of model decoupling. In Section 3, we introduce the graph management of Seraph, including the graph sharing and lightweight fault tolerance. Sections 4 and 5 introduce synchronous and asynchronous computing engines respectively. We describe implementation details in Section 6. Section 7 evaluates the performance of Seraph through comparing with Giraph and Spark. We then describe related work in Section 8 and conclude in Section 9.

2 Motivation

In this section, we first introduce the practical usage of existing graph computing systems, and then reveal their inefficiency under real workloads. Finally, inspired by these issues, we propose a decoupled computation model to address the above inefficiency.

2.1 Background: Practical Graph Computation

Recently, in social network companies like Facebook [9] and Renren [12], graph systems are widely used to process a large number of daily graph jobs across different products. Typical jobs include friend recommendation, advertisements, variants of Pagerank, spam detection, kinds of graph analysis, label propagation, clustering, etc. [8].

With the increasing number of jobs running on the same platform, it easily occurs the case of multiple jobs running concurrently. To demonstrate this scenario, we collect one month job logs from the computing cluster of a social network company. Fig. 1 depicts one week's job distribution from Nov. 1, 2013, which shows that a significant number of jobs are executed concurrently every day. At peak time, there are more than 20 jobs pending on the platform. We also depict the complementary cumulative distribution of job concurrency of every second in Fig. 1. As it shows, more than 83.4 percent of time has at least two jobs executed concurrently. On average, the number of concurrent jobs is 8. Furthermore, we have manually checked the graph datasets that each job uses, observing that more than 58.6 percent of these jobs use the same (or at least a large common part of) graph, e.g., the different snapshots of the evolving graph.

However, the existing graph processing systems are all designed for single job, thus are much inefficient when processing many concurrent jobs. Those inefficiencies are generally fall in three aspects: *limited concurrency*, *high fault tolerance cost* and *inefficient scheduling*.

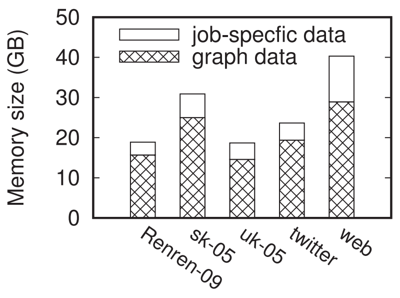

Fig. 2. The runtime memory usage of Giraph.

Limited Concurrency. We notice that current graph systems could support very limited concurrent jobs due to high memory consumption. Taking Giraph as an example, Fig. 2 shows the memory usage of Giraph job running Pagerank algorithm on various graph datasets [13], [14]. From the figure we can see that the graph data occupy the majority of memory as compared with vertex values and combined messages (job-specific data), and the percentages it accounts for are varying from 71 to 83 percent for different datasets. Due to the heavy memory cost, it is hard to execute many concurrent jobs in a commodity cluster. For example, in our experiments, a cluster with 16 nodes (each with 64 GB memory) could only support at most four concurrent Giraph jobs over a graph with 1 thousand million vertices.

High Fault Tolerance Cost. Existing fault tolerance solution for graph systems has both storage and performance issues. Specifically, existing graph systems, such as Pregel, Giraph, GraphLab and PowerGraph, all rely on checkpointing and rollback-recovery to achieve fault tolerance [1], [2], [3], [5]. Each checkpoint needs to store all the data required by computation in that step to persistent storage. Those data include graph, exchanging data (e.g., messages) and computation states. The huge data volume often introduces significant storage challenge. For example, in Giraph, running a single PageRank job on a 100 gigabytes graph with checkpointing per iteration, the total data saved in HDFS are more than 9 terabytes. Moreover, saving the large checkpoint data involves a long execution latency. Fig. 3 shows the execution time of running PageRank and SSSP algorithm on a 25 million vertices graph with different checkpointing frequency (CF). As it shows, the overall execution time is increased by 52.7 percent due to checkpointing when CF=5 for Pagerank. This case is especially worse when with many concurrent jobs, which will cause heavy I/O pressure.

Inefficient Scheduling. Most existing graph systems take less consideration of the job's runtime memory consumption (besides the part of graph data). For example, in a

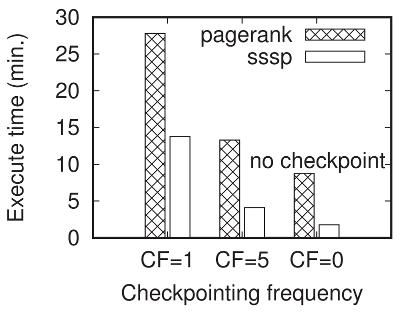

Fig. 3. The latency incurred by checkpoint mechanism.

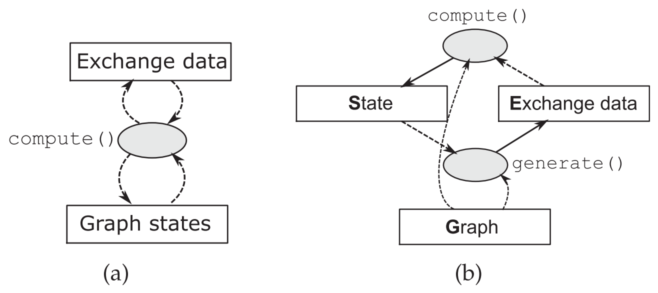

Fig. 4. Abstracted computation model of graph system. (a) Existing computation model, (b) GES model.

message-passing based asynchronous graph computation, without a carefully designed controlling mechanism, the messages could be generated too fast to be consumed, which can quickly run out of system memory and lead to the crash of all other jobs that co-located in the same server (i.e., with out-of-memory errors). Thus, memory management is much more important for a multi-job system, due to the resource sharing, than the single-job systems. However, the existing computation model leaves less space to allow the system to independently control the message generation, since the message generating logic is mixed with the computation logic. Thus, a more flexible computation model is beneficial to fully leverage the scheduling benefits.

2.2 Model Decoupling

To understand the aforementioned issues, we extend the current vertex-centric `compute()` model from the perspective of data management and data transformation, as Fig. 4a shows. In this model, the *graph states* data indicates not only the values on each vertex, but also the graph structure data. Besides, the *exchanging data* means the intermediate data (e.g., messages) which is usually generated during the computation. The computation is completed through applying the `compute()` function on each vertex to update both the two types of data, until the graph states meet a certain termination condition.

However, our observation shows that this tightly-coupled model is the key factor causing the aforementioned inefficiencies. Specifically, this model combines the graph structure data with the vertex value data as a tightly-coupled unit, which leads the concurrent jobs unable to reuse the common graph structure data (the *concurrency* issue). Moreover, due to without explicitly defining the data dependencies among the three types of data, it incurs a heavy checkpoint overhead (the *fault tolerance* issue). Finally, defining both the computing logic and message generating logic in the single `compute()` function makes it hard to control the message generating, which will heavily affect the memory usage and computation performance (the *scheduling* issue).

Decoupled Model. To efficiently address these costs, we abstract a novel fine granularity graph computation model, denoted as **G**raph-**E**xchange-**S**tate (GES) model. In GES model, a graph computation program is based on three types of data: *graph*, *exchanging information* and *state*. The graph **G** is commonly static data describing the graph structure and some inherent graph properties, such as the edge weight. It can be expressed as a tuple of vertices set, edges set and optional weights, i.e., $\mathbf{G} = (V, E, W)$. The state data $\mathbb{S}$ is a set of vertex values associated with all the vertices in $\mathbf{G}$, which are used to record the current state of a program. The exchanging data $\mathbb{E}$ is used to share the states among vertices.

Generally, a graph computation can be described as a series of transformations or updates among the three parts of data, as illustrated in Fig. 4b. The following functions show the dependence among these data

$$\mathbb{S} \leftarrow f_c(\mathbf{G}, \mathbb{E}, \mathbb{S}) \tag{1}$$

$$\mathbb{E} \leftarrow f_g(\mathbf{G}, \mathbb{S}). \tag{2}$$

In each superstep, the function f_c takes the exchanging data $\mathbb{E}$ (messages) and the old state $\mathbb{S}$ as input and compute the new state $\mathbb{S}$. And the function f_g takes the state $\mathbb{S}$ as input to generate exchanging data for next superstep.

Advantages. The GES model has two major advantages: First, it decouples the whole runtime data into three parts: states, exchanging data and graph, which enables the job-level parallelism on a shared graph, i.e., multiple jobs jointly use one graph dataset in memory. Moreover, this model further describes the data dependencies among the three parts of data explicitly, which can be leveraged to design a light-weight fault tolerance mechanism: instead of checkpointing both the state data $\mathbb{S}$ and the exchanging data $\mathbb{E}$, we only checkpoint the tiny state data and regenerate exchanging data according to the data dependencies.

Second, it decouples the computation into two separated logical steps: *compute* and *generate*. This decoupling allows us to separately control the two transformation logics. Through carefully balancing the executing rates of *compute* and *generate* functions, we can not only optimize the memory consumption, but also improve the converge speed for asynchronous computation.

Discussion. Compared with the PowerGraph's gather-apply-scatter (GAS) decomposition [3], our GES model highlights the decoupling of data dependency for a vertex-oriented program. In particular, the GES model explicitly separates the graph data $\mathbf{G}$ from the job-specific exchanging data $\mathbb{E}$ and state data $\mathbb{S}$, which enables the design of graph sharing mechanism in Section 3. Moreover, the GES model explicitly defines the generation of exchanging data $\mathbb{E}$ from only the states $\mathbb{S}$ and graph data $\mathbb{G}$, which enables the design of lightweight fault tolerance mechanism in Section 4.3. In contrast, the GAS model does not make such data dependency decoupling, leaving room for performance improvement and cost saving.

2.3 System Overview

To implement the GES model, we design *Seraph*, a graph computation system that particularly supports multiple jobs running on a shared in-memory graph. Fig. 5 shows the high-level system overview. At the very beginning, Seraph loads the initial graph data into memory (Step 1). Then it begins to receive new graph updates (e.g., add/remote vertices/edges) and new submitted jobs (Steps 2 and 3). Once a new job arrived, Seraph will commit all the current updates to generate a latest graph snapshot, and start the job on it (Step 4). Finally, the checkpoint

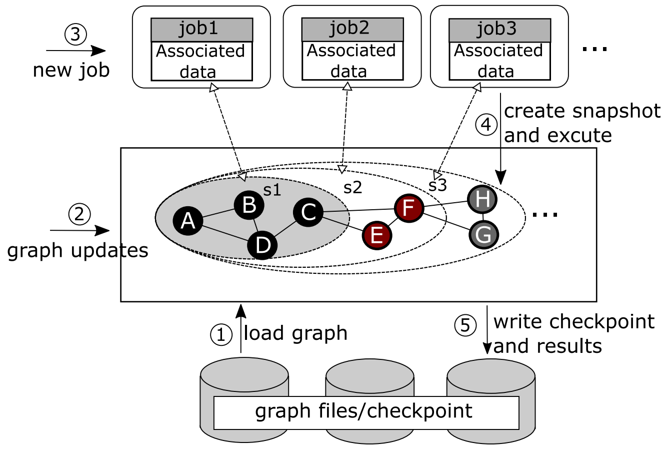

Fig. 5. The system overview of Seraph.

data and the job results will be written to persistent storage during the job execution (Step 5).

Since Seraph is a distributed system, the graph is partitioned and stored in a distributed manner. By default, Seraph uses hash-based partition to evenly assign the vertices to each worker. Certainly, sophisticated graph partitioning methods can be adopted for less communication overhead. However, graph partitioning is beyond the scope of this paper. Indeed, there are considerable literatures on this topic, e.g., PowerGraph [3], FENNEL [15], Giraph [5], which focus on providing multiple alternatives to partition the graph data. In fact, arbitrary partition methods can be easily applied in Seraph through a pre-processing step provided by user.

In the rest of this paper, we will introduce the details of how Seraph supports the graph sharing and the dynamic updating in Section 3. Seraph supports both synchronous and asynchronous computation. Both the two engines are based on the same decoupled computation model and provide the same programming interfaces. We will introduce each of them in Sections 4 and 5 separately. Besides of them, we will introduce how our light-weight fault tolerance works along within the three sections separately (i.e., Sections 3, 4, and 5).

3 Graph Data Management

In this section, we first introduce how Seraph supports graph sharing, as well as the mechanisms for graph mutation, graph updating and snapshot maintaining. Then we propose the lightweight fault tolerance mechanism for graph management.

3.1 Graph Sharing

The unique feature of Seraph is that concurrent jobs could share graph structure data to avoid memory waste. Specifically, Seraph only stores and maintains one graph data in memory. Instead of executing on a separate graph, each individual of the concurrent jobs uses the same shared graph by creating reference to it. Then each job creates its own job-specific data (vertex states and exchanging data) in the local memory space.

To efficiently support graph sharing, we have to address some conflicts caused by the graph sharing. First, the local

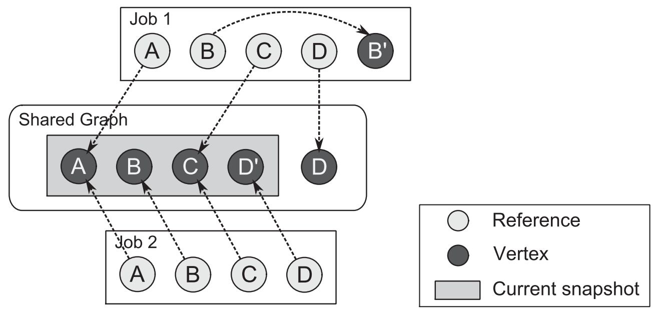

Fig. 6. Illustration of graph mutation and graph update in Seraph.

graph mutations must be isolated. Some graph algorithms may modify the graph structure during the computation (e.g., a K-core algorithm may delete some edges or vertices in each superstep). Without the isolation, the mutations would interfere other jobs due to using the same graph data. Second, we should maintain a consistent graph snapshot for each individual job according to its submission time. For example, the global updates of the underlying graph (e.g., the arrival of new vertices or edges) should be only visible to the jobs submitted later than the updates. In this section, we show how Seraph achieves these requirements.

Graph Mutation. To isolate the graph mutation for each job, Seraph adopts a "copy-on-write" semantic to isolate the local mutations of individual job. In particular, when a job needs to modify the edge list of a vertex (denoted as v), it first creates a `delta graph` object in local space and copies the corresponding edge list of v to it. Then the mutations are applied on that local copy. Meanwhile, the system changes this job's reference of v to the new one in the local graph, and the original vertex v (and its edge list) is still used by other jobs. Finally, once the job is completed, the copied local graph in memory will be released by Seraph.

Fig. 6 illustrates the example of graph mutation. Job 1 needs to modify vertex B's edges (the modification may be add or delete operation). It first copies the vertex B to local memory space, denoted as B', then modifies it locally and changes the reference of B to B'. After this, the later mutations on this vertex will be directly applied on the local data. Since jobs only mutate a small part of graph during the computation, the copy-on-write is an efficient way with little extra replication cost. Note that this kind of mutation is not applied on the shared graph, so it is transparent to other jobs.

Notice that Seraph targets the scenarios where multiple jobs share the same graph and incur little graph modification during the execution. Although the overhead of Seraph increases as jobs largely modify the base graph, the system only generates copies for the mutated vertices. So the worst-case overhead in Seraph will not exceed that in existing systems without graph sharing (i.e., loading an independent graph copy for each job).

Graph Update. Different from the graph mutation caused by running jobs, graph update are caused by the evolving of underlying graph (e.g., the new arrival of edges and vertices). Seraph guarantees that a new submitted job can see all the previous changes on the shared graph. Formally, when an update is committed at time t, it should be visible (or invisible) to jobs submitted after (or before) t.

Seraph achieves the above requirement through a snapshot mechanism. The graph managers in each worker of

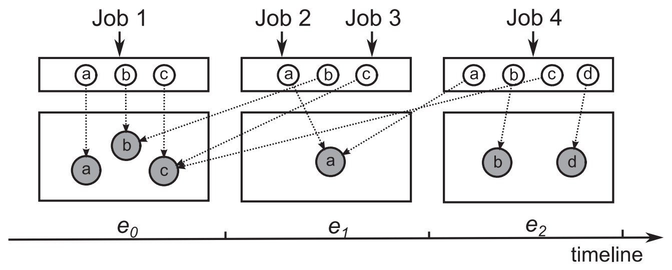

Fig. 7. Illustration of lazy snapshot generation in Seraph.

Seraph collaboratively maintain the graph snapshots. When an update arrives, Seraph incrementally creates a new snapshot as the up-to-date snapshot. When a job is submitted, it refers to the latest snapshot of graph. The mechanism of generating a consistent snapshot will be addressed in the following section.

Fig. 6 illustrates the update/snapshot process. When job 1 is submitted, it is executed on the graph snapshot $\{A, B, C, D\}$. During job 1's execution, the graph has been updated on vertex D. Since job 1 still uses vertex D, Seraph copies a new vertex D' from D and applies the update. Hence, a new snapshot $\{A, B, C\} \cup D'$ is formed. Later, job 2 is submitted, and it just refers to this new snapshot. Note that when job 1 is finished (the number of reference on D will be zero), the vertex D will be released by the graph manager.

Consistent Snapshot. A consistent snapshot of graph is critical for the computation in Seraph. Notice that the shared graph in Seraph is partitioned and maintained by multiple workers. The update on any portion of graph may incur inconsistence for the graph. An algorithm running on inconsistent graph would incur incorrect results or even runtime errors, e.g., sending messages to non-existed vertices or counting deleted edges.

Therefore, we propose a *lazy snapshot protocol*, where a global consistent graph snapshot is generated only when a new job is submitted. The intuitive idea is to delay the graph update commit till the next job is started, so that we can package multiple updates into a batch and reduce the number of committing transactions. Specifically, in Seraph, each graph update is sequentially appended into the master node, and the master node continuously distributes these updates to the corresponding workers (without committing). When a new job is submitted, the master packages all the remaining buffered updates and flush them to corresponding workers. Then the Seraph master will start a commit transaction and notify each worker's graph manager to enter into a new epoch. The updating processing is illustrated in Fig. 7.

A commit transaction is completed through a two phases commit protocol (2PC) that comes from traditional distributed systems [16]. Specifically, in the first phase, the master dispatches the remaining updates to the workers and receives acknowledgments from all workers. The worker node just caches but not applies the updates. In the second phase, the master sends committing messages to each worker and the worker nodes apply updates to graph. When a transaction is successfully submitted, all the pending jobs can be started on this latest graph snapshot. Considering that the 2PC protocol is executed within a computation cluster and is performed only once per job

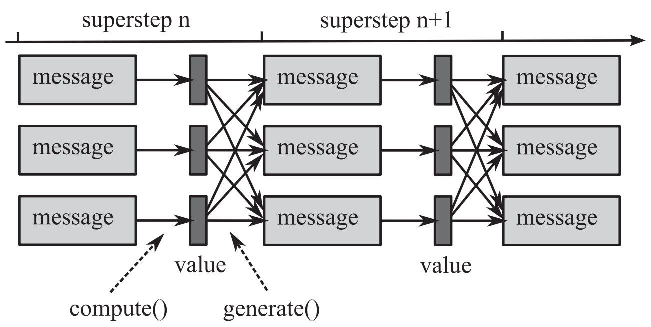

Fig. 8. The data transformation and computation model of Seraph.

submission, its overhead is quite minor. For example, in our experiment, the 2PC only takes hundreds of milliseconds per transaction, which can be ignored as compared with the job execution time (e.g., usually hundreds of seconds or more).

3.2 Fault Tolerance for Graph Data

Existing graph processing systems rely on checkpoint and rollback-recovery to achieve fault tolerance [1], [2], [3], [5]. However, checkpointing all the runtime data, including messages, states and graph data, into persistent storage will incur a heavy checkpoint cost and long checkpoint delay.

In contrast, Seraph is able to make lightweight fault tolerance based on the GES model. In this section, we only discuss how to make the graph data reliable, and for computational data (states and messages), it will be introduced in Sections 4 and 5.

Delta Graph Checkpointing. With the graph sharing mechanism, Seraph only needs to keep one graph data reliable for all jobs. However, different jobs may run on different snapshots, even there is only one graph. When failures occur, it needs to recover specific graph snapshot for each job.

Recall that once a new job is submitted at epoch e_i, there will be a *Delta-graph* generated, denoted as Δ_i (see Fig. 7). Graph snapshot in epoch e_n can be generated through the graph snapshot in epoch e_{n-1} and delta-graph in e_n, i.e.,

$$G_n = G_{n-1} \oplus \Delta_n, \tag{3}$$

where the operation "$\oplus$" has the following definition: for graph or delta-graph G_1 and G_2,

$$S_{G_1 \oplus G_2} = (S_{G_1} \setminus S_{G_2}) \cup S_{G_2}, \tag{4}$$

where S_G means the set of vertices in graph G.

Thus, when the underlying graph is changed in epoch e_n, Seraph only checkpoint the delta graph Δ_n. In Seraph, the worker will checkpoint the delta-graph at the beginning of each epoch. When failures occur, Seraph can easily reconstruct the corresponding snapshot for each individual job through each delta graph. The delta-graph is quite small compared with the whole graph data, thus the incremental checkpointing significantly reduces the checkpoint cost.

When a worker fails, the graph partition on that worker will be lost. Seraph assigns the graph partition to a standby worker and constructs underlying graph according to the checkpointed delta-graphs, as well as the specific snapshot for each running job. After that, the graph jobs restart and continue the computation from the failed step.

4 Synchronous Computation

This section introduces synchronous computation model in Seraph, as well as the fault tolerance mechanism for computational data.

4.1 Decoupled Computation Model

Seraph's computation model is derived from the BSP [7] model. In general, the graph computation consists of several supersteps, in each of which a vertex can collect messages and execute user-defined functions. In BSP, all these actions are coupled in a user implemented function, let's say `compute()`. This vertex-centric approach can express the behavior of a broad set of graph algorithms [17].

However, according to the GES model in Section 2.2, a better way to express the graph computation logic is decoupling the `compute()` function into two individual functions: `compute()` and `generate()`. Here, the new `compute` interface is for computing new vertex state based on received messages, and `generate` interface is used to generate the messages for the next superstep according to the new states. Fig. 8 shows the data transformation and computation model of Seraph.

According to the GES model, we expose two interfaces for user to write their applications: `compute()` and `generate()`. We argue that the new interfaces are very natural to use. To show how easily this programming model can express common graph algorithms, Fig. 9 illustrates the "compute-generate" implementations of Pagerank, SSSP and WCC algorithms.

Pagerank

```
//compute
compute(Msgs):
  sum = 0;
  for (m:Msgs)
    sum += m;
  setValue(sum*0.85 + 0.15);

//generate message
generate(value):
  sendMsgToNbrs(value/#nbrs);
```

SSSP

```
//compute
compute(Msgs):
  for (m:Msgs)
    minv = Min(minv, m);
  setValue(minv);

//generate message
generate(value):
  if (changed(value))
    sendMsgToNbrs(value + 1);
```

WCC

```
//compute
compute(Msgs):
  for (m:Msgs)
    maxv = Max(maxv, m);
  setValue(maxv);

//generate message
generate(value):
  if (changed(value))
    sendMsgToNbrs(value);
```

Fig. 9. The example of implementations of graph algorithms based on compute-generate model.

4.2 Generality Analysis

Before introducing the benefits of this model, we first give an analysis on its generality. In message-passing based graph computation, it generally involves two kinds of data: *values* (denoted as V) which are used to describe the computational states (i.e., the $\mathbb{S}$ in GES model), and *messages* (denoted as M) which describes the exchanging information between vertices (i.e., the $\mathbb{E}$ in GES model). The values are kept in memory all the time, however messages are temporal and will be released once being consumed.

Compared with vertex-centric *compute* model, the *compute-generate* model mainly has following differences: 1) program logic is split into two parts, compute and generate phases. 2) the compute phase takes messages as input and can only modify values. 3) the generate phase takes value as input and can only send messages. However, we argue that the constraints introduced in this model do not reduce the ability of problem abstracting. In other words, it has at least the equivalent expressiveness with the existing *compute* model.

Theorem 1. *The* compute-generate *model is equivalent with the* compute *in their ability of problem abstraction.*

Proof. For a specific vertex, we denote M as its incoming messages and V as its value . For the traditional *compute* model, we can define a function f which takes M and V as inputs, and generates outgoing messages M' and new values V', i.e.,

$$(M', V') = f(M, V).$$

Now for the *compute-generate* model, we constraint the program to be represented as a combination of two subsequent functions $f_{compute}$ and $f_{generate}$. The computation can be written as

$$V_t = f_{compute}(M, V)$$
$$M' = f_{generate}(V_t).$$

For graph algorithms, we can divide them into two cases: 1) the M' only depends on the V'. In this case, we can allow $V_t = V'$ to transform them from the *compute* model to *compute-generate* model. And to our knowledge, most typical algorithms fall into this case, like the algorithms we illustrate in Fig. 9. 2) Otherwise, we still can allow $V_t = (M', V')$, and function $f_{compute} = f$. Thus the function $f_{generate}$ only needs to return the M'. Based on this, we can prove that any algorithm written in *compute* model can be transformed to *compute-generate* model. □

4.3 Fault Tolerance for Computational Data

Section 3.2 has introduced the fault tolerance for graph data. Now let's look at how Seraph guarantees the computational data (i.e., vertex values and messages) reliable.

Specifically, given an graph algorithm that can be described by the GES model as follows:

$$M_i \leftarrow generate(V_{i-1})$$
$$V_i \leftarrow compute(M_i, V_{i-1}),$$

where M_i and V_i are the messages and values in ith superstep. Based on the above functions, Seraph can usually use the `generate` function to regenerate message data based on the value data of last superstep. Thus, we only need to checkpoint the value data (e.g., V_{i-1}) at each step to reduce checkpoint cost, since the size of message data is often much larger than that of value data.

When a job fails, since the graph data is decoupled from the job, only the values and the messages associated with the job will be lost. Seraph will restart the job on each worker, and roll back to last superstep of failure step. By running `generate()` function on that superstep to regenerate the lost messages, Seraph can quickly recover from the failure.

5 Asynchronous Computation

In this section, we illustrate how Seraph supports asynchronous computing based on the decoupled model introduced in Section 2.3.

5.1 Decouple-Based Asynchronous Computation

Derived from GES model, Seraph's asynchronous engine adopts a decoupled `compute-generate` programming model, with further enabling accumulative computation. Given that the algorithm allowing accumulative computation, a vertex can update its state based on every delta message, with no need to wait for all neighbors' messages. Furthermore, Seraph uses the message passing as the way of data exchanging to avoid the data inconsistence issue, which is introduced in [2].

Now we give a formal definition of the accumulative asynchronous model. In this model, each vertex's state is changed according to its neighbor's impact. We assume such impact can be accumulated from many small impacts. Formally, we have

$$S_v \oplus \sum_{i=1}^{n} \oplus I_v^i = S_v \oplus I_v^1 \oplus I_v^2 \oplus \ldots \oplus I_v^n, \tag{5}$$

where, S_v is the state of vertex v, I_v^i is the ith impact on vertex v. The accumulative operator $\oplus$ can combine multiple impacts into one or apply impact to vertex state. Based on this assumption, the computation for a vertex v can be written as

$$\begin{cases} S_v &= S_v \oplus \sum_{i=1}^{n} \oplus I_v^i \\ I_u &= g(S_v), \forall u \in N_v \end{cases}. \tag{6}$$

Specifically, every time vertex v is scheduled, it first combines all its impacts and applies to current state S_v. Function $g()$ is used to generate the impacts to all other vertices based on its own latest state. N_v denotes vertex v's neighbors.

Message Flooding Issue. Like in BSP model, a simple way to implement the above accumulative model is exposing an interface, like `compute()`, for user to implement the computation logic in Equation (6). However, this model works well for the synchronous scenario, but not the asynchronous one. Specifically, in synchronous model, all the generated messages in iteration i can be consumed in iteration $i+1$, so the worst case of memory footprint for message storage is $O(N_E)$, where N_E is the number of edges. However, in asynchronous model, each call of `compute()` will generate N (N is the number of neighbors for this vertex) messages, but only consuming one message. Thus, the number of messages can easily saturate system memory.

Pagerank-Async

```
//compute
compute(deltaMsg):
  accValue += deltaMsg;

//generate message
generate(accValue):
  v = accValue*0.85 + 0.15;
  value += v;
  sendMsgToNbrs(v/#nbrs);
  accValue = 0;
```

SSSP-Async

```
//compute
compute(deltaMsg):
  if (deltaMsg < value)
    setValue(deltaMsg);
    accValue = deltaMsg;

//generate message
generate(accValue):
  v = accValue + 1;
  sendMsgToNbrs(v);
```

WCC-Async

```
//compute
compute(deltaMsg):
  if (deltaMsg > value)
    setValue(deltaMsg);
    accValue = deltaMsg;

//generate message
generate(accValue):
  v = accValue;
  sendMsgToNbrs(accValue);
```

Fig. 10. The example of implementations of graph algorithms based on asynchronous compute-generate model.

Thus, a more reasonable designing is to decouple the `compute()` logic into two parts like our GES model, i.e., `compute()` and `generate()`. The `compute()` describes how to apply a delta message into vertex's value, and the `generate()` describes how to generate delta messages for other vertices based on the current value. Based on this decoupling, we can individually control the rate of each interface, so as to limit the total memory consumption. For example, if the total number of messages is beyond a threshold, we can adaptively slowdown the rate of generating message (`generate()`), and vice versa. Furthermore, this model naturally provides consistent programming interface with the synchronous model in Seraph, thus it is easy for user to transform their synchronous application into asynchronous one. Fig. 10 illustrates three asynchronous examples which are transformed from the synchronous model in Fig. 9.

5.2 Double-Queue (DQ) Based Framework

To enable the above decouple-based asynchronous computing model, we propose a double-queue based framework, seen in Fig. 11. In this framework, there are two continuously running components: one for `compute()` function and another for `generate()` function. Beyond these, there are two data queues for storing the runtime status. The message queue is for gathering all incoming delta messages, and the active vertex queue is for placing all active vertices' id.

During the running process, the *compute* component continuously fetches messages from the message queue and calls `compute()` interface to add the delta value to corresponding vertex's value. Meantime, it will check whether this vertex should be active. For example, for PageRank algorithm, if a vertex's value has been changed larger than a certain threshold, it needs to propagate its impact to neighbors. A vertex will be inserted to the active vertex queue only if it is not in the queue and it is active.

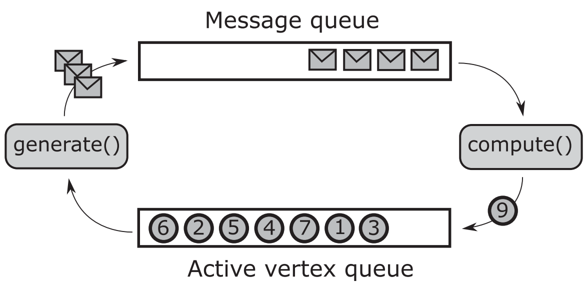

Fig. 11. Double-queues based asynchronous computing model.

The *generate* component continuously pops out a vertex from the active vertex queue and calls the `generate()` interface, so as to propagate its new impact to its neighbors through messages.

This framework provides very flexible controlling space which can be used to optimize the resource usage (i.e., memory) and performance. For example, dynamically adjusting the rate of computing and generating components can control the length of both message queue and active vertex queue, which can further affect the memory usage and computational performance. To obtain the optimal performance, we need to come up with the best control policy for this framework. Here, we denote the rates of computing and generating components as r_c and r_g, which are defined as the speed of message consuming by `compute()` and the speed of message generating by `generate()`, respectively. The maximal computing rate is denoted as $\overline{r_c}$, which is the message consuming rate when the CPU resource is fully utilized by computing component, i.e., $r_c \leq \overline{r_c}$. Assume the operations of enqueue and dequeue spend negligible time, thus we have following theorems.

Theorem 2. *In the DQ framework, let l_1 and l_2 be two constants with $l_1 < l_2$, then we have that the computation time T_1 with message queue Q_M's length of l_1 will be less than the computation time T_2 with that of l_2.*

Proof. Given a running DQ framework D_1, we denote the length of its message queue Q_1^M as l_1. Assume the last d messages M in Q_1^M are generated from vertex v. Now let's transform the D_1 to D_2, where we remove the messages M from Q_1^M, which turns to be Q_2^M, and push back v to the head of active vertex queue Q_1^V, which becomes Q_2^V. Thus the length of Q_2^M turn to be l_2, where $l2 < l_1$. To keep the length of Q_2^M to be l_2, v has to stay in Q_2^V for a certain time, i.e., t. During this time, there is a probability p that vertex v will be active again and its impact will be increased. Thus, after a same certain time, messages M_1 and M_2 in both D_1 and D_2 will be computed. However, M_2 will generate larger impact than M_1, which will make d_2 converge faster than d_1. □

This theorem shows that the less messages in the messaging queue (Q_M), the faster the computation will be.

Theorem 3. *In the double-queue framework, to obtain shortest computation time, the generating and computing rates should be equal to the maximal computing rate $\overline{r_c}$, which has $r_g = r_c = \overline{r_c}$.*

Proof. We define the `compute()` logic as basic operator in this framework, which is denoted as $\mathbb{C}$. Given that generating component is quite lightweight compared to computing one, the overall computation time is determined by the total number of basic operator and computing rate. Given an application p, we denote the total operator number as $N_o(p)$ and the computing rate as $r_c(p)$, thus the total computation time can be written as

$$T(p) = \frac{N_o(p)}{r_c(p)}. \tag{7}$$

Note that, if there are sufficient messages in the message queue Q_M, i.e., $r_g \geq r_c$, then $r_c(p) = \overline{r_c}(p)$.

Now let's consider the case where $r_g < r_c$, which means there will be insufficient messages in Q_M for `compute()` to consume. Thus r_c will decrease, which in turn increases $T(p)$, i.e.,

$$\frac{N_o(p)}{r_c(p) \downarrow} \Rightarrow T(p) \uparrow . \tag{8}$$

On the other hand, suppose that $r_g > r_c$, which means the `generate()` dequeues too fast from the active vertex queue Q_A. This will increase the length of the message the queue Q_M. According to Theorem 2, increasing Q_M's length will also increase the computation time, i.e., $T(p) \uparrow$.

Based on above analysis, either r_g is greater or less than r_c, it always increases the execution time. Thus, we can conclude that $r_g = r_c = \overline{r_c}$ is the best configuration. □

5.3 Message Controlling

In this section, we give the best message controlling policy which can minimize the execution time of the Double-Queue framework. According to Theorems 2 and 3, we can derive that a best controlling policy has at least the following three properties:

1) computing rate r_c always gets the maximal rate $\overline{r_c}$;
2) message generating rate r_g is equal to computing rate r_c;
3) length of message queue l_m is approaching to zero.

The first two properties are derived from Theorem 3, showing that computing component should fully utilize CPU resource, and the generating component's speed is determined by the computing component. According to Theorem 2, to minimize the execution time, we should reduce the length of message queue as short as possible.

Adaptive Message Controlling. To enable above controlling policy in DQ framework, we design a message controlling module in Seraph.

First, Seraph needs to control the length of message queue to a minimal value τ. According to Theorem 2, the less messages in the message queue, the faster the computation will be. However, in a distributed implementation, we have to reserve few buffered messages to tolerant the network jitter, variance of generating rate, and network latency, so as to make sure that the computing component could always get a message from the message queue. Thus we introduce a threshold τ as the minimal queue length. When a job is started, we don't control the message generating at first, so as to make sure there are enough messages for computing component to consume. When the number of messages is sufficient to fully utilize CPU resource, Seraph starts the controlling module. The controlling module first stops the message generating until it observed the cluster's CPU utilization is dropped down from 100 percent by a granularity of μ (e.g., $\mu = 5$ percent). At this time, the computing rate r_c is just slightly below $\overline{r_c}$, and the system configures the threshold τ as the current length of the message queue.

Second, according to Theorem 3, we should control the global generating rate r_g to be equal to r_c. During the initial period, each worker i periodically estimates its computing rate $r_c^{(i)}$ by itself and updates to master. In master side, there is a table which maintains all workers' latest computing rate $r_c^{(i)}$. Master also periodically calculates the computing rate of whole system by summing all worker's rate, i.e., $r_c = \sum r_c^{(i)}$. To let $r_g = r_c$, each worker's target generating rate $r_g^{(i)}$ should be r_c/N, where N is the number of workers. Master will broadcast this target value to all workers periodically, and each worker dynamically adjust its message generating rate $r_g^{(i)}$ according the target rate and current rate.

5.4 Fault Tolerance for Computational Data

According to the message controlling mechanism, the runtime data for asynchronous job in Seraph includes only vertex states and very few messages in the message queue. Compared with the synchronous job, this is quite lightweight. Thus, we simply adopts the Lamport consistent snapshot algorithm to generate runtime snapshots and periodically checkpoint them to disk. Thus, the job can continue to execute from the history checkpoint when it fails.

6 Architecture and Implementation

Seraph is implemented in Java. The communication between workers is based on Apache MINA [18]. Like Giraph, the underlying persistent storage is using Hadoop HDFS [19]. In the following, we give the Seraph's architecture, as well as the specific implementation details of graph management and job scheduling.

6.1 System Architecture

A Seraph cluster includes a single master (with a backup master to tolerate failures) and several workers. The master controls the execution of multiple jobs, and each worker executes part of individual jobs.

The master is mainly responsible for controlling the execution of multiple jobs and monitoring the state of workers. Specifically, *job controller* receives job submissions and dispatches jobs to workers holding the corresponding data. It also controls the job progress, including startup, superstep coordination and termination. The master also checkpoints the states of each job for fault-tolerance.

Several workers execute jobs and maintains its portion of the graph in memory. To share graph among multiple jobs, Seraph implements a graph manager to decouple graph structure data from job-specific data (e.g., vertex value), and only maintains one graph data in memory. The graph manager is also responsible for graph updating and snapshot maintaining.

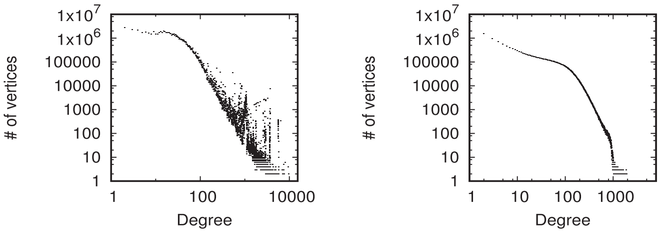

Fig. 12. The degree distribution of the *uk* graph (left) and *snp* graph (right) plotted in log-log scale.

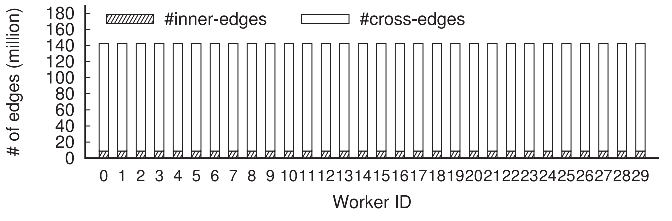

Fig. 13. Number of edges on each Seraph worker after partitioning *uk* graph into 30 parts.

At the startup, a worker registers with the master, and periodically sends heartbeats to demonstrate its continued availability. When a job is dispatched to a worker, Seraph starts a new *executor*, which is a separate thread used to manage this job's execution.

6.2 Graph Management

In Seraph, a graph in memory is stored as a set of `Vertex` objects. Each object includes vertex value, message queue and an adjacency list storing neighbor edges. Thus, Seraph extracts edge list from `Vertex` object and forms a new global `graph` set containing each vertex's edge list. Each job makes a reference to this unique graph object to get the graph structure. The reference pointing to the edge list of each vertex, which is far smaller than a whole edge list.

Hibernate Mode: Different with existing systems, Seraph is an online running platform which supports the updates on graph. To save memory when there is no jobs running, Seraph implements a hibernate mode. Once the system is idle for a period (e.g., longer than a certain threshold), Seraph allows all the workers to hibernate. Each worker will make a checkpoint of the full graph (instead of delta-graph). This graph will be considered as the new initial graph G_0, and all the old G_0 and previous delta graphs on HDFS will be deleted. After that, the memory occupied by the graph data will be reclaimed. In the hibernate mode, all graph updates will be cached on master node. When a new job arrives, master will wake up all the workers and apply the cached updates to the graph maintained by them. Note that the checkpoint is serialized in disk, so the waking up process is much faster than reloading the raw graph data from HDFS.

6.3 Job Scheduling

To better leverage the resource, Seraph implements a *job scheduler* to control job execution. The scheduler assigns each job a priority based on its submission order, e.g., job submitted earlier has a higher priority. Seraph guarantees that early jobs has more priority to use resource, job submitted later can exploit the idle resource. The job scheduler is implemented through performing flow control at two levels: controlling job execution and controlling the message dispatching.

Execution Controlling. In Seraph, job's execution is through performing each vertex's `compute()` and `generate()` operations. We package a bunch of vertices' operations of a job and the job ID into a `task` package, which composes the minimum execution unit in Seraph. Thus, each job is executed through submitting their tasks to executing threads pool. The submission rate of each job is customized based on the priority. Currently, the rate r of each job is set as inverse ratio to its priority level p, which is $r = 1/(p - p_h + 1)$, where the p_h is the highest priority among current running jobs.

Message Dispatching. Message dispatcher uses a priority queue to buffer messages from all jobs. Meanwhile, Seraph assigns each job a different threshold to control the max buffer size it can use, which is similarly setting as the inverse ratio to its priority. When a job achieves the maximum buffer size, its incoming flow path will be blocked and executor will be paused. This avoids the low-priority jobs to aggressively compete for bandwidth with high-priority jobs.

7 Experiments and Evaluations

7.1 Experimental Setup

Our experiments are conducted on a cluster with 16 machines. Each is with 64 GB of memory and 2.6 GHz AMD Opteron 4,180 Processor (12 cores). All these machines are connected with a gigabit switch. We choose one machine as the master and other 15 machines as workers.

We take some popular graph algorithms as benchmarks to evaluate Seraph's performance, including Pagerank, random walk, weakly connected component (WCC), single source shortest path (SSSP) and K-core. In our following experiments, we set PageRank's max iteration steps to 20. For random walk, we set 100 walkers for each vertex and the length of walk is 10 steps. For k-core, we set $k = 5$.

We run the benchmarks on a snapshot graph (denoted as *snp*) of Renren, which contains more than 25 million vertices and 1.4 billion relational edges, and a public web graph data set (i.e., *uk-2007-05*, denoted as *uk*) [13], [14]. Both the two datasets are power-law graphs with highly skewed degree distribution, as demonstrated in Fig. 12. Even though, we could observe that hash-based graph partitioning can evenly assign the edges to each workers. For example, Fig. 13 shows the number of inner partition edges and cross partition edges on each Seraph worker after partitioning *uk* graph into 30 parts. The result is similar for Renren graph. Since the storage, communication, and even computation complexity are always linear to the number of edges on a specific worker, we observe that the hash partition could achieve workload balance even in the presence of high-degree vertices.

7.2 Performance of Executing Concurrent Jobs

First, we compare Seraph with both a graph computing system (i.e., Giraph) and an in-memory general data processing system (i.e., Spark), as well as its graph extension, GraphX. Giraph is one of the most popular open-sourced Pregel

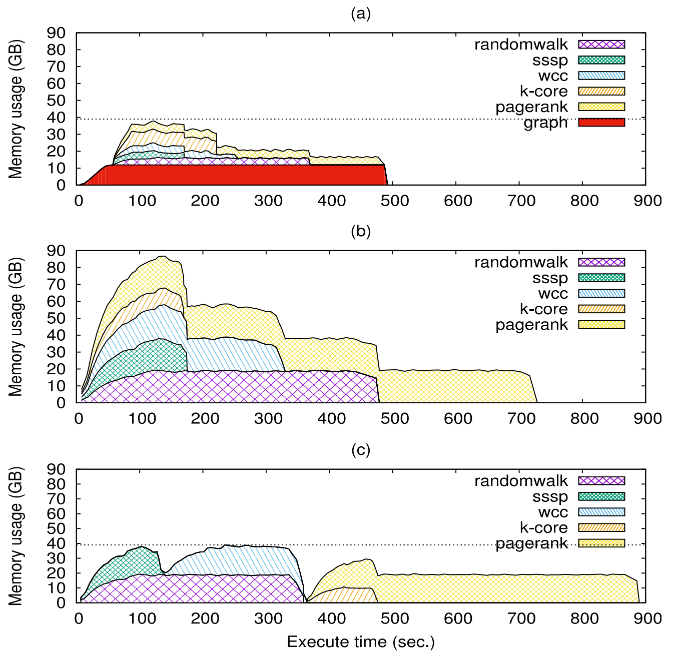

Fig. 14. The memory usage for five parallel graph jobs on: (a) Seraph with maximum memory resource of 40 GB, (b) Giraph with unlimited memory (heap usage is 87 GB) and (c) Giraph with same limitation of maximum 40 GB memory.

implementation. We use the Giraph 1.0.0. Spark is a general data-parallel system which supports in-memory data sharing through RDD (Resilient Distributed Dataset) [20]. The version of Spark and GraphX is 1.3.0.

To evaluate Seraph's performance with multiple concurrent jobs, we submit several graph analytic jobs with different algorithms (i.e., random walk, SSSP, WCC, k-core and PageRank) simultaneously to both Seraph and the baseline systems. For Spark, We have implemented all the five graph algorithms with the RDD. For each graph algorithm, we use three separated RDDs to repersent the graph, vertex states and exchanging data.

7.2.1 Memory Consumption

Figs. 14a and 14b show the memory usage during the execution of Seraph and Giraph, respectively. Seraph first loads the underlying graph, which occupies 11.86 GB memory. Then, Seraph executes multiple jobs in parallel over the shared graph. The peak memory usage of Seraph during the execution is 37.8 GB. Note that the K-core algorithm occupies more job-specific memory than others, because its graph mutation operations incur copy-on-write data in its local memory space.

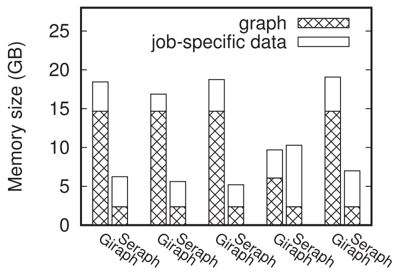

Fig. 15. The memory usage for jobs in Giraph and Seraph.

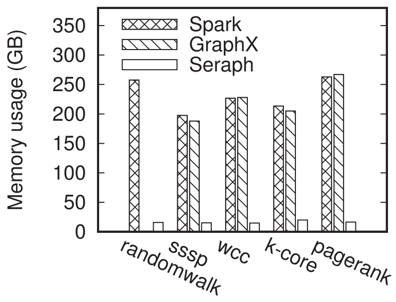

Fig. 16. The memory usage for jobs in Spark and Seraph.

For Giraph, each job has to work on a separate graph in the memory, so the total memory usage is much larger than that of Seraph. From Fig. 14b we see that the total memory rapidly increases with the number of parallel jobs, and its peak memory occupation is about 87 GB. Notice that K-core occupies less memory than other jobs, which is different from its behavior in Seraph. This is because that the algorithm can mutate its separate graph (e.g., deletes some vertices) directly.

Fig. 15 shows the memory occupation for each individual jobs, which includes memory occupation of graph data and job-specific data (e.g., states and messages). Since the graph data is shared among five jobs in Seraph, we consider that each job has $1/5$ graph memory. On average, Seraph reduce about 67.1 percent memory usage than Giraph for each job except K-core. K-core in Seraph incurs little more memory usage than Giraph due to the part of copy-and-mutated graph in its local memory space.

Now we compare the memory usage for Seraph, Spark and GraphX. Fig. 16 depicts the average memory usage for the three systems.[1] As the figure shows, Both Spark and GraphX uses an order of magnitude larger memory than Seraph. On average, the memory usage of Spark is more than $14\times$ larger than that of Seraph. The major reason is that Spark (or GraphX) will allocate new RDD for each data mutation, because RDD is immutable. However, most graph algorithms usually just mutate a little part of the whole dataset, such as SSSP and WCC algorithm. Thus, it will introduce many duplications in memory.

7.2.2 Execution Time

We first compare the execution time of Giraph and Seraph with both running five jobs in parallel, i.e., the case in Figs. 14a and 14b. In this case, Seraph can reduces the memory usage of Giraph by half, but still outperforms Giraph in terms of execution time, as shown in Fig. 17a, Although Seraph increases the execution time of K-core due to copy-on-write operations, Seraph reduces the overall execution time of five jobs by 28 percent than Giraph. The reduction is mainly brought by the job scheduling mechanism which avoids aggressive resource competition.

Next, we limit the peak memory usage of Giraph equal to that of Seraph (e.g., 40 GB). With this limitation, Giraph can only execute two jobs in parallel, with others pending until

1. We didn't provide the results of random walk on GraphX, because based on GraphX's edge-centric programming model, it is unable to write such an algorithm that has the similar execution pattern to the one in Seraph and Spark.

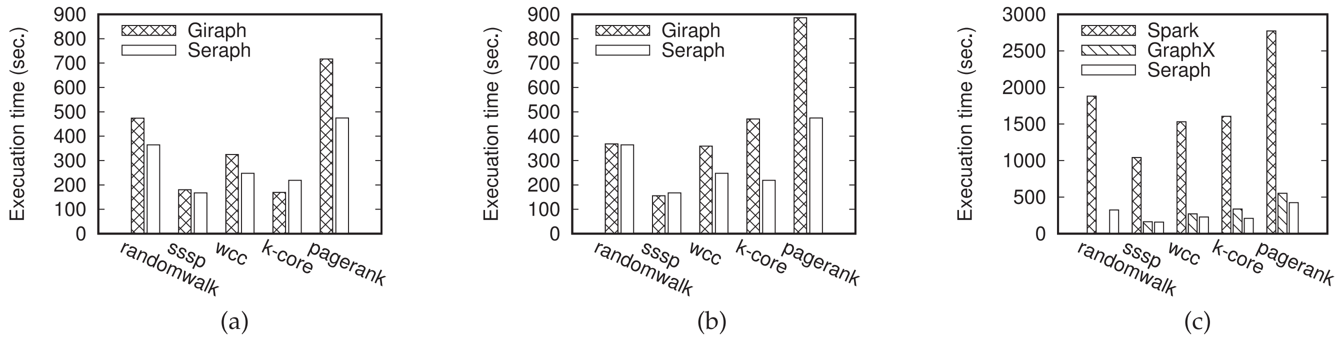

Fig. 17. The execution time comparison: (a) Giraph versus Seraph, both execute concurrent jobs with sufficient memory, the peak memory usage of Seraph and Giraph are 37.8 and 87 GB respectively. (b) Giraph versus Seraph, executing with limited maximum memory of 40 GB. (c) Spark versus Seraph, both with unlimited memory

the resource is available. Fig. 14c shows the memory usage of Giraph in this case. We see that the WCC, K-core and PageRank job need to be pending for a period, increasing the total execution time.

Given the same memory constraint, Fig. 17b shows the execution time of individual jobs executed in Giraph and Seraph, respectively. Note that the execution time includes the job pending time. We see that Seraph executes jobs much faster than Giraph. On average, it reduces the individual job completion time by 24.9 percent, as compared with Giraph. In term of the total completion time of five jobs, Seraph brings a reduction of 46.4 percent as compared with Giraph.

Now we compare the execution time of Spark, GraphX and Seraph. Fig. 17c shows the execution time of each job. We see that Seraph is more than $6\times$ faster than Spark in terms of average execution time. Even GraphX is carefully optimized for graph-specific computation, Seraph can still outperforms it by 20.4 percent. To understand the overhead of Spark, we analyze the time consumption of each operation in Spark, which is the *join* of two RDDs. Take the example of Pagerank execution, Spark separates the graph (*links*) and values (*ranks*) to share the graph for later iteration's use. However, it needs to join this two datasets in each iteration. In contrast, Seraph separates these data with an explicit connection using reference, which can avoid the overhead of joining.

7.3 Fault Tolerance

We now compare Seraph with Girpah in the fault-tolerant performance. Seraph checkpoints the vertex values at the end of each superstep. Based on the experience of [21], we set the checkpoint frequency (i.e., the number of supersteps for every checkpoints) as five for Giraph.

7.3.1 Checkpointing Cost

We evaluate the cost of checkpointing for both Giraph and Seraph. Fig. 18a shows the total completion time (including execution time and checkpointing time) of each job for both Giraph and Seraph. Overall, the total job completion time of Seraph is reduced by 58.1 percent than that of Giraph. As shown in the figure, the checkpointing in Giraph accounts for a significant part of total job completion time, e.g., 57.8 percent for the five jobs on average. In contrast, checkpointing time in Seraph occupy only 19.3 percent on average, even though Seraph checkpoints every step.

Note that the K-core in Seraph spends more time on the checkpointing than other jobs because Seraph uses the *delta-graph checkpointing* for K-core due to it mutates graph in each superstep. To evaluate the overhead of delta-graph checkpointing, we collect each superstep's checkpoint size in HDFS for both Giraph and Seraph. For Giraph, each checkpoint needs to save the whole graph data and other job-specific data. However, Seraph's *delta-graph checkpointing* only needs to checkpoint the changed part of graph in each superstep. Fig. 18b depicts the cumulative checkpoint size of K-core in each superstep. According to the figure, each superstep's checkpoint size of Giraph is about 5.8 GB. However, Seraph averagely just checkpoints 522 MB in each superstep. After the job finished, the size of total checkpoint of Giraph is $2.35\times$ that of Seraph.

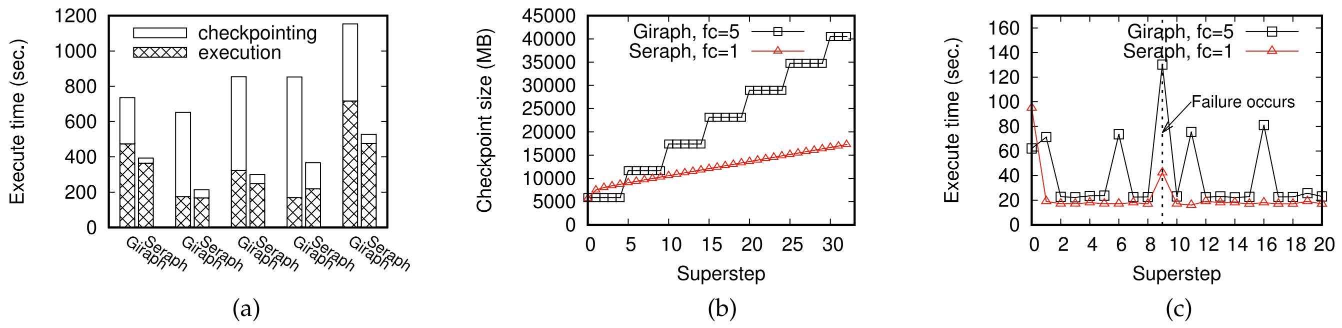

Fig. 18. The fault-tolerance comparison between Giraph and Seraph, Giraph makes checkpoint per five iterations and Seraph per iteration: (a) The execution time and checkpointing time for Giraph and Seraph. (b) Cumulative checkpointing data size of k-core job. (c) Illustration of checkpointing time and recovery time for Pagerank

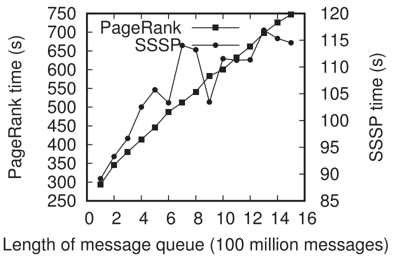

Fig. 19. The asynchronous execution time with varied length of message queue.

7.3.2 Recovery Time

We intentionally generate a failure to a running job (e.g., pagerank), and test the recovery time for both Giraph and Seraph. For fairness, we let failure occur in superstep 9, which is the average number between two checkpoint (superstep 6 and 11) for Giraph. Fig. 18c illustrates the checkpointing time and recovery time when runing Pagerank jobs on Giraph and Searph, respectively. Since Seraph checkpoints every superstep, it can recover from the last superstep. For Giraph, it checkpoints every five steps, thus has to roll back to the latest checkpoint in superstep 6. From figure we see that the recovery time for Seraph is reduced about 76.4 percent compared with Giraph.

7.4 Asynchronous Execution

7.4.1 Message Controlling

According to Theorem 2, to get the best performance, the message queue should be controlled as short as possible (but should be larger than the threshold τ). Based on the controlling strategy in Section 5.3, the threshold τ we obtained is 40 million messages when the CPU utilization downgrading unit is set as $\mu = 0.05$. Note that the 40 million messages is very small compared to the total number of messages in computation. In the first experiment, we control the message queue varying from 40 million to 1.6 billion of messages, and check its impact on the execution performance.

Fig. 19 shows the impact of message queue length to the computation performance for two algorithms, PageRank and SSSP on *snp* graph. We can see that the execution time is monotonically increasing with the size of message queue. For PageRank, with message controlling, it can gain around $3\times$ performance improvement over the worst case. For SSSP, it also can improve 26.3 percent, even there are few messages generated during the whole computation. This experiment shows that carefully choosing message queue size can greatly improve the performance.

Fig. 20 depicts the dynamic length of message queue during the whole computation of PageRank on *snp* graph. In warm-up stage (the first 20 seconds), the number of messages quickly goes up to 180 billion due to without controlling. After this stage, Seraph starts the controlling mechanism, and the length of message queue is quickly controlled to the minimum value (i.e., 40 million) until the end. This can prove that our controlling mechanism can adaptively adjust the total length of message queues to a constant number in distributed environment.

7.4.2 Comparison with Synchronous Engine

We compare the asynchronous engine with the synchronous engine in terms of computation time and memory usage.

Fig. 21 shows the execution time for the two algorithms on the two graph data sets with the two computation model. On average, the asynchronous model can reduce execution time by 59.2 percent than synchronous model. This is because the per iteration synchronization prevent the computation to observe the latest data, so as to affect the execution time. Fig. 22 is the average memory usage comparison for these two models with different algorithms. Compared with the synchronous model, our asynchronous model can control the size of message queue in a very small range, thus it can reduce memory usage by 32.8 percent.

7.4.3 Comparison with PowerLyra

We also compare Seraph with a state-of-the-art asynchronous system, PowerLyra, on a cluster with 10 `r3.2xlarge`

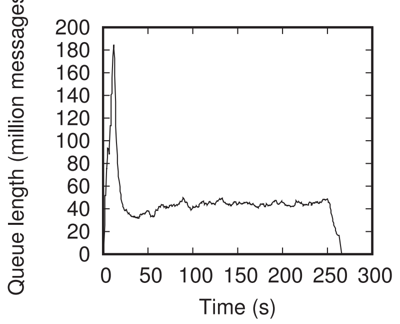

Fig. 20. The dynamic length of message queue under the controlling (PageRank).

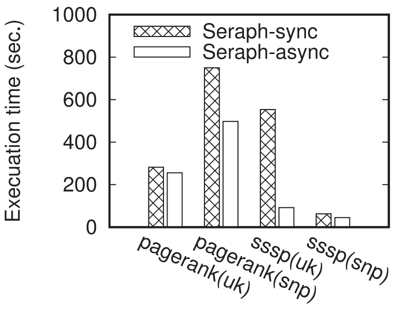

Fig. 21. The comparison of execution time between synchronous and asynchronous engine.

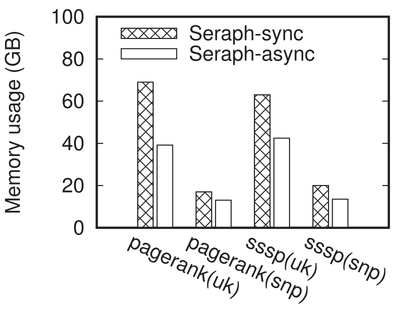

Fig. 22. The comparison of memory usage between synchronous and asynchronous engine.

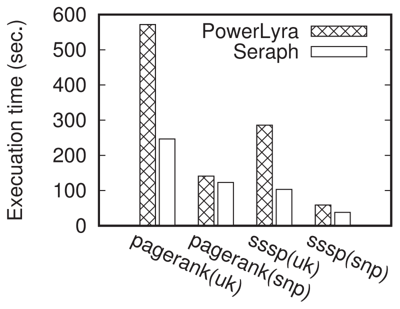

Fig. 23. The comparison of execution time between PowerLyra and Seraph.

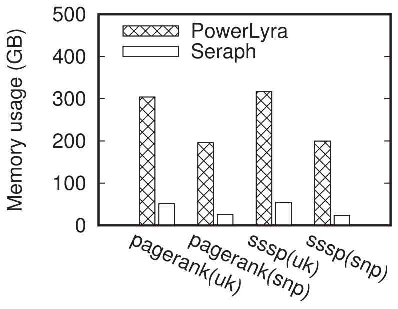

Fig. 24. The comparison of memory usage between PowerLyra and Seraph.

EC2 instances.[2] Fig. 23 shows the execution time comparison with different algorithms on two graph datasets. To be clarified, PowerLyra is implemented in C++ while Seraph is a JVM-based system. Even though, Seraph can significantly outperform PowerLyra with up to 176 percent performance improvement. This is mainly because PowerLyra's asynchronous scheduling involves heavy lock contention overhead (as the recent research [22] shown), whereas Seraph avoids this through using the incremental message-passing based model. Also, the message controlling mechanism used in Seraph could not only help improve the algorithms's convergence speed but also save memory overhead of storing the messages. As shown in Fig. 24, PowerLyra uses 7× more memory space than Seraph on average when they execute the same algorithms.

8 Related Work

Recently, several graph processing systems have been proposed, such as Pregel [1], Giraph [5], GPS [4], GraphLab [2] and PowerGraph [3]. These systems are based on BSP model [7] to implement a vertex-centric computation. Other type of graph computation systems targets multi-core environment, such as Grace [6] and Galois [23]. However, all of them suffer the problems of memory inefficient and high fault tolerant cost when executing multiple jobs in a parallel manner, as we explained in Section 2.

Some systems are proposed to support continuously updated graph. Neo4j [24] and HyperGraphDB [25] are graph databases. They both support simple graph traversals and queries. Due to focusing on the transaction, they are not suitable for graph computation. Kineograph [26] supports graph-mining on a stream of incoming data. Similar with Seraph, they both enable to capture the changing graph through constructing new snapshot continuously. However, Kineograph focuses on reducing the computation latency through incremental computing, without considering the case of concurrent jobs.

9 Conclusion

Seraph is a large scale graph processing system that can support multiple concurrent jobs running on a shared graph. The basic idea of Seraph is decoupling the computation model of current graph computing, which allows multiple concurrent jobs to share graph structure data in memory, as well as providing efficient scheduling mechanism. Our evaluation demonstrated that significant benefits on both execution performance and memory consumption can be gained from the decoupling model.

2. We use EC2 in this experiment because some software libraries on our cluster have some incompatibility issues with PowerLyra.

Acknowledgments

This work was supported by the National Natural Science Foundation under Grant No. 61472009 and State Key Program of US National Science Foundation under Grant No. 61232004, Shenzhen Key Fundamental Research Projects under Grant No. JCYJ20151014093505032. Zhi Yang is the corresponding author of this paper.

References

[1] G. Malewicz, et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[2] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A new parallel framework for machine learning," in *Proc. Conf. Uncertainty Artif. Intell.*, Jul. 2010, pp. 340–349.
[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, Oct. 2012, pp. 17–30.
[4] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proc. 5th Int. Conf. Sci. Statist. Database Manage.*, Jul. 2013, Art. no. 22. [Online]. Available: http://ilpubs.stanford.edu:8090/1039/
[5] (2016). [Online]. Available: http://giraph.apache.org/
[6] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan, "Managing large graphs on multi-cores with graph awareness," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, pp. 4–4.
[7] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.
[8] (2013, Aug.). [Online]. Available: http://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920
[9] (2016). [Online]. Available: http://www.facebook.com
[10] Y. Shen, G. Chen, H. V. Jagadish, W. Lu, B. C. Ooi, and B. M. Tudor, "Fast failure recovery in distributed graph processing systems," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 437–448, Dec. 2014. [Online]. Available: http://dx.doi.org/10.14778/2735496.2735506
[11] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: An efficient, low-cost system for concurrent graph processing," in *Proc. 23rd Int. Symp. High-Performance Parallel Distrib. Comput.*, 2014, pp. 227–238. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600222
[12] (2016). [Online]. Available: http://www.renren.com
[13] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proc. 20th Int. Conf. World Wide Web*, 2011, pp. 587–596.
[14] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. 13th Int. Conf. World Wide Web*, 2004, pp. 595–601.

[15] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming graph partitioning for massive scale graphs," in *Proc. 7th ACM Int. Conf. Web Search Data Mining*, 2014, pp. 333–342. [Online]. Available: http://doi.acm.org/10.1145/2556195.2556213

[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley, 1986.

[17] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 517–528.

[18] (2016). [Online]. Available: http://mina.apache.org/

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.

[20] M. Zaharia, et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Networked Syst. Des. Implementation*, 2012, pp. 2–2.

[21] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Comput. Syst.*, vol. 22, no. 3, pp. 303–312, Feb. 2006.

[22] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An experimental comparison of Pregel-like graph processing systems," *Proc. VLDB Endowment*, vol. 7, no. 12, pp. 1047–1058, Aug. 2014.

[23] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 456–471.

[24] (2016). [Online]. Available: http://neo4j.org

[25] B. Iordanov, "HyperGraphDB: A generalized graph database," in *Proc. Int. Conf. Web-Age Inf. Manage.*, 2010, pp. 25–36.

[26] R. Cheng, et al., "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 85–98.

Jilong Xue received the BS degree from Xidian University, in 2011. He is currently working toward the PhD degree at Peking University. His research interests include the areas of distributed computing systems, such as for large-scale graph processing, and machine learning.

Zhi Yang received the BS degree from the Harbin Institute of Technology, in 2005, and the PhD degree in computer science from Peking University, in 2010. He is currently an associate professor with Peking University. His research interests include the areas of security and privacy, networked and distributed systems, and data intensive computing.

Shian Hou received the BS degree from Peking University, in 2013. He is currently working toward the MS degree at Peking University. His research interest include the areas of graph computing systems.

Yafei Dai received the PhD degree in computer science from the Harbin Institute of Technology. She is currently a professor with the Computer Science Department, Peking University. Her research areas include networked and distributed systems, P2P computing, network storage, and online social networks. She is a member of the IEEE.