Front. Comput. Sci., 2016, 10(3): 462–476
DOI 10.1007/s11704-016-5485-7

RESEARCH ARTICLE

# iGraph: an incremental data processing system for dynamic graph

**Wuyang JU, Jianxin LI (✉), Weiren YU, Richong ZHANG**

School of Computer Science and Engineering, Beihang University, Beijing 100191, China

**Abstract** With the popularity of social network, the demand for real-time processing of graph data is increasing. However, most of the existing graph systems adopt a batch processing mode, therefore the overhead of maintaining and processing of dynamic graph is significantly high. In this paper, we design iGraph, an incremental graph processing system for dynamic graph with its continuous updates. The contributions of iGraph include: 1) a hash-based graph partition strategy to enable fine-grained graph updates; 2) a vertex-based graph computing model to support incremental data processing; 3) detection and rebalance methods of hotspot to address the workload imbalance problem during incremental processing. Through the general-purpose API, iGraph can be used to implement various graph processing algorithms such as PageRank. We have implemented iGraph on Apache Spark, and experimental results show that for real life datasets, iGraph outperforms the original GraphX in respect of graph update and graph computation.

**Keywords** big data, distributed system, in-memory computing, graph processing, hotspot detection

## 1 Introduction

In recent years, social media such as Facebook, Twitter and Weibo, have become the most popular information sharing platforms, where news, social behaviors and opinions propagate rapidly. The increasing popularity of these social networks creates a new demand for processing real-time dynamic graph. Analysis of these data in real-time manner has significant business value. For example, people all over the world visit Facebook, and more or less click the advertisements shown on it. Based on the immediate analysis of the Facebook social graph (e.g., user relationships, info about users, and connections between users and photos), Facebook can timely adjust the strategy and budget for each advertiser, meanwhile provide more accurate advertisements. Such analysis can be summarized into three steps: 1) process the update of the evolving graph; 2) run the user-defined algorithm (usually incremental algorithm); and 3) get feedback. To keep up with the fast changing graph, the whole process should be completed as soon as possible.

There are several challenges for processing dynamic graph in such a scenario. The first is that modifications of the graph structure tend to be time-consuming in existing systems, especially for inserting new nodes and new edges [1]. The second is that real-time processing is not easy to be achieved, because most of the existing graph-parallel systems [2–10] process graph in a global and batch mode. Along with the huge scale of social graphs, the time cost of each iteration is unacceptable [11]. The third is that the time-sensitive feature of dynamic graph may lead to workload imbalance, because some of the graph nodes are more frequently updated than others in a particular period of time [12–14].

In order to address the above challenges, we propose iGraph, an distributed graph processing system designed to timely process dynamic graph with its continuous updates. iGraph maintains two parts of data, one is the main graph, the other is a sequence of blocks of update data. Our main

Received November 15, 2015; accepted February 19, 2016

E-mail: lijx@act.buaa.edu.cn

contributions are as follows:

- A vertex-cut and hash-based graph partition strategy is introduced. As a result, the evolving graph can be distributed across partitions, and when a stream of incoming data arrives, the graph structure can be updated finegrainedly and rapidly.
- A graph computation engine is designed, which supports incremental graph algorithms. We adopt a vertex-based graph computation model, which abstracts the logic of most of the incremental graph algorithms, to provide easy-to-use interfaces.
- A workload rebalancer based on hotspot is designed. As incremental processing mainly focuses on the changed part of the graph, the time-sensitive feature may cause workload imbalance. The rebalancer evaluates the computation cost of hotspots with their heat rate and the number of adjacent nodes, then uses greedy algorithms to rebalance them.
- We implement iGraph on Apache Spark [15] and GraphX [16], which provides a lineage-based and overhead-negligible fault tolerance mechanism through its data structure RDD. Results show that iGraph is up to 10x faster than the original GraphX in respect of both graph update and graph computation.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 gives some necessary background knowledge. Section 4 presents an overview of iGraph. Section 5 describes how the graph partition strategy works. Section 6 introduces our graph computation engine. Section 7 presents the programming model of iGraph. Section 8 reports the results of experiments on clustering and incremental graph algorithms. Finally, we conclude this paper in Section 9.

## 2 Related work

Our work draws inspirations from many characteristics of the earlier graph processing systems and incremental processing systems.

• **Graph processing systems** In recent years, online social networks are growing more and more rapidly. A large number of research focuses on fast processing on large-scale graph [17]. Pregel [2] adopts the BSP model [18] and a vertex-based graph computation model. Giraph[1] regards a worker node in Pregel as a mapper in hadoop to avoid the I/O overhead. Based on a shared memory abstraction, GraphLab [5,19] adopts an asynchronous programming model for graph computation. PowerGraph [8] further introduces the powerlow degree distribution in real-world graphs and proposes edge-cut partition strategy to improve the efficiency of processing social graphs. Based on an extended version of the graph visitor pattern, Pearce et al. [6] propose an asynchronous graph computation model in a multi-core scenario. Kang et al. [7] designs a large number of highly-scalable graph algorithms to execute on Hadoop. GraphX [16] is a distributed in-memory graph computation framework built on top of the widely used large-scale data processing engine Apache Spark. It unifies graph-parallel and data-parallel computation and achieves a balance among expressiveness, performance, and ease of use. Some work [20–22] has been proposed to ensure the locality of the graph computation. However, most of the above systems concentrate more on batch graph processing. For social networks applications which update and compute frequently on the graph, these systems will cause a noticeable overhead.

• **Incremental processing systems** Incremental processing widely exists in areas of data mining, text processing, graph processing, and other data or computing intensive applications. Systems like Haloop [23], Yahoo!'s continuous bulk processing (CBP) [24], DryadInc [25] and Nectar [26] adopt a data-parallel programming model along which introduce new primitives to store and reuse the result of prior state for incremental processing. The output of a stage is redirected with loopback as its input. They take the prior state as an explicit input combined with the new input. Incoop [27] is an incremental computation framework extended from MapReduce. It designs an incremental HDFS, improves the scheduling performance, and provides a transparent solution for users. To create web search index incrementally, Google's Percolator [28] introduces an event-driven programming model, which is an application structured as a series of observers. When user-specified data changes, observers are triggered to modify other data. Naiad [29] designs *timely dataflow* based on a directed graph, on which stateful dataflow vertices send and receive logical timestamps to track the progress in the iterative algorithms. However, most of the above systems are complex for users to implement their algorithms, and they mainly focus on the data computation model, but not the graph computation model.

• **Incremental graph processing systems** Although most of the graph processing systems focus on large-scale dis-

[1] http://giraph.apache.org

tributed graph processing on a static graph, several systems have been designed to solve the problem of incremental computation on a dynamic graph. X-stream [30] and GraphChi [31] introduce an edge-centric processing model for graph computation, and support addition of edges between different computation stages. However, they are designed for single-machine processing. Kineograph [32] is a distributed in-memory graph processing system, which adopts *epoch commit* protocol to maintain incremental snapshots of the graph. The graph-mining algorithms are executed on a static underlying graph. We extends the above systems with a more flexible incremental computing engine which can handle applications with different computing request frequency. Moreover, the engine has a workload rebalance module which takes advantage of hotspot detection to make the algorithms execute more efficient on an evolving social graph.

## 3 Background

In order to have a better understanding of the motivation of our system, we first give a brief introduction to Apache Spark and GraphX which are most relevant to iGraph.

- **Apache spark** Spark is a MapReduce-like data-parallel computation engine. Several features of it make it different from traditional MapReduce engines. First, Spark supports general computation directed acyclic graph (DAG), much more efficient than Hadoop's two-stage mode. Second, the computing engine can address the loss of any workers and can automatically re-execute the lost tasks. Third, Spark provides an in-memory storage abstraction called resilient distributed datasets (RDDs) to avoid reading and writing large amounts of data to the disk, and automatically reconstructs lost partitions upon failures.

- **GraphX** GraphX is a layer on top of Spark that provides a graph data structure composed of Spark RDDs, and it provides an API to operate on those graph data structures. GraphX comes with the standard Spark distribution, and extends it with some GraphX-specific API. GraphX implements a notion called the property graph, where vertices and edges can have arbitrary sets of attributes associated with them. The attribute could be something as simple as the age of a person or something as complex as an XML document, image or video. GraphX stores graph's edges in one table and vertices in another. This allows graph algorithms implemented in GraphX to efficiently traverse graphs either as graphs (i.e., along edges from one vertex to another) or as tables of edges or vertices. The latter mode of access permits efficient bulk transforms of edge or vertex data. Although GraphX stores edges and vertices in separate tables as RDBMS' schema-based design, internally it has special indexes (i.e., the *routing table*) to rapidly traverse the graph.

## 4 System overview

The overview of iGraph is shown in Fig. 1. The *Row data processor* keeps on analyzing the incoming update data (e.g., microblogs) in a batch manner [33], which means that we process the update data every time interval. Upon analyzing, a time signal will be sent to the *application scheduler* to synchronize, and two kinds of data will be generated and then sent to the corresponding module.

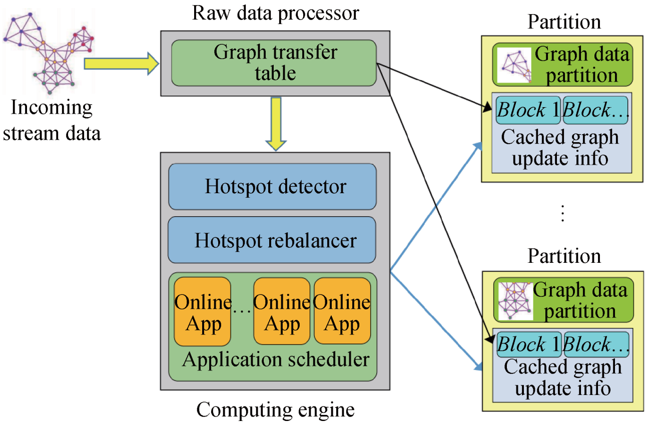


**Fig. 1** System overview

On the one side, with the *graph transfer table* and a hash-based method (the initial graph is divided into *graph data partition* with the same hash-based partition method), update operations will be packaged into blocks and then sent to the corresponding partition (blocks and partitions are one-to-one correspondence). We will further explain the detail of the graph partition method in Section 5. Blocks of update data will be held sequentially in the *cached graph update info*, which is a infrastructure to make our system support incremental graph computation. The details of it will be elaborated in Section 6.1. On the other side, all the changed vertices with the info of each changed frequency (i.e., updated times in this time interval) will be sent to the *hotspot detector*, in which a standard score method is adopted to detect the hotspots of the graph. The above method focuses on detecting nodes of the social graph which change frequently (e.g., a new edge involving the node is added or the attribute of the node is changed) in a sequential of several time intervals, because such nodes are very likely to be altered continuously in the next period of time. We will demonstrate the hotspot detection method in Section 6.3.1.

Periodically, the *hotspot rebalancer* is triggered to equalize the computation cost of all the partitions. The scheduler uses the heat rate and the number of adjacent edges of a hotspot to approximately evaluate the computation cost. When all the cost values are evaluated and collected, a global optimization algorithm based on greedy strategy is executed on them to work out the best placement of all the hotspots. The design of the algorithm is in consideration of both computing resource usage and data locality. When the best deployment of the hot nodes is obtained, there will be data communication among partitions. In this case, some of the hot nodes with their associated data will be transferred from one partition to another. After that, each hotspot with their related edges is placed in the same partition, and the workload of each partition is approximately equal. Potentially this will disorganize partitions resulting from the hash-based function. To address this challenge, the scheduler will send the vertices' IDs with its changed index to the graph transfer table in the raw data processor. Due to our hash-based graph partition method, the subgraph stored in each partition will still be updated in a fine-grained and high-speed manner with negligible overhead. We will analyze the exact process of the scheduler and the algorithm in Section 6.3.2.

The application scheduler is mainly designed to make our system support applications that simultaneously execute computing requests on the graph with different frequencies. For example, application $A_1$ is running a community detection algorithm on the graph in every two time intervals, while application $A_2$ is running an algorithm to determine whether the graph is connected in every three time intervals. Such kind of discordance will lead to chaos of graph update and graph computation. The application scheduler records the number of time intervals between the executions of an application. The sequential blocks in the cached graph update info provide necessary information to achieve the scheduling, and the block release time is specified by the scheduler. The philosophy of the scheduler will be further explained in Section 6.1.

## 5 Graph partitioning

In order to maintain a graph in a form that can be easily and efficiently updated, first we make a decision between vertex-cut partitioning method and edge-cut partitioning method. Then we introduce a hash-based graph partitioning method [34] to support fine-grained and rapid update. This method is applied to both the initial graph and the graph update data. The great improvement in graph update efficiency is the key factor that can guarantee the incremental graph processing and hotspot rebalance.

### 5.1 Vertex-cut partitioning

The traditional data-parallel computation is usually processed separately. However, graph processing bases on a computation module where each vertex or edge needs the context of its neighbors. Therefore, the data partition strategy is a key factor to achieve high-efficient processing in a graph computation system. The main idea of a sophisticated strategy is to minimize communication between partitions and overhead of graph storage, meanwhile ensure balanced computation.

The main goal of a vertex-cut method is to assign edges equally to partitions in the cluster and allow vertices to span multiple partitions. The overhead of communication and storage depends on the amount of vertices that are cut. Therefore, the method aims at assigning edges to partitions evenly meanwhile minimizing the number of partitions spanned by each vertex to ensure balanced computation and low overhead respectively. Experiments in [8] show that the vertex-cut partitioning method performs much better than edge-cut method in reducing the above overhead in the scenario of large-scale real-world graph processing.

### 5.2 Hash based partitioning

To achieve fine-grained and rapid updating, each graph update operation must be immediately located to the partition storing the corresponding graph data. Therefore, we adopt a two-step partitioning method [34] which utilizes the source vertex ID as well as the destination vertex ID of an edge to create hash values [16].

The proposed partitioning method adopts the following computing model, given an $N \times N$ sparse edge adjacency matrix $\mathbf{E}$ (source vertex ID on the horizontal axis and destination vertex ID on the vertical) and $P$ partitions organized as a $K \times L$ matrix $\mathbf{R}$, the main goal is to evenly assign the edges to these partitions while minimizing the number of partitions spanned by each vertex. The partitioned adjacency matrix maps onto a mesh of partitions. Therefore, the communication along a matrix row or column is confined to a subset of partitions, hence the total communication cost is limited. The first step of the method is to partition $\mathbf{E}$ row-wise into $K$ parts using a hash-based function whose hash value is generated with the source ID. We use a big prime number in the generator of the hash value to achieve an approximately even assignment. After finishing the first step, we obtain $K$ submatrices $\mathbf{E}_k$ for

$k = 1, 2, \ldots, K$, and each one has roughly equal number of edges. Second, each submatrix $\mathbf{E}_k$ for $1 \leqslant k \leqslant K$ is partitioned column-wise into $L$ parts using a similar hash-based function. After that, edges are equally distributed to the partitions and in the worst case, a vertex is replicated in $K + L - 1$ partitions.

## 6 Computing engine

The evolving graph reflects the interaction of users, proposing the need to study and develop models on it. The analysis of such graph requires executing graph mining algorithms on the graph in every period of time. A straightforward approach is to invoke the graph mining algorithm on the whole graph in each time period. However, with the increasing size of the graph, this is becoming infeasible. Hence, there is a need for processing the graph incrementally, to save the computation costs.

To meet the demand of fast response for applications on the dynamic graph, iGraph regards incremental graph processing as a fundamental computing pattern. We design an application scheduler to coordinate applications with different computing request frequency and control the use of system resource. In order to support incremental algorithms, we introduce a vertex-based graph computation model, which is highly fit to most of the scenarios. In addition, the workload rebalancer is invoked periodically to further improve the computing and network efficiency.

### 6.1 Application scheduler

The application scheduler is in charge of merging the graph update data into the main graph (stored in the graph data partition), triggering computing task for applications and releasing blocks in the cached graph update info. It is mainly designed to support the following scenario: multiple applications are executing simultaneously on the system and each of them sends a computing request for different time intervals. We also adopt the idea of batch processing in the scheduler to improve the efficiency of graph update and task execution.

When registered to the iGraph, application has to provide the frequency of the execution (FE), i.e., the number of time intervals between invocations of this application. For each application, the scheduler creates a data structure to record the FE, the last invocation time (LIT) and the next invocation time (NIT, i.e., add the FE to the LIT). As for graph update, the scheduler records the last time when the graph update data is merged into the main graph (LMT). For each time interval, the scheduler first detects if any applications are invoked, which is achieved by examining the NIT of every application. Nothing would happen if there are no computing requests, otherwise the scheduler will do the following things for those invoked applications:

1) The scheduler merges blocks ranging from the LMT to the current time in the cached graph update info to generate a block of union. Then the union is merged into the main graph. This union-merge pattern takes advantage of batch processing, together with our fine-grained update explained in Section 5, the update operations can be completed instantly. After finishing the update operations, the LMT is set to the current time.
2) Then the scheduler invokes applications one by one. For each of the execution, the initial set of changed vertices is generated from analyzing the union of blocks ranging from the LIT to the current time in the cached graph update info. Upon execution, the LIT and the NIT are updated by adding FE.
3) After finishing all the invocations, the scheduler traverses the LIT of every application to get the most remote one. The blocks prior to it are released because they will never be used again.

Figure 2 illustrates a simplified example of a scheduling process on two applications, $A_1$ and $A_2$. They are invoked every two and three time intervals respectively (shown with the light grey and black two-way arrows). $t_0$ is the initial point, so the LMT as well as the LIT of both are $t_0$, the NIT is $t_2$ and $t_3$ respectively. There are no invocations at $t_1$, so the scheduler do nothing. At $t_2$, an invocation of $A_1$ is triggered, the scheduler first updates the graph with a union of blocks ranging from $t_0$ to $t_2$ (Block 1 and Block 2), then the LMT is set to $t_2$. After that, $A_1$ executes with an initial changed vertices set generated from blocks between $t_0$ and $t_2$ (Block 1 and Block 2). The LIT and NIT of $A_1$ is set to $t_2$ and $t_4$ separately. Since LIT of $A_2$ is still $t_0$, no blocks will be freed at $t_2$. At $t_3$, $A_2$ is invoked, after that, the LMT is set to $t_3$ and the LIT and

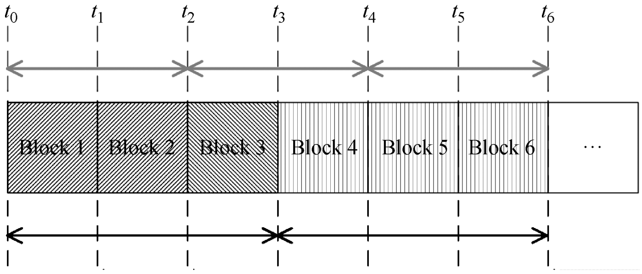


**Fig. 2** Scheduling on multiple applications

NIT of $A_2$ changes to $t_3$ and $t_6$, respectively. This time the scheduler regards $t_2$ as the most remote LIT, therefore Block 1 and Block 2 (filled with left slash) will be released. Similarly, the subsequent process of this example can be inferred. We want to note that Block 3 (filled with right slash) will be freed at $t_3$, and Blocks 4–6 (filled with vertical line) will be released at $t_6$. With the help of the scheduler, applications work harmoniously and the resource of machines can be used rationally.

### 6.2 Model of incremental processing

To support incremental graph computation, we illustrate the general incremental processing model we adopt in Fig. 3. Initially, computing request is triggered by the application scheduler as we mentioned above. Then each partition analyses the update operations in a number of (i.e., NIT minus LIT) data blocks in the cached graph update info to detect the changed vertices. After that, the user-defined update function in the application will be executed on the vertices set to generate a new value for each of them. In the next step, a user-defined filter, which compares the variation between the old value and the new value of a vertex with a user-specified threshold, is called on the vertices set to generate a subset which change significantly. If there are no vertices meeting the condition of the filter, this task execution of the application is finished. Otherwise, that all the vertices change significantly will propagate messages with state-change information to a user-specified subset of its neighbors (for some algorithms the update is only need to be propagated to an individual neighbor or neighbors of a certain type). Vertices that receive messages will be detected as changed vertices in the next iteration of the computation. All of the above operations are implemented in a batch manner of RDD, which ensures high-efficiency in-memory computing as well as fast recovery.

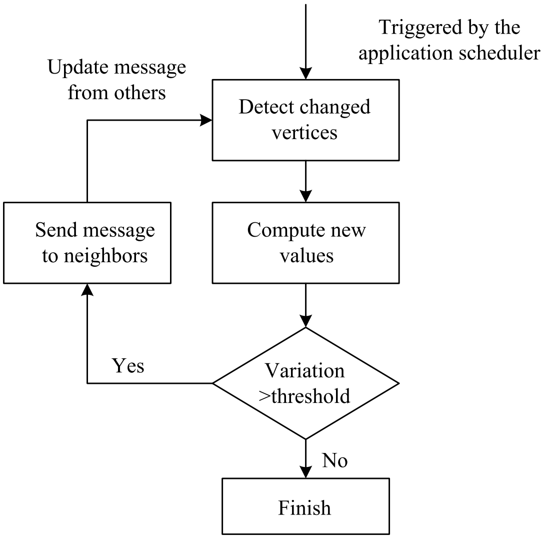

Triggered by the application scheduler
Update message from others
Detect changed vertices
Send message to neighbors
Compute new values
Yes
Variation >threshold
No
Finish

**Fig. 3** Model of incremental graph processing

### 6.3 Workload rebalancer

Social graph is usually time-sensitive, some of the graph nodes are updated much more frequently than others in a particular period of time. In the above incremental computing model, the workload is proportional to the update frequency. Therefore, the system is usually in a state of workload imbalance. Figure 4 illustrates this phenomenon in our real production environment. We use four machines as a cluster and run an application which detects events (incremental algorithm) from microblogs we crawled from Weibo in real time. The graph model we use is that each node represents a word (word segmentation is executed on each microblog as a preprocess), and a link between two nodes means that these two words appear in the same microblog. We collect the system logs of 6 pm–8 pm on January 13 ($t_1$) and January 17 ($t_2$). After statistic analysis, the states of workload are shown in Figs. 4(a) and 4(b) respectively. The black figures in the circle represent the resource usage of the 20% hottest nodes (our heat rate equation will be shown later), and the light grey figures represent the 80% rest. We can draw two conclusions from Fig. 4:

- Workload imbalance, which is caused by the uneven distribution of the hotspots, is a common state of the system. The execution time of the most loaded machine can be several times more than the least one, therefore some stragglers appear in this situation.
- The 20% of the hottest data engages more than 80% of the computation resource. It means that we only need to transfer a small amount of data (i.e., the hotspot with associated data) among partitions to achieve the workload rebalance.

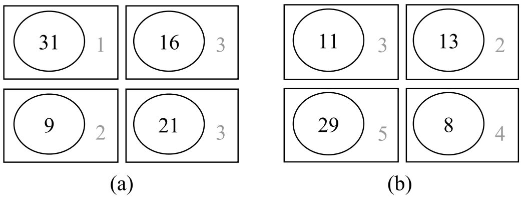


**Fig. 4** Workload imbalance. (a) State of workload in $t_1$; (b) state of workload in $t_2$

#### 6.3.1 Hotspot detector

Taking the efficiency and accuracy of hotspot detection into consideration, the hotspot detector and the rebalancer are triggered every quite long period of time (e.g., every 30 minutes). The changed vertices information blocks received from the row data processor during this time period will be merged into one block. The block contains a table of vertex ID as well as the sum of the update times from these blocks in this period ($UT$). Then we adopt a standard score method to detect the hotspots of the graph. We define $HR(i, t)$ as the heat rate of node $i$ at time $t$, and $UT(i, t)$ as the number of updated times of node $i$ from $t - 1$ to $t$. Then $HR(i, t + 1)$ is computed as follows:

$$HR(i, t + 1) = \lambda \cdot HR(i, t) + UT(i, t + 1), \tag{1}$$

where $\lambda < 1$ is a decay factor of heat rate. The standard score of node $i$ at time $t$ can be computed as follows:

$$score = \frac{HR(i, t) - \mu(i, t)}{\sigma(i, t)}, \tag{2}$$

where $\mu(i, t)$ and $\sigma(i, t)$ are the mean and the standard deviation of $HR$, respectively. If $score$ is larger than a user-specified $T$, the node will be flagged as a hotspot. If it is smaller than a lower bound $B$, the node will be regarded as cold data and is not likely to be updated in a future period of time.

The hotspot detector uses a map to store the ID of a vertex (key) with a tuple containing the latest $HR$, number ($N$), sum ($Sum$) and the square sum ($SSum$) of its $HR$ series (value). Substituting these three values to Eq. (2), we can get:

$$score = \frac{HR(i, t) - \frac{Sum}{N}}{\sqrt{\frac{SSum}{N} - (\frac{Sum}{N})^2}}. \tag{3}$$

While processing data received from the raw data processor at time $t + 1$, it first checks whether the vertex ID exists in the map. If the ID is not found, it will be inserted into the map with an initial tuple ($UT$, 1, $UT$, $UT^2$) ($UT$ is obtained in the receiving data). Otherwise, the detector first computes new $HR$, $N$, $Sum$ and $SSum$ ($HR(i, t + 1)$ with Eq. (1), new $N = N + 1$, new $Sum = Sum + HR(i, t + 1)$ and new $SSum = SSum + HR(i, t + 1)^2$). After that, $score$ can be obtained with Eq. (3). If $score$ is larger than $T$, the vertex ID (key) with its $HR$ (value) will be added into $Map_{hotspot}$ and then sent to the rebalancer for further scheduling. The tuple ($HR$, $N$, $Sum$, $SSum$) will be updated to the new one. Else if $score$ is less than $B$, it will also be sent to the rebalancer in an $Array_{remove}$ and then removed from the map with its tuple to reduce resource usage. Else, the detector will only update the tuple of the vertex.

### 6.3.2 Hotspot rebalancer

The rebalancer uses the heat rate ($HR$) and the number of adjacent edges ($AEdges$) of a hotspot to evaluate the approximate computation cost. The cost is computed as follows:

$$cost = HR \cdot AEdges^{\alpha}, \tag{4}$$

where $\alpha$ represents that the change of this vertex can be spread to more than its neighbors (e.g., neighbors of its neighbors or even far more). Empirically, we set the value to 1.5 for our event detection application. What we want to note is that "edge" is the atomic computing unit in MapReduce-like graph processing system, and each computation will finally be converted to iterations of traverse of corresponding vertices' associated edges. After receiving $Map_{hotspot}$ from the detector, the scheduler will evaluate and then record every vertex's ID with computation cost in $Map_{cost}$. Then a global optimization algorithm based on greedy strategy is executed on them to work out the best placement of all the hotspots. The design of this method considers both computation cost and data locality. In our scheduler, we do not take into account the computation cost of the rest cold data in each partition, because their influence is negligible from our previous analysis. After working out the best deployment, for each hotspot, the associated edges will be transferred to the target partition. Such aggregation guarantees our model of computation cost evaluation and reduces network flow, because there is not any data communication of some hotspots from different partitions.

Algorithm 1 is used for the initial hotspot distribution.

**Algorithm 1** Initial hotspot distribution

```
Input: $Map_{hotspot}$, number of partitions $n$
Sort the vertices in $Map_{hotspot}$ with the computation cost in descending order.
Create $Map_{result}$ to record the hotspots' vertex ID ($vid$) with their corresponding partition ID ($pid$) and create $Array_{cost}$ initialized with $n$ zeros to record the total computation cost of assigned hotspots in each partition.
for all ($vid$, $cost$) $\in Map_{cost}$ do
    Traverse the $Array_{cost}$ to find the partition with lowest cost ($p_{lowest}$).
    Add ($vid$, $p_{lowest}$) to $Map_{result}$.
    $Array_{cost}[p_{lowest}]$ += $cost$.
end for
Output: $Map_{result}$
```

Figure 5 illustrates the process of the initial distribution algorithm. Sorted elements are assigned to the partition with the lowest cost one by one in descending order. When we get the $Map_{result}$, the data communication between partitions will be started. The $Map_{result}$ will be sent to all the partitions, then each of the partition traverses its data to determine the involved edges (one of the vertices of an edge is a hotspot). These edges will be distributed to $n-1$ blocks, each of them is corresponding to their target partition. As a result, all the adjacent edges of a hotspot will be copied to one specified partition. The data in the original partition will be marked as unavailable and will not participate in computation until it becomes active again. Considering computing efficiency and data locality, it can greatly improve the efficiency and reduce the network traffic in the shuffle phase of future computation. For subsequent rebalance, we design Algorithm 2, which mainly considers the data localization.

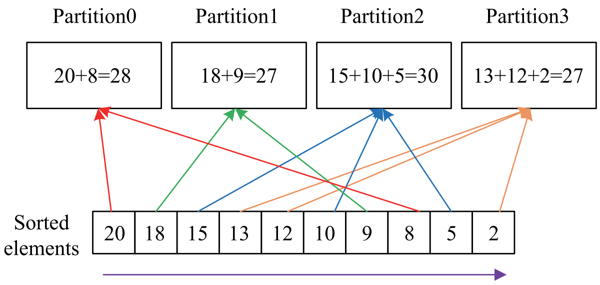
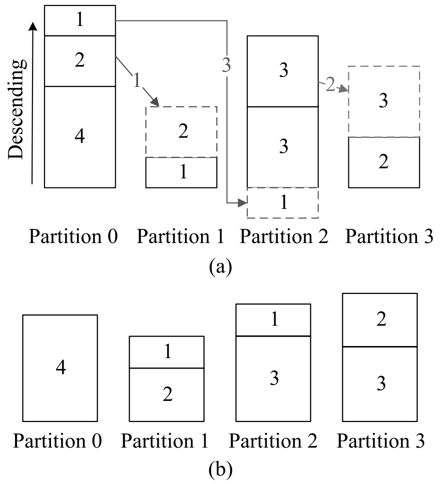


**Fig. 5** Initial hotspot distribution

**Algorithm 2** Subsequent hotspot distribution

```
Input: $Map_{hotspot}$, $Map_{result}$ of last time
1: Create $Map'_{result}$ and $Array_{cost}$ initialized with $n$ zeros.
2: for all $(vid, cost) \in Map_{cost}$ do
3:     if $vid \in Map_{result}$ then
4:         Get $(vid, pid)$ from $Map_{result}$.
5:         Add $(vid, pid)$ to $Map'_{result}$.
6:         $Array_{cost}[pid]$ += $cost$.
7:     else
8:         Define $p_{lowest}$ as the partition with lowest total cost.
9:         Add $(vid, p_{lowest})$ to $Map'_{result}$.
10:        $Array_{cost}[p_{lowest}]$ += $cost$.
11:    end if
12: end for
13: Sort vertices with computation cost in descending order.
14: while true do
15:     Find $p_{lowest}$ and $p_{highest}$.
16:     Define $k$ as the least number satisfying that the sum of cost of most
        $k$ vertices in $p_{highest}$ is higher than $p_{lowest}$'s total cost.
17:     if $k$ == number of vertices in $p_{highest}$ then
18:         break
19:     end if
20:     Define $vid_{k+1}$ as the most $k+1$ cost vertex in $p_{highest}$ and $cost_{k+1}$ as
        its cost.
21:     Update $Map'_{result}$ to $(vid_{k+1}, p_{lowest})$.
22:     $Array_{cost}[p_{highest}]$ -= $cost_{k+1}$.
23:     $Array_{cost}[p_{lowest}]$ += $cost_{k+1}$.
24:     Sort vertices in $p_{lowest}$.
25: end while
Output: $Map'_{result}$
```

Since Steps 2–12 of Algorithm 2 are easy to understand, we use Fig. 6 to illustrate Steps 13–25. In Fig. 6(a), rectangles with solid border are the initial state, vertices of each partition are sorted according to their cost. For the first time, partition 0 has the highest total cost and partition 1 has the lowest. In Step 16 we get $k = 1$, because the most costly vertex (4) in partition 0 is higher than the total cost of partition 1 (1). Along with $k < 3$ (number of vertices in partition 0), the second most costly vertex in partition 0 will be transferred to partition 1. Then vertices in partition 1 will be resorted. Similarly, the vertex of 3 and 1 cost will be transferred in later process. The result is shown in Fig. 6(b).



(a)

(b)

**Fig. 6** Subsequent hotspot distribution. (a) Process of the distribution; (b) result of the distribution

This greedy-based method ensure an approximate optimal distribution with minimal data transfers. The $Map'_{result}$ will direct a new round of data communication, the same as we mentioned above. One more thing we want to note is that all the vertices in $Array_{remove}$ with their adjacent edges will be removed from the existing partition and restored to their corresponding original partition. All the edges involving vertices in $Array_{remove}$ will be distributed to $n$ blocks with the hash-based graph partitioning method introduced above. Then each of the block will be merged into its corresponding partition to update the involved data marked as unavailable in the past as well as to insert new data, after that the unavailable data will be set as active again.

Data communication between partitions will lead to chaos of partition process (edges may not stay in their specified partition anymore), therefore we create the graph transfer table in the raw data processor to record every transferred vertex ID as well as its target partition ID. After generating the placement of the hotspots, the rebalancer sends the $Map_{result}$ to the raw data processor to recreate the table. While distributing one of the graph update operations, the processor first checks if the vertices involved exist in the table. If found, the operation will be distributed to the corresponding partition recorded in the table, else it will be distributed with the hash-based function. If both of the vertices of an edge within an operation are in the table with different moved target, the operation will be sent to both of the corresponding partitions.

## 7 Programming model

There are two main computation models for graph parallel processing, the bulk synchronous parallel (BSP) [2,18] model and the gather-apply-scatter (GAS) [8] model. iGraph supports the above two models in order to cope with various graph mining algorithms that need for different kinds of inter-vertex communications. Generally, in the BSP model vertices push updates to a specified set of neighboring vertices (for some algorithms the update is only need to be propagated to an individual neighbor or neighbors of a certain type), while in the GAS model vertices proactively pull data from a number of stated neighbors. To make the incremental algorithms execute efficiently in the distributed environment, iGraph supports the following aspects in the two models at this specific application scenario:

- Each vertex can send an update of variation to some of its neighbors, i.e., each vertex only needs to send the difference between its current and previous ones.
- A user-specified function is used to determine whether the change of a vertex is significant. If true, the change will be notified to a specified vertices of its neighbors, and then the computation propagates in the graph.

iGraph is designed with flexible and humanized user interfaces, where applications can customize functions and properties to achieve highest integrating degree with our system. The user interfaces will be shown later and here we want to introduce several functions instantiated and customized by our system automatically. 1) The raw data processor always runs a function that parses the incoming data in a time interval to an RDD. After analyzing, the RDD is further transformed to a new one consisting of a set of graph update operations. This function is the initial point of the system. 2) There is a function running on the application scheduler which is always receiving the message from the raw data processor to record time and then trigger the following scheduling. 3) A similar function runs on the hotspot rebalancer, which is in charge of time management and task triggering.

### 7.1 User interface

There are a set of functions as well as a number of properties for applications to define in order to implement a graph processing algorithm. Our programming abstraction inherits some of the GraphX operators, and further encapsulates and extends them through generating a list of specialized user interfaces. From the perspective of the total design and the system modules, there are several other important attributes and functions of the total system shown as follows.

```
// Attributes of the overall system

// The initial graph, VD and ED are types of
    vertex and edge attribute
val graph:  Graph[VD, ED]

// Update data is processed every time interval
val timeInterval:  Int

// The number of time intervals between
    invocations of hotspot rebalance
val rebalanceFrequency:  Int

// Functions of the overall system

// Register an app to the Application Scheduler.
    The scheduler records its properties and
    return an AppId
def registerApp:  (App, Type) => AppId

// Unregister an application with its AppId and
    release related resources
def unregisterApp:  (AppId) => Boolean
```

From the point of view of an application, the attributes and the common functions of it are listed below.

```
// Attributes for of application

// The number of time intervals between
    invocations of this app, it will be sent to
    the Application Scheduler
val computingFrequency:  Int

// Common functions of an application

// Compute new vertex value with the original
    value and the receiving inbound messages of
    type A
def computeNewVD: (VD, A) => VD

// Merge two incoming messages into one message.
```

```
   It should be commutative and associative.
def combineMessages:  (A, A) => A

// Detect whether the change between a vertex's
   old and new value exceed the threshold. If
   true, it will send message.
def judgeChange:  (VD, VD) => Boolean
```

There are disparate necessary functions for the BSP model and the GAS model because their actuating logics are different. For an application that adopts the BSP model, the specific function is displayed as follows.

```
// Applied to out edges of vertices that receive
   messages, and then returns messages of type A
   to their neighbors
def sendMsg:  Triplet[VD, ED] => Iterator[(VID,
A)]
```

For an application that uses the GAS model, the involved function is summarized as follows.

```
// Execute on each edge triplet adjacent to a
   vertex. The returning accumulator is then
   merged using the combineMessages function
def gatherMsg:  (VID, EdgeTriplet[VD, ED]) => A
```

### 7.2 Example of incremental PageRank

We have implemented an application of incremental PageRank on iGraph. We describe it in this section.

Over the last decade, PageRank [35] has been widely used as a very effective measure of reputation for both web graphs and social networks. The key idea is that the rank of a page depends upon the ranks of the pages pointing to it, thus a page has high rank if it is pointed to by many highly ranked pages. The rank of a page $m$ can be written as follows:

$$\mathrm{PageRank}(m) = p + (1 - p) \cdot \sum_{(n,m)\in G} \frac{\mathrm{PageRank}(n)}{\mathrm{OutDeg}(n)}, \tag{5}$$

where $p$ is the random reset probability and OutDeg($n$) is the number of hyperlinks on page $n$.

The incremental PageRank we implement will only propagate the delta of the vertices' PageRank in iterative stage, if the change of the old PageRank and the new one is more than the threshold that we have set. For example, we set the threshold to 0.02 and if page A has a PageRank of 1 000 and in the later stage it has a new PageRank of 1 010 or 990, it will not propagate the delta to its neighbors and we will only propagate the message if and only if it changes at least to 1 020 or 980. We show an simplified implementation of the incremental PageRank with the BSP model as follows, which only contains the definition of necessary functions.

```
// Compute the new PageRank of a vertex with the
   old value and the sum of delta values from its
   neighbors
def computeNewVD (oldV, deltaSum) = {
  oldV + (1.0 - p) * deltaSum
}

// Combine delta values from neighbors
def combineMessages (message1, message2) = {
  message1 + message2
}

// Judge if the change of the old PageRank and the
   new one will be propagated
def judgeChange (oldV, newV) = {
  abs((newV - oldV) / oldV) >= threshold
}

// Send the difference of the new and the previous
   PageRank to its neighbors
def sendMessages (edge) = {
  Iterator((edge.dstId, delta * edgeWeight))
}
```

We want to note that we can adjust the accuracy of the result and the computation time by adjusting the `threshold` in the `judgeChange` function.

## 8 Experimental evaluation

We have implemented iGraph on Apache Spark version 1.3.0 and GraphX. Due to the characteristics of in-memory storage and lineage based fault recovery of RDD, we can set the time interval of the batch computation to a quite small value (e.g., 10 seconds). Therefore, the real-time feature of our system will not be destroyed. We have evaluated iGraph on three different kinds of tasks to test performance improvement of graph update, incremental algorithm and the hotspot rebalancer. We benchmark iGraph against the original GraphX. We want to note that we set the graph partition number as 16 (4×4 in the hash-based partitioning method) and 32 (4×8 in the hash-based partitioning method), and the results of all experiments are averages of ten runs.

All experiments are conducted on a cluster with five machines, each of them is equipped with an Intel Xeon E5-2650 CPU (eight-core, 2.00GHz) and 32GB memory and all the machines are connected by a Gigabit switch. One of them is master and the other four are workers, and each worker is allocated with 16GB memory. All the machines ran 64-bit version of Debian squeeze (Linux kernel version 2.6.32) with Java JRE version 1.7.0.

### 8.1 Graph update

The efficiency of graph update has great influence on the incremental graph processing and the hotspot rebalancer. Therefore, it should support the scenario that the evolving graph has high update throughput. We set the system time interval as 60 seconds and do this experiment on three different real-world graph datasets, their main characteristics are summarized as follows:

- Google web graph This graph consists of about 875K nodes and nearly 5.1M edges. Nodes represent web pages, and directed edges represent hyperlinks between them.
- LiveJournal friendship social network This graph models friendship of users of LiveJournal, a free on-line community with almost ten million members. It consists of about 4.8M nodes and nearly 68.9M edges. Nodes represent users of LiveJournal, and directed edges represent friendship between them.
- Friendster friendship social network This graph models friendship of users of Friendster, an on-line gaming network. It consists of about 65.6M nodes and nearly 1.8B edges. Nodes represent users of Friendster, and directed edges represent friendship between them.

Initially, the graph is loaded and partitioned with the hash-based method. To simulate the real scenes, we produce a streaming data generator which can randomly produce and send a specified number of graph update messages in a time interval to our system. In addition, we can customize the ratio of different kinds of update message, including update of vertex attribute, update of edge attribute and insertion of new edge. In our experiment, we set the ratio of the above three kinds of message to be 25%, 25% and 50% respectively. The generator sends 10 000 and 100 000 messages to test the performance of graph update in different scenarios.

Figure 7 shows the efficiency improvement of iGraph. It averagely reduces 73%, 86% and 93% of the graph update time on the above three datasets respectively. To update a graph, the original GraphX first merges the update message into RDDs of edges and vertices of the main graph respectively, and then recreates the routing table, which is logically a map from a vertex ID to the set of edge partitions that contain adjacent edges, to obtain the new graph. The recreation of routing table is time consuming, and the computation cost is proportional to the amount of edges and vertices. This kind of graph update is based on a global manner, whose time cost is not greatly influenced by the number of update messages. As shown in Fig. 7(b), original GraphX can not cope with graph update in the LiveJournal dataset because the update time is longer than the time interval, leading to *data accumulation*. In iGraph, we design index storage of edges and vertices in partitions of edge RDD and vertex RDD. With the help of the hash-based partitioning method, it can substantially reduce the time cost of graph update, i.e., update messages first can be rapidly sent to the corresponding partition with the partitioning method and then random-accessly applied to the target data. Moreover, the routing table can be updated fine-grainedly. From the above, iGraph achieves the

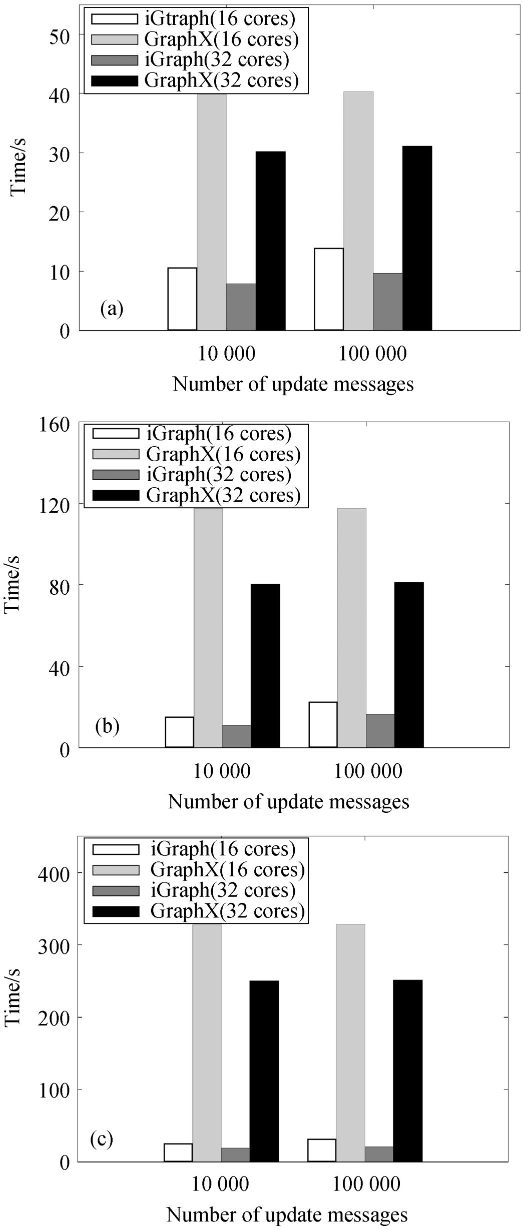


**Fig. 7** Graph update time. (a) Google web graph; (b) LiveJournal friendship social network; (c) Friendster friendship social network

rapid and fine-grained update. In addition, the time cost of this kind of update is not affected much by the amount of edges and vertices of the main graph, however it will increase with the growing number of update messages. We set the number of partitions to 16 and 32 respectively to test the scale-out effect of our system. Results show that the graph update time in 32 cores can be reduced at least 30% compared with that in 16 cores.

### 8.2 Incremental processing

We use the incremental PageRank, introduced in Section 2, to test the performance improvement and network flow reduction of the computing engine which supports incremental algorithm over the original GraphX which can only execute a non-incremental version. As for the parameters in the algorithm, we set `p` as 0.15 and `threshold` as 0.001 to ensure a sufficiently accurate result. We execute the algorithm after the graph is updated with 100 000 messages in the above experiment and record the execution time and network flow in the cluster.

Figure 8 demonstrates the improvement of iGraph's computing engine which supports incremental algorithms. Since the incremental algorithms reuse the previous results of the last iteration, the computing time and network flow can be greatly reduced. The larger the graph size is, the more time will be used to recompute PageRank in the non-incremental version. Figure 8(a) shows the time benefit of incremental computation. With the increasing size of the three datasets, the incremental PageRank averagely reduces 78%, 86% and 94% of the computing time respectively in the scenario of 16 cores, as well as 77%, 84% and 94% in that of 32 cores. Compared with the scenario of 16 cores, the computing time in 32 cores is reduced at least 41%.

Compared to the non-incremental version, the input data of the incremental one is smaller and the time of iteration is less. Therefore, the network performance can be improved. Similarly, the improvement will be grater as the graph size grows. Figure 8(b) shows that the network flow decreases 80%, 89% and 91% for the three datasets respectively in the scenario of 16 cores, as well as 80%, 88% and 90% in that of 32 cores.

### 8.3 Effect of workload rebalancer

We do this experiment on a dataset of one day's microblogs we crawled from Weibo. The number of total microblogs we crawl in one day is about 10M. In this experiment, the application we use detects events from continuously incoming microblogs. First, each microblog is segmented to words with a dictionary consisting of about 200K words. Then each microblog is transformed to a small graph, where each vertex represents a word and a link between two nodes, meaning that these two words appear in the same microblog. The next step is to merge these small graphs to the main graph constructed by all the words in the dictionary. Finally, the algorithm is executed, regarding these changed data as input. We run this application with the workload rebalancer on and off respectively to test the reduction of graph processing time and network flow. We set the system time interval as 10 minutes and run this event detection application every time interval. The workload rebalancer is invoked every 3 time intervals (i.e., 30 minutes) and the decay factor $\lambda$ is set to 0.5.

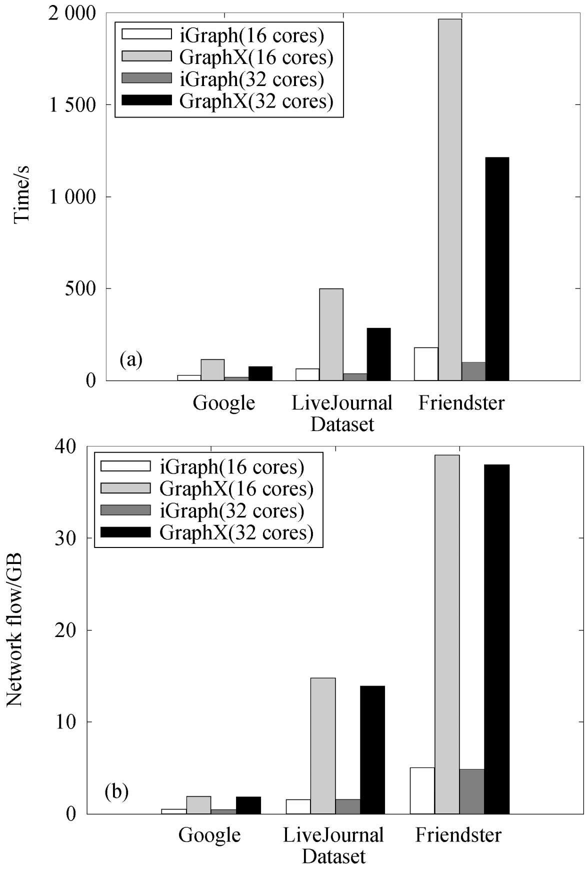


**Fig. 8** Resource cost in graph processing. (a) Time cost in computing PageRank; (b) network flow cost in computing PageRank

Figure 9(a) shows the improvement of the processing time cost. The $x$-axis represents the 24 hours of a day, and the $y$-axis represents the execution time of the event detection algorithm in each time interval. Since the workload is approximately equally distributed to each partition, the completion time of each task will be reduced. Averagely, with the help of the workload rebalancer, the time cost of jobs is decreased by 11%. We want to note that the figure for iGraph with rebalance includes the flow of data communication while rebalancing. Therefore, at the time point when the rebalancer is invoked (i.e., 00:30, 01:00, 01:30, etc.), the figure in the green polyline is sometimes higher than the red polyline. However, because the logic of the rebalance algorithms is relatively straightforward and hotspots only account for a small portion of vertices, obtaining the best placement of the hotspots takes very little time. As a result, the figure for iGraph with rebalance tend to be less than that without rebalance.

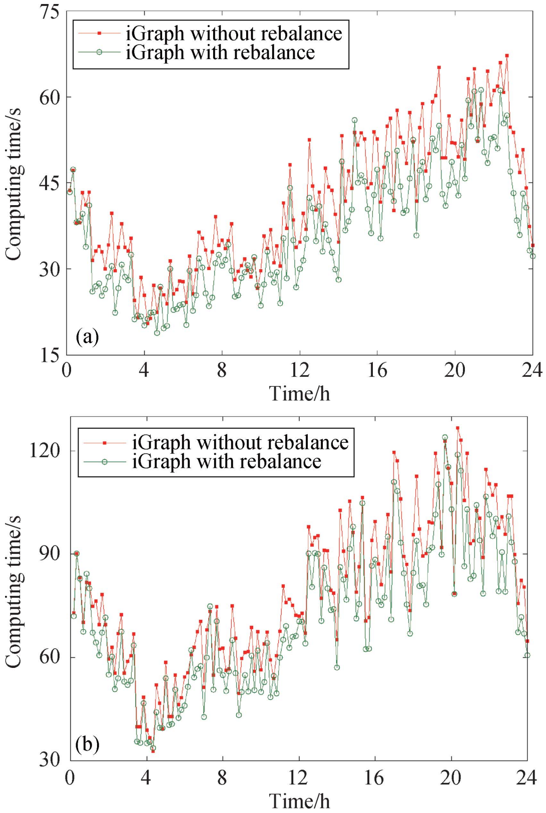


**Fig. 9** Effect of workload rebalancer. (a) Improvement of graph computing; (b) reduction of network flow

Figure 9(b) demonstrates the increase of network performance. Because for each hotspot, we aggregate its associated edges to the same partition, the data communication in the computing phase will be reduced. Therefore, we can see the decrease of network flow in the cluster. Due to the workload rebalancer, the network flow in the cluster is reduced by 12% averagely. Similarly, the figure for iGraph with rebalance includes the flow of data communication while rebalancing. Therefore, the figure in the green polyline is sometimes higher than the red polyline. However, as we take data locality of the hotspots into consideration, in most cases, the figure for iGraph with rebalance is less than that without rebalance in certain degree.

## 9 Conclusions

Processing dynamic graph in a real-time manner is a new demand created from many applications on social networks. This paper introduces iGraph: a distributed graph processing system to timely process dynamic graph with its continuous updates. To achieve fine-grained and high-efficient update, we adopt a hash-based method in partitioning the initial graph as well as the update data. We design a computing engine that supports incremental graph computation. A workload rebalancer is designed to further improve the efficiency of the engine, which detects hotspots and rebalances them with greedy algorithms, using their valuated workload. Our experimental results with real datasets demonstrate that iGraph is up to 10× faster than existing approaches both in the update time and computation time of the graph.

There are several topics which need to be further investigated. First, a variety of graph algorithms should be developed based on the executing logic of iGraph's incremental computing engine. Second, the interfaces of iGraph will be extended, e.g., the hash function adopted in iGraph's partitioning process should be defined by users based on the practical applicants.

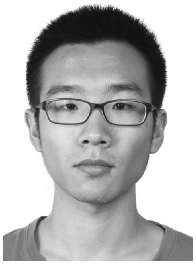
**Acknowledgements** This work was supported by the National Natural Science Foundation of China Program (Grant No. 61472022), China National Program on key Basic Research Project (2014CB340300) and China MOST project (2012BAH46B04).

## References

1. Shao Y, Cui B, Ma L. PAGE: a partition aware engine for parallel graph computation. IEEE Transactions on Knowledge and Data Engineering, 2015, 27(2): 518–530
2. Malewicz G, Austern M H, Bik A J C, Dehnert J C, Horn I, Leiser N, Czajkowski G. Pregel: a system for large-scale graph processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. 2010, 135–146
3. Salihoglu S, Widom J. GPS: a graph processing system. In: Proceedings of the 25th International Conference on Scientific and Statistical Database Management. 2013, 22:1–22:12
4. Power R, Li J Y. Piccolo: building fast, distributed programs with partitioned tables. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation. 2010, 293–306
5. Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein J M. Graphlab: a new framework for parallel machine learning. In: Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence. 2010, 340–349
6. Pearce R A, Gokhale M, Amato N M. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In: Proceedings of the ACM/IEEE International Conference for High Performance

Computing Networking, Storage and Analysis. 2010, 1–11

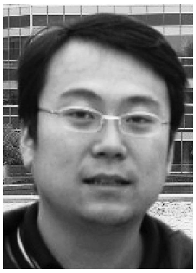
7. Kang U, Tsourakakis C E, Faloutsos C. PEGASUS: a peta-scale graph mining system. In: Proceedings of the 9th IEEE International Conference on Data Mining. 2009, 229–238
8. Gonzalez J E, Low Y, Gu H, Bickson D, Guestrin C. Powergraph: distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation. 2012, 17–30
9. Ching A, Edunov S, Kabiljo M, Logothetis D, Muthukrishnan S. One trillion edges: graph processing at Facebook-scale. Proceedings of the VLDB Endowment, 2015, 8(12): 1804–1815
10. Yan D, Cheng J, Lu Y, Ng W. Blogel: a block-centric framework for distributed computation on real-world graphs. Proceedings of the VLDB Endowment, 2014, 7(14): 1981–1992
11. Zhang Y, Liao X F, Jin H, Lin L, Lu F. An adaptive switching scheme for iterative computing in the cloud. Frontiers of Computer Science, 2014, 8(6): 872–884
12. Zheng X L, Zhong Y G, Zeng D, Wang F Y. Social influence and spread dynamics in social networks. Frontiers of Computer Science, 2012, 6(5): 611–620
13. Kumar R, Novak J, Tomkins A. Structure and evolution of online social networks. In: Philip S Y, Han J, Faloutsos C, eds. Link Mining: Models, Algorithms, and Applications. New York: Springer, 2010, 337–357
14. Yan D, Cheng J, Lu Y, Ng W. Effective techniques for message reduction and load balancing in distributed graph computation. In: Proceedings of the 24th International Conference on World Wide Web. 2015, 1307–1317
15. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauly M, Franklin M J, Shenker S, Stoica I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation. 2012, 15–28
16. Gonzalez J E, Xin R S, Dave A, Crankshaw D, Franklin M J, Stoica I. Graphx: graph processing in a distributed dataflow framework. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation. 2014, 599–613
17. Ma S, Li J, Hu C M, Lin X L, Huai J P. Big graph search: challenges and techniques. Frontiers of Computer Science, 2016, 10(3): 387–398
18. Cheatham T, Fahmy A F, Stefanescu D C, Valiant L G. Bulk synchronous parallel computing — a paradigm for transportable software. In: Proceedings of Annual Hawaii International Conference on System Sciences. 1995, 268–275
19. Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein J M. Distributed graphlab: a framework for machine learning in the cloud. Proceedings of the VLDB Endowment, 2012, 5(8): 716–727
20. Pujol J M, Erramilli V, Siganos G, Yang X, Laoutaris N, Chhabra P, Rodriguez P. The little engine(s) that could: scaling online social networks. ACM SIGCOMM Computer Communication Review, 2011, 41(4): 375–386
21. Mondal J, Deshpande A. Managing large dynamic graphs efficiently. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. 2012, 145–156
22. Yang S, Yan X, Zong B, Khan A. Towards effective partition management for large graphs. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. 2012, 517–528
23. Bu Y, Howe B, Balazinska M, Ernst M D. Haloop: efficient iterative data processing on large clusters. Proceedings of the VLDB Endowment, 2010, 3(1): 285–296
24. Logothetis D, Olston C, Reed B, Webb K C, Yocum K. Stateful bulk processing for incremental analytics. In: Proceedings of the 1st ACM Symposium on Cloud Computing. 2010, 51–62
25. Popa L, Budiu M, Yu Y, Isard M. Dryadinc: reusing work in large-scale computations. In: Proceedings of Workshop on Hot Topics in Cloud Computing. 2009
26. Gunda P K, Ravindranath L, Thekkath C A, Yu Y, Zhuang L. Nectar: automatic management of data and computation in datacenters. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation. 2010, 75–88
27. Bhatotia P, Wieder A, Rodrigues R, Acar U A, Pasquin R. Incoop: MapReduce for incremental computations. In: Proceedings of the 2nd ACM Symposium on Cloud Computing. 2011
28. Peng D, Dabek F. Large-scale incremental processing using distributed transactions and notifications. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation. 2010, 251–264
29. Murray D G, McSherry F, Isaacs R, Isard M, Barham P, Abadi M. Naiad: a timely dataflow system. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles. 2013, 439–455
30. Roy A, Mihailovic I, Zwaenepoel W. X-stream: edge-centric graph processing using streaming partitions. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles. 2013, 472–488
31. Kyrola A, Blelloch G E, Guestrin C. Graphchi: large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation. 2012, 31–46
32. Cheng R, Hong J, Kyrola A, Miao Y, Weng X, Wu M, Yang F, Zhou L, Zhao F, Chen E. Kineograph: taking the pulse of a fast-changing and connected world. In: Proceedings of the 7th ACM European Conference on Computer Systems. 2012, 85–98
33. Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I. Discretized streams: fault-tolerant streaming computation at scale. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles. 2013, 423–438
34. Çatalyürek Ü V, Aykanat C, Uçar B. On two-dimensional sparse matrix partitioning: models, methods, and a recipe. SIAM Journal on Scientific Computing, 2010, 32(2): 656–683
35. Page L, Brin S, Motwani R, Winograd T. The PageRank citation ranking: bringing order to the web. Technical Report. 1999



Wuyang Ju received the BE degree from the School of Mathematics and Systems Science at Beihang University, China in 2013. He is currently a master candidate in the Department of Computer Science, Beihang University. His research interests include distributed storage system and graph processing.

Jianxin Li is an associate professor at the School of Computer Science and Engineering, Beihang University, China. He received his PhD degree from Beihang University in 2008. He was a visiting scholar in machine learning department of Carnegie Mellon University, USA in 2015, and a visiting researcher of MSRA in 2011. His current research interests include data analysis and processing, distributed systems, and system virtualization.

Weiren Yu received the BE degree from the School of Advanced Engineering at Beihang University, China in 2011. He is currently a PhD candidate in the Department of Computer Science, Beihang University since 2011. His research interests include distributed machine learning systems, scalable graphical models and graph mining models for emerging event detection on social media.



Richong Zhang received his BS degree and MASc degree from Jilin University, China in 2001 and 2004, respectively. In 2006, he received his MS degree from Dalhousie University, Canada. He received his PhD form the School of Information Technology and Engineering, University of Ottawa, Canada in 2011. He is currently an associate professor in the School of Computer Science and Engineering, Beihang University, China. His research interests include artificial intelligence and data mining.