Big Data Research 4 (2016) 59–69

Contents lists available at ScienceDirect

Big Data Research

www.elsevier.com/locate/bdr

# epiCG: A GraphUnit Based Graph Processing Engine on epiC

Yanyan Shen [a], Qingchao Cai [b,*], Wei Lu [d], Dalie Sun [c], Zhongle Xie [b]

[a] Shanghai Jiao Tong University, PR China
[b] National University of Singapore, Singapore
[c] Harbin Institute of Technology, PR China
[d] Renmin University of China, PR China

ARTICLE INFO

*Article history:*
Received 31 December 2015
Received in revised form 21 April 2016
Accepted 22 April 2016
Available online 9 June 2016

*Keywords:*
epiCG
epiC
Vertex-cut partitioning
Distributed graph processing systems
Big Data

ABSTRACT

A large number of specialized graph processing systems have been developed to cope with the increasing demand of graph analytics. Most of them require users to deploy a new framework in the cluster for graph processing and switch to other systems to execute non-graph algorithms. This increases the complexity of cluster management and results in unnecessary data movement and duplication. In this paper, we propose our graph processing engine, named epiCG, which is built on top of epiC, an elastic data processing system. The core of epiCG is a new unit called GraphUnit, which is able to not only perform iterative graph processing efficiently, but also collaborate with other types of units to accomplish any complex/multi-stage data analytics. epiCG supports both edge-cut and vertex-cut partitioning methods, and for the latter method, we propose a novel light-weight greedy strategy that enables all the GraphUnits to generate vertex-cut partitioning in parallel. Furthermore, unlike existing graph processing systems, failure recovery in epiCG is completely automatic. We compare epiCG with several prevalent graph processing systems via extensive experiments with real-life dataset and applications. The results show that epiCG possesses high efficiency and scalability, and performs exceptionally well in large dataset settings, showcasing its suitability for large-scale graph processing.



## 1. Introduction

The increasing demand of graph analytics is the inevitable consequence of the growing scale and importance of graph data. Big graph examples including social network graphs, email and instance message graphs typically involve billions of vertices and edges. For example, Facebook has over 1.5 billion monthly active users at present. In recent years, a large number of specialized distributed graph processing systems such as Pregel [1], PowerGraph [2], Giraph [3] and GPS [4] have been proposed to handle complex graph analytics tasks. These specialized systems gain their popularities for two reasons. First, they follow the vertex-centric programming model introduced by Pregel that allows users to express various graph algorithms in a natural way. Second, most graph processing systems are designed for iterative computation and are in-memory processing systems [5] since they hold the graph data in memory during iterations. Therefore, such systems outperform the general-purpose distributed systems such as MapReduce [6,7] and its open-source implementation Hadoop [8] that typically flush data to the distributed file system at the end of each iteration and reload data into memory in the beginning of the next iteration.

### 1.1. Issues and opportunities

While specialized graph processing systems are efficient for graph analytics, the specialization itself is a double-edged sword. We observe that most graph processing systems require users to establish a new framework in the cluster and conduct necessary configuration before running. This is always a daunting job for cluster managers. In particular, given various types of data analytics applications, we cannot afford to set up a new system for each of them. Furthermore, graph analytics may only be one part of a complex analytics job and we have to switch to other systems if the graph processing system is unsuitable for the analytics tasks afterwards. For example, consider a top $k$ PageRank application which tries to find $k$ web pages with highest PageRank values. This application involves two tasks, one for calculating PageRank for all the web pages and another for computing $k$ pages with the highest PageRank values. Intuitively, the first task can be easily handled by graph processing systems such as Pregel, while MapReduce-based

* Corresponding author.
*E-mail addresses:* shenyy@sjtu.edu.cn (Y. Shen), caiqc@comp.nus.edu.sg (Q. Cai), uqwlu@ruc.edu.cn (W. Lu), sdl@hit.edu.cn (D. Sun), zhongle@comp.nus.edu.sg (Z. Xie).

http://dx.doi.org/10.1016/j.bdr.2016.04.002
2214-5796/

systems such as Hadoop are more suitable to solve the second one. However, switching among different systems will introduce complexity and increase the job execution time due to data movement.

As opposed to the specialized graph processing systems, GraphX [9] was introduced as an embedded graph processing framework on top of a general-purpose dataflow system, namely Apache Spark [10]. To support graph processing, GraphX defines a set of graph operators that are implemented based on the standard dataflow operators (e.g., map, join, group-by) in Spark. While the implementation of GraphX allows complex/multi-stage analytics tasks to be handled in a unified system Spark, it hinders all the optimizations proposed for the specialized graph processing systems and requires a bunch of dataflow optimizations to align its performance with the specialized ones. Furthermore, GraphX represents graph data as two collections (i.e., vertex collection and edge collection) and performs multi-way join over these collections to construct the view of a graph. This requires additional transformation workload since graph data is always represented in an adjacency list.

**Opportunity:** Is there any unified distributed platform which supports complex analytics tasks for various data types (e.g., graph and non-graph data), while retaining the optimizations and advantages of the specialized graph processing system?

Another issue about graph processing systems is the high communication overhead caused by cross-machine message forwarding. Pregel adopts the edge-cut based graph partitioning which distributes vertices among the compute nodes and allows edges to span across the nodes. Network communication overhead is incurred when a vertex in one compute node wants to send a message to its neighbor in another node. It is worth noting that the overhead becomes more significant for *natural* graphs which follow power-law degree distribution. Consequently, PowerGraph [11] proposed vertex-cut based graph partitioning. The idea is to randomly distribute edges among the compute nodes and allow vertices to span across the nodes. Unlike edge-cut partitioning, the communication cost produced by a vertex-cut is restricted to the total number of compute nodes spanned by the vertices. However, when performing a vertex-cut partitioning, PowerGraph requires one compute node to load the entire graph into the main memory, execute the partitioning algorithm and forward edges to the compute nodes accordingly. This limits the size of the graph that can be processed by the vertex-cut partitioning algorithm.

**Opportunity:** Can we implement a light-weight, distributed vertex-cut partitioning method for graph processing?

The third issue is fault tolerance. Most existing distributed graph processing systems adopt checkpoint-based recovery mechanism. That is, a checkpoint that records graph status is made periodically (e.g., every 10 iterations). Once a compute node fails or reports an exception, the job will resume its execution from the latest checkpoint. However, to our best knowledge, existing distributed graph processing systems such as GPS [4], Giraph [3] and PowerGraph [2] cannot recover from failures automatically. When a failure occurs, the job will be terminated and the system requires users to manually restart the job from the latest checkpoint. GraphX [9] achieves fault tolerance by leveraging the features (e.g., lineage) from the dataflow system Spark. However, their approach can hardly be adapted to the advanced Pregel-like graph processing systems.

**Opportunity:** Can we achieve automatic failure detection and recovery in the distributed graph processing systems?

### 1.2. Our solution and contributions

To address the above three challenging issues, we propose our graph processing engine, epiCG. We build epiCG on top of epiC [12], an elastic data processing system proposed for large-scale data analytics. epiC adopts an Actor-like programming model that is able to execute any number of computations (called *units*). In epiC, users can process data with different computation models by defining their own units, e.g., MapUnit and ReduceUnit for MapReduce model, and SelectUnit, JoinUnit and AggregateUnit for relational model. The implementation of epiCG on top of epiC leverages the flexibility and extensibility of epiC and obtains the following benefits.

- Reusability: implementation cycle of our graph engine is shortened by leveraging existing components provided in epiC.
- One-size-fits-all: users do not need to configure the cluster to run a new system for graph applications. A complex analytics task can be divided into multiple stages, each of which is handled by a different type of unit (e.g., MapUnit, ReduceUnit) in epiC.

We build epiCG by implementing a new computation unit called *GraphUnit* in epiC. We define two types of GraphUnits. The first one is called *slave* GraphUnit which manages a subset of the graph data and performs iterative graph computation in parallel with other slave GraphUnits. The second type is called *master* GraphUnit which coordinates the computation activity among all the slave GraphUnits. It is worth mentioning that 1) the graph APIs provided by epiCG align to those defined in Pregel due to their expressiveness and simplicity, and 2) GraphUnit is independent of other types of units in epiC, thus allowing all the optimizations proposed for the specialized graph processing systems to be implemented in epiCG seamlessly.

However, the basic unit programming for GraphUnit is inefficient due to the following two reasons. First, epiC employs a disk-based generic programming model (see details in Section 2). Like MapReduce-based systems, during each iteration, GraphUnit will go through three phases: data loading, one-iteration computation and data flushing. This incurs unnecessary data movement across all the iterations. Second, computation units in epiC cannot communicate with each other directly. Instead, if one GraphUnit wants to send a messages to another GraphUnit, it has to first send the message to the master node in epiC and the master node will then forward the message to the corresponding unit. Apparently, the master would become the bottleneck due to the high volume of the messages involved in most graph applications.

To solve both problems, we implement GraphUnit as an in-memory computation unit. During the computation, all the slave GraphUnits are able to hold graph data in memory, thus eliminating unnecessary I/O cost across iterations. We also allow all the GraphUnits to communicate with each other directly via messages, so that the master node in epiC will not get overwhelmed in receiving and forwarding massive messages introduced by the graph computation. Furthermore, epiCG supports both edge-cut and vertex-cut partitioning methods and allows users to process different graph data with the most suitable partitioning methods. To generate a vertex-cut partitioning, we propose a light-weight partitioning method that allows every GraphUnit to generate a part of the vertex-cut in parallel with others. Finally, epiCG adopts the checkpoint-based recovery method and achieves automatic failure detection and recovery by maintaining the statuses of slave GraphUnits inside the master GraphUnit. When a slave fails, the master GraphUnit will repartition the graph based on all the remaining healthy slaves and instruct them to resume the computation since the latest checkpoint.

We summarize the contribution of this paper as follows.

- We develop a novel graph processing engine epiCG on top of epiC. We build epiCG by implementing a new unit called GraphUnit in epiC. GraphUnit is an in-memory computation unit that can handle iterative computation efficiently and collaborate with other kinds of units to accomplish any complex/multi-stage data analytics.

• epiCG supports both edge-cut and vertex-cut graph partitioning methods. To generate a vertex-cut partitioning, we propose a light-weight greedy strategy which uses multiple compute nodes to generate vertex-cut partitions collaboratively.

• epiCG adopts the checkpoint-based recovery method and leverages the remaining healthy compute nodes to perform recovery automatically upon a failure.

• We conduct comprehensive experiments to demonstrate the efficiency and scalability of epiCG, compared with advanced graph processing systems.

The rest of the paper is organized as follows. Section 2 provides background about epiC and distributed graph processing. Section 3 gives an overview of epiCG, followed by implementation details in Section 4. Section 5 presents fault tolerance in epiCG. Section 6 reports experimental results. We discuss related works in Section 7 and conclude the paper in Section 8.

## 2. Preliminaries

We first provide the background of epiC, the underlying platform of epiCG. We then briefly review the key ideas of specialized graph processing and discuss the two graph partitioning methods: edge-cut partitioning and vertex-cut partitioning.

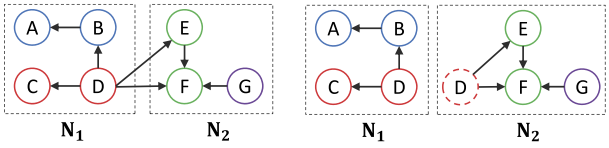
### 2.1. The epiC system

epiC [12] was proposed to address data variety challenge in Big Data. It adopts the Actor-like programming model and provides a simple yet efficient *unit* interface to support various computation models. In epiC, users can express different computation logics by defining different units. Each processing unit performs computation in parallel with other units and communicates with other units through message passing. Messages, however, cannot be sent directly between two units; each message is sent to the master network (which is responsible for routing messages) and then forwarded to the corresponding units. A unit becomes active if it receives messages from the master network. After parsing the message, the unit will load data from the underlying storage and apply the user-defined computation logic to process the data accordingly. When the unit completes its computation, it flushes the output to the storage and becomes inactive until it receives a new message.

epiC adopts the master-worker architecture. There is only a single master node in epiC (this is different from the nodes in the master network which are mainly responsible for message routing). The master node runs a master daemon which monitors the healthy statues of the slaves and commands workers to execute tasks. Each worker node establishes a worker tracker daemon. The worker tracker manages a pool of worker processes which accept and execute assigned unit tasks. In epiC, we assign one unit task to one worker process.

### 2.2. Distributed graph processing

The key idea of distributed graph processing is a vertex-centric programming model proposed by Pregel [1], which abstracts graph algorithms as the computation for every vertex and message exchange between different vertices. Typically, the execution of a graph job consists of three phases: data loading, iterative computation and the output. In the data loading phase, an input graph is loaded from the underlying storage system and distributed among the compute nodes. During iterative computation, each compute node sequentially scans its received vertices and executes a user-defined *compute* function for each of them. Every vertex can update its value and send messages to other vertices during the computation. Pregel-like systems follow Bulk Synchronous Parallel (BSP) model [13] and all the compute nodes will proceed to the next iteration synchronously, while some systems such as PowerGraph [2] perform computation in an asynchronous manner. Finally, in the output phase, compute nodes flush the results (e.g., the computed values of the vertices) to the storage system. Essentially, distributed graph processing distributes graph data and parallelizes computation tasks among a cluster of compute nodes, thus accelerating the processing significantly.




(a) Edge-cut of $\mathcal{G}$ (b) Vertex-cut of $\mathcal{G}$

**Fig. 1.** Two graph partitioning methods.

**Graph partitioning.** We denote the input graph to the distributed graph processing systems by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ contains all the vertices and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ represents directed edges[1] among vertices. Typically, there are two approaches to distribute input graph to multiple compute nodes, using edge-cut and vertex-cut partitioning methods respectively. The *edge-cut* partitioning method distributes the vertices among compute nodes and allows edges to span across the nodes. An optimal edge-cut partitioning minimizes cross-node edges (to reduce network communication cost) as well as balances the computation workload (i.e., the number of vertices) among different compute nodes. However, the complexity of finding an optimal edge-cut partitioning is NP-hard. Consequently, most existing graph systems adopt efficient greedy strategies such as distributing vertices to the compute nodes in a round-robin or hash manner. Fig. 1(a) gives an example of edge-cut partitioning, where we distribute a 7-vertex (A–G) graph $\mathcal{G}$ over two compute nodes $N_1, N_2$. This edge-cut partitioning results in two cross-node edges, D→E and D→F.

In contrast, the *vertex-cut* graph partitioning distributes the edges among compute nodes and allows vertices to span across the nodes. We say that a vertex has multiple copies if it spans across multiple nodes. One copy is selected as the *master* vertex, while others are *mirrors*. Fig. 1(b) shows a vertex-cut partitioning for $\mathcal{G}$. There is no cross-node edge and vertex D has two copies in $N_1, N_2$, respectively. The communication cost using vertex-cut is mainly caused by synchronizing all the copies of the same vertex, which is proportional to the total number of compute nodes spanned by the vertices. It has been theoretically and experimentally proved in [2] that vertex-cut outperforms edge-cut in many real-life graphs which have power-law degree distribution. However, the partitioning algorithm proposed in [2] requires a compute node to load the entire input data into memory and assign the $(i + 1)$-th edge based on the assignment of the previous $i$ edges. This results in large memory footprint and restricts the size of the graphs to be processed.

## 3. Overview of epiCG

In this section, we present the computation model and the architecture of epiCG.

### 3.1. Computation model

epiCG adopts the vertex-centric programming model of Pregel and applies a user-defined *compute* function for each vertex. Specifically, every vertex carries two statuses: active and inactive.

[1] For indirected graphs, we can represent each undirected edge $(u, v)$ by two directed edges $\langle u, v\rangle$ and $\langle u, v\rangle$.

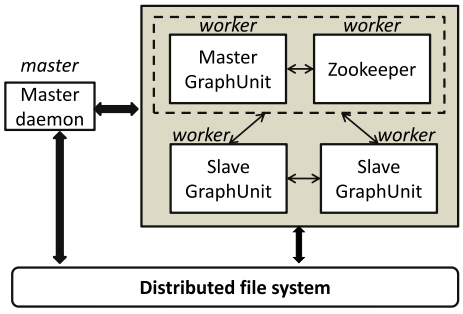



**Fig. 2.** The Architecture of epiCG.

During the computation, a vertex can process the messages sent by other vertices, update its value and send messages to other vertices. If a vertex finds there is no message to send in the current iteration, it can call VoteToHalt() to inactivate itself. An inactive vertex will be automatically activated upon receiving messages from other vertices. When all vertices become inactive or there is no message to send at each vertex, the job is then terminated. epiCG follows the BSP model and the execution of an epiCG program is organized by *supersteps*. That is, a global synchronous point is reached when all the vertices finish computation in one superstep. After that, all the vertices proceed to the next superstep synchronously.

### 3.2. epiCG architecture

Fig. 2 shows the architecture of epiCG. epiCG deploys the distributed file system (DFS) as its underlying storage. Typically, DFS contains the initial graph data to be processed by epiCG and the final results produced by epiCG. epiCG follows the single master-multiple worker architecture of epiC. The master runs a master daemon which maintains the healthy statuses of all the workers and instructs workers to execute *unit* programs. We divide workers into three categories to execute *masterGraphUnit*, *slaveGraphUnit* and *Zookeeper*, respectively. The *masterGraphUnit* coordinates supersteps among all the workers who execute *slaveGraphUnit* and the *Zookeeper* maintains information shared among these workers, e.g., which worker has finished the execution of the current superstep, how many workers have dumped a checkpoint. Typically, given a set of workers, we choose one worker to run the *masterGraphUnit* program. The functionalities of the GraphUnits are listed as follows.

**MasterGraphUnit.** The *MasterGraphUnit* program performs two tasks:

1. partition and distribute the input graph among the workers that run *SlaveGraphUnit* program;
2. coordinate the *SlaveGraphUnit* workers to perform supersteps synchronously.

To perform these tasks correctly, *MasterGraphUnit* maintains several important objects.

- MasterPartitioner: generate the vertex-to-partition mapping for the input graph (see details in Section 4.1);
- MasterClient: notify workers of the newly computed global aggregated values;
- MasterAggregator: retrieve local aggregated values from the workers and generate the global aggregated ones.

**SlaveGraphUnit.** The *SlaveGraphUnit* program is responsible for the following four tasks.

1. load its assigned graph data and flush computation results from/to the storage system;
2. loop over vertices and execute *compute()* function;
3. forward messages generated during the computation;
4. generate aggregated values and write to the zookeeper.

*SlaveGraphUnit* maintains four important objects:

- WorkerServer: retrieves and manages the graph data that is assigned to the worker;
- WorkerPartitioner: maintains partition information for the vertices residing in the worker;
- WorkerClient: forwards messages to the zookeeper and other workers;
- WorkerAggregator: computes aggregated values and writes to the zookeeper.

Once a graph job is submitted to epiCG, all the workers will be activated immediately. At the beginning of the execution, epiCG establishes pairwise connections between GraphUnits. This is different from epiC where units cannot communicate with each other directly, but rely on the message service provided by the master network. As most graph applications such as PageRank and shortest path computation involve a large number of messages, setting up direct connections between units allows them to communicate with each other more efficiently and prevents the master network being the bottleneck.

## 4. Implementation details

The implementation of epiCG addresses two important problems in graph processing: graph partitioning and iterative graph computation. epiCG supports both edge-cut and vertex-cut graph partitioning methods, which allows users to perform graph computation with the most appropriate partitioning method. In this section, we first introduce the graph representation in epiCG and our light-weight vertex-cut generation algorithm. We next present how epiCG performs iterative computation using two partitioning methods.

### 4.1. Graph partitioning

#### 4.1.1. Partition-based graph representation

Consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. epiCG first divides all the vertices into partitions and distributes partitions among the workers that execute *slaveGraphUnit*. Let $\mathcal{P}$ be the set of all the partitions. For any partition $P_i \in \mathcal{P}$, we have $P_i = (\mathcal{V}_i, \mathcal{E}_i)$ where $\mathcal{V}_i \subseteq \mathcal{V}$ and $\mathcal{E}_i = \{(u, v) \in \mathcal{E} \mid u \in \mathcal{V}_i\}$. Typically, the number of partitions is much larger than the number of compute nodes and each compute node will be assigned with multiple partitions. The benefit of partition-based graph representation is to support dynamic repartitioning. For example, every *slaveGraphUnit* can collect statistics such as the execution time for each of its partitions and report this information to the *masterGraphUnit*, who may ask some of *slaveGraphUnits* to exchange partitions for the purpose of load balance. Instead of repartitioning the graph in the vertex granularity, partition-based repartitioning is obviously more cost effective.

Table 1 lists several important data structures managed by each *slaveGraphUnit* for partition maintenance. `Vertex` records the information for a vertex in the input graph, including the identity of the vertex, its value, outgoing edges, and status (active or inactive). For each vertex, we also attach two variables, *isMaster* and *#allEdges*, to support vertex-cut graph partitioning. *isMaster* is a boolean variable indicating whether the vertex is a master version; *#allEdges* records the total number of edges associated with the vertex in the original graph. `Partition` is a list of vertices and

**Table 1**
Graph-related objects maintained by each worker.

| Object | Description |
|---|---|
| `Vertex` | information of a vertex in the input graph |
| `Partition` | a set of vertex in a partition |
| `PartitionStore` | a set of partition residing in the worker |
| `PartitionOwner` | indicating a partition belongs to which worker |
| `PartitionOwnerList` | partition-worker mapping |

each partition is associated with an identifier "pid". We use `PartitionStore` to keep track of all the partitions assigned to the worker. In order to forward messages to the corresponding destination vertices, every worker needs to know in which worker and which partition every destination vertex resides. To do this, we maintain two mappings, one vertex-to-partition mapping $\varphi$ and one partition-to-worker mapping $\phi_p$ in each worker, where $\varphi$ indicates which vertex belongs to which partition and $\phi_p$ indicates which partition is assigned to which worker. If the worker wants to forward a message to vertex $u$, it can easily conclude that $u$ belongs to partition $\varphi(u)$ in the worker $\phi_p(\varphi(u))$.

By default, epiCG adopts a simple hash mapping method to generate $\varphi$ and $\phi_p$. That is, let $\mathcal{P}$ be the set of all the partitions. For a vertex $v$ and a partition $P \in \mathcal{P}$,

$$\varphi(v) = P \Leftrightarrow v.vid \equiv P.pid \pmod{|\mathcal{P}|} \quad (1)$$

Similarly, let $\mathcal{W}$ be the set of all the *slaveGraphUnit* workers. For a partition $P \in \mathcal{P}$ and a worker $W \in \mathcal{W}$,

$$\phi_p(P) = W \Leftrightarrow P.pid \equiv W.workerid \pmod{|\mathcal{W}|} \quad (2)$$

epiCG provides various partitioning methods to generate the two mappings $\varphi$ and $\phi_p$ including hash partitioning, block partitioning. We also allow users to provide their own mappings by overriding the partitioning method in the MasterPartitioner. Moreover, the partitioning can also be decided before graph loading by using advanced graph partitioning tools such as Metis [14]. In epiCG, we use `PartitionOwner` to keep information for a partition, i.e., to which worker this partition is assigned. Every worker maintains a list `PartitionOwnerList` (i.e., $\phi_p$) recording `PartitionOwners` for all the partitions. During the computation, every worker can easily forward a message to the destination vertex by referring to the vertex-partition mapping $\varphi$ and `PartitionOwnerList`.

The input graph of an epiCG job is stored as a plain file in the distributed file system. Typically, a graph file consists of a set of lines, each containing a vertex id and all the ids of its neighbors. In the beginning of graph loading, the *masterGraphUnit* will generate the PartitionOwnerList and inform *slaveGraphUnits* of the list via zookeeper. Initially, every *slaveGraphUnit* loads several data splits (i.e., lines) of the input graph and then iterates over the splits. For each line, it checks whether the vertex belongs to any of its partitions. If so, the vertex and its outgoing edges are appended to its local partition accordingly. Otherwise, *slaveGraphUnit* forwards this line to the corresponding compute node based on `PartitionOwnerList`.

#### 4.1.2. Light-weight vertex-cut generation

We observe that a cross-node edge is caused by assigning the source vertex and the destination vertex to two compute nodes. By creating a copy of the source vertex in the node which contains the destination vertex, the cross-node edge is dismissed. This inspires us to generate a vertex-cut partitioning by distributing edges based on the assignment of destination vertices and creating copies for a source vertex if the source and destination vertices are assigned to different nodes. More specifically, given a vertex $v$ and its outgoing edges $\mathcal{E}_v$, we generate a cut for $v$ in the following steps.

1. We split $\mathcal{E}_v$ into $n$ groups, $\mathcal{E}_1, \ldots, \mathcal{E}_n$, which satisfy: 1) $\bigcup_{i=1}^{n} \mathcal{E}_i = \mathcal{E}_v$ and $\forall i, j \in [1, n] \wedge i \neq j$, $\mathcal{E}_i \cap \mathcal{E}_j = \emptyset$, and 2) $\forall e_1 = \langle v, u_1 \rangle$, $e_2 = \langle v, u_2 \rangle \in \mathcal{E}_v, e_1, e_2$ belong to the same group *iff* $\phi_p(\varphi(u_1)) = \phi_p(\varphi(u_2))$.
2. We create a copy $v_i$ of vertex $v$ for each edge group $\mathcal{E}_i$ and replace with it the source vertex $v$ of each edge in this edge group. We then assign each copy to a compute node where at least one of the destination vertices of its edges reside.
3. We choose the copy assigned to node $\phi_p(\varphi(v))$ as the master vertex, and others as mirrors. If no copy is generated in $\phi_p(\varphi(v))$, we create a copy in partition $\varphi(v)$ as the master vertex.
4. We assign every copy to the corresponding partition. The master vertex is assigned to $\varphi(v)$. For each mirror, we only know the worker to which it is assigned. We then randomly choose a partition in that worker and assign the mirror to that partition. Note that assigning mirror to any partition in that worker will not affect computation cost and message forwarding cost.

For example, consider the graph $\mathcal{G}$ in Fig. 1(a). Suppose every vertex composes a partition and we have $B, C, D$ assigned to $N_1$ and $E, F$ assigned to $N_2$. To generate a cut for vertex $D$, we first split its four outgoing edges into two groups $\mathcal{E}_1 = \{D \to B, D \to C\}$ and $\mathcal{E}_2 = \{D \to E, D \to F\}$. We then create two copies $D_1, D_2$ of $D$, replace vertex $D$ of $\mathcal{E}_1(\mathcal{E}_2)$ with $D_1(D_2)$, and assign $D_1(D_2)$ to $N_1(N_2)$. Since $D$ is assigned to $N_1$, we choose copy $D_1$ as the master vertex and $D_2$ as its mirror. Finally, we obtain the cut of vertex $D$ in Fig. 1(b).

Thus far, we obtain a vertex-cut for each vertex. However, another important issue of vertex-cut generation is that we should avoid spanning vertices with low out-degrees (i.e., the total number of the outgoing edges). This is because the copies of these vertices increase network cost for synchronizing vertex values, while their contributions to reducing the cost of forwarding cross-node message is insignificant [15]. To address the problem, we set a threshold $\theta$ on the out-degrees of the vertices. If the out-degree of a vertex is no larger than $\theta$, we assign all the edges to the vertex without generating any copies for the vertex. Otherwise, we decompose it into several parts to obtain a vertex-cut.

Algorithm 1 provides the pseudo code of our vertex-cut generation algorithm. Given a vertex $v$, we first check whether its out-degree exceeds the threshold $\theta$ (line 3). If not, we assign all the edges to the vertex and the vertex will be forwarded to partition $\varphi(v)$ using Equation (1) (line 4). Otherwise, we start to assign the edges of $v$ among the workers (line 7–14). Specifically, we maintain a map function $N$ from workers to the copies indicating which worker owns which copy (line 7) and iterates over all the outgoing edges of the vertex (line 8). For each edge $\langle v, u \rangle$, we first compute the worker $W$ where $u$ resides (line 9). We then check whether $N$ contains worker $W$ (line 10–11). If so, we retrieve from $N$ the copy $v'$ that is assigned to $W$ and attach $\langle v', u \rangle$ to the edge list of $v'$. If $N$ does not contain worker $W$, we create a copy $v'$ and assign edge $\langle v', u \rangle$ to the copy. Besides, we add the pair $(W, v')$ to $N$ (line 13). We then choose master vertex and assign every copy to a partition (line 16–31). Furthermore, for each copy of $v$, we record the out-degree of $v$ in the original graph (line 28) and for the master version, we record a list of partition identifiers indicating the locations of the mirrors (line 32).

This is easy to verify that the vertex-cut partitioning produced by Algorithm 1 has the following properties.

**Property 1.** *Consider a vertex $v$ and the set $\mathcal{E}_v$ of its outgoing edges. Let $\{(v_1, P_1), \cdots, (v_n, P_n)\}$ be the list $M$ produced by Algorithm 1 and $\mathcal{E}_i$ be the set of edges associated with $v_i$, $1 \leq i \leq n$. We have:*

$$\forall \langle v_i, u \rangle \in \mathcal{E}_i, \phi_p(P_i) = \phi_p(\varphi(u))$$

**Algorithm 1:** GenerateVertexCut.

**Input** : $v$, a vertex
$\phi_p$, PartitionOwnerList
$n$, the number of partitions
**Output**: $M$: list of (copy, partition) pairs

```
1  M ← ∅;
2  E_v ← v.GetEdges ();
3  if |E_v| ≤ θ then
4      M ← {⟨v, GetPartition(v.vid)⟩};
5  else
6      /* generate vertex-cut for v */
7      N ← ∅;
8      foreach Edge ⟨v, u⟩ ∈ E_v do
9          W ← GetWorkerInfo(GetPartition(u.vid), φ_p);
10         v' ← N.get(W);
11         if v' = null then
12             v' ← CreateVertex(v);
13             N ← N ∪ {(W, v')};
14         v'.AddEdge(⟨v', u⟩);
15     /* select master vertex and assign copies to partitions */
16     W* ← GetWorkerInfo(GetPartition(v.vid), φ_p);
17     v_m ← null; P_m ← ∅;
18     foreach (W, v') ∈ N do
19         if W = W* then
20             v'.isMaster ← true;
21             M ← M ∪ {v', GetPartition(v.vid)};
22             v_m ← v';
23         else
24             v'.isMaster ← false;
25             P ← ChooseOnePartition(W);
26             M ← M ∪ {(v', P)};
27             P_m ← P_m ∪ {P.pid};
28         v'.#allEdges ← |E_v|;
29     if W ∉ N.keySet() then
30         v' ← CreateVertex(v);
31         M ← {v', GetPartition(v.vid)};
32     v_m.AddMirrorPartitionIds(P_m);
```

*This guarantees that forwarding messages from any copy to its neighbors does not incur network communication cost.*

Furthermore, in the graph loading phase, every *slaveGraphUnit* worker can apply Algorithm 1 to the vertices in its own data splits in parallel with other workers. After obtaining a vertex-cut for a particular vertex, the worker can forward each copy and its edges to the corresponding partition based on its computed list $M$. After all the workers finish shuffling vertex copies and edges, every worker obtains the following information:

- for every vertex $v$ in its own partitions, the worker knows the out-degree of the vertex in the original graph based on $v.\#allEdges$ and whether it is the master version based on $v.isMaster$.
- for a master vertex $v$ in its own partitions, the worker records in which partition of $v.MirrorPartitionIds$ each mirror of $v$ resides, and further decide to which worker this partition belongs using Equation (2).
- for each vertex in the graph, the worker knows the partition and worker where the master version of the vertex resides based on Equation (1) and (2).

In essence, an edge-cut partitioning can be regarded as a special case of vertex-cut partitioning where all the vertices has a single copy which is the master vertex. Hence, without loss of generality, we next describe how epiCG performs iterative graph computation based on vertex-cut partitioning.

```
public DoubleWritable produceMsg(Edge<LongWritable,NullWritable> edge) {

    int edges = getNumEdges();
    if(edges > 0) {
        double tmp = getValue().get()/edges;
        return new DoubleWritable(tmp);
    }
    else
        return null;
}
```

**Fig. 3.** `ProduceMsg(Edge e)` for PageRank.

### 4.2. Iterative computation

epiCG performs graph computation via a set of iterations, i.e., supersteps. In each superstep, every *slaveGraphUnit* worker undergoes the following three phases.

#### 4.2.1. Mirror message delivery

Every master vertex will forward its updated vertex value to all of its mirrors at the end of each superstep. Once a worker receives an updated vertex value, it will store the value into a `VertexValueUpdateCache`. At the beginning of a superstep, the worker will first update its mirrors with the new values received in the last superstep. Specifically, every worker will go through all of its residing vertices. If a vertex is a mirror and `VertexValueUpdateCache` contains a new value for the vertex, the worker will update its value accordingly. When all the values in `VertexValueUpdateCache` are processed, the worker will clear this cache.

After a mirror updates its value, it needs to produce and forward messages to its own neighbors. That is, a message that should be forwarded directly from the master vertex will now be produced and forwarded by the mirror. To achieve this, we define a new API, `produceMsg(Edge e)`, in `Vertex` class, which allows any mirror to generate and send messages to its own neighbors. In epiCG, if a user submits a graph job and chooses to use vertex-cut partitioning method, we require the user to define `produceMsg` function that will be executed by the mirrors. Fig. 3 shows an implementation of `produceMsg` for PageRank computation. A mirror retrieves its value via `getValue()` and computes the PageRank share of its neighbor, which is the vertex value divided by the out-degree of the vertex in the original graph.

One limitation of `produceMsge` function is that the message generated by a mirror can only rely on the value of the vertex, the out-degree of the vertex in the original graph (stored in `Vertex.#allEdges`) and the information (e.g., value) of the edge. This limitation, however, does not restrict the applicability of the proposed epiCG system, as it fits into the message model of a large range of graph analytics applications, including PageRank, breadth first search, graph keyword search, triangle counting, connected component computation, graph coloring, minimum spanning forest computation, k-means, shortest path, minimum cut, and clustering/semi-clustering. All these applications possess the property that the content of any message from $u$ to $v$ is determined by the latest value of $u$, the out-degree of the vertex (in the original graph) and the information of the edge $\langle u, v\rangle$.

Each message produced by a mirror will be forwarded to the master version of the destination vertex. According to Property 1, such master vertices reside in the same compute node as the mirror. Hence, each generated message will be appended by the worker to its local incoming message store without any network transmission.

#### 4.2.2. Master vertex computation and message delivery

After all the mirrors finish producing and forwarding messages, vertex computation starts. In this phase, every worker checks its partitions in the `PartitionStore`. For each partition, the worker loops over the vertices in it and performs computation for a vertex $v$ iff (1) $v$ is a master vertex; (2) $v$ receives at least one message sent by other vertices.

When a master vertex performs computation, it may produce messages. Note that a master vertex may only has a subset of the edges and hence can only send messages to its known neighbors. Every worker collects the messages produced by the master vertices during the computation and forwards them to the workers where the destination vertices reside accordingly. If the destination vertex of a message has multiple replicas, the message will only be forwarded to the master vertex, and none of the mirrors will receive any messages. In epiCG, all the workers forward their messages asynchronously.

#### 4.2.3. Vertex value synchronization

When a master vertex finishes its computation, it may produce a new vertex value and all of its mirrors must be informed of this new value. In epiCG, every worker is responsible for sending the updated value of a master vertex (obtained after vertex computation) to all the mirrors. Recall that every master vertex maintains a list `MirrorPartitionIds` recording the partitions where the mirrors reside. By checking this list, the worker forwards the updated vertex value to the mirrors via `syncVertexValueRequest` requests. To ensure the correctness of vertex value synchronization (i.e., every mirror must receive the updated vertex value successfully), the requests of `syncVertexValueRequest` are handled by a TCP-like *three-way handshake* protocol. When a worker $W_1$ forwards a `syncVertexValueRequest` request to another worker $W_2$, $W_1$ will wait for a *completion* signal sent by $W_2$. If $W_1$ does not receive the completion signal for a while, it will re-send the request. Any worker who receives a `syncVertexValueRequest` request will parse the request and put the updated value into `VertexValueUpdateCache` for the corresponding mirror. Every worker will continue its processing only if all of its requests are sent successfully and all of its received requests are handled properly.

## 5. Fault tolerance

Failure detection and recovery are two key problems to achieve fault tolerance. epiCG detects failures by asking every *slaveGraphUnit* worker to register its healthy status periodically. At the end of each superstep, the *masterGraphUnit* worker will check healthy statuses for all the slave workers. If a slave worker does not register its status over a time period, the master worker will regard it as *failed*.

For failure recovery, epiCG adopts one of the most widely used recovery mechanisms, namely checkpoint-based recovery. Specifically, all the *slaveGraphUnits* write a global checkpoint (recording graph status) periodically and restarts from the latest checkpoint upon a failure. However, we observe that in most existing distributed graph processing systems such as Giraph and GraphLab, the master will report error messages for the failed workers and terminate the job accordingly. To resume the execution of a failed job, existing systems require users to manually launch a new job that starts from the latest checkpoint. Apparently, such kind of implementation for failure recovery requires long downtime. Furthermore, it violates the fault tolerance requirement that the system should be able to recover from failures and resume normal execution automatically.

To achieve automatic recovery, upon slave worker failures, epiCG leverages the remaining healthy slave workers to continue the execution instead of terminating the job. Specifically, the master worker will create a `JobState` indicating: 1) the next superstep (i.e., the latest checkpointing superstep) to perform, and 2) every slave worker should load graph data from the latest checkpoint in the next superstep. The master worker then broadcasts `JobState` to all the healthy slave workers. At the beginning of next superstep, for edge-cut partitioning, the master worker will generate a new partition-to-worker mapping based on the remaining healthy slave workers and inform them of the new mapping via zookeeper; the slave workers will first load graph data from the latest checkpoint, exchange vertices and edges using edge-cut partitioning or vertex-cut partitioning based on the newly received partition-to-worker mapping. After that, all the workers redo the lost computation from the latest checkpointing superstep automatically. Note that even if healthy workers retain the up-to-date vertex values, they do need to redo the lost computation to reproduce messages which are demanded for the recovery of failed vertices. We also implement the parallel recovery mechanism proposed in [16] to accelerate the recovery process.

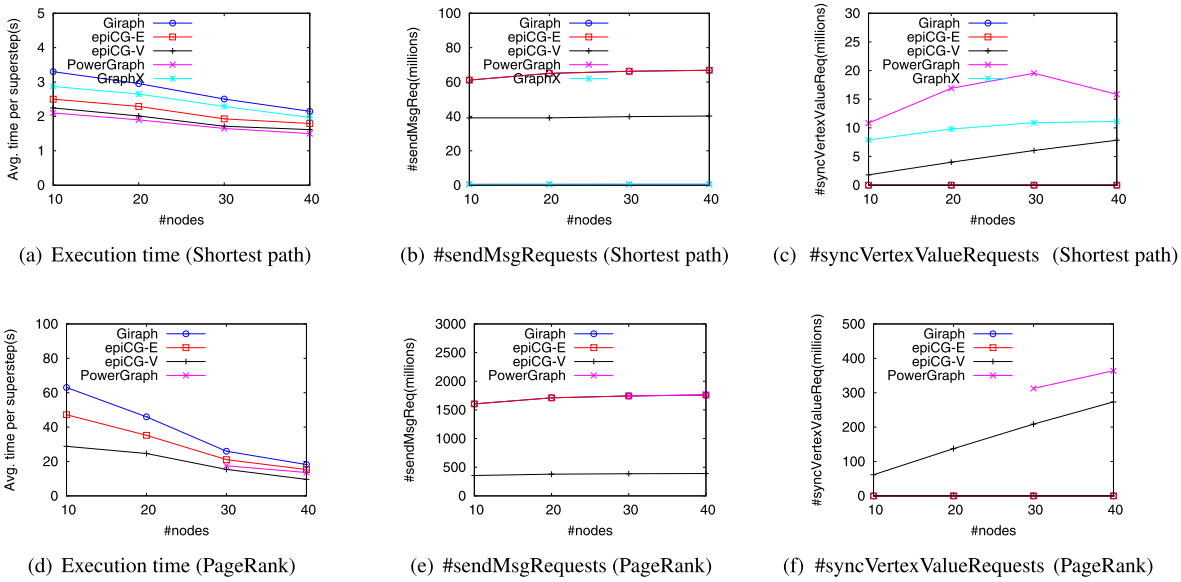
Note that the job completion time would increase dramatically when too many slave workers get failed. A user may prefer to terminate the job and restart with a fixed number of healthy slave workers to continue the execution. In epiCG, we provide a threshold `MinWorker` on the minimal number of healthy slave workers. `MinWorker` is set to 1 by default, but we allow users to set their own values for `MinWorker` via job configuration. Upon failures, the master worker will first check whether the number of remaining healthy slave workers is smaller than `MinWorker`. If not, it will perform automatic recovery as mentioned before. Otherwise, it will terminate the execution immediately.

## 6. Experimental evaluation

We evaluate the performance of epiCG by comparing it with several popular distributed graph processing systems [17], Giraph [3], PowerGraph [11] and GraphX [9]. Giraph employs edge-cut graph partitioning, and the other two use vertex-cut graph partitioning. It has been experimentally shown that PowerGraph is the most efficient graph processing engine on small graphs compared with several other popular graph processing systems such as Hama and GraphX [18]. In all the experiments, we use Giraph version 1.0.0 and PowerGraph version 2.2. Our comparisons include running time, communication cost, scalability and speedup.

### 6.1. Experiment setup

We ran all the experiments on our in-house clusters. The cluster consists of 72 compute nodes, each of which is equipped with one Intel X3430 2.4 GHz CPU, 8 GB of memory, two 500 GB SATA hard disks and gigabit ethernet. For each node in the cluster, we installed CentOS 5.5 operating system, Java 1.7.0 with a 64-bit server VM and Hadoop 0.20.203.0 [8]. Giraph runs as a Map-only job on top of Hadoop, and hence we made the following changes to the default Hadoop configurations: (1) the replication factor is set to 1; (2) each node is configured to run one map task; (3) the size of virtual memory for each map task is set to 4 GB. For Giraph, one node was selected as the master running Hadoop's NameNode and JobTracker, while the remaining compute nodes were the slaves running TaskTracker daemons. For PowerGraph/epiCG, we chose one node as the master/master worker and all the others to be the slaves/slave workers. We required every node to execute only one master or slave thread and set the virtual memory size for each thread to 4 GB. To make a fair comparison, we ran PowerGraph in the synchronous mode. Note that the gains in asynchronous mode is often compromised by the complex concurrency control and there is no dominating mode [19]. By default, we chose 31




(a) Execution time (Shortest path) (b) #sendMsgRequests (Shortest path) (c) #syncVertexValueRequests (Shortest path)

(d) Execution time (PageRank) (e) #sendMsgRequests (PageRank) (f) #syncVertexValueRequests (PageRank)

**Fig. 4.** Scalability.

**Table 2**
Dataset description.

| Dataset | Data size | #Vertices | #Edges | Avg. degree |
|---|---|---|---|---|
| Livejournal | 1.0 GB | 3,997,962 | 34,681,189 | 8.67 |
| Friendster | 31.16 GB | 65,608,366 | 1,806,067,135 | 27.53 |
| Ftiny | 1.9 GB | 12,739,496 | 221,933,535 | 17.42 |
| Fsmall | 3.7 GB | 17,694,120 | 428,192,865 | 24.19 |
| Fmedium | 7.5 GB | 25,126,704 | 862,648,522 | 34.33 |

compute nodes out of the 72 compute nodes for the experiments. For all the three systems, we used HDFS as the underlying storage system.

### 6.2. Benchmark tasks and datasets

We study the performance of different distributed graph processing systems using the following two benchmark tasks.

**1) Shortest path.** Shortest path computing is to select one vertex as the *source* and compute the shortest distances to the source for all the vertices. For all the three systems, we always use the same vertex as the source.

**2) PageRank.** The PageRank algorithm is an iterative graph processing algorithm. We refer the readers to the original paper [20] for the details of the algorithm. Without loss of generality, we run all the tasks for 10 supersteps and all the results are averaged over ten runs.

We conduct the experiments using several publicly available real-life datasets.[2] Table 2 provides the details for each of them.

**1) Livejournal.** Livejournal is an online social networking service that enables users to post blogs, journals, and dairies. It contains more than 4 million vertices (users) and over 30 million directed edges (friendships between users). We use this dataset to evaluate the execution of Shortest path tasks.

**2) Friendster.** Friendster is an online social networking and gaming service. It contains more than 60 millions vertices and 1 billion edges. We use it to evaluate the execution of PageRank tasks. To evaluate the speedup of various systems, we prepare three down-samples, **Ftiny, Fsmall, Fmedium**, of Friendster by randomly selecting a subset of vertices from the original Friendster dataset and only keeping the edges associated with the selected vertices.

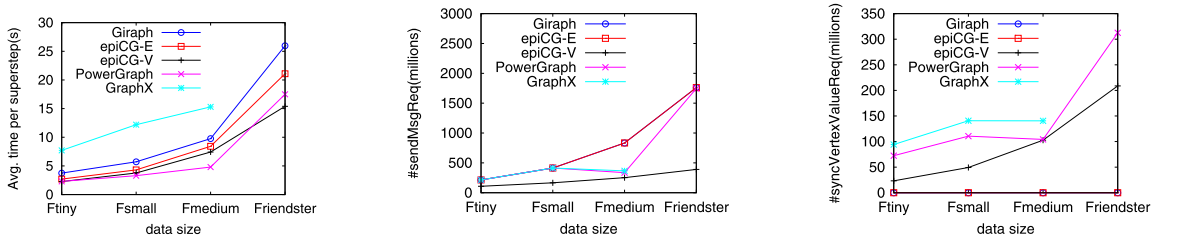
### 6.3. Results

For the experiments, we compare the performance of epiCG-E (using edge-cut), epiCG-V (using vertex-cut) with Giraph (using edge-cut), PowerGraph (using vertex-cut) and GraphX [9] over two **metrics**: running time and communication cost. For the communication cost, we calculate the number of cross-node messages (i.e., *sendMsgRequests*) to be forwarded during the computation as well as the number of cross-node messages (i.e., *syncVertexValueRequests*) to synchronize vertex values from master vertices to their mirrors. We measure the number of messages instead of total message size because each type of message has the same size across all the systems. The vertex-cut degree threshold is initially set to 60, and the impact of different values of $\theta$ on the performance of epiCG-V will be investigated in Section 6.3.3 in detail.

#### 6.3.1. Performance comparison

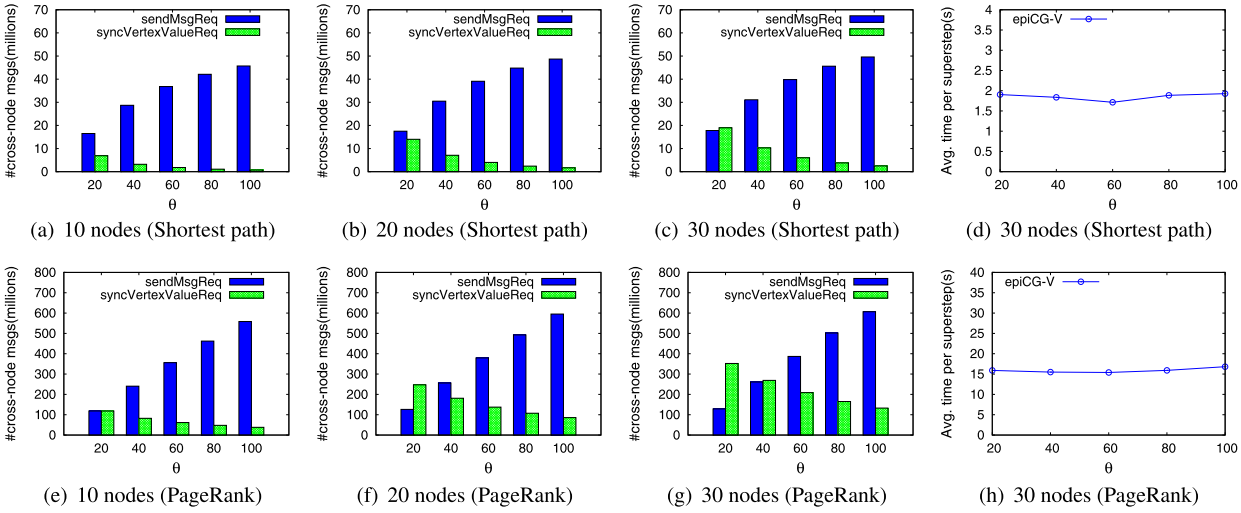
Fig. 4(a)–4(c) show the execution time per superstep and communication cost in Shortest path task for each of the five systems with varied number of compute nodes. As we can see, epiCG-V and PowerGraph perform best among all five systems as they incur fewer messages than the other three. Although the number of sendMsgRequests in epiCG-V is larger than PowerGraph, there is a little difference in the execution time between epiCG-V and PowerGraph. This is because such messages are sent asynchronously, and PowerGraph requires much more synchronous syncVertexValueRequests than epiCG-V, as shown in Fig. 4(c). epiCG-E performs slightly worse than epiCG-V, which is mainly due to that epiCG-V leverages vertex-cut to reduce the message forwarding cost during computation. As shown in Fig. 4(b), the number of sendMsgRequests in epiCG-V is only two-thirds as much as that in epiCG-E and Giraph. epiCG-E and Giraph require the same number of sendMsgRequests as they use the same partitioning function, and both of them do not generate syncVertexValueRequests as a result of edge-cut graph partitioning. However, Giraph experiences a longer execution time than epiCG-E. The reason is that Giraph is built on top of Hadoop, which is a MapReduce framework and does not suit graph computing very well. The high running time of GraphX can be attributed in a similar way, as it is built on Spark [21], which is an in-memory MapReduce framework.

[2] http://snap.stanford.edu/.




(a) Execution time (PageRank) (b) #sendMsgRequests (PageRank) (c) #syncVertexValueRequests (PageRank)

**Fig. 5.** Speedup.




(a) 10 nodes (Shortest path) (b) 20 nodes (Shortest path) (c) 30 nodes (Shortest path) (d) 30 nodes (Shortest path)

(e) 10 nodes (PageRank) (f) 20 nodes (PageRank) (g) 30 nodes (PageRank) (h) 30 nodes (PageRank)

**Fig. 6.** Effect of vertex-cut degree threshold $\theta$.

Fig. 4(d) provides the execution time per superstep in PageRank task. The execution time in all the graph systems decreases almost linearly with the number of compute nodes. Compared with Shortest path task, PageRank task requires 10x more execution time due to the large size of the Friendster dataset. We fail to get PowerGraph result for the cases with 10 and 20 nodes and GraphX results for all cases, because PowerGraph and GraphX generated a large number of mirrors and exhausted memory space. epiCG-V requires the least execution time in all scenarios with various numbers of compute nodes. On average, epiCG-V runs over 2x and 1.5x faster than Giraph and epiCG-E, respectively. This is because we tested PageRank task on Friendster dataset, which follows power-law degree distribution, and vertex-cut is able to reduce the communication cost significantly for such natural graphs. For 30 and 40 nodes, epiCG-V runs slightly faster than PowerGraph, which can be attributed to the 3x more sendMsgRequests and 1.5x more syncVertexValueRequests generated by PowerGraph than by epiCG-V, as shown in Fig. 4(e) and Fig. 4(f).

### 6.3.2. Graph size

We now study the effect of different input graph size on the performance of the four systems. To this end, we perform PageRank task against 4 datasets with various sizes, i.e., Ftiny, Fsmall, Fmedium and Friendster (see details in Table 2), using 30 compute nodes. Since GraphX is not able to run over the largest Friendster dataset, the corresponding result is thus absent. Fig. 5(a) shows the execution time per superstep required for each dataset. As shown in this figure, epiCG-V performs best among all five systems in the case of the largest dataset (i.e., Friendster), and only second to PowerGraph in the other cases. Furthermore, epiCG-V shows the best adaptivity to input graph size in the sense that the execution time of epiCG-V increase slowest as the graph size increases. In addition, for all datasets, epiCG-V requires less execution time than Giraph, epiCG-E and GraphX, and this gap becomes more significant with the increase of input dataset size.

Fig. 5(b) and 5(c) respectively provide the number of sendMsgRequests and that of syncVertexValueRequests generated for the four datasets. As we can see, epiCG-V incurs least sendMsgRequests and syncVertexValueRequests among all three vertex-cut systems, i.e., epiCG-V, PowerGraph and GraphX, and the total number of messages generated by epiCG-V is much less than that of the two edge-cut systems, i.e., Giraph and epiCG-E. Furthermore, as the size of input dataset increases, the increasing rate of generated messages in epiCG-V is slower than other systems, especially in the case of large graphs. This also explains the best performance of epiCG-V in the Friendster dataset case.

### 6.3.3. Vertex-cut degree threshold $\theta$

Finally, we study the effect of vertex-cut degree threshold $\theta$ on the performance of epiCG-V. Recall that $\theta$ decides whether we need to generate mirrors for a vertex. Fig. 6(a)–6(c) show how the number of cross-node requests forwarded in Shortest path task changes with $\theta$ in several scenarios with various numbers of compute nodes. In each scenario, when $\theta$ becomes larger, the number of sendMsgRequests also increases. In particular, the number of sendMsgRequests for $\theta = 100$ is around 3x larger than that for $\theta = 20$. This is because larger values of $\theta$ result in more vertices without mirrors and hence more sendMsgRequests forwarded directly from vertices to their neighbors. In contrast, the number of syncVertexValueRequests decreases as $\theta$ becomes larger, due to the fact that for larger values of $\theta$, more vertices have no mirrors and fewer number of requests is required for vertex value synchronization. Interestingly, when the number of compute nodes increases from 10 to 30, the number of sendMsgRequests for each particular value of $\theta$ remains constant while the number of syncVertexValueRequests increases significantly. This is because the neighbors of

a vertex can be assigned to multiple different compute nodes, and consequently, with more available compute nodes, the number of mirrors generated for a vertex is also likely to increase. Fig. 6(e)–6(g) demonstrate the communication cost for PageRank task. Compared with Shortest path task, PageRank incurs large numbers of sendMsgRequests and syncVertexValueRequests in all the cases. This is because PageRank task is performed over Friendster dataset, which is 30x larger than Livejournal dataset used for Shortest path task.

Fig. 6(d) and 6(h) provide the execution time per superstep for Shortest path and PageRank task, respectively. The number of compute nodes used for this experiment is 30, and we omit the results for the cases with 10 an 20 compute nodes, since they exhibit similar behaviors as in the 30-node case. It can be observed from the two figures that although the numbers of sendMsgRequests and syncVertexValueRequests vary significantly with the threshold $\theta$, the execution time keeps almost unchanged. This is easy to understand. On one hand, a small value of $\theta$ results in less sendMsgRequests being forwarded during computation, but more mirrors for vertex as well as the cost for mirror synchronization; on the other hand, a large value of $\theta$ incurs more sendMsgRequests but less synchronization cost. As a result, we conclude that our previous result for execution time of epiCG-V also holds for the value of $\theta$ other than 60, and one does not need to tune it for the best performance.

## 7. Related work

**Distributed graph processing.** There have been a large number of systems proposed for graph processing. Pregel [1] follows the Bulk Synchronous Parallel (BSP) model and introduces a vertex-centric programming model that allows users to express graph algorithms in a natural way. Later on, various implementation of Pregel have been developed. Giraph [3] treats the computation as a map-only job in MapReduce framework [6] where the input and output data are stored in HDFS. Bu et al. [18] proposed Pregelix, which runs iterative dataflows dealing with computations, to conduct graph analysis. Yan et al. [22] implemented Pregel+, a C/C++ graph system eliminating serialization cost introduced by Java. Graphlab [23] introduces a shared memory abstraction which allows the adjacent vertices and edges to be accessed by the local vertex and hence enables users to concentrate on the sequential computation by hiding the details of data movement between vertices.

Since synchronization is indispensable in BSP model, the stragglers, i.e., the workers who run much slower than others, can significantly slow down the synchronization process. Hence, Salihoglu et al. [4] introduced GPS which follows the same storage design with Giraph but provides two optimizations. The first one is to maintain dynamic partitions among workers, and the second one is to divide the adjacency list of high-degree vertices into different workers to balance the loads of workers. Graphlab [23] and PowerGraph [2] employ Gather-Apply-Scatter (GAS) model and can run in synchronous or asynchronous mode. Asynchronous mode migrates the stragglers by eliminating global synchronization cost. However, this increases system complexity due to the concurrency control for serializability.

Recent efforts focused on leveraging existing platforms for graph processing. Simmen et al. [24] introduced Aster 6 which provides SQL-like interface for graph analytics. Similarly, Fan et al. [17] proposed Grail, a syntactic layer for querying graph on top of RDBMS, which translates graph queries into SQL queries. GraphX [9] is built on Spark [10], which implements Pregel by leveraging general dataflow operators in Spark. To align the performance with the specialized graph processing systems, various optimizations for dataflow operators have been proposed. This is different from our work as we develop epiCG as one unit in epiC for graph processing and reply on epiC to leverage different units for complex analytics query processing. As an independent unit, epiCG allows us to implement all the optimizations proposed for the specialized graph processing systems.

**Graph partitioning.** Graph partitioning is critical to distributed graph processing. While various edge-cut partitioning methods [14, 25–27] have been proposed to balance the computation workload among multiple compute nodes and try to minimize the network communication cost, less attention has been paid to vertex-cut partitioning. PowerGraph [2] proposed a greedy strategy to generate vertex-cut partitioning, which require to use a single compute node to load the entire graph into memory, thus restricting the size of graph that can be handled. SBV-cut [28] and JA-BE-JA-VC [29] are two recent works for distributed vertex-cut partitioning. However, both of them requires iterative computation over the entire graph, which does not allow multiple compute nodes to generate a vertex-cut partitioning in parallel.

## 8. Conclusion

In this paper, we present our distributed graph processing engine epiCG. We develop epiCG as one extension of epiC to avoid extra configuration for a new system. epiCG supports both edge-cut and vertex-cut partitioning, and a light-weight approach is employed in epiCG to parallelize the generation of vertex-cut partitions. epiCG also allows automatic failure detection and recovery. The experiments on real-life datasets illustrate the high efficiency and scalability of epiCG, compared with three state-of-the-art distributed graph processing systems, Giraph, PowerGraph and GiraphX.

## References

[1] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: SIGMOD, 2010.
[2] K. Lang, Finding good nearly balanced cuts in power law graphs, Tech. Rep. YRL-2004-036, Yahoo! Research Labs, 2004.
[3] http://giraph.apache.org/.
[4] S. Salihoglu, J. Widom, Gps: a graph processing system, in: SSDBM, ACM, New York, NY, USA, 2013, pp. 22:1–22:12.
[5] H. Zhang, G. Chen, B.C. Ooi, K.-L. Tan, M. Zhang, In-memory big data management and processing: a survey, IEEE Trans. Knowl. Data Eng. 27 (7) (2015) 1920–1948.
[6] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, in: OSDI, 2004.
[7] F. Li, B.C. Ooi, M.T. Özsu, S. Wu, Distributed data management using mapreduce, ACM Comput. Surv. (CSUR) 46 (3) (2014) 31.
[8] http://hadoop.apache.org/.
[9] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, I. Stoica, Graphx: graph processing in a distributed dataflow framework, in: OSDI, 2014, pp. 599–613.
[10] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: HotCloud, 2010.
[11] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: distributed graph-parallel computation on natural graphs, in: OSDI, 2012.
[12] D. Jiang, G. Chen, B.C. Ooi, K.-L. Tan, S. Wu, epiC: an extensible and scalable system for processing big data, Proc. VLDB Endow. 7 (7) (2014) 541–552.
[13] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[14] G. Karypis, V. Kumar, Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0.
[15] R. Chen, J. Shi, Y. Chen, H. Guan, H. Chen, Powerlyra: differentiated graph computation and partitioning on skewed graphs, Tech. rep., 2013.
[16] Y. Shen, G. Chen, H.V. Jagadish, W. Lu, B.C. Ooi, B.M. Tudor, Fast failure recovery in distributed graph processing systems, Proc. VLDB Endow. 8 (4) (2014) 437–448.
[17] J. Fan, A.G.S. Raj, J.M. Patel, The case against specialized graph analytics engines, in: CIDR, 2015.
[18] Y. Bu, V.R. Borkar, J. Jia, M.J. Carey, T. Condie, Pregelix: big(ger) graph analytics on a dataflow engine, Proc. VLDB Endow. 8 (2) (2014) 161–172.

[19] C. Xie, R. Chen, H. Guan, B. Zang, H. Chen, Sync or async: time to fuse for distributed graph-parallel computation, in: PPoPP, 2015, pp. 194–204.
[20] L. Page, S. Brin, R. Motwani, T. Winograd, The pagerank citation ranking: bringing order to the web, Tech. rep., 1999.
[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, in: NSDI, 2012.
[22] D. Yan, J. Cheng, Y. Lu, W. Ng, Effective techniques for message reduction and load balancing in distributed graph computation, in: WWW, 2015, pp. 1307–1317.
[23] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein, Graphlab: a new parallel framework for machine learning, in: UAI, 2010.
[24] D. Simmen, K. Schnaitter, J. Davis, Y. He, S. Lohariwala, A. Mysore, V. Shenoi, M. Tan, Y. Xiao, Large-scale graph analytics in aster 6: bringing context to big data discovery, Proc. VLDB Endow. 7 (13) (2014) 1405–1416.
[25] D.A. Padua (Ed.), Encyclopedia of Parallel Computing, Springer, 2011.
[26] G. Karypis, V. Kumar, Parallel multilevel k-way partitioning scheme for irregular graphs, in: Supercomputing, 1996.
[27] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: Supercomputing, 1995.
[28] M. Kim, K.S. Candan, Sbv-cut: vertex-cut based graph partitioning using structural balance vertices, Data Knowl. Eng. 72 (2012) 285–303.
[29] F. Rahimian, A.H. Payberah, S. Girdzijauskas, S. Haridi, Distributed vertex-cut partitioning, in: DAIS, 2014, pp. 186–200.