Seraph: An Efficient System for Parallel Processing on a Shared Graph

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

Abstract

Motivated by the need to process very large graphs, there has been significant recent interest in designing graph processing systems. However, existing systems do not naturally support graph data sharing among parallel jobs, thus leading to inefficient use of memory.

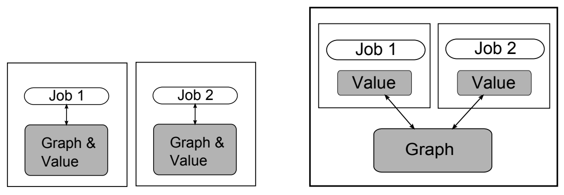
This paper introduces Seraph, a graph processing system that enables the job-level parallelism on a shared graph, i.e., multiple jobs jointly use one graph dataset in memory. Seraph adopts a copy-on-write semantic to isolate graph mutations, and uses a pull-based message delivery mode to reduce the actual memory usage specific to jobs. Our prototype demonstrates that Seraph significantly outperforms Giraph system in both memory usage and job completion time.

1. Introduction

Due to the increasing need to process large volumes of graph-structured data (e.g., social networks and web graphs), there has been significant recent interest in parallel frameworks for processing graphs, such as Pregel [7], Pegasus [1], GraphLab [6], PowerGraph [4], GPS [9], Giraph [2] and Grace [8]. All these systems are based on the Bulk Synchronous Parallel (BSP) computation model that operates on graph data [11].

In these systems, the actual cost of job execution is to store the large graph data in main memory. However, they do not allow multiple jobs using the same graph to share graph data in memory. In particular, systems usually combine graph data and *job-specific* vertex value together. As a result, individual jobs need to operate on separate graph data in memory, introducing inefficient use of memory, as illustrated in Figure 1(a).

The underlying reason for such inefficacy is that existing systems are developed for a single job, rather for batch-oriented jobs. But in practical online/offline graph analytics, multiple jobs may be submitted together. For example, a large number of applications run



(a) Existing systems (b) Seraph

Figure 1. Parallel computations executed using existing systems and Seraph. The size of value is much smaller than that of the graph data.

on the same platform of social networks (e.g., Facebook and Renren) easily generate jobs overlapping in time (i.e., concurrency). Another typical case is the system may accumulate many jobs during peak hours, and then submits them in a batch manner when the computation cluster is idle.

Given a batch of jobs, we have to replicate graph data in memory to execute jobs in parallel, leading to a high memory cost. Or alternatively, we have to use a single in-memory graph to execute jobs in a serial manner, leading to a low throughput due to underused resource. This weakness is more noticeable if we use the resource of cloud, since we have to spend more money either for larger memory (to store multiple graph data) or for longer computation time (to finish all the jobs).

To address these limitations, we have developed Seraph, a graph processing system that can support parallel jobs on a shared in-memory graph. The basic idea of Seraph is to maximize throughput through enabling the job-level parallelism at a low memory cost. Differing from existing systems, Seraph decouples the graph-structure data from the application data associated with jobs, thus allowing multiple parallel jobs to share graph data in memory, as illustrated in Figure 1(b).

To maximize parallelism, Seraph incorporates three new features: First, Seraph only needs to input graph data once for multiple jobs, providing a very fast startup

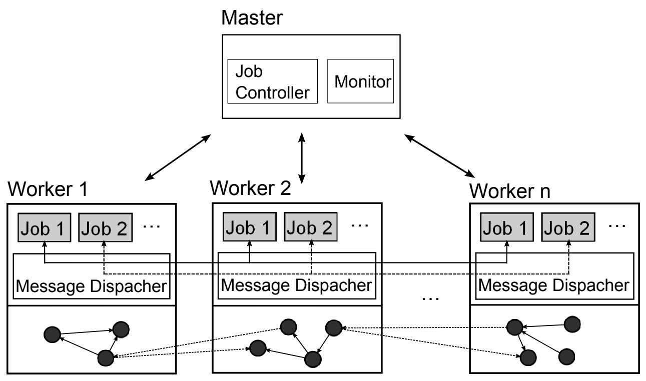


Figure 2. The system architecture of Seraph

time. Second, Seraph adopts copy-on-write semantic to isolate graph mutation. Once a job needs to change graph topology, Seraph would copy the corresponding local region for mutation, without affecting others. Finally, it adapts priority-based scheduling to reduce interference among jobs. In addition, Seraph also incorporates a pull-based messages delivery mode to reduce the actual memory usage specific to jobs, and graph partition to reduce the inter-machine communication.

We have implemented an initial Seraph prototype using the Java. We give the preliminary performance of this prototype by comparing it with Giraph. Given four parallel jobs, our experiments show that Seraph could reduce 63.0% memory usage and 13.3% average job execution time, as compared with Giraph.

2. Seraph Design and Implementation

In this section, we first present an overview of Seraph, and then detail the design of its key components.

2.1 Basic Architecture

Figure 2 gives the high-level architecture of a Seraph cluster, including a single master 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 dispatchs 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 store 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 following subsections.

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 a superstep. It loops through all vertices and calls `compute()`, typically executing an user defined external process.

The computation on vertexes requests the values of all their neighbors in the previous iteration. Instead of performing a barrier synchronization between each iteration, Seraph adopts a "*pull and consume*" asynchronous mode to actively pull neighbor values from remote machines and to execute computation immediately. This mechanism will save the memory for storing intermediate messages.

The *message dispatcher* is responsible for the message delivery for individual jobs. It labels messages (e.g., the neighbor value) with their job ID before sending them. When they are received at the destination worker, the dispatcher places the messages directly in the corresponding vertex's local buffer according to the job/vertex ID. To save memory, the local buffer is released after the computation on a vertex is finished.

2.2 Graph Data Sharing

The unique feature of Seraph is that parallel jobs could share graph structure data for memory saving, which is achieved by separating graph data from job-specific data. We now explain how Seraph enables this separation. In existing systems (e.g., Pregel and 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 are unique to all jobs and could be very large (e.g., Facebook users have an average number of 190 neighbors [10]). Seraph extracts edge list attribute from `Vertex` object and forms a new global `graph` object containing the edge list. After this change, each job just needs to create its own `Vertex` objects, and makes a reference to this unique `graph` object to get the neighbor list of a vertex.

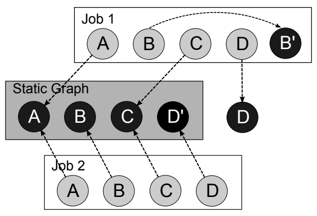


Figure 3. Illustration of maintaining snapshot in Seraph

Graph mutation. To efficiently support graph sharing, we must be able to isolate graph mutation. Some graph algorithms may modify the graph structure during the computation (e.g., a minimum spanning tree algorithm). Graph mutation would influence other jobs due to sharing the common graph data. Hence, Seraph adopts a "copy-on-write" semantic to isolate the local mutation of individual jobs. When the algorithm (i.e., a job) needs to modify the edge list of a vertex, it first creates a `local graph` object that copies the corresponding edge list, and then applies mutable operations on that local copy. The original vertex (and edge list) is still used by other jobs. This method only copies mutable vertices for the corresponding job, thus incurring little memory overhead. Figure 3 illustrates a simple example. Job 1 needs to modify vertex B's edges. It copies a new vertex B' from B, and modifies it locally.

Graph updating. Seraph also supports graph update arising from the change of the underlying graph (e.g., the arrival of new nodes/edges), instead of the graph mutation introduced by jobs. When a update is committed at time t, it should be visible (or invisible) to jobs submitted after (or before) t. Seraph achieves this through a snapshot mechanism. Graph managers of workers collaboratively maintain graph snapshot. When a update arrives, seraph incrementally creates an new snapshot as the up-to-date snapshot. When a job is submitted, it refer to the latest snapshot of graph, see Figure 3. 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.

2.3 Priority Based Job Scheduling

Parallel execution of jobs can improve system throughput due to full resource utilization (e.g., network and CPU). However, it would also incur resource competition. When some resource(e.g., bandwidth) becomes a bottleneck, all jobs experience the same expected slowdown due to aggressive resource competition.

To avoid this, Seraph implements a *job scheduler* to schedule jobs. The scheduler assigns each job a priority based on its submission order, e.g., jobs submitted earlier have a higher probability. The higher priority job gains resource access first, whereas the lower priority jobs gain access once the resource is not used by the higher one. Hence, seraph can guarantee that early jobs are finished as quickly as possible, while jobs submitted later can exploit the idle resource.

Job scheduler is implemented by controlling the message dispatcher and job execution progress. In particular, message dispatcher uses a priority queue to buffer messages from all the jobs. Each time, Seraph fetches message to send from the head of queue (i.e., the message of highest priority job). Meanwhile, Seraph monitors the size of message queue, when it is larger than a threshold (i.e., the resource is limited), some low priority jobs will be suspended and stop to produce messages.

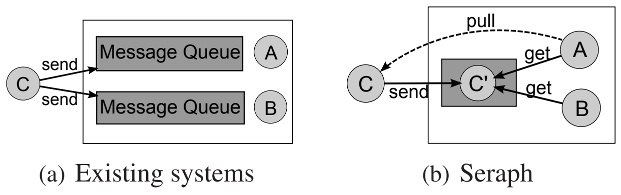


Figure 4. The message delivering mode in existing systems and seraph.

2.4 "Pull-and-Consume" of Message

All BSP-based computation needs values from neighbor vertices in previous superstep. In Pregel, each vertex has a message buffer to receive values sent by their neighbors. Before the computation, all vertexes should store received messages locally, which incurs a large memory overhead (see Figure 4(a)).

Rather than start computing after all vertexes received their neighbor messages (i.e., values), Seraph adopts a "*pull and consume*" mode to pull neighbor messages (values) for a vertex. The system performs `compute()` immediately after a single vertex has pulled all its neighbor values, after which the message buffer of this vertex is released. To avoid message delivering blocking the execution, a local cache is used to pre-fetch messages of some vertexes in advance.

Meanwhile, arrived messages are cache locally for other vertex's use. See in Figure 4(b), vertex A and B both need to obtain vertex C's message from remote machine. Vertex A first sends a *pull* message to C and caches the message locally. Later, B's computation also needs C's message. It can directly get from the local cache.

3. Experiments and Evaluations

We have implemented a Seraph prototype using Java. With this prototype, we present the preliminary results on Seraph's performance. Our experiments deploy Seraph on a cluster with 16 machines. Each machine has 64GB of memory and 2.6GHz AMD Opteron 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.

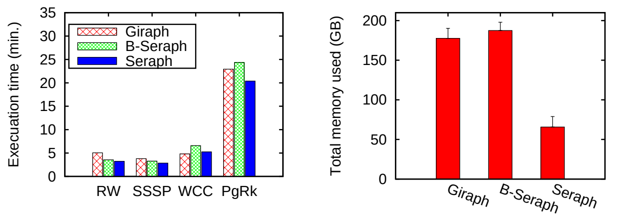
We use some popular graph algorithms as benchmarks to evaluate Seraph's performance, which include Pagerank, random walk, Weakly Connected Component (WCC) and Single Source Shortest Path (SSSP). All these algorithms have inherent different characteristics in resource usage. For example, PageRank is a network intensive application that needs little computation cost. In contrast, Random walk needs little network cost but a long-time per-vertex computation. In our following experiments, we set PageRank's max iteration steps as 30. For random walk, we set 10 walkers for each vertex and the length of walk is 10 steps .

In our preliminary experiments, we run the benchmarks on a snapshot of Renren network (at the end of 2008), the largest online social network of China. The snapshot graph totally contains more than 25 million vertices (users) and 1.4 billion relational edges.

Our preliminary experiments mainly want to reveal Seraph's performance in term of memory usage and job completion time. We compare Seraph with two systems:

- *Giraph*, a popular open source implementation of Pregel framework, with several additional features beyond Pregel.
- *B-Seraph* (Basic Seraph), we remove the Seraph's unique feature of graph data sharing, to serve as a baseline. The B-Seraph is much like a Pregel implementation.

We compare Seraph with Giraph/B-Seraph in two scenarios. The first scenario allows duplicate graph data in memory, where we run multiple Giraph (or B-Seraph) to execute jobs in a parallel manner. We want to examine how much memory Seraph can save through sharing graph data. The second scenario only allows a single graph data in memory, where Giraph (or B-Seraph) executes jobs in a serial manner. In this case, we want to examine to what extent can Seraph reduce job completion time and how much extra cost it introduces.



(a) Execution time (b) Memory used

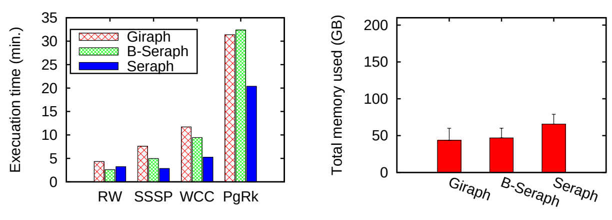
Figure 5. The comparison of job execution in sufficient memory scenario.

Duplicate Graph Data Scenario: We run random walk (RW), SSSP, WCC and PageRank using four parallel Giraph (or B-Seraph), but only using one Seraph due to its unique feature. Figure 5(a) and Figure 5(b) show job completion time and memory usage for three systems. From Figure 5(a), we see that all the three systems have similar performance in job completion time. Even though, Seraph still slightly outperforms others. It reduce about 13.3% and 16.0% in job completion time comparing with Girpah and B-Seraph, respectively. This reduction is brought by the job scheduler of Seraph, which avoids aggressive resource competition.

We now examine how much memory Seraph can save in this scenario. As shown in Figure 5(b), both Giraph and B-Seraph occupy more than 180GB memory in total. However, Seraph occupies only 65.7GB memory for four jobs due to sharing graph data, reducing 63.0% and 64.9% memory usage comparing with Giraph and B-Seraph, respectively.

Unique Graph Data Scenario: We next run random walk (RW), SSSP, WCC and PageRank using Giraph (or B-Seraph) in a serial manner, i.e., executing the later job after the former has been finished. This scenario is common in computing clusters with limited memory or very heavy load.

Figure 6(a) shows job completion time for the three systems. In this case, Seraph significantly outperforms Giraph and B-Seraph in job completion time. On av-



(a) Execution time (b) Memory used

Figure 6. The comparison of job execution in memory-limited scenario.

erage, it reduces the individual job completion time by 42.3% and 35.8%, as compared with Giraph and B-Seraph, respectively. In term of the total completion time of four jobs, Seraph brings a reduction of 55.2% and 44.4% as compared with Giraph and B-Seraph. Moreover, Figure 6(b) shows that Seraph introduces moderate additional memory cost to enable job parallelism (33% and 25% higher than Giraph and B-Seraph, respectively).

4. Related Work

Graph applications can be categorized into two classes: online query processing and offline graph analytics. Graph databases such as Neo4j [3], HyperGraphDB [5] and Sones GraphDB belong to the first category. They typically focus on efficient storage and retrieval of graph structured data, but lack of support for distributed, parallel graph computation. In contrast, Seraph focuses on iterative graph structured computation.

Recently, several notable projects have been developed for processing such large graphs on parallel machines including Pregel [7], Pegasus [1], GraphLab [6], PowerGraph [4], GPS [9], Giraph [2], and Grace [8]. In order to load the entire graph dataset in main memory, most of the systems (with the exception of the original GraphLab and Grace) have been designed to work on distributed memory parallel machines. However, in these systems, one loaded graph dataset only belongs to a single computation job, which introduces inefficient use of memory. In contrast, Seraph system allows multiple concurrent jobs to share one loaded graph dataset, which is designed to maximize throughput.

5. Conclusion and Ongoing 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 to maximize system throughput through enabling the job-level parallelism on a graph shared in memory. Seraph adopts a copy-on-write semantic to support isolation of graph mutation. To save memory, a pull-based messages delivery mode is used. We have implemented a Seraph prototype and demonstrate that it significantly outperforms current systems both in memory usage and job completion time.

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.

Acknowledgements

This work was supported by the National Basic Research Program of China (Grant No. 2011CB302305) and the National High Technology Research and Development Program of China (Grant No.2013AA013203).

References

[1] DEELMAN, E., ET AL. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program. 13*, 3 (July 2005), 219–237.

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

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

[4] GONZALEZ, J. E., ET AL. Powergraph: distributed graph-parallel computation on natural graphs. In *Proc. of OSDI'12* (2012).

[5] IORDANOV, B. Hypergraphdb: a generalized graph database. In *Pro. of WAIM'10* (2010).

[6] LOW, Y., ET AL. Graphlab: A new framework for parallel machine learning. *CoRR abs/1006.4990* (2010).

[7] MALEWICZ, G., ET AL. Pregel: a system for large-scale graph processing. In *Proc. of SIGMOD'10* (2010).

[8] PRABHAKARAN, V., ET AL. Managing large graphs on multi-cores with graph awareness. In *Proc. of USENIX ATC'12* (2012).

[9] SALIHOGLU, S., AND WIDOM, J. Computing strongly connected components in pregel-like systems. Technical report, Stanford University, 2013.

[10] UGANDER, J., KARRER, B., BACKSTROM, L., AND MARLOW, C. The anatomy of the facebook social graph. *CoRR abs/1111.4503* (2011).

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