Towards Scale-out Capability on Social Graphs

Haichuan Shang†‡ Xiang Zhao§ Uday Kiran‡† Masaru Kitsuregawa*‡
† National Institute of Information and Communications Technology, Japan
‡ University of Tokyo, Japan
§ National University of Defense Technology, China
* National Institute of Informatics, Japan
†\{shang, uday_rage, kitsure\}@tkl.iis.u-tokyo.ac.jp §xiangzhao@nudt.edu.cn

ABSTRACT
The development of cloud storage and computing has facilitated the rise of various big data applications. As a representative high performance computing (HPC) workload, graph processing is becoming a part of cloud computing. However, scalable computing on large graphs is still dominated by HPC solutions which require high performance all-to-all collective operations over torus (or mesh) networking. Implementing those torus-based algorithms on commodity clusters, e.g., cloud computing infrastructures, can result in great latency due to inefficient communication. Moreover, designing a highly scalable system for large social graphs is far from being trivial, as intrinsic features of social graphs, e.g., degree skewness and lacking of locality, often profoundly limit the extent of parallelism.

To resolve the challenges, we explore the iceberg of developing a scalable system for processing large social graphs on commodity clusters. In particular, we focus on the scale-out capability of the system. We propose a novel separator-combiner based query processing engine which provides native load-balancing and very low communication overhead, such that increasingly larger graphs can be simply addressed by adding more computing nodes to the cluster. The proposed system achieves remarkable scale-out capability in processing large social graphs with skew degree distributions, while providing many critical features for big data analytics, such as easy-to-use API, fault-tolerance and recovery. We implement the system as a portable and easily configurable library, and conduct comprehensive experimental studies to demonstrate its effectiveness and efficiency.

Categories and Subject Descriptors
H.2.4 [Database Management]: Systems

General Terms
Algorithms; Experimentation; Performance

Keywords
Graph Databases; Distributed Databases; Social Networks

1. INTRODUCTION
In an era of information explosion, big data analytics have become an increasingly important component of industry, which focuses on developing special parallel computational platforms to analyze various forms of massive data. The widespread usage of these platforms has changed the landscape of scalable parallel computing, which was long dominated by High Performance Computing (HPC) applications. The big data platforms support applications, different from those solved by HPC before in that (1) big data platforms rely on commodity cluster-based execution and storage infrastructure like MapReduce; and (2) the utility computing model introduced by cloud computing makes computing resources available to the customer as needed, such that large scale-out capability is mandatory for the systems on cloud.

As one of the most popular data-intensive applications, the rapidly growing social networks require an efficient and scalable graph processing platform. Unfortunately, MapReduce, the de-facto big data processing model, and its associated technologies such as Pig [19] and Hive [31] are often ill-suited for iterative computing problems on large graphs. As a result, several graph-parallel projects [1, 17, 18, 22, 24, 27, 30, 32, 36], have been proposed to facilitate data analysis on large graphs. These projects can be generally classified into two categories:

- The first category is 1D partitioning methods, which means that the vertices of a graph are divided into partitions along with their outgoing edges. 1D partitioning is widely adopted by graph-parallel projects. Different from its predecessor Parallel BGL [10] which exposes a whole graph to user-code as the interface, Pregel [18] introduces an easy-to-use interface, named vertex-centric Bulk Synchronous Parallel (BSP) [34] model. This model is similar to a BSP model which consists of a sequence of supersteps separated by global synchronization points until the algorithm terminates. In every superstep, a user-programmable function vertex.compute() is called exactly once for each vertex. The function vertex.compute() receives messages via the incoming edges from the previous superstep, processes the messages such as an aggregation, and sends messages via the outgoing edges for the next super-
step. Pregel inspires several research and open-source projects [1, 22, 24, 27, 30, 32, 36].

**Advantage.** The advantage of these projects is providing an easy-to-use infrastructure to reduce the costs of implementing distributed graph algorithms in terms of human time and effort, because the man-hour spent on developing well-optimized native code is usually unaffordable to enterprise IT infrastructure.

**Disadvantage.** Real-world graphs typically follow power law degree distributions [3, 5, 8, 23]. The systems based on 1D partitioning cannot scale-out to a large number of computing nodes when processing massive power law graphs, because the workers (i.e. the instances of an algorithm running on a computing node) in charge of the high degree vertices will take longer query processing time than the others, and thus exceedingly slow down the overall response time. In other words, increasing the number of computing nodes in a cluster cannot further reduce overall response time, because the elapsed time in which a worker processes the vertex with the highest degree determines the overall response time.

- The second category is 2D partitioning methods [7, 37], in which every vertex is managed by a group of computing nodes. In each superstep, the computing nodes in the same group communicate with one another to aggregate the received messages, and then send messages to the other groups along outgoing edges to the next superstep. The communication among the computing nodes in the same group is implemented by either AllReduce [37] or AllGather [7]. A scalable system based on the 2D partitioning approach is usually supported by special HPC hardware such as torus (or mesh) networking. For example, a 2D torus configuration places 16 computing nodes in a $4 \times 4$ grid. Each of the 16 computing nodes is directly linked to its four nearest-neighbors. Initially, the vertices of a graph is partitioned into four non-overlapping subsets. Each row of the computing nodes in the torus manages a subset of the vertices. The outgoing edges from a subset of the vertices are further partitioned into four edge subsets by their targets, and each edge subset is handled by a computing node. In a typical computation, every worker communicates vertically (Fold) to send messages along outgoing edges, and performs AllReduce or AllGather horizontally (Expand) to aggregate received messages for the same group of vertices.

**Advantage.** This approach is highly scale-out-capable to more than sixty thousand computing nodes [2] on HPC.

**Disadvantage.** This approach tackles degree-skew and scale-out capability as a trade-off of communication cost. Indeed, the communication cost of the Fold step is equal to 1D partitioning’s total communication cost in terms of total message size. The communication cost of the Expand step is the price we pay extra for load-balancing and scalability. The high-performance implementation [6, 20] of row-wise and column-wise collective communication including AllReduce and AllGather is critical to the scale-out capability. Due to lacking torus networking and high-performance implementation of AllGather operation, implementing this approach on commodity clusters can lead to inefficiency in the form of inefficient communication. In addition, it is fairly expensive to implement 2D graph algorithms, in terms of human time and effort, which conflict with the “ease-of-use” concept of big data.

Briefly, the existing systems provide the following choices to process large graphs.

1. Relying on the existing systems which are based on vertex-centric BSP model or MapReduce platforms. The systems, such as Apache Giraph [1] on Hadoop [35] and GraphX [36] on Spark[38], are very easy to use, but cannot handle large degree-skewed graphs derived from massive social networks.

2. Adopting the HPC solutions which provide high performance on large-scale clusters, but needs expensive HPC clusters with torus (or mesh) networking.

3. Extending collective communication to commodity clusters. These systems [9, 11] alleviate the effect of degree-skew on small-scale clusters, but the impact of concurrent network transactions can be significant for collective communication on large-scale clusters, and thus degrades the performance of these systems in processing large graphs.

<table>
<thead>
<tr>
<th></th>
<th>Degree-skew</th>
<th>Commodity</th>
<th>Scale-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D (Commodity)</td>
<td>×</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2D (HPC)</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>2D (Commodity)</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
</tr>
</tbody>
</table>

**Table 1: Existing systems against requirements**

Developing a system for processing large social graphs on large-scale commodity clusters means satisfying three requirements: 1. **Degree-skew:** the partitioning method is load-balanced when processing graphs with skew degree distributions; 2. **Commodity:** the system does not rely on costly networking such as torus; and 3. **Scale-out:** larger graphs can be addressed by simply adding more computing nodes to the system. As shown in Table 1, none of the existing approaches satisfy the above three requirements.

Motivated by this, we propose a novel system, named GraphSlice. GraphSlice’s query processing is based on a Separator-Combiner BSP model. We demonstrate that the above three requirements can be satisfied simultaneously by an elegantly designed query processing engine by which massive social graphs can be efficiently processed on commodity clusters. This system provides the following features.

1. Capturing the common patterns of graph algorithms such that API’s are sufficient to express a wide range of graph algorithms.

2. Achieving scalable performance in processing large graphs with skew degree distributions.

3. Synchronizing supersteps without all-collective communication which elegantly fit large-scale commodity clusters where the impact of concurrent network transactions is serious.

4. Relying on one-to-one communication in a BSP model, such that the proposed system is highly reliable and
fault-tolerant on commodity clusters and is able to be seamlessly integrated with existing commodity cluster-based infrastructures such as MapReduce.

The rest of this paper is organized as follows. Section 2 introduces the background knowledge. Section 3 gives the framework of our approach. Section 4 describes the query processing engine. Section 5 presents API of our system. Section 6 discusses system implementation. Section 7 reports the experimental results. The related work and conclusion is given in Section 8 and 9, respectively.

2. PRELIMINARIES

In this section, we briefly introduce the background knowledge on 1D partitioning and vertex-centric BSP model.

2.1 1D partitioning

The 1D partitioning (a.k.a. distributed adjacency list, source-vertex partitioning and horizontal partitioning) is the most popular partitioning method for graph processing in distributed environments. The vertices are partitioned to workers along with their outgoing edges. Figure 1 (a) and (b) illustrate the partitions of an example graph and its distributed adjacency list, respectively. In this example, the vertices are partitioned into three partitions \{A, B, C\}, \{D, E, F\} and \{G, H, I\} along with their outgoing edges.

![Figure 1: 1D partitioning](image)

2.2 BSP model

In a BSP model [34], the superstep concept is used for synchronizing the parallel execution of workers. A superstep typically consists of three parts.

- Computation: Every worker executes computation for the vertices it owns.
- Communication: The worker sends messages on behalf of the vertices it owns to the neighboring vertices.
- Barrier: The worker waits until all other workers have finished their computation and communication.

Then the algorithm moves to the next superstep. An algorithm may involves several supersteps which are executed one after another.

For example, a simple BFS algorithm on 1D partitions is illustrated in Figure 2. The BFS begins at the root vertex ‘C’ which is initialized by the vertex value ‘0’ as its distance to the root. In the Superstep 1, the worker, which owns the vertex ‘C’, sends the messages that contains the value ‘2’ for the next superstep. The algorithm iteratively updates vertex values if the received value is smaller than the current vertex value, and sends messages which contain the updated value plus ‘1’. The algorithm terminates when there is no update in a superstep.

![Figure 2: A simple BFS algorithm on 1D partitions](image)

2.3 Vertex-centric API

Pregel [18] proposes a vertex-centric API on BSP model, abbreviated as vertex-centric BSP. Writing a vertex-centric BSP program usually means overriding the virtual \texttt{Compute()} function in the \texttt{Vertex} class as shown in Algorithm 1, which will be executed at each active vertex in every superstep. The template defines three value types which are associated with vertices, edges and messages. The vertex-centric BSP model exposes all outgoing edges of a vertex to the \texttt{Compute()} function by providing an \texttt{OutEdgeIterator}.

A vertex-centric BSP implementation of Single Source Shortest Path (SSSP) algorithm is shown in Algorithm 2. The \texttt{ShortestPathVertex} class inherits from \texttt{Vertex} class and implements the \texttt{Compute()} function. The vertex value with each vertex is initialized to a constant INF which is larger than any feasible distance. In each superstep, each vertex first receives messages via its incoming edges, and calculates the minimum value among the received messages. If the minimum is less than the current vertex value, this vertex updates its value and sends the updated value via the outgoing edges, consisting of the updated vertex value added
to the edge weight. The algorithm terminates when there is no new updates in a superstep, after that the value of each vertex denotes the distance from the source to that vertex. This algorithm is amenable to optimization using a combiner as discussed in [18], which is omitted here due to the interest of space.

3. FRAMEWORK

In this paper, we propose a system GraphSlice that employs novel partitioning and query processing methods to tackle degree-skew and scale-out problems, while providing an easy-to-use API to the users. It consists of three parts:

1. Partitioning. The partitioning is managed by Separators which are containers of a subset of edges outgoing from a same vertex.

2. Query Processing. The query processing is based on a Separator-Combiner BSP model.

3. API. The API provides an easy Vertex-and-Edge view, while the concept of Separator and Combiner is usually hidden from the users.

The next three sections will introduce the system details.

4. PARTITIONING AND QUERY PROCESSING

In this section, we introduce the basic idea of the partitioning method, followed by the query processing model. More implementation details will be given in Section 6.

4.1 Partitioning

The vertices are firstly partitioned to workers according to a user-defined hash function. Then, edge partitioning is managed by Separators which are containers of a subset of edges outgoing from a same vertex. We provide two types of Separators to manage the outgoing edges of every vertex: PreSeparator and PostSeparator which are abbreviated for pre-communication separator and post-communication separator respectively. Every vertex’s outgoing edges are managed by either PreSeparator or PostSeparator. In particular, there is an underlying degree-based mechanism which manages high degree vertices by PostSeparators and low degree vertices by PreSeparators.

PreSeparator is very similar to the existing 1D partitioning method. It is hosted by the worker which owns the source vertex of the PreSeparator. PostSeparator is more tricky. If a vertex v is managed by PostSeparator, it is indeed managed by at most N PostSeparators on N different workers where N is the total number of workers in the system. Each PostSeparator on a worker manages a subset of the outgoing edges of v which emanate to the vertices owned by that worker.

For example, a weighted directed graph is shown in Figure 3. A user-defined hash function firstly divides the vertices into two partitions: {A,B,C} and {D,E,F}. The system estimates the degree of every vertex to determine the type of Separators. In this example, the vertex ‘C’, whose out-degree is four, is managed by PostSeparator, while the others are managed by PreSeparator. In particular, ‘C’ is managed by two PostSeparators, one for edges {‘CA’, ‘CB’} and the other for {‘CD’, ‘CE’}. It is worth-noting that these subsets are classified by the target vertex of each edge. For instance, the edges ‘CA’ and ‘CB’ are partitioned into the same subset, as the target vertices ‘A’ and ‘B’ are owned by the first worker. It is similar that the edges ‘CD’ and ‘CE’ are on the second worker which owns vertices ‘D’ and ‘E’.

4.2 Query Processing

The query processing engine employs a Separator-Combiner BSP model (SC-BSP). A Combiner is an instance of an aggregation function that aggregates the messages emanating to a same vertex. We illustrate the query processing procedure by an example of the SSSP algorithm on the graph in the previous example. The SSSP algorithm starts with a root at vertex ‘A’. In Superstep 1, the distance of root vertex ‘A’ is initialized to ‘0’; the root vertex ‘A’ sends messages to its neighbors ‘C’, ‘E’ and ‘F’; and finally these three neighbors update their distances to ‘1’, ‘2’ and ‘1’, respectively. We omit the execution of Superstep 1, but enlarge the details of the execution of Superstep 2 in Figure 4. There are two workers as depicted by the large gray rectangles. One of them contains the vertices ‘A’, ‘B’ and ‘C’, while the other contains the vertices ‘D’, ‘E’ and ‘F’. In Superstep 2, the vertices ‘C’, ‘E’ and ‘F’ are the frontiers which are marked in red color, as their distances have just been updated in Superstep 1.

As shown in the example in Figure 4, the low degree vertices ‘E’ and ‘F’ are managed by PreSeparators, while the high degree vertex ‘C’ is controlled by two PostSeparators {‘CA’, ‘CB’} and {‘CD’, ‘CE’}.

A superstep consists of three parts: pre-computation, communication and post-computation. In the pre-computation of Superstep 2, the vertex ‘C’ broadcasts its value ‘1’ to

---

Algorithm 2 Single Source Shortest Path (SSSP)

```java
class ShortestPathVertex extends Vertex<int, int, int>
{
    void compute(MessageIterator msgs) {
        int mindist = isSource(vertexID()) ? 0 : INF;
        For (;msgs.Done();msgs.Next())
            mindist = min(mindist, msgs.Value());
        If (mindist < getVertexValue())
            setVertexValue(mindist);
        OutEdgeIterator iter = getOutEdgeIterator();
        For (;!iter.Done();iter.Next())
            SendMessage(iter.Target(), iter.getEdgeValue() + mindist);
    }
}
```

---

Figure 3: A sample graph
all the workers since it is managed by PostSeparators. In the meanwhile, the vertices ‘E’ and ‘F’ call their PreSeparators to conduct edgeCompute() function (the API will be discussed in the next section). In the SSSP algorithm, edgeCompute() function adds the edge weight to the vertex value, and sends the sum to the target vertex of the edge. As the two edges ‘ED’ and ‘FD’ have the same target vertex ‘D’, the PreCombiner is called after the PreSeparators to combine the two messages. For the SSSP algorithm, the MinValueCombiner merges the messages by selecting the minimum value. Then, the communication part exchanges the four messages among two workers.

In the post-computation of Superstep 2, the PostSeparators of vertex ‘C’ traverse the outgoing edges of ‘C’ in which edgeCompute() function is called to add the edge weight to the vertex value for every edge. The messages produced by PostSeparators are combined by their destinations in the PostCombines. Then, these combined messages are sent to vertices in the next superstep. In the beginning of Superstep 3, the distances of vertices ‘B’ and ‘D’ are updated, and these two vertices will become the new frontiers.

The degree threshold, which determines whether a vertex is manged by PreSeparator or PostSeparator, is user-configurable. By default, the threshold is the number of workers. The default setting can minimize the total message size in communication for most applications. We observe that there are some existing graph algorithms which are optimized by degrees. The one which is similar to our proposal is [28]. Compared with the existing algorithm, our contribution is designing a degree-based mechanism in a parallel graph processing engine, while the complexity of the design is hidden from users. Users can use GraphSlice’s API to develop graph algorithms without an understanding of the underlying degree-based mechanism.

5. GRAPHSLICE’S API

In this section, we present an overview of GraphSlice’s API which includes the major classes and functions. The focus of GraphSlice’s API is providing an easy Vertex-and-Edge view to users, while the underlying degree-based mechanism for Separator and Combiner is hidden.

5.1 Basic API

The Vertex class of GraphSlice introduces three new features to the vertex-centric BSP model: (1) the enum Path which describes the message-passing paths between the vertices, (2) the template argument SeparationType which defines a user-specified value type for intermediate results, and (3) the concept of edge compute and EdgeHandler class which handles edge compute.

As shown in Algorithm 3, users override the virtual vertexCompute() method, which will be executed at each active vertex in every superstep. The SendMessage() method allows vertexCompute() to send messages to other vertices, and these messages will be received by the other vertices via MessageIterator in their next superstep. SendMessage() contains three arguments: (1) enum Path, which describes the destinations of this message, either the target vertex of the outgoing edges of the current vertex or all vertices in the graph, (2) SeparationType, which is the value type that will be passed to EdgeHandler, and (3) a user-defined subclass of the EdgeHandler class.

Algorithm 3 The Vertex API foundations

```java
template <VertexType, EdgeType, MessageType, SeparationType>

class Vertex {
    virtual void vertexCompute(MessageIterator msgs);
    long vertexID();
    boolean isSource(long vertexID);
    int superstep();
    VertexType getVertexValue();
    void setValue(VertexType value);
    enum Path {OutgoingEdges, AllVertices}
    void SendMessage(Path path, SeparationType separation, Class<? extends EdgeHandler > edgeHandler);
    void VoteToHalt();
    int getNumOutgoingEdges();
    int getNumIncomingEdges();
    OutEdgeIterator getOutEdgeIterator();
    OutEdgeIterator InEdgeIterator();
};
```

The EdgeHandler class is shown in Algorithm 4. Instead of exposing all outgoing edges in the Vertex class, the EdgeHandler class is responsible for edge compute. Users override the virtual edgeCompute() method which
receives a SeparationType value from its source vertex, produces a MessageType value and sends the MessageType value to its target vertex.

Algorithm 4 The Edge API foundations

template <EdgeType, MessageType, SeparationType>
class EdgeHandler {
    virtual MessageType edgeCompute(SeparationType separation);
    int superstep();
    EdgeType getEdgeValue();
    void setEdgeValue(EdgeType value);
}

An example of implementing the SSSP algorithm by overriding the Vertex class and the EdgeHandler class is illustrated in Algorithm 5. For representation simplicity, we set VertexType, EdgeType, MessageType and SeparationType to int which is abbreviated for Integer. By overriding the vertexCompute() function, the algorithm firstly calls the isSource() function to get an initial value for the distance. The distance is initialized to zero if the vertex is the source (i.e. root) of the algorithm, otherwise it is assigned with a constant INF which is larger than any feasible distance. In each superstep, every vertex processes the received messages to calculate the minimum value among all the messages. If the minimum value is less than the current vertex value, the vertex updates its value and sends the updated value via the outgoing edges by specifying OutgoingEdges as the Path in the SendMessage method. The class type ShortestPathEdge.class is also specified to indicate the EdgeHandler of this message.

The ShortestPathEdge class is the EdgeHandler in the SSSP algorithm. It adds the vertex distance to the edge weight, and returns the sum. By default, the returned value will be sent to the target vertex of the current edge as a received message in the next superstep. The algorithm terminates when there is no update to any vertex in a superstep, at which point the value of each vertex indicates the distance from the source to that vertex.

As shown in Algorithm 6, the Combiner class is usually used for reducing the communication of an algorithm. A combiner combines the messages which are sent to the same vertex. For example, in the SSSP algorithm, only the minimum value of the messages is useful to the vertex. The combiner has been proposed in Pregel. But the GraphSlice’s Combiner API is different from that of Pregel’s, due to their difference in query processing engines. In Pregel’s API, the combiner exposes all the messages, which are sent to the same vertex, as an iterator of messages to users. However, GraphSlice’s API in Algorithm 6 provides a combine() function which always combines the original message and a newly arrived message, because its SC-BSP based query processing engine combines messages in both pre-computation and post-computation.

An example of the minimum value combiner for the SSSP algorithm is shown in Algorithm 7. If the value of a newly arrived message msgToCombine is less than the original message originalMsg, the value of the original message will be replaced by the value of the newly arrived message.

5.2 Advanced API

In consideration of ease-of-use, the Separator class is usually hidden from users. We provide in Algorithm 8 a default separate() which is usually applicable for most applications. The default separate() function iterates the edges in the subset, calls the edgeCompute() function of each edge, and sends the returned MessageType value to the target of each edge.

Advanced users may override the separate() function to implement advanced algorithms. The Separator class provides a set of functions to users for implementing complicated algorithms. We briefly introduce some important functions. ThegetSourceVertexID() function returns the VertexID of the source vertex of this subset of edges. The isPreSeparator() function provides the type of the separator which can be either PreSeparator (true) or PostSeparator (false). The getNumOfSubsets() and getSubsetID() functions return the number of subsets which are responsible for the same source vertex and the ID of this subset, respectively.

6. IMPLEMENTATION

We are dedicated to implementing GraphSlice as an easy-to-use, easily maintainable and scale-out capable system on commodity clusters.

Indexing of Graph Structure. GraphSlice organizes the graph structure in the distributed memory of workers. The vertices are distributed to workers according to a user-defined Hash function. The distributed memory holds PreSepara-
Communication. There are two alternative implementations of PostSeparators. The first is indexing the workers in which a high degree vertex has PostSeparators, and sending \texttt{SeparationType} value to the indexed workers. The second is broadcasting the \texttt{SeparationType} value of a high degree vertex to all the workers. When the outgoing edges of a high degree vertex are distributed across more workers, the efficiency produced by such indexes will be reduced. We implement the prototype of PostSeparators as a broadcasting method in order to make the code easily readable and easily maintainable.

Online Updating. Some graph mining algorithms may need to change the graph’s topology in their algorithms. Online update of graph structure is supported with some constraints. GraphSlice supports setVertexValue, addVertex, removeVertex, addEdge and removeEdge functions in Vertex class. It also supports setEdgeValue function in EdgeHandler class.

Persistent Storage. GraphSlice accesses persistent storage only for input, output and checkpoints. Either general graph formats or a specific partitioned graph format can be used as an input. General graph formats are user-configurable, which include adjacency lists, edge lists and etc. When general graph formats are used as the input, GraphSlice needs to estimate the degree of every vertex to determine the type of separators. The estimation is typically calculated by randomly sampling of edges. After the estimation step, the graph structure are loaded into distributed memory, while the corresponding in-memory indexes are built on the fly.

GraphSlice is able to write the graph structure on disk in either general graph formats or the specific partitioned graph format. The specific partitioned graph format organizes the graph structure into two parts: vertices and separators. It directly writes the separators on each worker’s local disk. This format is used as the format in checkpointing for fault tolerance and recovery.

Fault Tolerance and Recovery. As the query processing of GraphSlice uses a BSP model, the mechanism of fault tolerance and recovery is based on checkpointing, which is similar to Pregel’s. According to a user-defined parameter $T_{CP}$, we make check points every $T_{CP}$ supersteps. In these check points, the modified graph data and the buffered messages are written to the persistent file system for failure recovery. The persistent file system adopts the hadoop distributed file system (HDFS) with a configurable replication factor; that is, the file content is replicated on the storage of multiple computing nodes for reliability.

7. EXPERIMENTS

We conduct comprehensive experiments to evaluate the performance of the proposed SC-BSP model against the vertex-centric BSP model on massive degree-skewed graphs. We implement both two models as an in-memory system based on Hadoop, Netty and Giraph on an existing HDFS cluster with 128 physical machines each of which is con-
The experiments in Figure 5 (a) evaluate the scale-out capability on degree-skewed social graphs. Scale-out capability means that larger graphs can be addressed by simply adding more workers to a system.

Both SC-BSP model and vertex-centric BSP model are initially allocated with 32 workers with 4GB memory on each worker. With 32 workers, SC-BSP can process graphs up to 0.5 billion edges. In contrast, vertex-centric BSP model only can process graphs up to 0.1 billion edges. Although both of the two systems own the same amount 32 × 4GB = 128GB of distributed memory, SC-BSP model can process larger graph than vertex-centric BSP model because of its native load-balancing technique. As the maximum degree vertex emanates more than 10% of edges, the vertex-centric BSP model needs to keep these edges in a single worker’s memory due to its vertex-centric property. SC-BSP model deploys PostSeparators as a container of edges, such that edges emanating from the maximum degree vertex are managed by the PostSeparators on different workers.

We increase the number of workers to examine if the system can address larger graphs. Vertex-centric BSP model fails to process any larger graph, as the maximum degree vertex consumes too much memory on a single worker. On the contrary, SC-BSP model can address larger graphs by simply adding more workers to the system. With 512 workers, SC-BSP model can process a graph with up to 2.1 billion edges.

Response Time. The response time is illustrated in Figure 5 (b). Because vertex-centric BSP model cannot process any graph larger than 0.13 billion edges, we only report its performance until this graph size. Within graph size they can address, both two systems shows scalable response time with respect to the number of edges.

Communication. The communication is measured by the average message size produced by each worker. Figure 5 (c) reports the comparison between two BSP models. It is not surprising that performance gap is up to 40 times, even when the graph size is relatively small. The PreSeparators of the SC-BSP model produce the same amount of messages as the vertex-centric BSP model. However, the PostSeparators exceedingly reduce the amount of messages that are produced by high degree vertices, because only the SeparationType value will be transferred in communication and all messages along with outgoing edges will be processed by PostSeparators in a same worker. In power law graphs, when increasing the size of graph, more and more edges will be connected to the existing high degree vertices. These increased edges do not produce extra messages in the SC-BSP model, thus the performance gap drastically increases with the graph size. We observe that the message size increases sublinearly with the graph size in the SC-BSP model.

Dynamic SC-BSP. When graphs are relatively small, allocating a large amount of workers may introduce cost overhead. We propose a simple optimization to reduce the cost overhead. This optimization dynamically determines the number of workers according to the graph size. In this experiment, we intuitively set this ratio to 2 million edges per worker. Therefore, we allocate 16 workers to the graph with 0.03 billion edges and 512 worker to the graph with 1.07 billion edges. For the graph with 2.14 billion edges, we still allocate 512 workers due to reaching the constraint on the maximal number of workers. Figure 6 shows Dynamic SC-BSP can reduce up to three-fourth of the response time on relatively small graphs.

8. RELATED WORK

This section discusses existing literature on graph-parallel computing systems, with a focus on partitioning methods of graphs and corresponding communication models of messages. As real-life graphs are often too huge to be loaded into the memory of single machines, graph-parallel computing systems divide the graph into partitions. Every worker loads a partition of the graph into its own memory, and communicates with the other workers to execute a parallel graph algorithm. Next, we organize the related work into two categories according to their partitioning methods.

8.1 1D Partitioning

Parallel BGL [10] is a generic C++ library for research. It implements 1D partitioning method on Boost Graph Library, and uses MPI for communication in a BSP model.
Pregel [18] is currently the most popular large graph processing framework. In Pregel, master is primarily responsible for coordinating workers, and also participates in the computation of aggregators, which is a mechanism for global communication. GPS [24] extended aggregator to a master compute mechanism, which was later adopted by Giraph [1]. In Giraph’s API, the MasterCompute class has a compute() function, in which users are able to read and change aggregated values from the previous superstep; additionally, each aggregator is assigned to one of the workers such that master does not have to perform any aggregation. [25] proposed to monitor the size of the subgraph of active vertices. When the active subgraph becomes small enough, it is sent to the master to perform the remaining computation serially. [25] also discusses the optimization techniques including merging vertices to supernodes, cleaning edges on demand, etc.

It has been noted that the vertex-centric BSP model is unsuitable for the graph applications that require coordination, e.g., graph coloring. Lately, Giraphx [30] presented a modification to the BSP model to meet the aforementioned requirement by categorizing vertices as border and internal vertices. The modification allows the direct read to worker’s memory for internal vertices. For border vertices, additional synchronization is implemented by adopting two alternative coordination mechanisms, e.g., dimming philosophers and token ring.

The vertex-centric BSP model is easy to program, but sacrifices the flexibility of a vertex being able to access the other vertices in the same worker. Two recent studies, Giraph-h++ [32] and VB-Partitioner [14], exploited the idea of sharing information in the same worker to improve efficiency. In particular, vertices in the same worker are processed in a compute() function together. Remarkable performance improvements can be achieved by carefully designing the algorithms, with, however, a trade-off of the ease-of-use. High-degree vertices in a single worker are likely impose a bottleneck when processing power-law graphs, which are commonly seen in real world. The PowerGraph [9, 16] proposed to use a randomized edge partitioning approach to balance workloads. As the outgoing edges of each vertex are stored across workers, a synchronization mechanism aggregates the vertex values across all workers for all vertices in the graph. Since the aggregation is an all-to-all collective communication and the size of aggregators is proportional to the number of vertices, the scale-out capability to the number of computing nodes is hence limited.

Among others, Trinity [29] and Facebook’s TAO [4] are data models for serving online queries on a large-scale graph. These systems answer graph queries with low read latency and high read availability. The applications of Trinity and TAO are quite different from the other systems which focus on high throughput offline processing. In contrast to the BSP models, [12] proposed an asynchronous model for graph processing. A comprehensive performance comparison and analysis can be found in [26].

8.2 2D Partitioning

2D partitioning was first proposed in [37] and adopted by [7, 33]. In 2D partitioning, the outgoing edges of a set of vertices are collectively owned by all the computing nodes in a row of mesh networking. We employ a level-synchronous BFS algorithm to describe how graph algorithms run on 2D partitioning. A 2D-BFS algorithm contains two steps in each superstep: (1) **Expand**: Every computing node sends the current frontier of vertices that it owns to other computing nodes in the same column as an AllGather operation, and inspects the outgoing edges of vertices which belong to the gathered frontier set; and (2) **Fold**: Every computing node exchanges newly-discovered vertices with the other computing nodes in the same row, and constructs the new frontiers for the next superstep.

The 2D solution handles load-balancing and scalability with a trade-off of communication cost. Indeed, the communication cost of the Fold step is equal to 1D partitioning’s total communication cost in terms of total number of messages. The communication cost of the Expand step is the price we pay extra for load-balancing and scalability. Due to lacking torus networking and the a high-performance implementation of AllGather operation, deploying these algorithms on commodity clusters and cloud computing infrastructures can lead to inefficiency in the form of expensive communication. Moreover, implementing 2D graph algorithms requires great human effort, conflicting with the “ease-of-use” concept of big data analytics.

Note that there is another line of research on graph computation on MapReduce. This category of systems, such as Pegasus [13], Cloud9 [15], Pig Latin [19], Hive [31] and SGC [21], rely on Map and Reduce tasks for graph processing. MapReduce is often ill-suited for graph computation problems [18], and consequently can lead to suboptimal performance and usability issues. Therefore, MapReduce-based systems are beyond the scope of this paper.

9. CONCLUSION

We introduce a parallel system called GraphSlice for large-scale graph processing on commodity clusters and cloud computing infrastructures. The major contribution of this system is enabling high parallelism and scale-out capability for graph processing in real-world applications, where graphs typically have power law degree distributions. The other usability aspects such as the easy-to-use API, fault-tolerance, recovery and portability make GraphSlice easily configurable and usable on an existing Hadoop cluster. The performance, scalability and scale-out capability of GraphSlice have been experimentally demonstrated, which is able to meet the requirements of large-scale graph processing.

10. REFERENCES
