室蘭工業大学
学術資源アーカイブ
Muroran Institute of Technology Academic Resources Archive

QuickSquad: A new single-machine graph computing framework for detecting fake accounts in large-scale social networks

メタデータ	言語: English 出版者: SPRINGER 公開日: 2020-11-18 キーワード (Ja): キーワード (En): Security of online social networks, Fake accounts, Sybil detection, Graph computing, Distributed system 作成者: JIANG, Xinyang, LI, Qiang, MA, Zhen, 董, 冕雄, WU, Jun, GUO, Dong メールアドレス: 所属:
URL	http://hdl.handle.net/10258/00010315

QuickSquad: A New Single-machine Graph Computing Framework for Detecting Fake Accounts in Large-scale Social Networks

Xinyang Jiang[1] · Qiang Li[1,4] ·
Zhen Ma[1] · Mianxiong Dong[2] ·
Jun Wu[3,4] · Dong Guo[1,4]

Received: date / Accepted: date

Abstract Graph-based approaches for fake account detection is one of the important means to fight against fake accounts' attacks on social networks. With the growth of the scale of social networks, more and more researchers begin to use the graph computing framework to boost their detection algorithms.

We make detailed analyses of social networks' graph data and state-of-the-art graph computing frameworks, and find that some techniques of the current graph computing systems are overgeneralized and suboptimal, which means they only focus on how to design a graph processing framework on general graphs but miss the optimization of social networks graphs. So, in this paper we propose QuickSquad, a graph computing system on a single server which is specific to the optimization of social networks graph structures. QuickSquad uses the method of "divide and rule" instead of overgeneralization. First, we divide the graph structure data into the heavy set and the light set according to the out-degree of vertices. Then, we 1) store them with different formats, 2) process them with edge-based updating and vertex-based updating appropriately in a two-phase processing model, 3) apply two selective scheduler strategies of different level, i.e. vertex-level and file-level, and 4) provide four cache priorities when the memory is not enough to cache all data. Finally, we implement two detection methods, dSybilRank and dCOLOR, on our system, and the experiments demonstrate that our system can increase the performance up to 5.91X (from 1.14X) compared with the performance of the current graph computing systems, like GridGraph.

1.College of Computer Science and Technology, Jilin University, Changchun, China;
2.Department of Information and Electronic Engineering, Muroran Institute of Technology, Japan;
3.School of Cyber Security, Shanghai Jiao Tong University, Shanghai, China;
4.Symbol Computation and Knowledge Engineer of Ministry of Education, Jilin University, Changchun Jilin, China.

Corresponding Author: Dong Guo(guodong@jlu.edu.cn).

Keywords: security of online social networks, fake accounts, sybil detection, graph computing, distributed system.

1 Introduction

With the development of the Internet and various mobile intelligent terminals, Online Social Network (OSN) platform develops rapidly[42][29][18][19]. According to the statistics, by January, 2017, the monthly active users of Facebook (the largest social network in the world) have reached 1.871 billion and there have been over 20 social network platforms with over 100 million monthly active users[1]. Online social networks have gradually substituted the traditional methods of social networks, such as email, to become a widespread method for making friends, working, living and entertaining. With the continuous increase of the number of social network users, enormous commercial opportunities are brought to the industry of media, advertisement, entertainment[62].

While social networks bring convenience to people's life and benefits to businessmen, they also have new and enormous potential safety hazards. To seek profit in social networks, attackers create fake accounts which do not correspond with any real users and/or embezzle compromised accounts which are overtaken by perpetrators. And, then they use these accounts to start some attacking behavior, such as sending spams or malicious URLs [6], conducting click fraud to obtain charged click of advertisements, spreading malware and even illegally obtaining users' private information [7,31], etc.

To reduce the potential safety hazard brought by attackers who use fake accounts, researchers have proposed various detection approaches [7,8,60,56,4, 38,20,55,58,15,52,61,47] in recent years, which are mainly divided into three kinds that are respectively based on users' behavior features, information content of users and the graph structure of social networks.

The first type of methods are based on behavior features. Those methods establish the behavior patterns that can distinguish different users based on the malicious attacking modes of fake accounts. Since normal users and fake users have different behavior patterns in social networks, users' behavior patterns can be used to detect fake accounts [54]. The second type of methods are based on content features [5]. Those methods try to find out the features of the users' information or interactive information between the users, such as machine learning algorithm training [26]. The above two types of methods both need a large amount of ground truth data to strengthen the detection models or need constant training of detection systems to improve the detection performance [60]. In addition, the above two kinds of methods have comparatively high false negative rates and false positive rates [2]. Compared with the above two methods, graph-based detection approaches have the advantages of good detection performance and simple feature capturing.

[1] https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/ 2017.02.20

Attackers can imitate the behavior of normal users. However, it is very difficult for them to establish good social relationship with normal users, because normal users will refuse to establish relationship with fake accounts. Such features make it difficult for attackers to imitate normal users' behavior to evade detection [54]. To sum up, graph-based approaches are widely applied.

However, with the growth of social networks, existing complicated graph-based detection approaches are hard to be scaled and applied to the detection of large-scale social networks in the real world. In addition, it is hard to use traditional big-data processing framework, such as MapReduce, to process unstructured graph data [24]. Hence, some researchers [8,4,56] try to apply the method of Pregel [44]/Giraph [22], which is a vertex-centric graph computing system to implement the detection of large-scale social networks. However, the design of existing graph computing systems is overgeneralized. Furthermore, these systems implement the optimization that is specific to a common graph, but not embrace any specific social networks. For example, GraphChi [34] and X-stream [50] adopt *edge-based updating*, while GridGraph [66] and Venus [12] adopt *vertex-based updating*. Through our analysis, the edge-based updating perform well in dealing with low-degree vertices, while the vertex-based updating performs well in dealing with high-degree vertices.

Through the observation of social networks' graph data, this paper proposes QuickSquad, a vertex-centric graph computing system on single servers, which is specific to the optimization of social networks' graph. QuickSquad also makes use of Scatter and Apply interfaces [23]. It follows the GAS(Gather, Apply, Scatter) model and can implement most of the current applications of detection on social network. QuickSquad implements the optimization specific to the power law features of social networks. QuickSquad divides the graphs into two non-intersecting sets according to the degree of vertices, which are light set and heavy set, and different strategies are applied to these two sets. First, different storage formats are applied which are light shard format and heavy shard format (§4.1.1). Second, edge-based updating and vertex-based updating are both applied in our processing models. Moreover, we divide the graph processing into two phases, that is streaming light phase(SLP) and streaming heavy phase(SHP). The vertex-based updating is only processed in SHP, while edge-based updating is divided into two parts which are executed respectively in SLP and SHP (§4.1.2). Third, different selective scheduler strategies are applied to optimize the graph system, which are selective scheduler strategies of file granularity and selective scheduler strategies of vertex granularity (§4.1.3). Finally, different priorities of cache are provided (§4.1.4).

The main contribution of our work are summarized as follows:

1. The state-of-the-art graph system on single machines is analyzed (§2.1) and some key techniques in optimizing out-of-core computing are summarized (§3.2 and §4.1.5). Based on our observation, these key techniques are overgeneralized when being applied.
2. The power-law distribution of social network graphs is taken into consideration. We have implemented QuickSquad which divides the graph into

two parts, so different strategies are applied in light set and heavy set to decrease the I/O amount of the system and to improve the system's performance (§4 and §5).

3. The existing graph-based fake account detection algorithms are analyzed and the related graph algorithms are roughly divided into two types. One is power iteration algorithms based on random walk, such as SybilRank [7]; the other is traversal algorithms based on community findings, such as COLOR/COLOR+ [63]. We put forward dSybilRank and dCOLOR algorithms, which improve the efficiency by transforming the original algorithms, SybilRank and COLOR, to vertex-centric parallel iterative graph algorithms (§5.3).
4. The performance of the above two types of detection algorithms in QuickSquad are evaluated. Experiment results show that QuickSquad performs well. For example, it takes QuickSquad 459s to process the network of 50 million vertices on a single server, showing better performance than SybilRank which requires 33 hours to process the data of 160 million vertices by using eleven m1.large clusters. Moreover, we also compare QuickSquad with existing graph computing systems, such as GridGraph, and the comparison results show that the performance of QuickSquad can be increased by 1.14 - 5.91 times in social networks' graph. (§6)

2 Related Work

2.1 Graph computing system

When distributed computation is processing structured data and flattened data, MapReduce model is widely used.

However, most iterative graph algorithms have multiple iterations, such as BFS, pagerank, label propagation and so on. It is pretty hard to implement them using MapReduce [16] Model directly, since they usually need a large number of complex operations (like map, reduce and join). Moreover, the iterative graph has the characteristic of repeatedly access and poor locality when accessing the partition of graph. Furthermore, MapReduce model needs a distributed file system to store the partition of graph data, which also makes the implementation perform poorly. In addition, graph data itself has the feature of being unstructured and presents poor locality of access and strong dependence among data, which makes MapReduce model that needs distributed file system to exchange data in iterations have poor performance. BSP(Bulk Synchronous Parallel) model proposed by Valiant (a Turing Award winner) [53], is a model that is more suitable for iterative graph processing. Most distributed graph computing systems at present such as [24,43,23,67] refer to the thought of BSP model and suggest thinking like a vertex (TLaV) [45], which is also called vertex-centric parallel iterative graph computing system.

There are many vertex-centric computing frames. They can be divided into distributed graph computing system and single server graph computing systems according to the difference of low-leveled hardware frames. While in a single server graph computing system, storage needs to be used in processing large graphs, such as disks for improving the scalability of a single server. Compared with distributed graph computing system, graph computing systems on single servers have low consumption without depending on the Network, so that it's easier to be managed and maintained. Therefore, we give priority to graph computing systems, when data sets do not exceed the single server's processing capacity. In this paper, we optimize and use graph computing systems on single servers to process the graph data of social networks. Plus, there are some paper focus on the GPU, such as [46,32,39,51]. But this paper is focus on CPU.

However, we find that most of state-of-the-art graph systems[34,50,66,65, 14,40] are overgeneralized, which means they don't get the optimal through the power law graph distribution. A social network's graph in the real world is subject to power-law distribution in the application. PowerLyra [11] system proposed by Shanghai Jiaotong University takes the characteristics of power-law distribution into consideration. PowerLyra finds that some systems [22, 43] distribute vertices evenly among machines and all the edges related to one vertex should be put in the machine in which the vertices exist. However, when partitioning the power-law distribution graphs, the above operation which only considers the locality of access will lead to the imbalance of computing and communication, since some vertices are of high degree. If edges are partitioned evenly to every machine [24,23], those vertices with low degrees (that are not supposed to be partitioned) will be partitioned, which will decrease the locality of access and lead to extra cost of communication. Therefore, PowerLyra treats the vertices of low degrees through distributing vertices evenly in order to avoid the extra cost of communication, while it treats the vertices of high degrees through distributing edges to avoid load imbalance.

Moreover, we observe that the cut technique of vertices is not only applied to different environment in an actual distributed graph computing system, but also applied to some techniques of single servers which use the out-of-core computing. Some of these techniques support dense graphs well, and some perform better on sparse graphs. Just as what is shown in §3.1, some real world graphs, especially most social networks are subject to power-law distribution. Therefore, we make reference to the thought of ”divide and rule” of PowerLyra for reference in our system and combine these technologies together. We will introduce the details in the next section.

2.2 Graph-based detection algorithm

The analysis method based on social network graphs deems a social network as an entire graph. Through analyzing the graph features, an effective detection algorithm is established. Although attackers can imitate the random behavior of normal users, it is difficult to establish a lot of good social relationships [54] with normal users and change the topology features of the whole social network. Hence, a lot of researchers use graph analysis algorithm to identify fake accounts. Some works, such as [15, 52, 54, 61, 47], is based on graphs, but their designs have low detection rate, or high algorithm complexity, which can only work in smaller social networks [59]. As a result, it is difficult to really apply them to fake account detection in large scale social networks [7].

After SybilRank [7], there are many insightful works. Using the million-user-scaled data from millions of users, integro [4] trains a classifier based on the features of accounts to separate the fake accounts from large scaled social networks through the advanced random walk algorithm to rank the accounts and then to predict the fake accounts. Transductive Sybil Ranking(TSR) [28] proposes a TSR approach capable of adjusting edge weights based on the spread of sampled trust leaks, which shows good performance in defending real attacks. SmartWalk [41], an adaptive-random-walk method, predicts the requisite length of random-walk-length through supervised learning tools. SybilRadar [48] is a Sybil detection mechanism based on graph-based structural features of OSNs to detect nodes with weak trust relationships against Sybil attacks. In contrast to SybilRank, SybilRadar assumes an OSN with weak trust and with graphs of a lot of attacking edges. Therefore, SybilRadar computes similarity values between a pair of nodes to predict the attacking edge. Moreover, to predict the community, SybilRadar uses a module optimization method called Louvain Method [3]. Lastly, to rank the suspicious nodes, each node in the OSN is assigned a degree-normalized landing probability of a modified short random walk. Hence, SybilRadar shows much better detection accuracy than other competitors. And they all focus on improving the detection performance and to some extent look down upon on detection effectiveness.

Some researchers have implemented some of the above algorithms on big-data distributed systems, including the traditional big data processing framework MapReduce. For example, SybilRank is implemented in MapReduce. SynchroTrap [8] is implemented in MapReduce and Giraph [22]. integro [4] is implemented in MapReduce and Pregel [44]. VoteTrust [56] is implemented in Giraph. But they simply implement those algorithms in open source systems without considering the optimization of the system.

3 Background and Motivation

This section discusses the motivation of QuickSquad from the perspective of data, algorithm and system, that is, features of the social networks' graph data, graph-based fake account detection algorithms, and graph computing frameworks.

3.1 Features of graph structure on social networks

Graph is a very important kind of data structure and it can be used to represent the complicated relationship among entries of the same kind. Graph data is

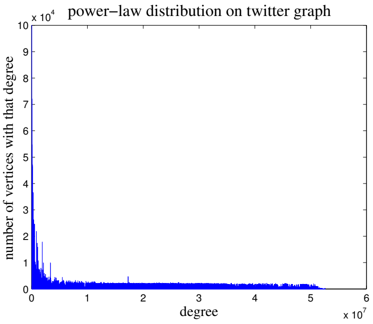

Fig. 1 Power-law distribution on Twitter

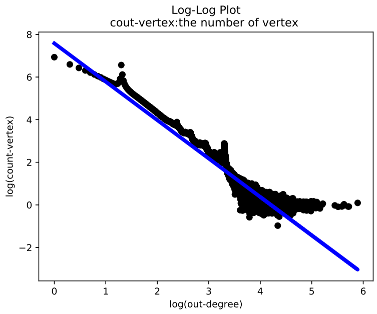

Fig. 2 The log-log plot of Twitter

unstructured so that its access is uncertain and its memory access suffers poor locality [24]; Moreover, graphs of social networks are diversified, some are based on unidirectional following, such as Twitter and Sina Weibo which are represented by directed graphs [54], and the others are based on bi-directional social relationship, such as Facebook and Friendster [57] which are expressed by undirected graphs [7,56]. In addition, these graphs are under continuous evolution. For example, graphs will be denser, for the growth of edges in the graph is superlinear compared with the growth of vertices, and the average distance between vertices will decrease continuously with graph evolution [36]. Graph will be subject to power-law distribution [21]. For example, a Twitter graph [33], shown in fig.1, plots the number of the vertices with a specific out-degree in twitter graph and those vertices are arranged from small to large according to their out-degree. The neighbors of most vertices account for only a small part of the graph, while a small number of vertices have many neighbors. Fig. 2 is the log-log plot of Twitter graph. According to the relationship of following among twitter users, the relationship between the following (out-degree) number of users and the number of users related with the number of followings (count-vertex) is calculated. Then logarithms of both out-degree and count-vertex are calculated, and a line is formed after fitting the logarithms. Graphs of some social networks will also present the small-world effect, transitivity, clustering, community structure [17] or other features [49]. These diversified graph data features bring more opportunities and challenges for the design of graph-based social network detection algorithms and also bring challenges for the design of big-data graph processing systems [24,37, 11,65]. For example, graph partition irrespective of power-law distribution features of the graph will cause load imbalance [11] and the partition being irrespective of graph's community structure will cause excessive network cost [37], etc. with the update value being k.

3.2 Optimization techniques for graph computing

A graph structure data can be considered as a kind of graph $G = (V, E)$, where V represents a set of users and E represents the relationship between the two users. In directed graphs, an edge $e(i, j)$ suggests that the user v_i follows the user v_j, while in undirected graphs, $e(i, j)$ and $e(j, i)$ are used simultaneously to show that v_i and v_j are bi-directional friends. There are $n = |V|$ vertices and $m = |E|$ edges, where, the edge number in an undirected graphs m is twice that in the actual graphs (as it is represented by two directed edges). The vertex-centric computing model is composed of a series of iterations which are called supersteps S. In a superstep, each vertex $v \in V$ will execute a self-defined *compute* function $F(v)$ and then $F(v)$ will be executed independently and parallelly. The *compute* function can be summarized into three phases: *gather*, *apply* and *scatter* stage, also called GAS model [24]. In the gather stage, current vertex v will collect the data updated in the previous superstep $S - 1$ from adjacent vertices and itself; in the apply stage, the vertex value will be computed and updated; in the last stage of scatter, the vertex will update the data on the edge and send them to the adjacent vertices for the next superstep $S + 1$.

At present, there are many general graph computing systems and there is also some work to implement the optimization according to the characteristics of graphs, such as PowerLyra [11], which optimizes distributed graph computing system PowerGraph [23] through the characteristics of the power-law distribution, using different computing and partitioning strategies through the vertices of different degrees, using the vertex-cut of multiple copies on vertices of low degrees. M-Flash [25] and Gemini [67] also try to use different strategies for vertices of different degrees, but they do not make full use of the power-law distribution of the graph to improve the system performance.

Some key optimization techniques that are applicable to vertex-centric graph computing systems on single servers. Unfortunately, most of them are ”one size fits all” design, but we find that some of them are substitutable for each other. For example, fine-granularity selective scheduling and coarse-granularity selective scheduling are two overgeneralized approaches and they can be used as alternative method in different situations.

Here, we discuss two pairs of techniques in detail and demonstrate why they can be combined to improve the performance for power-law distribution graph:

3.2.1 Edge-based Updating and Vertex-based Updating

When processing the *compute* function, QuickSquad will read data from disk at the gather phase and then write the data back to disk at the scatter phase. The amount of data of one vertex (which are read and written back) is related to the number of edges adjacent to it. Therefore, in the procedure of one iteration, system will read and write back the graph structure data whose total amount is related to $|E|$. So, we call this kind of updating model as

edge-based updating model in this paper. In edge-based updating model, the system will maintain a vertex value table and an edge value table in the whole implementation of algorithms, and the data will be exchanged between iterations through the edge value table. In scatter phase, the system produces the corresponding update for each adjacent edge of a vertex according to the vertex value, and writes it back to the storage. While in gather phase, the system will read the information of the related adjacent edges in the storage and recomputes the current vertex value according the updates. Such a way is adopted to update the system in GraphChi [34], X-stream [50] and NXGraph [14].

According to the work flow of edge-based updating, edge value table will generate the data whose amount is proportional to edges in every iteration. There are a large number of edges in a natural graph which will exert a strong impact on system performance by reading and writing repeatedly. **Vertex-based updating** (also called on-the-fly vertex updating) [66,12] is proposed, which directly writes the update into the accumulated value table of a vertex and then exchanges the data between iterations through the vertex value. When the updating of an algorithm satisfies Abelian law, namely the operation of updating is associative and commutative, the updating produced by every adjacent edge can be updated to the destination vertex by using the cumulative sum directly without the need to bring in edge value table additionally, thereby edge-related I/O is reduced in large amount. Though vertex-based updating needs to load and synchronize the vertex value table additionally, which also incurs additional vertex-related I/O, and the cost of this part is hidden because the number of most graph edges we process is much greater than the amount of vertices. Venus [12], GridGraph [66] and FlashGraph [65] all use the implementation strategy of vertex-based updating.

3.2.2 Partition Granularity and Vertex Granularity Selective Scheduling

One of the main steps in all iterations of an iterative graph computing algorithm is accessing graph structure data. Algorithms have different access patterns [65] and sometimes one iteration only needs to access parts of the graph data, so selective scheduling strategy is needed to load the needed data and skip those useless data. Theoretically, selective scheduling strategy can visually bring the following three kinds of benefits during graph computing:

1. Reduce the I/O amount of reading-in edges;
2. Reduce the number of edges in graph building (if graph building is needed);
3. Reduce the number of traversal edges when computing.

Selective scheduler with partition granularity, also known as coarse granularity, is implemented in some systems. It uses a subgraph as a scheduling unit and each subgraph has an activity state which decides whether the subgraph is necessary to be visited in this iteration. To implement the coarse granularity selective scheduler, they only need to record the active state of each vertex in a global scheduler [34]. And they divide the vertex table into several disjoint intervals (represent subgraphs) and the corresponding edges of each vertex congregate in some disjoint intervals. Finally, whether the edge set related to the interval is scheduled will be determined by interval states.

It is simple to implement the selective scheduling of coarse granularity, but it also has limitations. For example, in some extreme cases, even there is only one active vertex in an interval, all corresponding edge sets shall be scheduled in the whole interval. FlashGraph[65] and Graphene[40] use the scheduling method of vertex-granularity to effectively reduce I/O by observing the active state of every vertex rather than the active state of the whole interval. The scheduling of vertex-granularity will lead to a great amount of random I/O which decays the performance of storage, so FlashGraph and Graphene both work on SSD(solid state disk) and use some techniques to merge these I/O requests. In details, FlashGraph uses file system SAFS(set-associative file system)[64] and Graphene uses the I/O merging and I/O deduplication.

3.3 Graph-based detection approaches for fake accounts

Finally, we will discuss two different types of typical detection approaches based on graph structures used in this paper.

SybilRank [7] uses an early-terminal random walk algorithm to detect the sybil attacks effectively on social networks of bilateral relationship. Firstly, SybilRank assumes that normal users in social networks compose a well connected (or fast mixing) graph. Hence, a random walk algorithm can be deemed as an irreducible and aperiodic Markov chain [17] in the whole network, Like TrustRank [27] and SybilRadar [48], which also uses power iteration [35]. Before the algorithm starting, some identified normal users are assigned with a positive number as a trust value and these normal users are called trust seeds, and others are users to be detected and initialized to be 0. In each round of iteration, each user propagates its current trust value to its neighbors. Secondly, it is assumed that fake accounts have fewer opportunities to keep bilateral relationship with normal users (different from the unilateral followings of digraphs) and the relationship established is usually dispersed. Hence, we apply early-terminal technology before converging, which will lead to better results than that in the state of stationary distribution. The results got above form a ranklist, based on which suspicious accounts are identified. In addition, early-termination also improves the efficiency of the algorithm and reduces algorithm's iterations. Hence, SybilRank has higher efficiency and Cao et al. implemented it in Hadoop MapReduce [16] platform. On the other hand, SybilRank is also the foundation of a lot of subsequent work. However, experiment shows that SybilRank still needs more than one day to process a large-scale graph. For example, it takes 33 hours to process an artificially generated 160 Million network graph, which indicates that there is still a large gap between the effective detection efficiency and the actual use of SybilRank.

COLOR [63] algorithm is a traverse-based single-vertex detection algorithm which is subject to the following two assumptions. Firstly, to widely spread malicious information, fake accounts will establish relationship with more users and these users interact rarely. Namely, malicious accounts will attempt to establish relationship with users in different social groups, while social relationship between normal users will be concentrated on specific social groups. Secondly, normal users will not usually or deliberately interact with malicious accounts. Therefore, COLOR algorithm identifies suspicious accounts through scanning neighboring vertices of each detected user and through observing the interactive relationship between neighboring vertices. COLOR algorithm firstly traverses each neighboring vertex of the vertex to be detected, colors each neighboring vertex with a different color, and then begins recursive coloring from each neighboring vertex. To improve coloring efficiency, COLOR algorithm only colors vertices in the a distance within k from the vertex to be detected (k is the coloring distance). When colors of all vertices do not change any more or the colored distance of each color exceeds a certain limit, coloring is stopped. Finally, coloring is summarized by the algorithm and the reliability evaluation of detected users is provided according to the statistical structure.

Since this paper attempts to increase the efficiency of graph-based detecting algorithms by graph computing systems, but not to attempts discuss the algorithm performance in fake accounts detection, we choose SybilRank for that it is based on community detection algorithm and random walk algorithm. Much of the following work (such as TrustRank [27], VoteTrust [56], SmartWalk [41], SybilRadar [48]) can be seen as the improved version of SybilRank, and they are improved from the perspective of trust seed selection [48], assignment operations [48] on trust seed [27] or changing the mode of trust value propagation [56, 41]. Other detection methods that were conducted at the same time with SybilRank, such as SybilInfer [15], SybilGuard [61], SybilLimit [60] are all based on the random walk algorithm. In addition, SybilRank was implemented on the big data processing framework, MapReduce [16], which has been applied on a social network in the real world, that is, detecting the fake accounts on Tuenti (the largest OSN in Spain). Among this kind of algorithms, we choose the SybilRank algorithm finally.

Furthermore, COLOR is a single-vertex algorithm based on traversal, which uses the graph computing method which is different from that of SybilRank. Therefore, we choose it as another optimization method to prove the versatility of our system.

In conclusion, the above two types of algorithms are used as examples and are modified to be parallel vertex-centric algorithms. Note that, in this paper, we assume that all detection algorithms are applied to the vertex, that is to say, the computing result (excluding the intermediate result) is the specific attribute value of each node, and its calculation characteristic is in line with Abelian group [12].

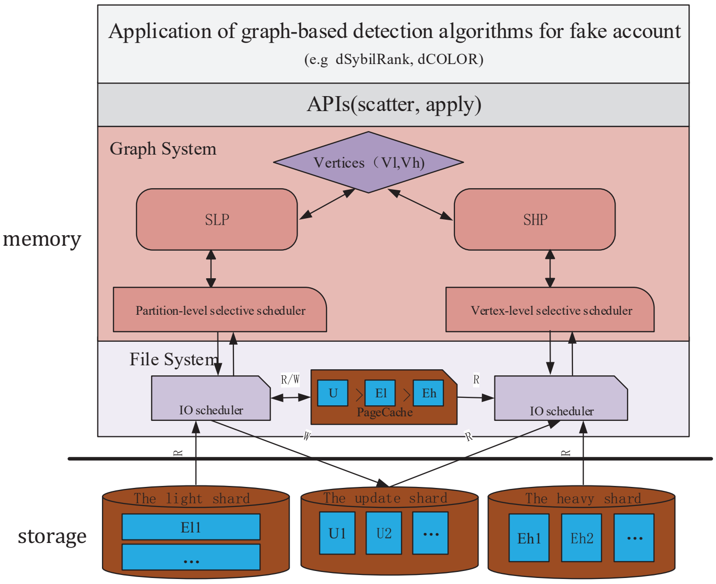

Fig. 3 Architecture of QuickSquad

4 Overview

In this section we will introduce QuickSquad which, just like other graph computing systems, is based on the idea of vertex-centric parallel graph computing and uses the out-of-core graph processing technology to expand its ability to process large-scale social network data. Unlike other systems, QuickSquad takes the features, like power-law distribution, of social network graphs into consideration.

4.1 Optimizing of power-law distribution graph

When designing our system, characteristics like power law distribution are considered to optimize the existing system. According to the out-degree of the vertices, the social network graph is divided into 2 disjointed sets, namely, heavy vertex set V_h and light vertex set V_l. These two represent the vertex set with high out-degrees and the vertex set with low out-degrees, respectively. E_h is the heavy edge set, which is the set of all outgoing edges related to V_h, i.e. the source vertex v_i of every $e(i,j) \in E_h$ belongs to V_h. Similarly, we define E_l as the light edge set, which is the set of all incidence edges related to V_l. Thus, two disjointed partitions named $G_h(V_h, E_h)$ and $G_l(V_l, E_l)$ can be obtained. By using different strategies on G_h and G_l, to processing of social network graphs can be accelerated including the graph storage format, graph processing model, selective scheduling strategy and cache policy. The overall structure of our system is shown in Fig.3.

4.1.1 storage format

The original edge data is pre-processed, and divided into E_h and E_l, which use different storage formats residing on disk. To store E_l on disk, we divide the vertex set evenly into P disjointed sets which are known as source vertex subintervals. And, then all edges in E_l fall in P light shards which are related to the P source vertex subintervals. An edge will be put into a shard if and only if the source of the edge is in the corresponding subinterval. Please note that we need sorted-edges only when selective scheduling requires vertex-level(§4.1.3), so we don't need sorted-edges in light shard, thus increasing the speed of pre-processing and decreasing the time of edge sorting. For E_h, the vertex ID values are also divided into Q equal disjointed sets which are called the destination vertex subintervals. Edges whose destination vertices are in the same destination subinterval are placed in the same file and sorted by the source vertex ID and the file is known as the **heavy shard**. Note that, when selecting the best value of Q, M(the size of internal memory) should be greater than $|Vl| + |Vh|/Q$, because the memory should cache all V_hs and at least one V_l to ensure the performance of the system. As for the interval division, consecutive ID values are segmented into P or Q equal parts, so as to improve the locality of access.

4.1.2 processing model

In this paper, we introduce two updating modes (§3.2.1), which are fit for graphs of different density respectively. Edge-based updating(EUP) performs better in sparse graph, by which QuickSquad processes G_l. While vertex-based updating(VUP) performs better in dense graphs by which QuickSquad processes G_h. So we put forward a two-phase processing strategy by combining EUP and VUP to process the network graph with power-law. This strategy does not process G_l and G_h sequentially or vice versa directly. In fact, we firstly execute the scatter stage of EUP on all edge e in G_l, which is called **streaming light phase (SLP)**. Then in **streaming heavy phase (SHP)**, all target vertex subintervals are processed one by one, and VUP and gather stage of EUP are executed on every target vertex at the same time. Processing VUP and gather stage of EUP at the same time on one subinterval can eliminate the cost of repeated data loading, and it will also increase the locality of visits.

The system executes this two-phase processing model in each iteration. In SLP, the edge-based update strategy is adopted to handle P E_l and generate updates. As is shown in Fig.4(a), the affiliated attribute of each light shard and its corresponding vertex is read in order. Each edge e is traversed and the corresponding update value u is calculated through the user-defined vertex function. There are Q U_l files which is classified by destination vertex subintervals and according to the destination vertex value of e, update value u is written into the corresponding file U_l. File U_l can be seen as the intermediate

algorithm 1 Main procedure of our system

```
Require: $V_h$, $V_l$, $D$, $E_h$, $E_l$
Ensure: $D$ of the results
for $i = 1$ to $iterations$ do
    Map $D$ to $V_h$ and $V_l$
    // SLP
    for $i = 1$ to $P$ do
        load $V_l^{(i)}$, $E_l^{(i)}$ in $SourceInterval[i]$
        for each $e \in E_l^{(i)}$ do
            compute $u$ from scatter($e$)
            append the $u$ to corresponding $U_l$
            if memory is full then save all $U_l$ buffers
            end if
        end for
    end for
    // SHP
    load $V_h$
    for $i = 1$ to $Q$ do
        // step 1
        load $D^{(i)}$, $U_l^{(i)}$ in $DestinationInterval[i]$
        for each $u \in U_l^{(i)}$ do
            apply($u$) to $D^{(i)}$
        end for
        // step 2
        load $E_h^{(i)}$ in $DestinationInterval[i]$
        for each $e \in E_h^{(i)}$ do
            direct_apply($e$) to $D^{(i)}$
        end for
        save $D^{(i)}$
    end for
end for
```

result between the first phase and the second phase. SLP can be seen as the first half phase of edge-based updating (§3.2.1).

In SHP, as is shown in Fig.4(b), is mainly composed of two tasks. One is to accumulate all update u in the Q update file U_l (which is generated in SLP) to the destination vertex subinterval. The other is to generate the update u of E_h and directly write the update u on the destination vertex subinterval (without introducing update files) In detail, QuickSquad will process the destination vertex subintervals (whose number is Q) one by one. First, the subinterval to be processed should be loaded from disks to memory. Then the loaded subinterval should be updated, which is divided into two steps:

1) Step 1: to process U_l (generated in SLP), by updating each update u in U_l to the destination vertex subinterval.

2) Step 2: to process each edge e in E_h by using the vertex-based updating strategy to update the data from the source vertex in V_h (which is already cached in the memory) of the edge directly to the destination vertex.

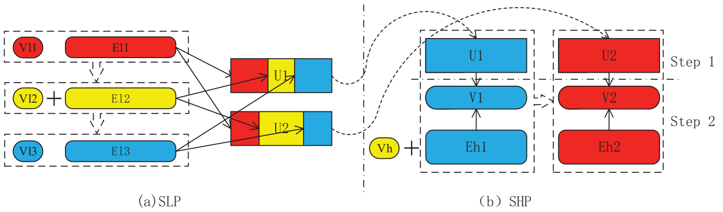

Fig. 4 The flow chart of processing model. Vl and El represent the vertices and edges in light shard respectively. Vh and Eh represent the vertices and edges in heavy shard respectively. U represents the update U_l.

Finally, after SLP and SHP, the updated V data will be mapped to V_l and V_h for the next round of iteration. The pseudo-code of the main streaming is provided in Algorithm 1.

I/O analysis: This paper adopts the analytical approach similar to I/O complexity [1] to analyze the I/O complexity of scanned data in the first iteration. In SLP, as is shown in Fig.4(a), V_l and E_l are loaded sequentially to generate the corresponding update u and then written in the corresponding U_l in sequence, where V_l is generated by mapping loaded V. As V and E_l are both read in order, and more than one file is written in U_l orderly at the same time, the total I/O number of SLP will be $|V|+|E_l|+|U_l|$. SHP mainly includes two steps, as is shown in Fig.4(b). To enhance the locality of data, two steps are applied continuously to Q V-intervals. In step 1, V and U_l of corresponding intervals (U_l in the light phase) need to be loaded. In step 2, all V_h and E_h of corresponding intervals need to be loaded. When finishing the two steps, the calculated result V is written to the storage again. As we assume that the system memory can cache V_h and V of one interval at the same time, and that step 1 and step 2 are executed continuously when handling each interval, the system merely needs to load V_h and the corresponding V only once. Therefore, the read amount in the streaming heavy phase is $|V| + |U_l| + |V_h| + |E_h|$, the written amount is V, the total amount of I/O is $2 * |V| + |U| + |V_h| + |E_h|$. In summary, the amount of all loaded I/O in an iteration by the system is $3 * |V| + |E| + 2 * |U_l| + |V_h|$, in which, $|E| = |E_l| + |E_h|$, $|V| = |V_l| + |V_h|$.

4.1.3 selective scheduling

QuickSquad uses two different scheduling strategies in E_l and E_h, that is, the selective scheduling strategy of partition granularity and that of vertex granularity. As is known that the graph computing system supports various algorithms, different accesses may be presented by different algorithms when accessing the graph structure data [65]. For instance, in the execution process of some traversal algorithms such as the breadth first search(BFS), only part of the graph structure data need to be accessed sometimes, while effective scheduling strategies can help to reduce I/O and the computing cost to some extent [34,66]. Therefore, we use the scheduling strategy of partition granularity for E_l. In other words, if and only if there is at least one vertex being active in a source vertex interval, we will load all E_l files that are related to this source vertex subinterval. And then, we'll process these files even though we know that most computation makes no contribution to the final result. In E_h, the scheduling strategy of vertex granularity is used, so the state of each vertex has to be scanned according to the sequence of source vertices on E_h. For active vertices, statistics about the position of their adjacent edges should be obtained before merging adjacent or close edges and then read them together in the memory. There are some reasons for using such bimodal I/O scheduling strategy. Edges in E_l are rather sparse. If the vertex granularity scheduling strategy is applied, numerous small random reading will be caused. Moreover, the state of corresponding vertices needs to be scanned for the scheduling strategy of vertex granularity, the cost of which may be greater than the cost of reducing I/O and the computing amount caused by the vertex granularity scheduling strategy.

4.1.4 cache policy

Since the access to graph structure data is random and of poor predictability, the actual bandwidth of storage devices access like disks will be decreased, so data should be cached to the memory in an efficient way. In case the memory fails to cache the entire data , frequently-used or system built-in cache strategies may be used, such as LRU whose performance is poor. A simple strategy is adopted by QuickSquad, through which different caching priorities are set for the data, that is, data is cached according to the sequence $V_h > V_l > E_l > E_h$. There are three reasons for setting such priority access. First, the magnitude order of vertices is smaller than that of edges, and the vertex access is more frequent than the edge access. Second, vertices with high out-degrees are more frequently accessed than that with low out-degrees. Finally, it is determined by the execution mode and scheduling mode of the system that the access times of E_l (including U_l) in the first iteration is 3 times that of E_h (read twice and write once).

4.1.5 summarize

Table 1 summarizes several typical optimization techniques used by graph computing system on single server including the update models used in algorithm processing, the total amount of I/O of each round of iteration involved (assuming the entire graph structure data is scanned once in an iteration), and the selective scheduling strategy. Our system is compared with other similar work, like I/O total amount. For instance, comparing with X-Stream, when the amount $2 * |U|$ of the updated data related to edges is reduced to $2 * |U_l| + |V_h| + |V|$, $|U_l|$ and $|V_h|$ can be rarely controlled in the power-law distribution graph. In §6.2.1, we are going to provide the values we measured in an actual social network graph. It is worth noting that FlashGraph is a semi-external model which assumes that the entire vertex data can be cached

Table 1 Comparison of the Key Techniques of other Graph Computing Systems. L represents light shard and H represents the heavy shard.

	Upd. Mod.	I/O Amount
GraphChi [34]	Edge	$\|V\| + 2 * \|E\| + \|U\|$
X-Stream [50]	Edge	$2 * \|V\| + \|E\| + 2 * \|U\|$
GridGraph [66]	Vertex	$(2 * \|V\| + \|E\| + P * \|V\|)$
FlashGraph [65]*	Vertex	$\|E\|$
QuickSquad	L:edge H:vertex	L: $\|V\| + \|E_l\| + \|U_l\|$ H: $\|V_h\| + 2 * \|V\| + \|E_h\| + \|U_l\|$ Total:$(3 * \|V\| + \|E\| + \|V_h\| + 2 * \|U_l\|)$

	Selective Scheduler	API	Abelian Law
GraphChi [34]	Partition-level	GAS	No
X-Stream [50]	-	GAS	No
GridGraph [66]	partition-level	A	Yes
FlashGraph [65]*	vertex-level	GAS	No
QuickSquad	L:partition-level H:vertex-level	GAS	Yes

by the system, so there's only the edge data for the I/O total amount of one iteration.

4.2 Programming interface

Just like other graph computing systems, we provide a vertex-centric programming interface. Users can implement user-defined vertex functions including the *scatter* function and *apply* function. Each vertex is scheduled in parallel by the system, and whether to execute the *scatter* function and / or *apply* function will be determined in accordance with the vertex state.

scatter(e, u)**:** *scatter* function can generate update $u = (u.target, u.value)$ by accessing edge $e = (e.source, e.target)$ as well as the attribute of its vertex $D(e.source)$ and / or $D(e.target)$. In SLP, the u generated by *scatter* function will be written into a buffer area associated with $u.target$. When SLP is completed or the memory is full, u will be written into the corresponding file U_l. However, in SHP, the u generated by *scatter* will be directly regarded as input by *apply* without generating a buffer zone or file of intermediate results.

apply(u)**:** *apply* function plays a role in updating the update u (generated by scatter) to the corresponding vertex. *Apply* function only works in the SHP phase, being responsible for handling the updates generated in SLP phase and SHP phase. For SHP phase, since no additional files are needed for saving updates and we provide *direct_apply*(e) function to update the computing result of e directly to vertices, *direct_apply* is an optimized method of *scatter* and *apply* function, which helps to reduce extra memory and computing expenses caused by generating u in SHP phase.

5 Implementation

We use C++ language to implement the system we designed, a total of 3700 lines including 2800 lines of graphic processing engines and about 900 lines of preprocessing tools. We also give some examples of the real implementation of algorithms, including dSybilRank, dCOLOR and some other basic algorithms.

5.1 Preprocessing

Before preprocessing, we need a format conversion tool to convert the input graph structure data into a binary edge list. Each edge e is represented by a pair of ID value $< srcID, dstID >$ that are remapped and counted from zero.

In the preprocessing phase, we first scan the edge list once to calculate the out-degree of each vertex and will divide the vertices into two sets: V_l and V_h according to the vertex degree. Then, we scan the edge list once and put the edges into P temporary E_l files and Q E_h files in accordance with certain rules. If an edge's source node srcID belongs to V_h, the edge will be put into the corresponding temporary E_h file according to the $dstID$ value of the edge, otherwise it will be put into the corresponding E_l file according to the $srcID$ value of the edge. Finally, by using the external sorting algorithm, edges inside the P temporary E_h files will be sorted according to $srcID$ and then be written into E_h file.

5.2 Computing

Before computing, the system will be initialized, including the meta-data of the processed graph. Then the system will execute the computing process in a circular manner according to the user-defined iteration condition.

The computing process mainly consists of two execution phases, SLP and SHP. Each round of iteration begins with SLP which loads the state set V of vertices. V is read-only at the moment that represents the initialization or the vertex state of the last iteration. By scanning the edge e in E_l, the state belonging to V_l will be found in V through srcID of the edge to update the state and generate update $u < dstID, D(srcID) >$. Since E_l is sorted according to $srcID$, the reading of V is performed in the sequence of ID value though it's not continuous. After the completion of SLP, SHP begins.

In SHP, each target subinterval of V is processed one by one. As is mentioned in §4.1, processing of a subinterval has two phases. First, QuickSquad loads U_l generated in the streaming light phase and processes update $u < dstID, value >$ in U_l to $V(dstID)$ through a user-defined update function. Second, QuickSquad loads the E_h file. Then each edge $e < srcID, dstID >$ in E_h generates a $u < dstID, D(srcID) >$, where $D(srcID)$ is the related attribute, and then directly updates the U_l to $V(dstID)$. In each phase, by selective scheduler (if necessary), the system issues the I/O requests to load the needed data through the main thread. The main thread will distribute the data to other threads to be processed respectively, so the user-defined function should guarantee the multi-thread safe, for which we provide a number of atomic operation interfaces.

We provide the interface of the *function objects* of C++ language, users can use the lambda function [30], a functor or function pointer to implement their user-defined vertex functions so as to implement the algorithm.

Finally, after finishing the computing process, the updated vertex state set V can synchronize V_h by remapping.

5.3 Applications

In this section, we will give two examples of graph-based accounts detection implementation on QuickSquad, called dSybilRank and dCOLOR. Note that we tend only to optimize the part which involves the graph processing in the detection algorithm. dSybilRank and dCOLOR algorithms are only the reimplementation of the distributed version of SybilRank and COLOR, whose aim is to improve the detection efficiency but not the detection performance. Therefore, we don't discuss the detection accuracy in the paper, such as true positive rate, false positive rate and even AUC if we need.

5.3.1 dSybilRank algorithm

dSybilRank is an improved algorithm of SybilRank, which make reference to the idea of SybilRank and uses Random Walk. It is worth noting that SybilRank is the process of conducting the entire detection, including generating the trust value through the power iteration, sorting the trust value and separating the suspicious users from normal users through the sequence of sorted users. We only optimize the power iteration involved in the graph structure. Though dSybilRank also uses Random Walk, it uses vertex-centric iteration, which can make the most of the parallelism of the iterative graph algorithm on QuickSquad, instead of the power iteration to compute the trust-rank.

5.3.2 dCOLOR algorithm

dCOLOR algorithm is an improved algorithm of COLOR. COLOR algorithm is a single-vertex detection algorithm through a recursive way ((§3.3). In order to improve the efficiency, although COLOR algorithm proposes two heuristic pruning strategies, the efficiency of recursive way is not high in parallel processing. In order to improve the parallelism of the detection, we change it into an iterative computing method, and use the method of vertex activation to color. The vertex activation means that only the initialized or the activated vertices in the last iteration will be dealt with in the current iteration, during which vertices needed to be dealt with in the next iteration will be activated.

First, at the beginning of the algorithm, we initialize the vertices to be detected as active vertices, see Alg.2. In each iteration, in order to reduce the system access to too many unnecessary vertices, we will color the destination vertex only when the current source vertex has the color that does not exist in the destination vertex, and we will activate the destination vertex only when the destination vertex is within a fixed distance from the vertex to be detected, that is assuming the original COLOR algorithm scanning the vertices within a fixed distance from the vertex to be detected. We provide DCOLOR's programming interface about scatter and apply on QuickSquad, see Alg.3 and Alg.4.

algorithm 2 dCOLOR's init($startID$) function

```
Require: $startID$, $direct_neighbors$,
Ensure: updated update $u$
for $v$ in $direct_neighbors$ do
    $active[v] = true$
end for
```

algorithm 3 dCOLOR's scatter(e, u) function

```
Require: edge $e$, pointer of update $u$
Ensure: updated update $u$
u.target = e.target
u.value = e.source
```

algorithm 4 dCOLOR's apply(u) function

```
Require: update $u$, pointer of $colors$
Ensure: updated update $u$
if $active[u.value] == true$ and $(colors[u.value] \cup colors[u.target]) \setminus colors[u.target]$ is not nil then
    if $u.target$ is in $direct\_nerghbors$ then $next\_active[u.target] = active$
    end if
    $colors[u.target] = colors[u.value] \cup colors[u.target]$
end if
```

6 Evaluation

In order to test the performance of QuickSquad on some real large-scale social network graphs, we implement different detection algorithms and detect them on different data sets. At the same time, in order to demonstrate the scalability of the system under different hardware conditions, we also test it under the condition of different hardware configurations.

Data set: For directed graph, we use Twitter [9] as the experimental data set, which has 54981152 users, including more than 40000 fake accounts and 1963263832 following (unidirectional) relationships [10]. We also use the data of Sina Weibo[2] crawled by our own: 0.5 million nodes of users, 23420 fake account in it and with the 259579862 following relationships of these users.

For undirected graph, we use Friendster [57], an on-line gaming network, which has 65608366 nodes and 1806067135 edges.

Furthermore, we use 40 trust seeds on both Twitter graph and Friendster graph, and use 10 trust seeds on Sina Weibo graph. We choose the trust seeds randomly, according to the several vertices with the highest degrees and make sure that they are the same when different algorithms(like SybilRank, dSybilRank) are working.

Algorithms: We implement the algorithms of dSybilRank, dCOLOR and breadth-first-search (BFS) algorithm. The dSybilRank and dCOLOR are the two algorithms described in §4.2, and BFS is the basic algorithm of dCOLOR. BFS scans from one vertex or a set of vertex, and each iteration marks the non-visited vertex in the adjacent side of the current active vertex as the active vertex of the next iteration. The algorithm doesn't finish until there is no new vertex to be marked. The visiting mode of BFS is often to traverse only certain edges, so it can be used to test the performance of the selective scheduling strategy like the dCOLOR algorithm.

Testbed: All our experiments are performed in a single server with 2 Intel Xeon E3-1230 V2 CPUs (3.30GHz, 4 cores per CPU), 4 DDR3-1600 memory of 8GB , and 2 3TB hard disks of 7200 rpm. We use 8 cores, 32GB memory and 3TB hard disk under the default configuration.

Methodology: The purpose of the experiments is to achieve the efficiency of the system, so we use the total run time of the implementation as the parameter for performance comparison. We measure the running time according to two circumstances: in the task with long execution cycle, we can only guarantee the test having no abnormity for only once; in the task with short execution cycle, we use the method of obtaining the average value by multiple measuring.

Our evaluation will answer the follow questions:

1. Does a distributed version of the algorithm implemented on QuickSquad (like dCOLOR) perform better than centralized one (like COLOR)? And how well does dCOLOR perform when threads are of different numbers? (§6.1)
2. How well does QuickSquad perform in social networks with power-law distribution when compared with existing graph systems? (§6.2)
3. Why can storage format and two-phase processing model perform better on powerlaw graph? (§6.2.1)
4. How do selective scheduling and cache policy help QuickSquad improve its performance? (§6.2.2)

[2] we catch these data using crawler at Jan, 2015 in our lab and evaluate the fake account by ourself.

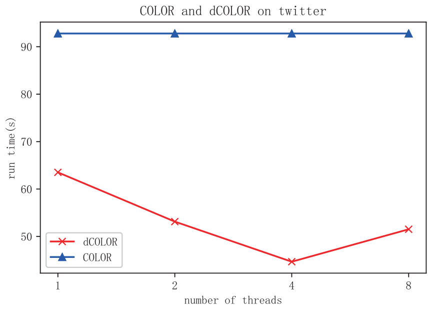

Fig. 5 Experiment Results of dCOLOR/COLOR with different numbers of threads

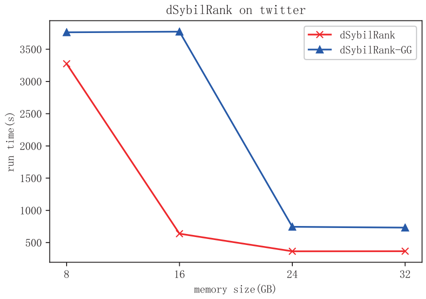

Fig. 6 Experiment Results of dSybilRank with different memory sizes(GB)

6.1 Comparison between dCOLOR and COLOR

We compare the performance of dCOLOR algorithm and COLOR algorithm. Fig.5 shows the performance comparison between dCOLOR algorithm and COLOR algorithm when they use different numbers of threads. COLOR algorithm is implemented based on single-cored depth-first-search algorithm. For acquiring the attribute of a single vertex, the average time of computing 10 vertices by COLOR algorithm is 92.79 seconds (the average value of 10 random vertices when processing the same graph), and in order to improve the parallelism, dCOLOR algorithm uses the breadth-first-search algorithm, which takes as fast as 44.68 seconds too compute 10 vertices. Compared with the single core COLOR algorithm, 8-thread dCOLOR algorithm uses the breadth-first-search algorithm with higher parallelism, but coloring it still has a large number of critical areas and there will be a deadlock, so the thread execution process needs a large amount of cost in mutual exclusive operation. This is why the performance of the dCOLOR algorithm does not obtain linear growth when the number of threads increases.

6.2 Performance comparison with other graph computing systems

The performance improvement of dSybilRank based on vertex-centric graph computing framework compared with SybilRank based on MapReduce framework can not directly prove that the optimization of QuickSquad has improved the performance. In order to verify the performance improvement implemented by our system specific to the optimization of social network graphs, we also make a comparison with GridGraph [66] whose performance is now better in every aspect, in addition to comparing with the implementation of traditional algorithms. We implement dSybilRank algorithm (marked as dSybilRank-GG) in GridGraph. At the same time, because GridGraph can not directly implement the algorithm which is secure in multiple threads, we use BFS algorithm

Table 2 Contrast between the I/O amount of QuickSquad and that of GridGraph. P represents the number of partitions selected by related graph

Social Network	# of v in V_h	# of v in V	V_h/V	# of e in E_l
Sina Weibo	270433	500000	54.09%	35745638
Twitter Graph	20960978	54981152	38.12%	71416245
Friendster	25205669	65608366	38.42%	24474341
Social Network	**# of e in E**	**E_l/E**	**I/O Amount**	**I/O Amount in GG**
Sina Weibo	259579862	13.77%	332841571	269579862($P = 20$)
Twitter Graph	1963263832	3.63%	2292000756	3612698392($P = 30$)
Friendster	1806067135	1.36%	2077046584	3774318115($P = 30$)

to detect the performance of selective scheduling (the implementation in GridGraph is BFS-GG).

6.2.1 I/O account analysis

QuickSquad uses a two-phase processing model to get the direct benefit that is the reduction of the total I/O in the power-law distribution graph. GridGraph applies the vertex-based update execution model through 2-D partition [66]. The total amount of I/O in one of its iterations is $2 * |V| + |E| + P * |V|$, among which P is the interval number generated when GridGraph segments vertices. QuickSquad makes full use of the characteristics of the power-law distribution of social networks to eliminate the parameters related to P, which are converted into a one-time access to a light edge and a heavy vertex, that is $3*|V|+|E|+2*|U_l|+|V_h|$. The size setting of P in GridGraph affects whether the data can be well cached in LLC (Last Level Cache) or not, affects as well as the scheduling granularity. If P is too small, the scheduling granularity will be large, and it can not be friendly cached in LLC or memory size. Therefore, the size of P is generally related with the ratio of the total amount of the data being dealt with to the memory of the machine [66]. We know that in the power-law distribution graph, $|U_l|$ and $|E_l|$ are directly proportional, while $|E_l|$ and $|V_h|$ may be two relatively small parameters after the proper cut. For example, the actual size of V_h is 20960978 in Twitter graph after cut, while the E_l size is only 71416245, accounting for only 3.63% of the total edges. While in Sina Weibo data, after cut, the actual size of V_h is 270433 and E_l is only 35745638, accounting for only 13.77% of the total edges. Therefore, our system increases its scalability after eliminating the influence of P. In addition, QuickSquad can better adjust of the selective scheduling granularity, even though P is 1(the minimum value). We can also make $2 * |U_l| + |V_h|$ stand at its lowest value, even at zero (in the extreme case), via proper cut.

6.2.2 selective scheduling and cache policy analysis

QuickSquad uses a selective scheduling and cache policy for social network graphs. In Twitter graph, the total number of edges after compression is

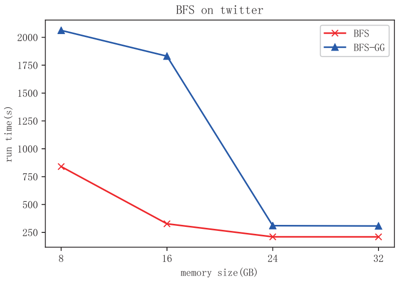

Fig. 7 Experiments Results of BFS

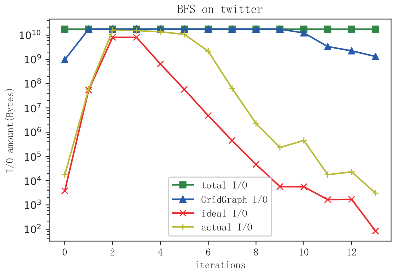

Fig. 8 Experiment Results of BFS's I/O Amount

15.6GB. In a full memory mode, GridGraph and QuickSquad both require about 22GB memory to deal with the graph (including the memory required by operating system). We measure the performance change of two systematic Twitter graphs from 8GB to 32GB memory, of which 8G represents a small memory, 16GB represents a medium memory, but it is still not completely cache the required graph data into memory, while 24GB and 32GB represent the infinite increase in memory. Fig.6 shows, in Twitter graphs, the performance comparison of dSybilRank implemented in QuickSquad and dSybilRank-GGs implemented in GridGraph. QuickSquad can use different caching strategies according to the relationship between the data and the available memory when memory is low, while GridGraph uses a unified way (one size fits all) to cache data. Therefore, the greater the memory is , the better performance QuickSquad can get than GridGraph when memory can't cache all the data. When memory is enough (or infinite), only the mode of execution is optimized, so it can be said that QuickSquad is more scalable when the data size is larger than the available memory.

Fig.7 shows, in the twitter graph, the performance comparison of BFS implemented by QuickSquad and BFS-GG implemented on GridGraph, and BFS algorithm is the basis of some traverse-based detection algorithms [13]. The result of the performance comparison of BFS algorithm is the difference of caching policies and the impact of selective scheduling policies. QuickSquad uses a selective scheduling strategy of a dual mode, which can make the system in each iteration focus on loading and computing the useful data, and in the case of low memory, useful data can be loaded by making full use of disk bandwidth. Therefore, comparing Fig.6 with Fig.7, we find that the optimization of BFS algorithm is better than that of dSybilRank algorithm in the case of low memory.

Fig.8 shows that if all the graph structure data are read from the storage (i.e. no cache data in memory), the comparison is made among the actual data size when reading Twitter in each iteration by BFS algorithm in two systems, and the data amount required by an ideal circumstance (ideal I/O, that is, when not reading any redundant data) and the total I/O. It can be seen that QuickSquad uses selective scheduling strategy of dual modes, whose performance is closer to the ideal I/O than GridGraph. It should be noted that the amount of ideal data is very difficult to be implemented well in an ideal circumstance, because the ideal data in the actual circumstance is randomly dispersed in the file, and not reading the redundant data leads to a large number of random overhead of I/O, especially in the storage of disks with seek time [1].

7 Conclusion

With the continuous development of social networks and their potential commercial value, attackers attempt to reap the benefits by new methods, such as fake accounts. The graph-based detection algorithm is one of the effective ways to detect fake accounts. However, with the continuous expansion of the data scale, the scalability and computing efficiency of the existing detection algorithms need to be improved. This paper puts forward a computing system on single machines, which specific to the feature optimization of social networks' graph structures by analyzing and studying the features of social networks' graph features, by the existing large-scale graph computing systems, by the existing algorithm of fake account detection based on graphs, and by the algorithm implemented on the system , including two kinds of detection algorithms and breadth-first-search traversal algorithm. The algorithm implemented in our system can significantly enhance the performance in contrast to the traditional implementation including the single-core algorithm implementation and even the implementation of MapReduce distributed framework. Moreover, the performance of the system can be improved by an average of 1.76 times compared with the existing system.

For future work, we are to improve from three aspects: First, now we only implement a single-node version, which is limited to extending to deal with social networks of ten billions nodes, like FaceBook. QuickSquad will be extended to support a distributed cluster which can be easily extended to have the feature of multiple nodes share memory [24]. Second, we would like to support the dynamic and mutable graph on QuickSquad, which means the graph can dynamically add/delete vertices or edges when we process them. Third, we will put forward a graph-based detection algorithm, which can not only improve the detection efficiency by making full use of such graph computation system as QuickSquad, but also improve the detection performance. However, current work tries to improve the algorithm's efficiency on the premise of ensuring the algorithm's validity, while algorithm's detection accuracy performance can't be optimized.

Acknowledgements We would like to thank the four anonymous reviewers for their insightful comments and suggestions, which have helped to improve the quality of our paper. This work is supported by the National Natural Science Foundation of China under Grant No.61772229 and No.61472162, the Open Fundation of Symbol Computation and Knowledge Engineer of Ministry of Education, JSPS KAKENHI under Grant Number JP16K00117, and KDDI Foundation.

References

1. Alok Aggarwal, Jeffrey Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. Muhammad Al-Qurishi, Mabrook Al-Rakhami, Atif Alamri, Majed Alrubaian, Sk Md Mizanur Rahman, and M Shamim Hossain. Sybil defense techniques in online social networks: A survey. *IEEE Access*, 5:1200–1219, 2017.
3. Vincent D Blondel, Jean-Lou Guillaume, Renaud Lambiotte, and Étienne Lefebvre. The louvain method for community detection in large networks. *J of Statistical Mechanics: Theory and Experiment*, 10:P10008, 2011.
4. Yazan Boshmaf, Dionysios Logothetis, Georgos Siganos, Jorge Lería, Jose Lorenzo, Matei Ripeanu, and Konstantin Beznosov. Integro: Leveraging victim prediction for robust fake account detection in osns. In *NDSS*, volume 15, pages 8–11. Citeseer, 2015.
5. Jian Cao, Qiang Fu, Qiang Li, and Dong Guo. Discovering hidden suspicious accounts in online social networks. *Information Sciences*, 394-395(Supplement C):123 – 140, 2017.
6. Jian Cao, Qiang Li, Yuede Ji, Yukun He, and Dong Guo. Detection of forwarding-based malicious urls in online social networks. *International Journal of Parallel Programming*, 44(1):163–180, Feb 2016.
7. Qiang Cao, Michael Sirivianos, Xiaowei Yang, and Tiago Pregueiro. Aiding the detection of fake accounts in large scale social online services. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 197–210, 2012.
8. Qiang Cao, Xiaowei Yang, Jieqi Yu, and Christopher Palow. Uncovering large groups of active malicious accounts in online social networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 477–488. ACM, 2014.
9. Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *In Proceedings of the 4th International AAAI Conference on Weblogs and Social Media (ICWSM)*, Washington DC, USA, May 2010.
10. Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and P Krishna Gummadi. Measuring user influence in twitter: The million follower fallacy. *ICWSM*, 10(10-17):30, 2010.
11. Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.
12. J. Cheng, Q. Liu, Z. Li, W. Fan, J.C.S. Lui, and C. He. Venus: Vertex-centric streamlined graph computation on a single pc. In *Proceedings of the IEEE 31st International Conference on Data Engineering*, ICDE '15, pages 1131–1142, 2015.
13. S. Cheng, G. Zhang, J. Shu, Q. Hu, and W. Zheng. Fastbfs: Fast breadth-first graph search on a single server. In *2016 IEEE International Parallel and Distributed Processing Symposium, 2016, Chicago, IL, USA, May 23-27, 2016*, pages 303–312, 2016.
14. Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. Nxgraph: an efficient graph processing system on a single machine. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 409–420. IEEE, 2016.
15. George Danezis and Prateek Mittal. Sybilinfer: Detecting sybil nodes using social networks. In *NDSS*. San Diego, CA, 2009.
16. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
17. Xiaoheng Deng, Genghao Li, Mianxiong Dong, and Kaoru Ota. Finding overlapping communities based on markov chain and link clustering. *Peer-to-Peer Networking and Applications*, 10(2):411–420, Mar 2017.
18. Mianxiong Dong, Kaoru Ota, and Anfeng Liu. Rmer: Reliable and energy-efficient data collection for large-scale wireless sensor networks. *IEEE Internet of Things Journal*, 3(4):511–519, 2016.
19. Mianxiong Dong, Kaoru Ota, Anfeng Liu, and Minyi Guo. Joint optimization of lifetime and transport delay under reliability constraint wireless sensor networks. *IEEE Transactions on Parallel & Distributed Systems*, (1):1–1, 2016.

20. Manuel Egele, Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Compa: Detecting compromised accounts on social networks. In *NDSS*, 2013.
21. M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 251–262, 1999.
22. Apache Giraph. `http://giraph.apache.org/`.
23. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, 2012.
24. J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, 2014.
25. Hugo Gualdron, Robson Cordeiro, Jose F Rodrigues Jr, Duen Horng Polo Chau, Minsuk Kahng, and U Kang. M-flash: Fast billion-scale graph computation using block partition model. *arXiv preprint arXiv:1506.01406*, 2015.
26. Supraja Gurajala, Joshua S White, Brian Hudson, Brian R Voter, and Jeanna N Matthews. Profile characteristics of fake twitter accounts. *Big Data & Society*, 3(2):2053951716674236, 2016.
27. Zoltán Gyöngyi, Hector Garcia-Molina, and Jan Pedersen. Combating web spam with trustrank. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 576–587. VLDB Endowment, 2004.
28. János Höner, Shinichi Nakajima, Alexander Bauer, Klaus-Robert Müller, and Nico Görnitz. Minimizing trust leaks for robust sybil detection. In *International Conference on Machine Learning*, pages 1520–1528, 2017.
29. Yanling Hu, Mianxiong Dong, Kaoru Ota, Anfeng Liu, and Minyi Guo. Mobile target detection in wireless sensor networks with adjustable sensing frequency. *IEEE Systems Journal*, 10(3):1160–1171, 2016.
30. Jaakko Järvi, Gary Powell, and Andrew Lumsdaine. The lambda library: unnamed functions in c++. *Software: Practice and Experience*, 33(3):259–291, 2003.
31. Yuede Ji, Yukun He, Xinyang Jiang, Jian Cao, and Qiang Li. Combating the evasion mechanisms of social bots. *Computers & Security*, 58:230–249, 2016.
32. Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment*, 11(3):297–310, 2017.
33. H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, 2010.
34. A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, 2012.
35. Amy N Langville and Carl D Meyer. Deeper inside pagerank. *Internet Mathematics*, 1(3):335–380, 2004.
36. Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2, 2007.
37. Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.
38. Changchang Liu, Peng Gao, Matthew Wright, and Prateek Mittal. Exploiting temporal dynamics in sybil defenses. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 805–816. ACM, 2015.
39. Hang Liu and H Howie Huang. Enterprise: Breadth-first graph traversal on gpus. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.
40. Hang Liu and H Howie Huang. Graphene: Fine-grained io management for graph computing. In *FAST*, pages 285–300, 2017.

41. Yushan Liu, Shouling Ji, and Prateek Mittal. Smartwalk: Enhancing social network security via adaptive random walks. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 492–503. ACM, 2016.
42. Yuxin Liu, Mianxiong Dong, Kaoru Ota, and Anfeng Liu. Activetrust: secure and trustable routing in wireless sensor networks. *IEEE Transactions on Information Forensics and Security*, 11(9):2013–2027, 2016.
43. Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.
44. G. Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, 2010.
45. R. R. McCune, Tim Weninger, and G. R. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for distributed graph processing. *CoRR*, abs/1507.04405, 2015.
46. Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.
47. Abedelaziz Mohaisen, Nicholas Hopper, and Yongdae Kim. Keep your friends close: Incorporating trust into social network-based sybil defenses. In *INFOCOM, 2011 Proceedings IEEE*, pages 1943–1951. IEEE, 2011.
48. Dieudonne Mulamba, Indrajit Ray, and Indrakshi Ray. Sybilradar: A graph-structure based framework for sybil detection in on-line social networks. In *IFIP International Information Security and Privacy Conference*, pages 179–193. Springer, 2016.
49. Mark EJ Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.
50. A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013.
51. Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on gpus: A survey. *ACM Computing Surveys (CSUR)*, 50(6):81, 2018.
52. Nguyen Tran, Jinyang Li, Lakshminarayanan Subramanian, and Sherman SM Chow. Optimal sybil-resilient node admission control. In *INFOCOM, 2011 Proceedings IEEE*, pages 3218–3226. IEEE, 2011.
53. L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.
54. Bimal Viswanath, Ansley Post, Krishna P Gummadi, and Alan Mislove. An analysis of social network-based sybil defenses. *ACM SIGCOMM Computer Communication Review*, 40(4):363–374, 2010.
55. Gang Wang, Tristan Konolige, Christo Wilson, Xiao Wang, Haitao Zheng, and Ben Y Zhao. You are how you click: Clickstream analysis for sybil detection. In *Proc. USENIX Security*, pages 1–15. Citeseer, 2013.
56. Jilong Xue, Zhi Yang, Xiaoyong Yang, Xiao Wang, Lijiang Chen, and Yafei Dai. Votetrust: Leveraging friend invitation graph to defend against social network sybils. In *INFOCOM, 2013 Proceedings IEEE*, pages 2400–2408. IEEE, 2013.
57. Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.
58. Zhi Yang, Christo Wilson, Xiao Wang, Tingting Gao, Ben Y Zhao, and Yafei Dai. Uncovering social network sybils in the wild. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(1):2, 2014.
59. Haifeng Yu. Sybil defenses via social networks: a tutorial and survey. *ACM SIGACT News*, 42(3):80–101, 2011.
60. Haifeng Yu, Phillip B Gibbons, Michael Kaminsky, and Feng Xiao. Sybillimit: A near-optimal social network defense against sybil attacks. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 3–17. IEEE, 2008.
61. Haifeng Yu, Michael Kaminsky, Phillip B Gibbons, and Abraham Flaxman. Sybilguard: defending against sybil attacks via social networks. *ACM SIGCOMM Computer Communication Review*, 36(4):267–278, 2006.

62. C. Zhang, M. Dong, K. Ota, and M. Guo. A social-network-optimized taxi-sharing service. *IT Professional*, 18(4):34–40, July 2016.
63. Jiahao Zhang, Qiang Li, Xiaoqi Wang, Bo Feng, and Dong Guo. Towards fast and lightweight spam account detection in mobile social networks through fog computing. *Peer-to-Peer Networking and Applications*, Jun 2017.
64. D. Zheng, R. Burns, and A. S. Szalay. Toward millions of file system iops on low-cost, commodity hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 69:1–69:12, New York, NY, USA, 2013. ACM.
65. D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 45–58, 2015.
66. X. Zhu, W. Han, and W. Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference*, USENIX ATC'15, pages 375–386, Santa Clara, CA, 2015.
67. Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA*, 2016.