Seraph: an Efficient, Low-cost System for Concurrent Graph Processing

Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou and Yafei Dai
Department of Computer Science, Peking University
Beijing, China
{xjl, yangzhi, quzhi, hsa, dyf}@net.pku.edu.cn

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 single job, existing systems incur great inefficiency in memory use and fault tolerance. Motivated by this, in this paper we introduce Seraph, a graph processing system that enables efficient job-level parallelism. Seraph is designed based on a decoupled data model, which allows multiple concurrent jobs to share graph structure data in memory. 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. Moreover, Seraph adopts an incremental checkpoint/regeneration model which can tremendously reduce the overhead of checkpointing. We have implemented Seraph, and the evaluation results show that Seraph significantly outperforms popular systems (such as Giraph and Spark) in both memory usage and job completion time, when executing concurrent graph jobs.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of systems; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

Keywords

Graph processing; Concurrent jobs; Graph sharing; Fast recovery

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 [19], GraphLab [18], PowerGraph [13], GPS [27], Giraph [11] and Grace [25]. These systems allow users to easily process large-scale graph data based on certain computation models (e.g., BSP [32] model).

HPDC'14, June 23–27, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2600212.2600222.

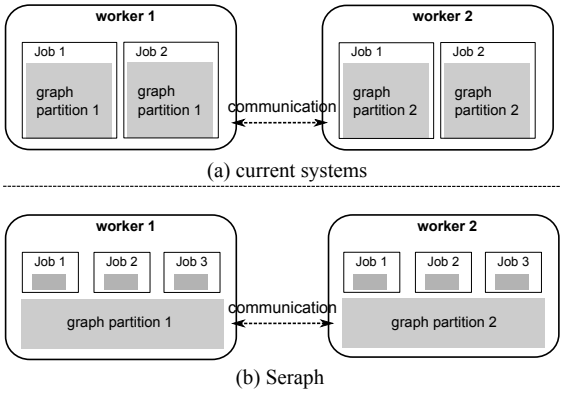

Figure 1: The manner of executing concurrent jobs for (a) existing graph systems and (b) Seraph

Recently, online social networks begin to leverage these graph systems to process their daily computation and analytics [28]. For example, Facebook [10] uses Apache Giraph [11] to process various graph algorithms across their many products, including typical 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 of social networks, it easily generates jobs overlapping in time (i.e., concurrent jobs). Our measurements on a large Chinese social network's computing cluster confirm that more than 83.4% of time has at least two graph jobs executed concurrently.

Although existing systems can process single graph job efficiently, they incur high cost when handling multiple concurrent jobs. Specifically, existing systems do not allow multiple jobs to share the graph data in memory. In particular, these systems usually tightly combine graph structure and job-specific vertex value together. As a result, each individual job needs to maintain a separate graph data in memory, introducing inefficient use of memory, as illustrated in Figure 1(a). Moreover, to tolerate failures, multiple concurrent jobs have to periodically checkpoint the large volume of memory data into disk, which significantly incurs I/O bottleneck and delays the computation.

In this paper, we try to solve these problems through decoupling the data model and computing logics in current systems. First, through decoupling graph structure data and job-specific data, concurrent jobs can share one graph structure and thus greatly save the memory occupation. Second, through decoupling the computation into more specific processes, we only need to checkpoint a small

amount of necessary data which can be used to recover the whole computation states.

Based on this decoupled model, we propose a new graph processing system, called Seraph, which can efficiently execute multiple concurrent graph jobs. Seraph enables multiple jobs to share the same graph structure data in memory, as illustrated in Figure 1(b). Each job running in Seraph only needs to maintain a small amount of job-specific data.

To maximize job-level parallelism, Seraph incorporates three new features: First, it adopts a "copy-on-write" semantic to isolate the graph mutations from concurrent jobs: once a job needs to modify the graph structure, Seraph copies the corresponding local region and applies the mutations, without affecting other jobs. Second, Seraph uses a *lazy snapshot protocol* to maintain the continuous graph updates and generates consistent snapshot for each new submitted job. Finally, Seraph implements an efficient fault tolerant mechanism, which uses *delta-graph checkpointing* and *state regeneration* to efficiently tolerate both job-level and worker-level failures.

We have implemented Seraph system in Java based on our design. Seraph adopts the master-slave architecture. Each Seraph instance can load a graph and execute one or more jobs concurrently on it. Graph mutation is allowed for each job during computation and system guarantees the isolation. If there is no job running, Seraph will enter hibernate mode to save memory occupation. In addition, to avoid the aggressive resource competition, Seraph implements a job scheduler to control the execution progress and message flow of each concurrent job.

We evaluate the overall performance of Seraph by comparing it with popular systems, including Giraph (graph-parallel system) and Spark (shared memory data-parallel system). Our experiments show that Seraph could reduce 67.1% memory usage and 58.1% 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, we shows that Seraph is nearly $6\times$ faster than Spark when executing graph algorithms, and can reduce memory usage of Spark by $14\times$.

This paper is organized as follows. Section 2 describes the background and our observation on graph computation. In Section 3, we introduce the design of Seraph. The graph sharing and lightweight fault tolerance are introduced in Section 4 and Section 5 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. BACKGROUND

In this section, we first introduce the practice usage of graph computing system in online social network, and then reveal the inefficiency of existing systems through measuring the real workload. Finally, we address the inherent inefficiency through abstracting the data model of current systems.

2.1 Graph Computation in Practice

Recently, several popular graph-parallel computation systems, such as Pregel [19], Giraph [11] and PowerGraph [13], have been widely used in the enterprises like online social networks (e.g., Facebook [10] and Renren [26]). Typically, these systems are deployed as an open platform in a company, and used to process large number of daily graph jobs across different products. These jobs include friend recommendation, advertisements, variants of Pagerank, spam detection, kinds of graph analysis, label propagation, clustering even some third-party developer's jobs [28].

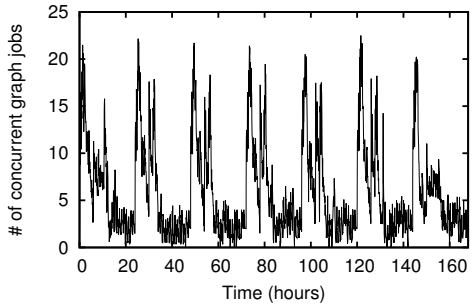

Figure 2: One week's workload of graph computation in real social network cluster

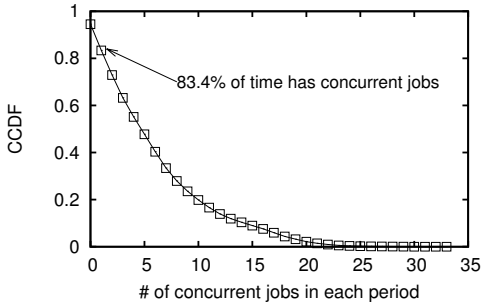

Figure 3: The distribution of graph jobs running on one social network's computing cluster

With increasing number of jobs running on the same platform of social networks, it easily generates jobs overlapping in time (i.e., concurrency). To demonstrate this scenario, we collect one month job execution logs from a large Chinese social network. Figure 2 depicts one week's workload (number of jobs at each time) of graph computation from Nov 1, 2013. The stable distribution shows that a significant number of jobs are executed concurrently every day. At peak time, there are more than 20 jobs submitted to the platform. We also depict the complementary cumulative distribution of the number of concurrent jobs in each time period (one second) in Figure 3. As it shows, more than 83.4% of time has at least two jobs executed concurrently. The average number of concurrent jobs is 8.7.

However, the existing graph processing systems are all designed for single job execution, which are much inefficient when processing multiple concurrent jobs. The inefficiencies mainly fall in two aspects: *inefficient use of memory* and *high cost of fault tolerance*.

Inefficient memory use: For most existing graph-parallel systems, one vital cost of job execution is to maintain the large graph data in main memory. Taking Giraph[1] as an example, Figure 4 shows the memory usage of Giraph job when running Pagerank algorithm on various graph datasets [1, 2]. From the figure we see that the graph data occupies majority of memory as compared with vertex values and messages (job-specific data), and its proportions are varying from 71% to 83% for different datasets. Due to the heavy memory cost of graph computation, it is hard to execute many concurrent graph jobs in a resource-limited cluster. For ex-

[1]Giraph is one of most popular open source graph computing system, which is currently used in Facebook.

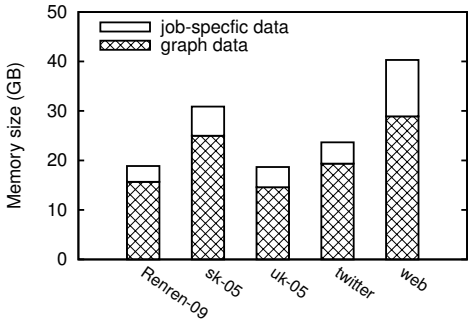

Figure 4: The memory usage of Giraph

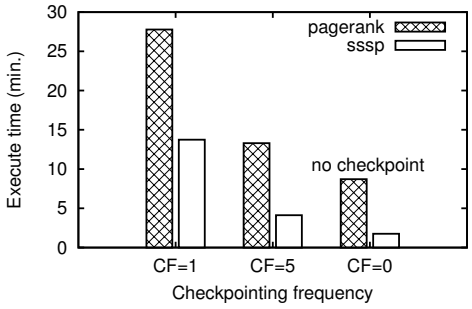

Figure 5: The latency incurred by checkpoint mechanism

ample, in our experiments, a powerful HPC cluster with 16 nodes (each with 64GB memory), could only support at most 4 concurrent Giraph jobs over a graph with 1 thousand million vertices.

Notice that almost all concurrent jobs are executed on same underlying graph. For example, most graph computations and analytics in Facebook are executed on a same friendship graph [28]. Thus, the graph structure data is duplicated in memory when running concurrent jobs. The memory usage can be reduced by allowing multiple parallel jobs to share graph data in memory. To do so, we need to decouple the graph structure data from the job-specific data and resolve conflicts when graph mutations occur. We will address the graph sharing functionality in Section 4.

High fault tolerance cost: The existing graph systems, such as Pregel, Giraph, GraphLab and PowerGraph, rely on checkpointing and rollback-recovery to achieve fault tolerance [19, 11, 18, 13]. When failures occur, program rollbacks to the latest checkpoint and continues the computation. Each checkpoint needs to store all the data required by computation in that step.

However, for existing graph systems, the computation in each superstep takes all the data (graph data and other computation states) as program input, leading to a high cost of checkpointing. First, the checkpoint size is very large. All the graph, vertex states and exchanging data should be saved in a persistent storage. In Giraph, running a single Pagerank job on a 100 gigabytes graph with checkpointing at each superstep, the total data saved in HDFS are more than 9 terabytes, which is really a heavy cost for real cluster.

Second, saving the large checkpoints involves a great executing latency. Figure 5 shows the execution time of running Pagerank and SSSP algorithm on a 25 million vertices graph with different checkpointing frequency (CF). As the figure shows, the total overhead of execution time is increased by 218.8% due to checkpointing when CF=1, and 52.7% when CF=5 for Pagerank. Similar result is showed for SSSP. The heavy cost of checkpointing makes most users prefer to disable the fault tolerance mechanism.

In section 5, we will propose delta-graph checkpointing and state-regeneration mechanisms to address these checkpoint overheads.

2.2 Data Model of Graph Computation

To fundamentally address the inherent reason that cause the problems of existing systems, we abstract the data model of current graph computations. Although most of these systems use different computation models, e.g., BSP model for Pregel [19], and GAS model for PowerGraph [13], they are similar from the perspective of data management. Specifically, we find that existing systems are based on the same abstracted data model, as showed in Figure 6.

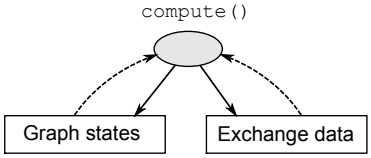

Figure 6: Data model of graph computation

In this model, the graph states are composed of two parts: graph structure and associated vertex values on the graph. The graph structure, simply denoted as *graph*, is made up of vertices and edges. The *values* are commonly associated with vertices. During the computation, there will be some exchanging data (e.g., *messages*) generated to assist the computation. For example, the exchanging data in Pregel is message data [19], and in GraphLab is replicas of vertices [18]. In this model, The computation is realized through executing user-defined program `compute()` on each vertex. In each superstep, the computation takes both the graph states and exchange data as input, and updates both of them during the execution. The computation of a graph algorithm continuously mutates the graph states until some termination condition is satisfied.

However, this tightly-coupled model inherently causes the inefficiency of existing systems when running concurrent jobs (the problems of memory inefficiency and high fault-tolerance cost). In this model, since the graph states combine graph structure and *job-specific* vertex value together, the system do not allow concurrent jobs (running on same graph) to share one common in-memory graph, leading to a high memory waste. Meanwhile, for fault tolerance, the system has to checkpoint a large amount of graph computation state, including graph structure, vertex state and the exchanging data. leading to a high checkpoint/recovery cost.

3. SERAPH DESIGN

To avoid the high memory usage and fault tolerance cost, we first present a novel fine-granularity data model to support graph computing. Based on this model, we then propose an efficient, low-cost graph computing system.

3.1 Decoupled Model

Essentially, the high cost of memory use and fault tolerance are both caused by the tightly-coupled model in Figure 6. As we have explained, the graph structure and vertex states are tightly-coupled, making it impossible to share the graph data over concurrent jobs. Moreover, the tightly coupled computing program requires the system to record all in-memory data, leading to a high checkpoint cost.

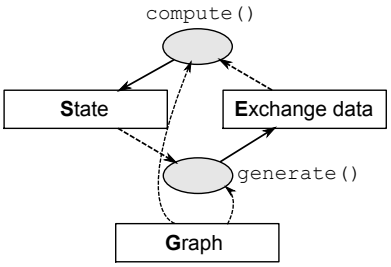

Figure 7: Data model of graph computation

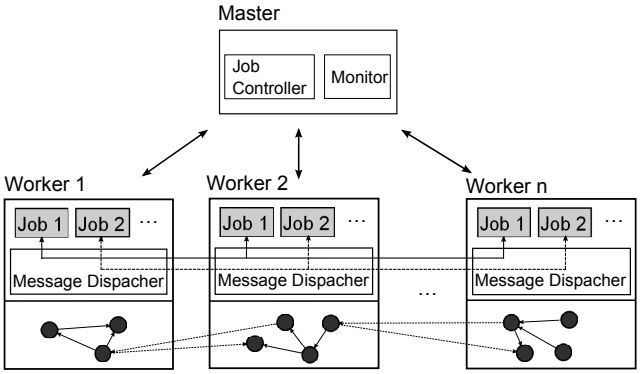

Figure 8: The system architecture of Seraph

To efficiently avoid these costs, we propose a novel fine granularity data model for the graph computation, called **GES** (**G**raph-**E**xchange-**S**tate) model. In GES model, a graph computation program is based on three types of data: **graph**, **exchanging information** and **state**. The graph $\mathbf{G}$ is commonly static data describing the graph structure and some inherent graph properties, such as 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 set of vertex values associated with each vertex, 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 between these three parts of data, as illustrated in Figure 7. The following functions shows the dependence between these data.

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

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

In each superstep, the function f_c takes the exchanging data $\mathbb{E}$ (messages) 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. Taking the Pagerank algorithm as an example, the algorithm is executed through several iterations (supersteps). Within a superstep, each vertex first does a computation which takes received messages from other vertices as input and updates its own new value (**state**). Then based on the new value, generates and sends its impacts (**exchanging data**) to its outgoing neighbors. Based on GES model, the computation in each superstep can be described as,

$$s_i \leftarrow \alpha \sum m_{ji} + (1 - \alpha) \ / \ N \tag{3}$$

$$m_{ij} \leftarrow s_i \ / \ |e_i|, \forall j \in e_i \tag{4}$$

where s_i is the state of vertex i, m_{ij} is the message from vertex i to j and e_i is neighbor set of vertex i.

The GES model has two major advantages: First, it decouples data into the job-specific data (i.e., states and exchanging data) and static data (i.e., graph), thus enabling the job-level parallelism on a shared graph, i.e., multiple jobs jointly use one graph dataset in memory. This can greatly reduce the memory occupation due to the fact that the size of graph data is very large, as compared with job-specific data. In particular, suppose that an algorithm working on a graph with N_v vertices and N_e undirected edges, the total memory size of the *graph* structure is $O(N_v + 2N_e)$. The states data is associated with each vertex, so the total space for states data is $O(N_v)$. The worst case of the memory usage for exchanging data is that each vertex obtains all the state of their neighbors. In this case, the memory space for exchanging data is $O(N_e)$. As N_e is usually much larger than N_v (i.e., $N_e \gg N_v$), we see that graph structure occupies at least $2/3$ of the total memory usage.

Second, the GES model decouples the computation program into two separated processes: *compute* and *generate*. With this decoupling, exchanging data in any superstep can be generated by state data in previous step according to the GES model. Thus, to recover a job in case of failures, we only need to checkpoint the small amount of *state* data, which makes the checkpoint very low-cost.

3.2 System Overview

To implement the GES model, we design *Seraph*, a graph computation system that can support parallel jobs running on a shared in-memory graph. Figure 8 shows the high-level architecture of a Seraph. 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 global 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. We shall detail these functions in the Section 4.

At startup, a worker registers with the master, and periodically sends a heartbeat to demonstrate its continued availability. When a job is dispatched to a worker, a new executor is created and invoked. The executor is a generic component that performs computation in a superstep. It loops through all vertices and call user defined external programs.

The computation on each vertex is simply separated into two sub functions: `compute()` to updates the new vertex value, and `generate()` to generate messages to neighbors for next superstep. Through this decoupling, the value and message in runtime memory can be generated separately. Based on this characterises, we design a lightweight fault tolerant mechanism. We detail the design of its key components in Section 5.

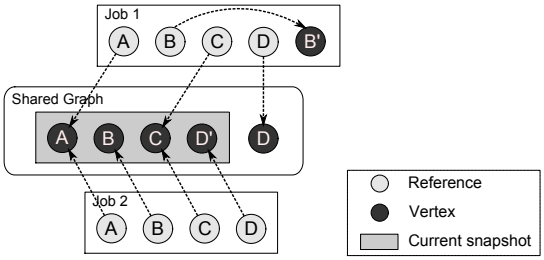

Figure 9: Illustration of graph mutation and graph update in Seraph

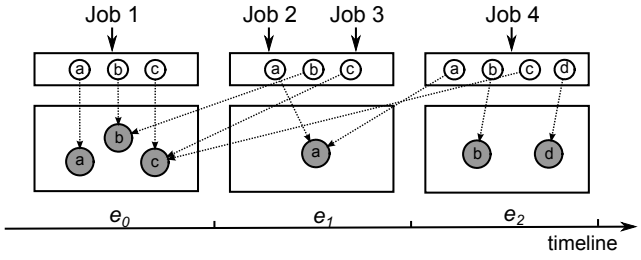

Figure 10: Illustration of lazy snapshot generation in Seraph

4. 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 a reference to it. Then each job creates its own job-specific data (vertex states and exchanging data) in local memory space.

To efficiently support graph sharing, we have to address some conflicts caused by the graph sharing. First, the local 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 isolation, the mutations would interfere other jobs due to using a common graph data. Second, we should maintain a proper 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.

4.1 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 the algorithm (i.e., a job) needs to modify the edge list of a vertex, it first creates a `delta graph` object in local space and copies the corresponding edge list to it. Then the mutations are applied on that local copy. Meanwhile, the job changes its reference of this vertex to the one in local graph. The original vertex (and edge list) is still used by other jobs. This method only copies mutated vertices for the corresponding job, thus incurring little memory overhead. After job got finished, these copied vertices in local memory will be released.

Figure 9 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, noted as vertex B', then modifies it locally and change 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 little part of graph during the computation, the copy-on-write is 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.

4.2 Graph Update

Different from graph mutation caused by running jobs, graph update are caused by the changing of underlying graph (e.g., the new arrival of edges and vertics). Seraph guarantees that a new submitted job can see all 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. Graph managers at individual workers collaboratively maintain graph snapshot. 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 following subsection.

Figure 9 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, 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 finished (the number of reference on D will be zero), the vertex D will be released by graph manager.

4.3 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.

However, existing methods for generating consistence are too costly for Seraph. First, to implement the consistence in a distributed system, it needs to assign a global timestamp [5, 17] or sequence number [6] on each update unit (e.g., edge in graph system). However, assigning a timestamp on each edge will double the memory occupation of graph data. Moreover, it needs a complex distributed algorithm to generate snapshot, such as Chandy-Lamport algorithm [4]. In fact, the traditional mechanism provides a consistence at any moment, which is unnecessary in our system. Seraph only needs a consistent snapshot of graph when a new job is submitted. This relaxed requirement enables us to design more low-cost snapshot protocol.

Therefore, we present a *lazy snapshot protocol*, where a global consistent snapshot of graph is generated only when a new job is submitted. Our idea is packaging the updates in batch, so as to save the memory of storing timestamps. In Seraph, each update is sequentially appended into the master node. When a new job is submitted, the master will notify each worker's graph manager to step into a new epoch, and package all local buffered updates into a transaction and dispatch each update to corresponding workers, as illustrated in Figure 10. Here, we define that all the updated vertices in epoch n compose a *delta-graph* Δ_n. A transaction is updated through a two phases commit protocol. In the

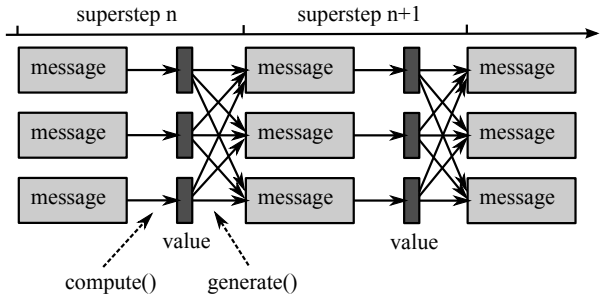

Figure 11: The data transformation and computation model of Seraph

first phase, master dispatchs updates to workers and receives acknowledgements from all workers. The worker node just caches but not applies the updates. In second phase, master sends confirm messages to each worker and worker nodes apply updates to graph. When a transaction is successfully submitted, all the pending jobs can be dispatched to workers and begin their computation.

5. LIGHTWEIGHT FAULT TOLERANCE

Current graph processing systems rely on checkpoint/rollback-recovery to achieve fault tolerance [19, 11, 18, 13]. For example, in Pregel or Giraph, user can optionally choose to set checkpoint at the end of each superstep, and the program will roll back to the latest checkpoint when failures occur. However, existing graph computation systems need to checkpoint all the values, messages and graph data into persistent storage, incurring a heavy checkpoint cost and long checkpoint delay.

In contrast, Seraph is able to make lightweight fault tolerance based on the decoupled data model. Through sharing the graph, Seraph only needs to keep one graph copy to tolerate failures of multi-jobs. Also, we propose a delta-checkpointing mechanism to checkpoint the graph incrementally. Further, through decoupling computation model, Seraph can recover messages from previous values, and thus prune the checkpointing traffic caused by the messages. In the following, we will introduce all of these designs in details.

5.1 Computation model

Seraph's computation model is derived from the Pregel. In general, the graph computation consists of several supersteps, in each of which a vertex can collect messages and execute user-defined functions. In Pregel, all these actions are coupled in a user implemented function: `compute()`. This vertex-centric approach can express the behavior of a broad set of graph algorithms [35].

However, according to the GES model in section 3.1, a better way to express the behavior of graph computation is separating the `compute()` function into two individual functions: `compute()` and `generate()`. Here, the new `compute` interface is to compute new values based on received messages from last superstep, and `generate` interface is to generate the messages for the next superstep according to the new values. Figure 11 shows the data transformation and computation model of Seraph. To show how our model can easily express typical graph algorithms, Figure 12 illustrates the "compute-generate" implementation of Pagerank, SSSP and Wcc algorithms.

After this separation, Seraph can use the `generate` function to regenerate message data based on the value data of last step. Specifically, for Pregel model, if superstep i fails, the system recoveries through rerunning the `compute` function:

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

where M_i and V_{i-1} are the messages and values checkpointed in last superstep. However, by adding the `generate` interface, the recovery process changes as:

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

Thus, we only need to checkpoint a small amount of 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.

5.2 Delta graph checkpointing

After checkpointing values for each job, Seraph can recover the job-specific data (including message and value data), so as to tolerate any job-level failures. Now, we begin to look at how Seraph recovers the shared graph data when a worker fails. Notice that each job is running on a different snapshot of graph. When failures occur, Seraph need to recover the corresponding 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 Figure 10). Graph snapshot in epoch e_n can be generated though the graph snapshot in epoch e_{n-1} and delta-graph in e_n:

$$G_n = G_{n-1} \oplus \Delta_n \tag{5}$$

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

$$S_{G_1 \oplus G_2} = (S_{G_1} \setminus S_{G_2}) \cup S_{G_2} \tag{6}$$

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 reduce the checkpoint cost.

5.3 Failure Recovery

The failures in Seraph are classed into two kinds: job failure and worker failure. For job failure, since the graph is decoupled with job, only the values and 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 message, Seraph can quickly recover from failure.

When a worker fails, the graph partition on that worker will be lost. Seraph assigns the graph partition to a standby worker and construct 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.

6. SERAPH IMPLEMENTATION

We have fully implemented Seraph in Java. The communication between the workers in Seraph is based on Apache MINA [20]. Like Giraph, the underlying persistent storage is using Hadoop

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)
    minValue = Min(minValue, m);
  setValue(minValue);

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

Wcc

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

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

Figure 12: The example of implementations of graph algorithms based on compute-generate model

HDFS [30], and all the checkpointing data are stored in HDFS. Seraph only loads a single copy of graph into main memory to support multiple concurrent jobs. In the following, we give the specific implementation details of graph management and job scheduling.

6.1 Graph Management

Graph sharing: We have introduced the design of graph sharing in section 4, now we explain how Seraph implements this mechanism through decoupling graph and job-specific data.

In existing systems (e.g., Giraph), a graph in memory is stored as a set of `Vertex` objects. Each `Vertex` object includes vertex value, message queue and an adjacency list storing neighbor edges. Notice that the edge list is unique to all jobs and could be very large (e.g., Facebook users have an average number of 190 neighbors [31]). Thus, Seraph extracts edge list from `Vertex` object and forms a new global `graph` set containing each vertex's edge list. After this change, each job only needs to make 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 from existing systems, Seraph is an online running platform due to supporting the updates on graph. However, in the case of no jobs running, the memory used for storing the graph is wasted. To overcome this problem, we implement a hibernate mode for Seraph.

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 storing graph 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.

6.2 Job Scheduling

Parallel job execution can improve the system throughput due to full resource utilization (e.g., network and CPU). However, aggressive resource competition might increases job execution time. To avoid this, 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. The higher priority job gains resource access firstly, whereas the lower priority job gains access once the resource is not used by the higher one. Hence, Seraph can guarantee that early jobs are finished as quickly as possible, meanwhile, 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 dispatch, as shown in Figure 13.

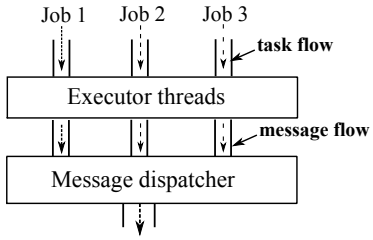

Figure 13: Two level flow control model for job scheduling in Seraph

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 flow 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 dispatcher. Message dispatcher uses a priority queue to buffer messages from all jobs. In each step, Seraph fetches messages from the head of queue (i.e., the message of highest priority job), and send them to the corresponding worker. 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

Our experiments deploy Seraph on a cluster with 16 machines. Each machine has 64GB of memory and 2.6GHz AMD Opteron

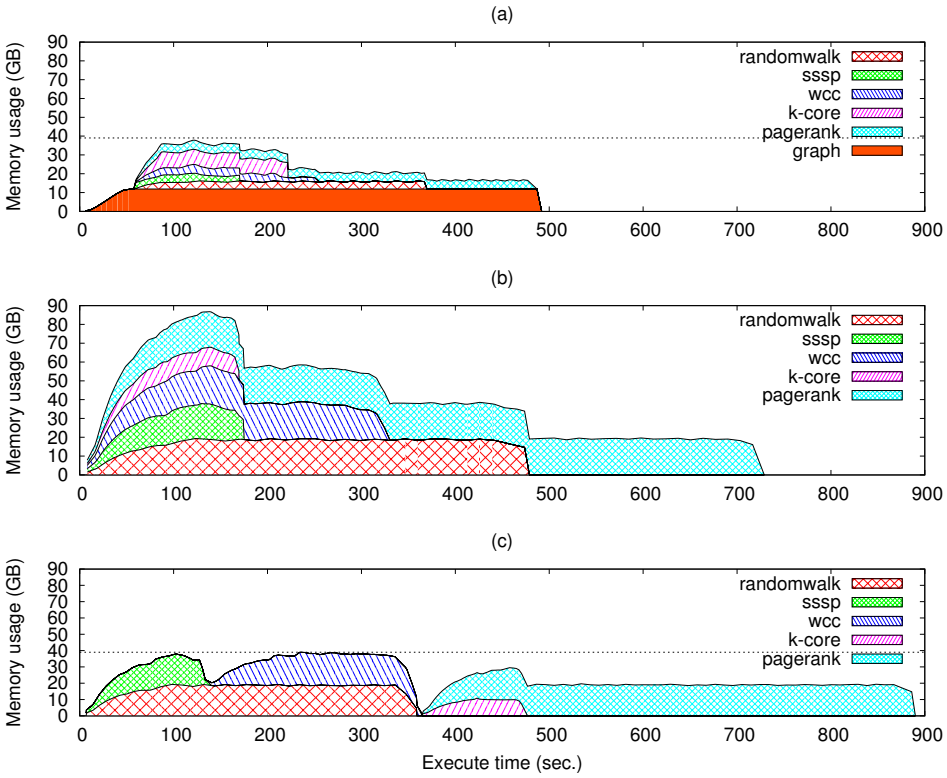

Figure 14: The memory usage for 5 parallel graph jobs on: (a) Seraph with maximum memory resource of 40GB, (b) Giraph with unlimited memory (heap usage is 87GB) and (c) Giraph with same limitation of maximum 40GB memory

4180 Processor (12 cores). All these machines are connected by a gigabit switch. We use 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 and K-core. All these algorithms have different characteristics in resource usage. For example, PageRank is network-intensive, whereas random walk is computation-intensive. Among them, K-core is the only one which mutates graph structure during the computation. 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 graph from history snapshot of Renren network, one of the largest online social network of China. The snapshot graph totally contains more than 25 million vertices (users) and 1.4 billion relational edges.

7.1 Comparison with Giraph

We first examine the performance of Seraph through comparing it with other graph-parallel computing systems. We choose the latest version of Apache Giraph (version 1.0.0), one of the most popular open-source Pregel implementation, as an comparison baseline. In this section, we first compare their computational efficiency (including memory usage and execution time), and then compare their fault-tolerance performance.

7.1.1 *Computational Efficiency*

To compare their computational efficiency for concurrent jobs, we first run all the five different jobs (random walk, SSSP, WCC, K-core and PageRank) simultaneously. Note that we only need to run a single Seraph instance to execute these jobs due to its graph-sharing feature. In contrast, to use Giraph executing jobs in a parallel manner, each job needs to initiate a Giraph instance.

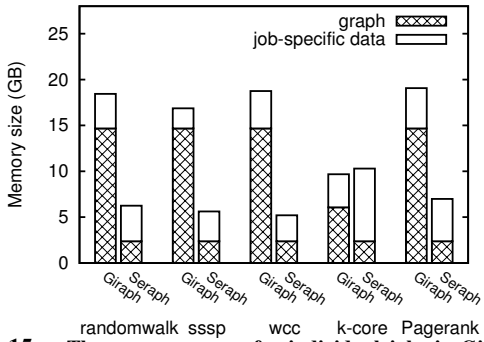

Figure 15: The memory usage for individual jobs in Giraph and Seraph

Memory usage: Figure 14(a) and 14(b) show the memory usage during the execution of Seraph and Giraph, respectively. Seraph first loads the underlying graph, which occupies 11.86GB memory. Then, Seraph executes multiple jobs in parallel over the shared graph. The peak memory usage of Seraph during the execution is

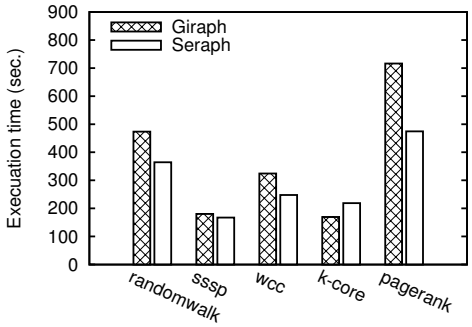

Figure 16: The execution time for concurrent jobs with sufficient memory, the peak memory usage of Seraph and Giraph are 37.8GB and 87GB, respectively

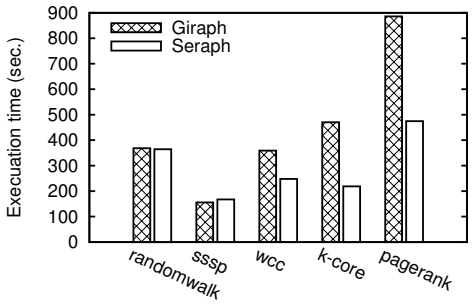

Figure 17: The comparison of execution time for each job executed with limited memory, both set maximum memory as 40GB

37.8GB. 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.

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 Figure 14(b) we see that the total memory rapidly increases with the number of parallel jobs, and its peak memory occupation is about 87GB. 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.

Figure 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% 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.

Execution time: We first compare the execution time of Giraph and Seraph with both running 5 jobs in parallel, i.e., the case in Figure 14(a) and (b). 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 Figure 16, 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% than Giraph. The reduction is mainly brought by the job scheduling mechanism which avoids aggressive resource competition.

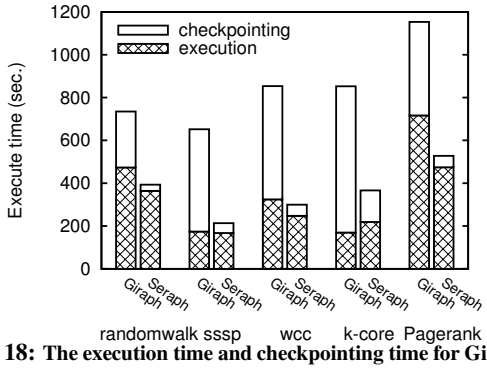

Figure 18: The execution time and checkpointing time for Giraph and Seraph

Next, we limit the peak memory usage of Giraph equal to that of Seraph (e.g., 40GB). With this limitation, Giraph can only execute two jobs in parallel, with others pending until the resource is available. Figure 14(c) shows the memory usage of Giraph in this case. We see that the wcc, K-core and pagerank job needs to be pending for a period, increasing the total execution time.

Given the same memory limitation, Figure 17 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%, as compared with Giraph. In term of the total completion time of five jobs, Seraph brings a reduction of 46.4% as compared with Giraph.

7.1.2 Fault Tolerance

We now compare Seraph with Girpah in the fault-tolerant performance. Seraph checkpoints the vertex values at the end of each superstep. Due to heavy checkpoint cost, Giraph requires to set the checkpoint frequency (i.e., the number of supersteps for every checkpoints), which is a tradeoff between the execution delay and recovery time. Frequent checkpoint reduces the recovery time, but will delay the execution due to checkpoint cost. Based on the experience of [7], we set the checkpoint frequency as 5 for Giraph.

Checkpointing cost: Making checkpoint will delay the job's execution due to writing state data into disk. We evaluate this cost of checkpointing for both Giraph and Seraph. In our experiments, Giraph makes checkpoint every 5 supersteps, whereas Seraph makes checkpoint every superstep. Figure 18 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% 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% for the five jobs on average. In contrast, checkpointing time in Seraph occupy only 19.3% 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. Figure 19 depicts the cumulative checkpoint size of K-core in each superstep. According to the figure, each superstep's

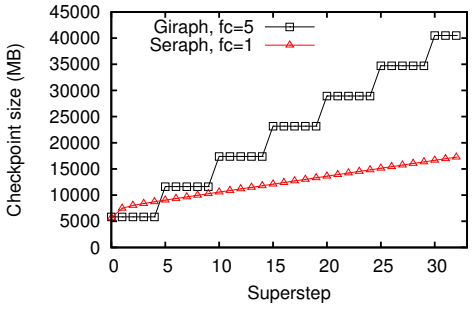

Figure 19: Cumulative checkpointing size of K-core job, Giraph checkpoints every 5 superstep and Serpah checkpoints at every supersteps

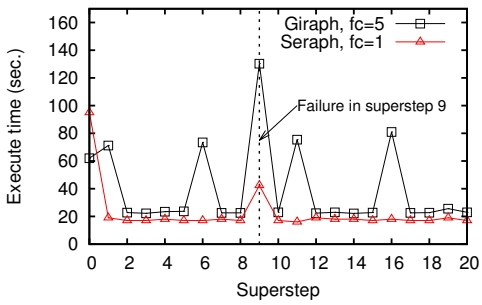

Figure 20: Illustration of checkpointing time and recovery time running Pagerank jobs, Giraph checkpoints every 5 superstep and Serpah checkpoints at every supersteps

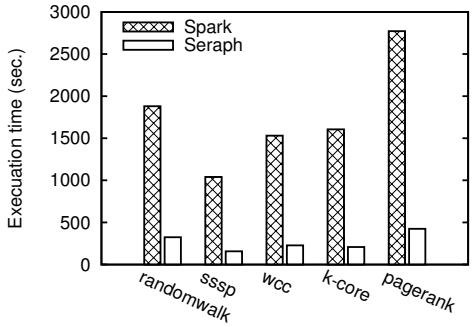

Figure 21: The comparison of execution time of graph algorithms for Spark and Seraph

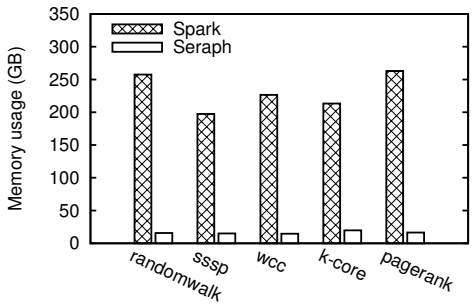

Figure 22: The comparison of memory usage of graph algorithms for Spark and Seraph

checkpoint size of Giraph is about 5.8GB. However, Seraph averagely just checkpoints 522MB in each superstep. After the job finished, the size of total checkpoint of Giraph is 2.35× that of Seraph.

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 occurs in superstep 9, which is the average number between two checkpoint (superstep 6 and 11) for Giraph. Figure 20 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 5 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% compared with Giraph.

7.2 Comparison with Spark

In this section, we compare the performance of Seraph with popular in-memory data-parallel system, Spark [36]. Spark is a data-parallel system which shares in-memory data through RDD (Resilient Distributed Dataset) and tolerate failures using lineage. Spark can also compute graph algorithms. Spark puts all data in memory and naturally separates each of dataset using RDD, and the RDD can be shared during the computation.

We have implemented all the five graph algorithms in Spark. For each implementation, we use three separated RDDs to repersent the graph, vertex states and exchanging data. To achieve the best performance of Spark, we partition each RDD through the operation `partitionBy()` to ensure the data with same key can be co-located in same worker. The graph RDD in each job is set as `persist()` to share for each iterations, except for K-core algorithm. Note that K-core needs to mutate the graph in each iteration.

In our experiments, we use the latest version of Spark (0.8.1) which is deployed on the same cluster as Seraph. We let Spark use all the cores of our cluster (the total number of cores is 192).

Execution time: First, we compare the execution time of Spark and Seraph. Figure 21 shows the execution time of each job. We see that Seraph significantly outperforms Spark. Seraph is more than 6× faster than Spark in terms of average execution time. To understand why Spark is slow, we analyze the time consumption of each operation in Spark. We find the most time-consuming operation in each jobs is *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, in each iteration, it needs to join this two datasets before computation. In contrast, Seraph separates these data but with an explicit connection using reference, which can avoid the overhead of joining.

Memory usage: Spark is an in-memory system which put all datasets (RDD) in main memory by default. To test the memory usage for Spark, we give Spark sufficient memory (each worker with 40GB) when executes each job, and we monitor the usage of memory on each worker using Ganglia[2].

Figure 22 depicts the average memory usage comparison for Spark and Seraph. As the figure shows, Spark uses an order of magnitude larger memory than Seraph. On average, the memory usage

[2]Ganglia is a distributed monitoring system for high-performance computing systems

of Spark is more than $14\times$ larger than that of Seraph. The major reason of large memory occupation in Spark is that Spark will allocate new RDD for each mutation, since RDD is immutable. However, most graph algorithms usually just mutates a little part of the whole dataset, such as the value RDDs in SSSP and Wcc algorithm, and the graph RDD in k-core algorithm. Thus, there are many duplication in Spark. However, Seraph only mutate the changed data which avoids the overhead.

8. RELATED WORK

Seraph is based on a primitive idea in our previous workshop paper [37], with significantly redesigning the system based on decoupled data model and adding lightweight fault tolerance, lazy snapshot protocol and other optimizations. This section summarizes the recent studies on the efforts of processing large graph.

Graph-parallel computing systems: Recently, several projects have been developed for processing large graphs on parallel machines or multi-cores, such as Pregel [19], Giraph [11], GPS [27], GraphLab [18] and PowerGraph [13]. These systems are based on BSP model [32] to implement a vertex-centric computation. There are other systems try to optimize the graph computation. Sedge [35] aims to minimize the inter-machine communication by graph partitioning, and Mizan [16] implements dynamic load balancing mechanism to solve the straggler problem. Other type of graph computation systems targets multi-core environment, such as Grace [25] and Galois [21]. 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.

Graph database and online graph processing system: Some graph storage and management systems are proposed to manage and support efficient queries on the continuously updated graph structure data. Neo4j [12] is a highly scalable, robust (fully ACID) native graph database, and HyperGraphDB [15] is another graph database based on a powerful knowledge management formalism known as directed hypergraphs. They both only support some simple graph traversals and relational-style queries. Due to they focus on the transaction maintaining, they are not suitable for batched graph computations.

Another class of graph systems is online graph processing systems, which can manage continuous incoming graph, support online querying and execute offline graph processing on it. Trinity [29] is a graph engine that supports both online query processing and offline analytic on large graphs. However, Trinity cannot address the problems of concurrent job execution, such as inefficient memory use, and mutation conflicts on a shared graph.

Another online graph system is Kineograph [6], which takes a stream of incoming data to construct a continuously changing graph. It also supports some graph-mining algorithms to extract timely insights from the fast-changing graph structure. 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.

In-memory data parallel computation systems: A bunch of in-memory data parallel computing system are proposed to process large scale data. Spark [36] is a general-purpose and fault-tolerant data-flow abstraction which focuses on the in-memory data sharing through RDD and the fault tolerant using lineage. Same with Seraph, Spark implements very low cost fault tolerance through recomputing on history dataset. However, the dataset in Spark (RDD) is inherently immutable, which incurs many inefficiency when computing graph algorithms. For example, SSSP only mutates a little part of vertex values in most supersteps, however Spark has to create a new RDD in each step, leading to a significant duplication. This problem also exist in GraphX [34], which is another graph computing framework built on the top of Spark engine. Furthermore, both Spark and GraphX can only share RDD or RDG for the operations in inner job, and do not allow them to share the underlying graph structure data at the job level.

There are also some systems allow memory-sharing. Piccolo [24] allows computation running on different machines to share distributed, mutable data via a key-value table interface. However, it has not isolated the job mutations when multiple graph jobs sharing the graph. Piccolo uses Chandy-Lamport (CL) distributed snapshot algorithm [4] to perform checkpointing. However, the cost is high due to checkpointing all in-memory data for graph jobs. Other class of in-memory data sharing systems includes Twister [9] and HaLoop [3], which are based on iterative MapReduce [8]. What's more, Presto [33] is a distributed system that extends R, which can shares sparse structured data (e.g., sparse matrices) to multiple cores. However, these frameworks only perform data sharing inner the jobs.

Other memory shared systems, such as distributed shared memory (DSM) systems [22], in-memory key-value stores like RAMCloud [23], and in-memory file system Tacyon [14], can share in-memory data to different jobs. However, compared with Seraph, they don't provide high-level programming model or specific interface for graph computation. For example, Tacyon provides the operations on files like other disk-based file systems. In addition, all these systems implement fault tolerance through checkpointing all in-memory data, which is highly expensive. In contrast, Seraph combines checkpointing and re-computation to implement a very low cost fault tolerance.

9. CONCLUSION AND FUTURE WORK

This paper introduces Seraph, a large scale graph processing system that can support parallel jobs running on a shared graph. The basic idea of Seraph is decoupling the data model of current graph computation, which allows multiple concurrent jobs to share graph structure data in memory. 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. Moreover, Seraph adopts an incremental checkpoint/regeneration model which can tremendously reduce the overhead of checkpointing.

In the future, we shall study the load balance mechanism given the multiple jobs sharing the same graph, i.e., how to migrate data by taking into account the different loads of jobs on each worker. We also attempt to make Seraph a real-time graph processing platform, where the system keeps up with continuous updates on the graph, and performs incremental graph computation for multiple jobs running on the platform.

Acknowledgement

We would like to thank the anonymous reviewers for their comments. This work was supported by the National High Technology Research and Development Program ("863" Program) of China (Grant No.2013AA013203), the National Basic Research Program of China (Grant No. 2011CB302305), and the State Key Program of National Natural Science of China (Grant No. 61232004). Zhi Yang is the corresponding author of this paper.

10. REFERENCES

[1] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[2] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW 2004*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[3] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI '06*, 2006.

[6] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *EuroSys '12*, 2012.

[7] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, Feb. 2006.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[9] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, 2010.

[10] `http://www.facebook.com`.

[11] `http://giraph.apache.org/`.

[12] `http://neo4j.org`.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI '12*, October 2012.

[14] L. Haoyuan, G. Ali, Z. Matei, B. Eric, S. Scott, and I. Stoica. Tachyon: Memory throughput i/o for cluster computing frameworks. In *7th Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '13, 2013.

[15] B. Iordanov. Hypergraphdb: A generalized graph database. In *Proceedings of the 2010 International Conference on Web-age Information Management*, WAIM'10, 2010.

[16] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *EuroSys '13*, 2013.

[17] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010.

[19] 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 '10*, 2010.

[20] `http://mina.apache.org/`.

[21] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[22] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, Aug. 1991.

[23] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, Jan. 2010.

[24] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[25] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, 2012.

[26] `http://www.renren.com`.

[27] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Scientific and Statistical Database Management*. Stanford InfoLab, July 2013.

[28] `http://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920`.

[29] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, 2013.

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, 2010.

[31] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *CoRR*, abs/1111.4503, 2011.

[32] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[33] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: Distributed machine learning and graph processing with sparse matrices. In *EuroSys '13*, 2013.

[34] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, 2013.

[35] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, 2012.

[36] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12*, 2012.

[37] Y. Zhi, X. Jilong, Q. Zhi, H. Shian, and D. Yafei. Seraph: An efficient system for parallel processing on a shared graph. In *7th Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '13, 2013.