QuickPoint: Efficiently Identifying Densest Sub-Graphs in Online Social Networks for Event Stream Dissemination

Hai Jin, *Fellow, IEEE*, Changfu Lin, *Student Member, IEEE*, Hanhua Chen, *Member, IEEE*, and Jiangchuan Liu, *Fellow, IEEE*

Abstract—Efficient event stream dissemination is a challenging problem in large-scale *Online Social Network* (OSN) systems due to the costly inter-server communications caused by the per-user view data storage. To solve the problem, previous schemes mainly explore the structures of social graphs to reduce the inter-server traffic. Based on the observation of high cluster coefficients in OSNs, a state-of-the-art social piggyback scheme can save redundant messages by exploiting an intrinsic hub-structure in an OSN graph for message piggybacking. Essentially, finding the best hub-structure for piggybacking is equivalent to finding a variation of the densest sub-graph. The existing scheme computes the best hub-structure by iteratively removing the node with the minimum weighted degree. Such a scheme incurs a worst computation cost of $O(n^2)$, making it not scalable to large-scale OSN graphs. Using alternative hub-structure instead of the best hub-structure can speed up the piggyback assignment. However, they greatly sacrifice the communication efficiency of the assignment schedule. Different from the existing designs, in this work, we propose a QuickPoint algorithm, which removes a fraction of nodes in each iteration in finding the best hub-structure. We mathematically prove that QuickPoint converges in $O(log_a n)(a > 1)$ iterations in finding the best hub-structure for efficient piggyback. We implement QuickPoint in parallel atop Pregel, a vertex-centric distributed graph processing platform. Comprehensive experiments using large-scale data from Twitter and Flickr show that our scheme is 38.8× more efficient compared to existing schemes.

Index Terms—Event stream dissemination, piggyback, densest sub-graph, online social networks

1 Introduction

Since the emergency of *Online Social Network* (OSN) applications [16], [27], [39], [44] in the last decade, such as Facebook, Twitter, and Tumblr, billions of people have started to use OSNs for information sharing through the social links [6], [13], [24], [28], [47]. In OSNs, relevant data of a user is the data of her own and that of her neighbors (e.g., followers' tweets, friends' status updates, etc.). For sharing the relevant data of a user, an OSN system provides two basic operations for a user: 1) sharing events among her friends, such as short text messages, photos, and videos; and 2) browsing the event stream, a list of recent events shared by her friends. In popular OSNs, event stream browsing produces a majority proportion of requests (e.g., 70 percent of the page views of Tumblr [1]), dominating the workload of the systems. Popular OSN systems commonly use the *materialized view* [14], where views for assembling event streams are formed based on a per-user style. A user's view contains the events generated by herself and those shared by her friends. Due to the complex user interconnections, event stream dissemination in large-scale OSNs incurs costly inter-server communications across datacenters [26], [30]. The problem becomes particularly acute under heavy datacenter loads [36], [37].

To support event stream dissemination, a straightforward scheme is to leverage a push or a pull based strategy. A push-based scheme updates a user's events, whenever generated, to all her friends' views. For example, in Fig. 1a, the user A sends a newly generated event to her friends B, C, and D. With the push-based strategy, a system achieves local semantic, i.e., a user can read the events shared by her friends in her own view [36]. Such a scheme is efficient for read-dominating workloads, i.e., an event data is written once and read frequently. In contrast, a pull-based strategy only writes a user's newly generated events to her own view. The events are subsequently pulled by her friends on demand. For example, in Fig. 1b, the user C fetches the recent events shared by her friends A and D, whenever C browses the event stream. The pull-based strategy reduces the cost of write during event stream dissemination, and thus is especially efficient for a user with frequent event writes but followed by rare reads. However, the event stream browsing could be costly. By combining both strategies, a hybrid scheme [38] assigns either the push or the pull strategy to each social link according to how frequently the pair of end users of the social link generate/browse events between each other. For example, in Fig. 1c, the scheme assigns the push strategy to the link $A \rightarrow B$, as B

- *H. Jin, C. Lin, and H. Chen are with Big Data Technology and System Lab, Services Computing Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {hjin, lcf, chen}@hust.edu.cn.*
- *J. Liu is with the School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada. E-mail: jcliu@cs.sfu.ca.*

Manuscript received 1 June 2018; revised 22 Oct. 2018; accepted 1 Nov. 2018. Date of publication 15 Nov. 2018; date of current version 8 Jan. 2020. (Corresponding author: Hanhua Chen.) Recommended for acceptance by W. Zhang. Digital Object Identifier no. 10.1109/TKDE.2018.2881435

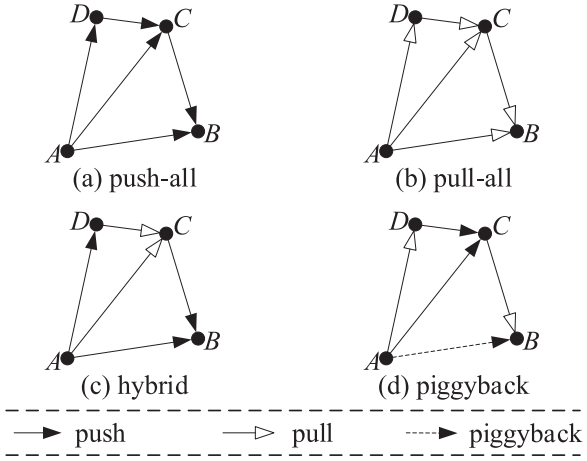


Fig. 1. Event dissemination strategies in OSNs.

browses events more frequently than A generating events. Meanwhile, the scheme assigns the pull strategy to the link $A \rightarrow C$, as A generates events more frequently than C browsing events. The hybrid scheme achieves better efficiency by precisely assigning the push or the pull strategy to each social link according to the actual frequency of event dissemination among users.

Recently, Gionis et al. [18] argue that the above per link assignment is not necessary. They show that the communications along some links can be saved while the event dissemination can be successfully achieved. Consider a simple example in Fig. 1d, if we assign the push strategy to the link $A \rightarrow C$ and pull to $C \rightarrow B$, we can save the messages from A to B, by piggybacking the messages from A to B onto the messages from A to C. Whenever B fetches the updates from C, B can also obtain the recent updates of A through C. In short, with C acting as a hub node, B can obtain the recent updates from A with no messages transmitted through $A \rightarrow B$. With such a piggyback strategy, a potential large amount of redundant messages can be saved, as a recent research shows that the social graphs usually have very high cluster coefficient [42] (i.e., the high possibility that a user's two friends are also friends) and such a triangle relation among A, B, and C in Fig. 1d is common.

Using the piggyback strategy is not difficult. However, how to fully exploit links for piggybacking to minimize the communication cost over OSNs is not trivial. Finding the best piggyback assignment is extremely difficult due to the enormous solution space of assigning the push, pull, or piggyback strategies across all links of the large-scale social graph. Gionis et al. prove that finding the best piggyback assignment is NP-hard [18]. To solve the problem, they design CHITCHAT, a greedy algorithm based on a *hub-structure* centred on each user (We will review the detail of the hub-structure in Section 3). A good hub-structure should leverage as many links as possible for piggybacking to minimize the inter-server traffic incurred by users' events dissemination. Gionis et al. show that finding the defined best hub-structure is equivalent to computing a variation of the densest sub-graph [12]. The CHITCHAT algorithm computes the best hub-structure by removing the node with the minimum *weighted degree* iteratively, requiring a computation cost of $O(n^2)$ [18] in worst case. To avoid the prohibitively costly computation for finding the best hub-structure, Gionis et al. construct an alternative hub-structure. They accordingly propose the PARALLELNOSY [18] algorithm, which can efficiently compute the alternative structure with an edge-centric parallel implementation using MapReduce [15]. However, their results show that the communication cost obtained by using the alternative structure greatly increases by 58 percent compared to that obtained by finding the best hub-structure.

To solve the problem, in this work, we propose the QuickPoint algorithm, which efficiently computes a densest hub-structure by allowing each iteration to remove a fraction of nodes. We mathematically prove that QuickPoint has an upper bound of the number of iterations as $O(log_a n)(a > 1)$, where a is a constant scaling the convergence rate of finding the densest hub-structure. We implement QuickPoint in parallel using Pregel [31], a vertex-centric distributed graph processing platform. We conduct comprehensive experiments to evaluate the performance of our design using large-scale traces from real-world social network systems. Results show that our scheme achieves a 38.8× improvement in efficiency compared to existing schemes.

All in all, the contributions of this work are threefold.

- We propose a novel event stream dissemination algorithm by quickly detecting the best hub-structure for efficient piggyback in OSNs.
- We mathematically prove that the upper bound of the number of iterations of our algorithm for identifying the best hub-structure is $O(log_a n)(a > 1)$.
- We implement the algorithm in parallel on top of Pregel, a real-world distributed graph processing platform.

The rest of the paper is structured as follows. Section 2 discusses the related work. Section 3 describes the preliminaries of the hub-structures. Section 4 presents our algorithm in detail. Section 5 introduces the parallel implementation. Section 6 presents how this design copes with the dynamic updates of social graphs. Section 7 evaluates the performance of our design. Section 8 concludes the paper.

2 Related Work

Event stream dissemination occupies a major part of the workloads in popular OSN systems [23], [46]. Recently, the problem of how to efficiently disseminate event streams across online social networks has attracted a lot of research interest [4], [18], [32], [38]. Existing schemes can be classified into two types: link-based [4], [38] and structure-based [18].

The link-based schemes assign each social link an event dissemination strategy, i.e., push or pull, and can be classified into three sub-types including push-all, pull-all, and hybrid. A push-all scheme assigns all links of the social graph a push strategy [36]. With the push-all scheme, an event newly generated by a user is pushed to all her friends' views. Thus, relevant events (e.g., news feed or followers' tweets) can be efficiently read from a user's own view when she browses the event stream in a real time style. The problem of the push-all scheme is the high cost of write during event stream dissemination. On the contrary, a pull-all scheme assigns each social link a pull strategy. With this scheme, a user's events are only written to her own view. When a user browses the event stream, the system pulls all the events data from her friends' views on demand. Such a scheme reduces the write cost. It is inefficient for systems with read dominating workloads [34].

The key issue of the link-based scheme is how to choose a pull or a push strategy for different social links. Silberstein et al. [38] present the Feeding Frenzy system, and propose a hybrid scheme, which selects a push or a pull strategy for a social link according to how frequently the two users at both ends of the social link share or browse events through the social link. They measure the statistical workloads of each user. The statistical information of event stream workloads of a user includes a production rate and a consumption rate, representing the average frequency a user generating events and browsing the event stream, respectively. By comparing the rates of production and consumption at the two ends of a social link, they adaptively select the preferable strategy for the link. For example, given a social link $u \rightarrow v$, they assign the push strategy to $u \rightarrow v$ if the production rate of u is smaller than the consumption rate of v; otherwise they assign the pull strategy to $u \rightarrow v$.

Bao et al. [4] present the GeoFeed system, a location-aware news feed system that follows the hybrid design of the Feeding Frenzy system [38]. GeoFeed extends Feeding Frenzy by enabling users to share events with spatial extent and considering their locations when disseminating news feed for them. Compared to the event stream dissemination addressed in this paper, the problem solved by Bao et al. [4] is quite different.

Another kind of schemes are based on an unique structure feature of OSNs, i.e., the high clustering coefficient [42]. The scheme can save redundant messages through a social link using a piggyback strategy. Gionis et al. [18] show that a greedy heuristic to fully exploit links for piggybacking needs to compute the best hub-structure through all the users repeatedly. Moreover, finding the best hub-structure for each user takes a costly computation time of $O(n^2)$ in the worst case, making such a scheme not scalable to large-scale OSN graphs. Although turning to alternative hub-structures can speed up the process, they significantly increase the communication cost of event stream dissemination [18], due to the sacrifice of the quality of the obtained piggyback structure. In this work, we propose a novel efficient algorithm to quickly find an approximate best hub-structure, which improves the quality of the obtained piggyback structure while greatly accelerates the convergence speed of the algorithm. We implement our algorithm in parallel on top of Pregel [31].

Mondal et al. [32] present EAGr, a system supports continuous ego-centric aggregate queries over the large-scale social graph. Mondal et al. address the problem of reducing the communication cost for a given ego-centric query among a subset of users in a social graph. In this work, our design addresses a different problem of minimizing the communication cost for event stream dissemination among all users in a social graph.

3 Densest Sub-Hub-Structure for Piggyback

In this section, we formalize the event stream dissemination problem to fully exploit links for piggybacking throughout the network, following the framework of Ref. [18].

An OSN user commonly consumes an order of magnitude more content than what she generates [34]. This results in reading dominated workloads in an OSN system [8]. Therefore, popular OSN systems commonly rely on in-memory distributed key-value stores, e.g., Memcache in Facebook [34], [37]. On top of the in-memory stores, event stream dissemination is based on the mechanism of materialized per-user view [18]. In OSNs, the relevant events of a user are the events shared by her friends and the events generated by herself.

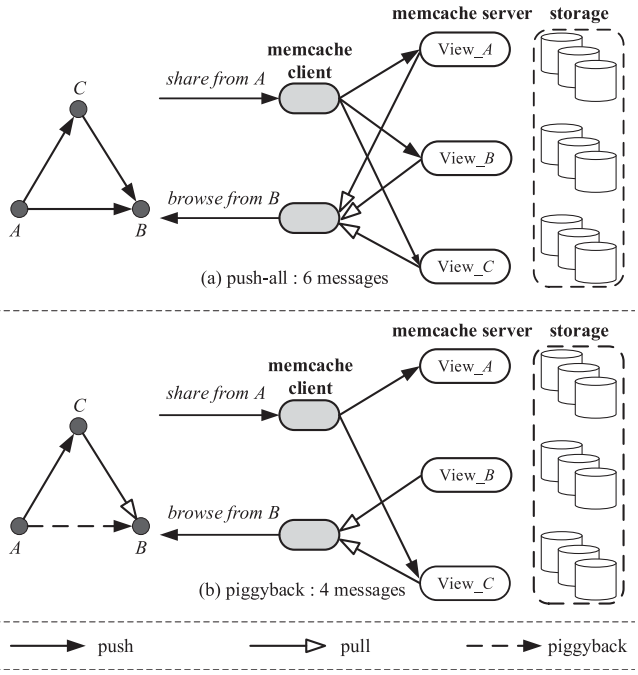


Fig. 2. Piggyback over OSNs.

Fig. 2 shows an example of the request flows of the event stream in an OSN system, where the views of the three users A, B, and C (the social links among them form a triangle structure consisting of links $A \rightarrow B$, $A \rightarrow C$, and $C \rightarrow B$) are distributed to different servers (e.g., Memcache servers) using hash partitioning. In the figure, we consider a simple case that A shares a new event and B browses the relevant event stream. With the traditional push-all scheme (see Fig. 2a), which assigns each social link a push strategy, the system produces a total number of six messages between the Memcache clients and servers. In contrast, Fig. 2b illustrates the piggyback scheme, which assigns the push strategy to the link $A \rightarrow C$ and pull to $C \rightarrow B$. It is clear that only four messages are needed. Due to the fact that OSNs commonly feature a high cluster coefficient [42], the triangle structure in the above example is a common case in a social graph. Therefore, carefully designing the assignments for those triangle structures following the piggyback strategy has a great potential to reduce a large amount of communication cost during event stream dissemination.

In this paper, we narrow down our study to social graphs, in which we can identify a large number triangle structures due the high cluster coefficient nature. Before giving the problem definition, we introduce two important concepts: *piggybacked* and *piggyback assignment*. For simplicity, we use the notations H and L to denote the sets of push links and pull links, respectively. Table 1 lists the notations used in the problem statement.

Definition 1 (Piggybacked). *Given a social graph $G(V, E)$, we call a link $u \rightarrow v \in E$ is piggybacked by a hub node w if there exists a node $w \in V$ such that $u \rightarrow w \in H$ and $w \rightarrow v \in L$.*

TABLE 1
Notations

Notations	Description
H	the set of push links
L	the set of pull links
(H, L)	a piggyback assignment
$r_p(u)$	the production rate of u
$r_c(v)$	the consumption rate of v
$g(u)$	the weight of u
$c(H, L)$	the communication cost of (H, L)
$G(X, w, Y)$	a hub-structure centred on w
$E(X, w, Y)$	the set of links in $G(X, w, Y)$
$g(X, w, Y)$	the total weight of nodes in $X \cup Y$
$G(X_w, w, Y_w)$	the largest hub-structure of w
$G(X^*_w, w, Y^*_w)$	the densest sub-hub-structure of w
$G(X^*_{w*}, w^*, Y^*_{w*})$	the global densest sub-hub-structure

Definition 2 (Piggyback assignment). *Given a social graph $G(V, E)$, for each link $u \to v \in E$, a piggyback assignment (H, L) must satisfies that either: 1) $u \to v \in H$, or 2) $u \to v \in L$, or 3) $u \to v$ is piggybacked by a hub node $w \in V$.*

In the following, we discuss how to compute the communication cost denoted by $c(H, L)$ for a given piggyback assignment (H, L). Indeed, $c(H, L)$ is the total communication cost generated by all the push links in H and all the pull links in L. The communication cost of each push or pull link $u \to v$ depends on the statistical workloads of both the users u and v. Formally, the statistical workloads of a user u can be modeled using two rates, the production rate $r_p(u)$ and the consumption rate $r_c(u)$. The production rate $r_p(u)$ represents the average frequency that user u generates events, while the consumption rate $r_c(u)$ quantifies the average frequency that user u browses the event stream. It is not difficult to see that the communication cost through a push link $u \to v$ is $r_p(u)$, because the user u pushes a new event to v's view whenever u produces the event. Similarly, the communication cost through a pull link $u \to v$ is $r_c(v)$, because the user v pulls u's events to v's view whenever v browses the event stream. Thus, we compute the communication cost $c(H, L)$ by Eq. (1),

$$c(H, L) = \sum_{u \to v \in H} r_p(u) + \sum_{u \to v \in L} r_c(v). \tag{1}$$

Based on Eq. (1), we present the event stream dissemination problem as below.

Problem 1 (Event stream dissemination problem). *Given a social graph $G(V, E)$, the production rate $r_p(u)(u \in V)$ and the consumption rate $r_c(u)(u \in V)$, find a piggyback assignment (H, L) that minimizes the communication cost $c(H, L)$.*

Due to the enormous solution space of the piggyback assignment that assigns the push, the pull, or the piggyback strategy across all links of the large-scale social graph, solving the event stream dissemination problem is extremely difficult over the large-scale social graph.

Let's take a closer look at the example shown in Fig. 2b. The node C works as a hub saving the messages through the link $A \to B$. In the entire social graph, some other links can also be piggybacked by the hub node C. Obviously, we can abstract a dedicated piggyback structure, a hub-structure centred on C. More generally, Fig. 3 shows an example of such a hub-structure $G(X, w, Y)$. In the hub-structure, w is the hub node; $X = \{x_1, x_2, \ldots, x_p\}$ is the set of users whose events are consumed by w; and $Y = \{y_1, y_2, \ldots, y_q\}$ is the set of users who consume w's events. By assigning each link $x \to w$ $(x \in X)$ a push strategy and each link $w \to y$ $(y \in Y)$ a pull strategy, all the messages through links from X to Y (marked as dotted lines in Fig. 3) can be saved. For example, in Fig. 3, node y_4 can achieve updates from x_2 and x_3 by pulling those updates from w, while it is not necessary for x_2 and x_3 to send their updates to y_4 through the links $x_2 \to y_4$ and $x_3 \to y_4$.

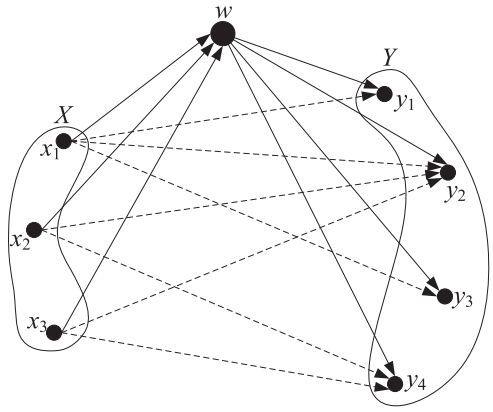


Fig. 3. Hub-structure for piggyback.

It is not difficult to see that saving messages from X to Y is only one side of the issue of reducing communications. On the other side, as aforementioned, due to the actual workloads among users, assigning all links $x \to w$ $(x \in X)$ and $w \to y$ $(y \in Y)$ using the above policy may still be inefficient. Considering the communication cost of the hub-structure shown in Fig. 3, a user $x \in X$ has a weight $g(x) = r_p(x)$ because the communication cost through the push link $x \to w$ is $r_p(x)$. Similarly, a user $y \in Y$ has a weight $g(y) = r_c(y)$ because the communication cost through the pull link $w \to y$ is $r_c(y)$. For simplicity, the user w has a weight of zero. Intuitive, a good hub-structure should reduce the communication cost from X to Y as much as possible, while achieve the total weight of all links $x \to w$ $(x \in X)$ and $w \to y$ $(y \in Y)$ as small as possible. Mathematically, the following density function computes the quality of a hub-structure $G(X, w, Y)$,

$$D(X, w, Y) = \frac{|E(X, w, Y)|}{g(X, w, Y)}, \tag{2}$$

where $E(X, w, Y)$ denotes the set of links including $x \to w$ for each $x \in X$, $w \to y$ for each $y \in Y$, and $x \to y$; $g(X, w, Y)$ denotes the total weight of all nodes in $X \cup Y$.

For simplicity, we use the notion $G(X_w, w, Y_w)$ to denote the largest hub-structure centred on w. It is not difficult to find that $X_w = \{x \,|\, x \to w \in E\}$ and $Y_w = \{y \,|\, w \to y \in E\}$. Thus, each hub-structure $G(X, w, Y)$ is a sub-graph [5], [21], [29], [41], [43] of $G(X_w, w, Y_w)$, i.e., $X \subset X_w$ and $Y \subset Y_w$. Through all the possible hub-structures $G(X, w, Y)$, we call the hub-structure that has the maximum density value of Eq. (2) as the densest sub-hub-structure of $G(X_w, w, Y_w)$.

Definition 3 (Densest sub-hub-structure). *Given the largest hub-structure $G(X_w, w, Y_w)$ centred on the hub node w, its densest sub-hub-structure $G(X^*_w, w, Y^*_w)$ holds that $(X^*_w, Y^*_w) = \arg\max_{X \subset X_w, Y \subset Y_w} D(X, w, Y)$.*

For a given social graph $G(V, E)$, each node $w \in V$ has its own densest sub-hub-structure $G(X^*_w, w, Y^*_w)$. Through all the densest sub-hub-structures $G(X^*_w, w, Y^*_w)$, we call the one with the maximum density value of Eq. (2) as the global densest sub-hub-structure.

Definition 4 (Global densest sub-hub-structure). *Given a social graph $G(V, E)$, its global densest sub-hub-structure $G(X^*_{w^*}, w^*, Y^*_{w^*})$ centred on the hub node w^* holds that $w^* = \arg\max_{w \in V} D(X^*_w, w, Y^*_w)$.*

The basic idea of the CHITCHAT algorithm [18] is to find the global densest sub-hub-structure in each iteration and reasonably assign all the links of the global densest sub-hub-structure using piggyback principle. Thus, each iteration of the greedy algorithm can leverage as many links as possible for piggybacking and achieve the communication cost of the piggyback assignment as small as possible.

Algorithm 1. CHITCHAT

Input: The social graph $G(V, E)$, the production rate $r_p(u)$ and the consumption rate $r_c(u)$ for all $u \in V$.
Output: The piggyback assignment (H, L).

1: $H \leftarrow \varnothing, L \leftarrow \varnothing, Z \leftarrow E$; // Z: the uncovered link set
2: **For** each $w \in V$
3: form largest densest sub-hub-structure $G(X_w, w, Y_w)$;
4: $(X^*_w, Y^*_w) \leftarrow$ densestSubHubStructure(w, H, L, Z);
5: **do**
6: select $G(X^*_{w^*}, w^*, Y^*_{w^*})$ from all $G(X^*_w, w, Y^*_w)$;
7: $H \leftarrow H \cup \{u \rightarrow w^* | u \in X^*_{w^*}\}$;
8: $L \leftarrow L \cup \{w^* \rightarrow v | v \in Y^*_{w^*}\}$;
9: $Z \leftarrow Z \backslash E(X^*_{w^*}, w^*, Y^*_{w^*})$;
10: **For** each $G(X_w, w, Y_w)$ with links in $E(X^*_{w^*}, w^*, Y^*_{w^*})$
11: update links in $E(X^*_{w^*}, w^*, Y^*_{w^*})$ as ASSIGNED;
12: $(X^*_w, Y^*_w) \leftarrow$ densestSubHubStructure(w, H, L, Z);
13: **while** $Z = \varnothing$
14: **return** (H, L).

Algorithm 1 introduces the CHITCHAT algorithm in detail. The algorithm is a typical serial program. It first forms the largest hub-structure $G(X_w, w, Y_w)$ and computes its densest sub-hub-structure $G(X^*_w, w, Y^*_w)$ for each node $w \in V$ (lines 1-2). Then, it greedily selects the global densest sub-hub-structure $G(X^*_{w^*}, w^*, Y^*_{w^*})$ from all $G(X^*_w, w, Y^*_w)$ (line 6). After selected, the algorithm makes the following assignments (lines 7-8): the push strategy to all the links $x \rightarrow w^*$ ($x \in X^*_{w^*}$) and the pull strategy to $w^* \rightarrow y$ ($y \in Y^*_{w^*}$). Thus, each link $x \rightarrow y$ if exists can be piggybacked by the hub node w^*. All the links in $E(X^*_{w^*}, w^*, Y^*_{w^*})$ will be removed from the uncovered link set Z (line 9). For each hub-structure $G(X_w, w, Y_w)$ which contains links in $E(X^*_{w^*}, w^*, Y^*_{w^*})$, the algorithm will mark the status of those contained links as ASSIGNED. These ASSIGNED links will not be counted in $E(X, w, Y)$ (see Eq. (2)) when computing the densest sub-hub-structure in the subsequent iterations. The algorithm will end when Z becomes empty and return the piggyback assignment (H, L).

It is not difficult to find that the main computation cost of CHITCHAT is spent on repeatedly computing the densest sub-hub-structure. The CHITCHAT algorithm achieves the *densestSubHubStructure* function by greedily removing the node with the minimum *weighted degree* $d(u)/g(u)$ [18] until the remaining hub-structure becomes empty. Here, $d(u)$ represents the degree of node u. During the process, the sub-hub-structure which maximizes the density value of Eq. (2) is used as the approximate densest sub-hub-structure. However, such a process will take a worst computation cost of $O(n^2)$, making the CHITCHAT algorithm not scalable for large-scale OSN graphs. Thus, it is unrealistic to implement the CHITCHAT algorithm on top of a typical distributed graph processing system, such as Pregel.

4 Algorithm

In this section, we first introduce the QuickPoint algorithm, which can efficiently find an approximate densest sub-hub-structure of a given hub-structure $G(X_w, w, Y_w)$. Then we formally prove that the upper bound of the number of iterations of QuickPoint is $O(log_a n)(a > 1)$, the time complexity of QuickPoint is $O(|E(X_w, w, Y_w)|/k)$, and the approximation of QuickPoint is $2a$.

4.1 QuickPoint Algorithm

As aforementioned, the expensive computation cost of finding the densest sub-hub-structure in CHITCHAT [18] is due to the operation of removing the node with minimum *weighted degree* $d(u)/g(u)$ [18] in each iteration, where $d(u)$ and $g(u)$ represent the degree and communication cost of node u, respectively. For a hub-structure with n nodes, the CHITCHAT [18] algorithm needs a number of n iterations with a worst computation cost of $O(n^2)$ to find the densest sub-hub-structure. To address the problem, we propose the QuickPoint algorithm, which can quickly find the densest sub-hub-structure of a given hub-structure within $O(log_a n)(a > 1)$ iterations. Note that, in this paper, the QuickPoint algorithm is dedicated to reduce the communication cost of disseminating event stream for OSN applications. We do not aim to design a common approach to quickly find the densest sub-hub-structure for all applications. One can modify the QuickPoint algorithm for different applications where computing densest sub-hub-structure dominating the computation cost.

The key motivation of the QuickPoint algorithm is to remove a fraction of nodes in each iteration to accelerate the process of finding the densest sub-hub-structure. To achieve it, in each iteration, it is critical to choose an appropriate threshold that considers both the structure of current sub-hub-structure and the communication cost of nodes. A straightforward approach is to use the average weighted degree, i.e., $\sum_{u \in G} d(u) / \sum_{u \in G} g(u)$. Note that G denotes the current sub-hub-structure $G(X, w, Y)$ for simplicity. Such a threshold can guarantee the convergence of the algorithm because at least one node will be removed, e.g., the node with the minimum weight degree. To further accelerate the process of the QuickPoint algorithm, we multiply the above threshold by a constant $a(a > 1)$. It is not difficult to obtain that $\sum_{u \in G} d(u) = 2|E(X, w, Y)|$ according to graph theory knowledge. Thus, we can simplify the threshold as $2aD(X, w, Y)$. Intuitively, a large value of a will make the QuickPoint coverage quickly. However, it may incur a potential large accuracy lose in the communication cost (see Eq. (1)) due to the coarse-grained process of removing nodes. Since the CHITCHAT algorithm contains a series rounds of computing the densest sub-hub-structure, it is extremely difficult to predict the relationship between the communication cost and the constant a. According to the

experiment results, we recommend to choose a constant a between 1.2 and 1.5, which can achieve a short running time while obtain an accuracy loss within 5 percent.

Algorithm 2. QuickPoint

Input: The hub-structure $G(X_w, w, Y_w)$, and the weight $g(u)$ for each node u in $X_w \cup Y_w$.
Output: The set of nodes of the approximate densest sub-hub-structure.

1: $S \leftarrow X_w \cup \{w\} \cup Y_w$;
2: $S^* \leftarrow S$; /* S^*: the set of nodes of the approximate densest sub-hub-structure */
3: **while** $S \neq \varnothing$ **do**
4: $\rho(S) \leftarrow |E(S)|/g(S)$; /*$\rho(S)$: the density of the current sub-hub-structure*/
5: $X' \leftarrow \{x \,|\, x \in S \cap X_w, d_S(x)/g(x) \leq 2a\rho(S)\}$;
6: $Y' \leftarrow \{y \,|\, y \in S \cap Y_w, d_S(y)/g(y) \leq 2a\rho(S)\}$;
7: $R \leftarrow X' \cup Y'$;
8: **if** $\rho(S) > \rho(S^*)$ **then**
9: $S^* \leftarrow S$;
10: **end if**
11: $S \leftarrow S \backslash R$;
12: **end do**
13: **return** S^*.

Algorithm 2 presents the process of QuickPoint in detail. Given a hub-structure $G(X_w, w, Y_w)$ with $g(u)$ denoting the weight of each node $u \in X_w \cup Y_w$. Let S denote the set of nodes of the current sub-hub-structure and begin with $X_w \cup \{w\} \cup Y_w$ (line 1). For a given constant $a > 1$, the algorithm computes the density of the current sub-hub-structure by $\rho(S) = |E(S)|/g(S)$ in each iteration (line 4), where $|E(S)|$ denotes the total number of links in the current sub-hub-structure, and $g(S)$ denotes the total weight of all nodes in S. For simplicity, we use the notation $d_S(u)$ to denote the degree of the node in the sub-hub-structure generated by all nodes in the set S. The algorithm selects all nodes with *weighted degree* [18] smaller than the threshold $2a\rho(S)$ (lines 5∼6). Thus we obtain $R = \{u \in S \,|\, d_S(u)/g(u) \leq 2a\rho(S)\}$ (line 7), the set of nodes to be removed in the current iteration. Then, the algorithm removes all the nodes in R from S (line 11), as well as all the links associated with them. Such an operation repeats until S becomes empty. Finally, the algorithm returns S^* (line 13), the set of nodes of the approximate densest sub-hub-structure that maximizes the density through all the sub-hub-structures during the process.

4.2 Upper Bound of Iterations of QuickPoint

We give the formal proof of the upper bound of the number of iterations of QuickPoint following the analysis of Bahmani [3]. Given an unweighed undirected graph, Bahmani et al. use a greedy algorithm to find an approximate densest sub-graph. Differently, the density in [3] is defined as half of the average degree of the graph. However, we prove the upper bound of QuickPoint in the directed hub-structure, where each node has a weight (see Section 3).

Theorem 1. *Given a hub-structure $G(X_w, w, Y_w)$ and a constant $a > 1$, the upper bound of the number of iterations of QuickPoint is $O(log_a n)(n = |X_w \cup Y_w|)$.*

Proof. For simplicity, we use the notation S to denote the set of nodes in the current sub-hub-structure. It is not difficult to see $2|E(S)| = \sum_{u \in S} d_S(u)$, which can be simplified as

$$2|E(S)| = \sum_{u \in R} d_S(u) + \sum_{u \in S \backslash R} d_S(u), \tag{3}$$

where R denotes the set of nodes to be removed in the current iteration, i.e., $R = \{u \in S | d_S(u)/g(u) \leq 2a\rho(S)\}$. □

In the following, we first prove that $|R| \geq 1$, which means at least one node will be removed in the current iteration. For simplicity, we use the notation u^* to denote the node with minimum weighted degree $d_S(u)/g(u)$. Thus, for each node $u \in S$, we have that $d_S(u)/g(u) \geq d_S(u^*)/g(u^*)$. By summing up the degree of all $u \in S$, we obtain that $\sum_{u \in S} d_S(u) \geq d_S(u^*)/g(u^*) \sum_{u \in S} g(u)$, which can be further simplified as $d_S(u^*)/g(u^*) \leq \sum_{u \in S} d_S(u) / \sum_{u \in S} g(u) = 2\rho(S) < 2a\rho(S)$. So far, we show that node u^* will be removed in the current iteration, i.e., $u^* \in R$. Thus, $|R| \geq 1$.

According to the definition of R, for each node $u \in S \backslash R$, we have that $d_S(u)/g(u) > 2a\rho(S)$. This equals to $d_S(u) > 2a\rho(S)g(u)$. Furthermore, we can change Eq. (3) as $2|E(S)| > \sum_{u \in S \backslash R} d_S(u) > 2a\rho(S) \sum_{u \in S \backslash R} g(u)$, which can be simplified as

$$\sum_{u \in S \backslash R} g(u) < \left(\sum_{u \in S} g(u) \right) \Big/ a. \tag{4}$$

Indeed, the above equation demonstrates that, after an iteration, the total weight of all the remaining nodes is at most a fraction of the total weight of all nodes in current sub-hub-structure. Thus, the QuickPoint algorithm will end when the total weight of all the remaining nodes is less than the minimum node weight, denoted by $g_{min} = \min_{u \in X_w \cup Y_w} g(u)$. At the begin, the hub-structure $G(X_w, w, Y_w)$ has a total weight of all nodes as $g_{total} = \sum_{x \in X_w} g(x) + \sum_{y \in Y_w} g(y) \leq n * g_{max}$. Here, g_{max} denotes the maximum node weight, i.e., $g_{max} = \max_{u \in X_w \cup Y_w} g(u)$. Therefore, the QuickPoint terminates within a number of iterations of $log_a((g_{max}/g_{min})n)$. Since g_{max}/g_{min} is a constant for $G(X_w, w, Y_w)$, QuickPoint terminates in $O(log_a n)$ iterations. Thus proved.

It is not difficult to find that the skewness between the weight value of nodes, i.e., g_{max}/g_{min}, has an important impact on the convergence rate of the QuickPoint algorithm. Thus, QuickPoint may not be very suitable for hub-structure with highly skewed weight value of nodes.

4.3 Time Complexity of QuickPoint

In this section, we discuss the time complexity of QuickPoint. The main running time of QuickPoint is spent on finding the nodes to delete and updating the degree of the remaining nodes for each iteration. It is not difficult to find that we can achieve both: 1) finding the selected nodes for each iteration and 2) updating the degree of the remaining nodes in parallel. For simplicity, we use the notion k to denote the parallelism degree for these two operations.

Theorem 2. *Given a hub-structure $G(X_w, w, Y_w)$ and the parallelism degree k, the time complexity of QuickPoint is $O(|E(X_w, w, Y_w)|/k)$.*

Proof. For simplicity, we use the notation $d_G(u)$ to denote the degree of node u in $G(X_w, w, Y_w)$. It is not difficult to

find that the degree of node u in any sub-hub-structure generated by QuickPoint is no more than $d_G(u)$. □

According to lines 5 and 6 of the Algorithm 2, it is simple to obtain the nodes to delete in current iteration. Thus, we focus on accounting the running time to update the degree of the remaining nodes. To further improve the running time performance of QuickPoint, we maintain lists of nodes with the same weighted degree. In each iteration, we only need to update the degree of neighbor nodes of the deleted nodes.

For each deleted node u, it incurs a worst updating time of $d_G(u)/k$ because we can update the degree of u's neighbor nodes in parallel. Thus, the running time of QuickPoint is at most $\sum_{u \in G} d_G(u)/k = 2|E(X_w, w, Y_w)|/k$. Thus proved.

4.4 Approximation of QuickPoint

In this section, we give the formal proof of the achieved approximation of QuickPoint. For simplicity, we use the notations W^* to denote the node set of the densest sub-hub-structure $G(X_w^*, w, Y_w^*)$, i.e., $W^* = \{X_w^* \cup w \cup Y_w^*\}$; and $\rho(W^*)$ to denote the density value (see Eq. (2)), i.e., $\rho(W^*) = D(X_w^*, w, Y_w^*)$.

Theorem 3. *Given a hub-structure $G(X_w, w, Y_w)$ and a constant $a > 1$, QuickPoint achieves a $2a$-approximation of the densest sub-hub-structure $G(X_w^*, w, Y_w^*)$, i.e., $\rho(S^*) \geq \rho(W^*)/2a$.*

Proof. Since W^* is the set of nodes in the optimal densest sub-hub-structure, for each node $u \in W^*$, we have that $\rho(W^*) \geq \rho(W^* \backslash \{u\})$, which can be represented as

$$\frac{|E(W^*)|}{g(W^*)} \geq \frac{|E(W^*)| - d_{W^*}(u)}{g(W^*) - g(u)}. \tag{5}$$

□

We can simplify the above equation as $\rho(W^*) = |E(W^*)|/g(W^*) \leq d_{W^*}(u)/g(u)$. Since QuickPoint removes at least one node, there must exist an iteration in which a node $u \in W^*$ is first removed. We assume S is the set of nodes of the sub-hub-structure formed in the selected iteration, and R is the set of nodes removed in the selected iteration. It is clear that $u \in R \cap W^*$, and $W^* \subseteq S$. Since $W^* \subseteq S$, we can obviously have $d_{W^*}(u) \leq d_S(u)$. So far, we have $\rho(W^*) \leq d_S(u)/g(u)$. With the definition of R, we have $d_S(u)/g(u) \leq 2a\rho(S)$. Thus, we have $\rho(W^*) \leq 2a\rho(S)$. It is not difficult to see that $\rho(S) \geq \rho(W^*)/2a$.

Since S^* is the set of nodes of the approximate densest sub-hub-structure that maximizes the density through all the sub-hub-structures generated during the process of QuickPoint, we have that $\rho(S^*) \geq \rho(S) \geq \rho(W^*)/2a$. So far, we prove that QuickPoint guarantees a $2a$-approximation of the optimal densest sub-hub-structure.

5 Parallel Implementation

5.1 Overview

To solve the event stream dissemination problem, a straightforward approach is to apply the QuickPoint algorithm to implement the densestSubHubStructure function of CHITCHAT (see lines 4 and 12 in Algorithm 1). It is not difficult to see that the process of finding the densest sub-hub-structure for each user is inherently vertex-centric, we can greatly improve the scalability of CHITCHAT by implementing the QuickPoint algorithm on top of a vertex-centric graph processing platform to solve the event stream dissemination problem.

Based on the above observation, we choose Pregel as the baseline platform because Pregel is a widely-used and efficient vertex-centric graph processing platform. Pregel is based on the *Bulk Synchronous Parallel (BSP)* computation model. Specifically, it distributes vertices onto different machines of a cluster to perform the computation in parallel in a vertex-centric way. The computation consists of iterations called *supersteps*. In each superstep, vertices receive messages from relevant neighbors in the previous superstep, execute their local computation functions in parallel, and send messages to their relevant neighbors. At the end of each superstep, vertices synchronize their states.

According to the BSP computation model of Pregel, there are two main challenges to implement the QuickPoint algorithm to solve the event stream dissemination problem. The first challenge is how to assign strategies for links appearing in multiple users' hub-structures in parallel. The assignment for those links will impact the accuracy of our design. To solve it, we propose a simple but effective approach by greedily locking a link to the hub-structure with the maximum density through all the hub-structures containing the link. The second challenge is how to reduce the communication messages between supersteps for updating link states for hub-structures. To achieve it, we present an optimization technique using the Bloom filter [7] to effectively reduce the message size.

5.2 Parallelizing QuickPoint Using Pregel

In this section, we introduce how we implement QuickPoint to solve the event stream dissemination problem on Pregel in detail. The implementation can be divided into two parts: the *pretreatment* and the *link assignment*. The pretreatment part gathers information of the hub-structures for computing. The pretreatment part contains a phase called *ConstructStructure*. In the ConstructStructure phase, each vertex w constructs its largest hub-structure $G(X_w, w, Y_w)$ centred on w in parallel. The link assignment part makes assignments for links and runs in iteration containing three phases: the *LockLinks* phase, the *AssignLinks* phase, and the *UpdateLinks* phase. The LockLinks phase, for each vertex w, locks any unassigned link $u \rightarrow v$ in $G(X_w^*, w, Y_w^*)$ to only one densest sub-hub-structure. The AssignLinks phase assigns the locked links in different densest sub-hub-structures in parallel. In this phase, each vertex w gathers all links locked in $G(X_w^*, w, Y_w^*)$, and achieves its *locked densest sub-hub-structure* $G(X_w^L, w, Y_w^L)$. If the locked densest sub-hub-structure $G(X_w^L, w, Y_w^L)$ has a positive benefit, we assign the push strategy to the link $x \rightarrow w$ for each $x \in X_w^L$, the pull strategy to $w \rightarrow y$ for each $y \in Y_w^L$, and the piggyback strategy to each link $x \rightarrow y$.

The benefit of the locked densest sub-hub-structure $G(X_w^L, w, Y_w^L)$ is defined by $\sum_{u \rightarrow v \in G_L} min\{r_p(u), r_c(v)\} - (\sum_{x \in X_w^L} r_p(x) + \sum_{y \in Y_w^L} r_c(y))$, where G_L is an abbreviation of $G(X_w^L, w, Y_w^L)$. Then we update $g(x)$ to zero for each $x \in X_w^L$ and update $g(y)$ to zero for each $y \in Y_w^L$. The UpdateLinks phase updates the state of links appearing in different locked densest sub-hub-structures. In this phase, any other hub-structure $G(X_{w'}, w', Y_{w'})$ centred on w' ($w' \neq w$) containing some links in $G(X_w^L, w, Y_w^L)$ is selected to update the state of those links ASSIGNED. After that, we compute the densest sub-hub-structure of the selected hub-structure. The process of the parallel implementation will terminate after all the

locked densest sub-hub-structures without having a positive benefit. The rest of the unassigned links will be assigned as either a push or a pull strategy following the hybrid strategy [38]. For simplicity, we call X_w the incoming neighbor set of w. Similarly, we call Y_w the outgoing neighbor set of w. Algorithms 3 and 4 describe the two phases on top of Pregel in detail.

Algorithm 3. Pretreatment

```
1: public void ConstructStructure (Iterable < QPMessage > messages){
2:   if (getSuperstep() == 1){
3:     SendMessageTo(X_w, r_c(w));
4:     SendMessageTo(Y_w, r_p(w));}
5:   if (getSuperstep() == 2){
6:     int u = getVertex(messages);
7:     g(u) ← getRate(messages);
8:     SendMessageTo(Y_w, Y_w);}
9:   if (getSuperstep() == 3){
10:    int w = getId();
11:    int u = getVertex(messages);
12:    Y_u ← getNeighbor(messages);
13:    store links cross u and Y_u ∩ Y_w;}}
```

ConstructStructure Phase. In this phase, the algorithm gathers and stores the essential information to construct the largest hub-structure $G(X_w, w, Y_w)$ centred on w in parallel. The essential information includes the production rate $r_p(x)$ for each $x \in X_w$, the consumption rate $r_c(y)$ for each $y \in Y_w$, and all the pairs of links cross X_w and Y_w. Initially, each vertex w only stores her own production rate $r_p(w)$, consumption rate $r_c(w)$, the set of incoming neighbors X_w, and the set of outgoing neighbors Y_w. The algorithm gathers the essential information for the vertex w by the following three supersteps.

In the first superstep, each vertex w gathers the production rate $r_p(x)$ for each $x \in X_w$ and the consumption rate $r_c(y)$ for each $y \in Y_w$ by sending a message including the production rate $r_p(w)$ or the consumption rate $r_c(w)$ to its corresponding vertices. Specifically, the vertex w sends $r_p(w)$ to each $y \in Y_w$, because w belongs to the incoming neighbor set X_y of the vertex y. Similarly, the vertex w sends $r_c(w)$ to each $x \in X_w$.

In the second superstep, each vertex w stores the production rate $r_p(x)$ for each $x \in X_w$ and the consumption rate $r_c(y)$ for each $y \in Y_w$ after receiving the messages from the previous superstep. Since links cross u ($u \in X_w$) and Y_w are equivalent to links cross u and $Y_u \cap Y_w$, the algorithm obtains all the pairs of links cross X_w and Y_w by gathering the outgoing neighbor set Y_u for each $u \in X_w$. To get Y_u ($u \in X_w$) for each vertex w, the vertex w sends Y_w to each $y \in Y_w$, because w belongs to the incoming neighbor set X_y of the vertex y.

In the third superstep, each vertex w first computes $Y_u \cap Y_w$ ($u \in X_w$) and then stores all the pairs of links cross u and $Y_u \cap Y_w$ after receiving the message with Y_u from the vertex u in the previous superstep.

LockLinks Phase. In this phase, we compute the densest sub-hub-structure $G(X_w^*, w, Y_w^*)$ of each vertex w using the QuickPoint algorithm in parallel. Then, the algorithm locks each unassigned link $u \to v$ in $G(X_w^*, w, Y_w^*)$ to only one densest sub-hub-structure. Specifically, the algorithm locks $u \to v$ to the densest sub-hub-structure that maximizes the density through all the densest sub-hub-structures, which contain $u \to v$. The algorithm achieves $G(X_w^*, w, Y_w^*)$ by the following two supersteps.

Algorithm 4. Link Assignment

```
1: public void LockLinks (Iterable < QPMessage > messages){
2:   if (getSuperstep == 1){
3:     (X_w^*, Y_w^*) ← QuickPoint(X_w, w, Y_w); /*w: vertex id*/
4:     for each unassigned link u → v in G(X_w^*, w, Y_w^*) do
5:       SendMessageTo(u, ρ(w));}
6:   if (getSuperstep == 2){
7:     (u, v) ← getLink(messages);
8:     select the vertex w* maximize ρ(w*) for u → v;
9:     SendMessageTo(w*, u → v);}}
10:  public void AssignLinks (Iterable < QPMessage > messages){
11:    obtain G(X_w^L, w, Y_w^L);
12:    make assignment for G(X_w^L, w, Y_w^L);
13:    update the weight for each node in X_w^L ∪ Y_L;}
14:  public void UpdateLinks (Iterable < QPMessage > messages){
15:   if (getSuperstep == 1){
16:     SendMessageTo(X_w ∪ Y_w);}
17:   if (getSuperstep == 2){
18:     (u, v) ← getLink(messages);
19:     P ← getVertexContainLink(u, v);
20:     SendMessageTo(P, u → v);}
21:   if (getSuperstep == 3){
22:     (u, v) ← getLink(messages);
23:     update the state of u → v as ASSIGNED;}}
```

In the first superstep, each vertex w computes its densest sub-hub-structure $G(X_w^*, w, Y_w^*)$ using the QuickPoint algorithm. After that, we spread each unassigned link $u \to v$ in $G(X_w^*, w, Y_w^*)$ to its corresponding vertices, if $G(X_w^*, w, Y_w^*)$ has a positive benefit. Specifically, the process performs as follow: the vertex w spreads each unassigned link $x \to w$ ($x \in X_w^*$) by sending $\rho(w)$ to the vertex x, spreads each unassigned link $w \to y$ ($y \in Y_w^*$) by sending $\rho(w)$ to the vertex w, and spreads each link $x \to y$ by sending $\rho(w)$ to the vertex x, where $\rho(w)$ is the density of the densest sub-hub-structure $G(X_w^*, w, Y_w^*)$ computed by Eq. (2).

In the second superstep, each vertex w locks the link $w \to y$ to only one densest sub-hub-structure for each $y \in Y_w^*$, if $w \to y$ is currently unassigned. For all the vertices u spreading $w \to y$ and its density $\rho(u)$ to w in the previous superstep, the vertex w locks $w \to y$ to the vertex w^* that maximizes the density through all the densest sub-hub-structures of those vertices. Then, the vertex w sends $w \to y$ to the vertex w^*.

AssignLinks Phase. In this phase, the algorithm computes the locked densest sub-hub-structure $G(X_w^L, w, Y_w^L)$ for each vertex w, and assigns links in $G(X_w^L, w, Y_w^L)$ using piggyback strategy. Then, we update the weight of the locked link $u \to v$ to zero, since it is free to leverage $u \to v$ after assigning the push or the pull strategy to $u \to v$. The AssignLinks phase takes one superstep.

In the superstep, each vertex w gathers all links locked in its densest sub-hub-structure $G(X_w^*, w, Y_w^*)$. The vertex w locks each $w \to y$ ($y \in Y_w^*$) into $G(X_w^L, w, Y_w^L)$ if w receives a message containing $w \to y$ from the vertex w. The vertex w locks each $x \to w$ ($x \in X_w^*$) into $G(X_w^L, w, Y_w^L)$ if w receives a message containing $x \to w$ from the vertex x. Then, the vertex w locks

each $x \to y$ into $G(X_w^L, w, Y_w^L)$ only if: 1) w receives a message containing $x \to y$ from the vertex x; 2) both $x \to w$ and $w \to y$ have been locked in $G(X_w^L, w, Y_w^L)$. At last, the algorithm obtains the locked densest sub-hub-structure $G(X_w^L, w, Y_w^L)$. If $G(X_w^L, w, Y_w^L)$ has a positive benefit, we assign the push strategy to the link $x \to w$ for each $x \in X_w^L$, the pull strategy to $w \to y$ for each $y \in Y_w^L$, and the piggyback strategy to each link $x \to y$. Then, the algorithm updates $g(x)$ to zero for each $x \in X_w^L$ and updates $g(y)$ to zero for each $y \in Y_w^L$.

UpdateLinks Phase. In this phase, the algorithm updates the state of those links in the largest hub-structure $G(X_w, w, Y_w)$ that appear in any $G(X_u^L, u, Y_u^L)$. The algorithm implements the updating process by performing the following four supersteps.

In the first superstep, each vertex w sends a request to all the vertices u ($u \in X_w^L$) and v ($v \in Y_w^L$) for gathering Y_u and X_v, respectively.

In the second superstep, each vertex w sends X_w to the vertices u ($u \in X_w^L$) if receiving a messages u in the previous superstep. Similarly, w sends Y_w to the vertex v ($v \in Y_w^L$) if receiving a messages from v in the previous superstep. After that, w sends the update messages to those vertices, whose densest sub-hub-structures contain some links in any $G(X_w^L, w, Y_w^L)$. For each $u \in X_w^L$, the vertex w sends the update messages of $u \to w$ to the vertices u, w and each vertex $w' \in Y_u \cap X_w$. For each $v \in Y_w^L$, the vertex w sends the update messages of $w \to v$ to the vertices v, w and each vertex $w' \in X_v \cap Y_w$. The vertex w sends the update messages of $u \to v$ to the vertices u, v and each vertex $w' \in Y_u \cap X_v$, for each link $x \to y$ in $G(X_w^L, w, Y_w^L)$.

In the third superstep, each vertex w updates the state of ASSIGNED to those links that are contained in the messages from the previous superstep.

5.3 Optimization

In the parallel implementation of QuickPoint, we need to send the incoming neighbor set X_w or the outgoing neighbor set Y_w to different vertices. Such a transmiting operation will incur a large amount of traffic of a vertex containing a large number of neighbors in its hub-structure. To address the problem, we design an optimization technique using the Bloom filter [7] to represent both the incoming and outgoing neighbor sets of a vertex in the social graph.

Specifically, when the vertex w sends the incoming neighbor set X_w or the outgoing neighbor set Y_w to any other vertex w' ($w' \neq w$), we send the corresponding Bloom filter of X_w or Y_w to w' instead. On one hand, we effectively reduce the traffic generated during the parallel implementation of QuickPoint due to the high space-efficiency of the Bloom filter. On the other hand, we can also speed up the process of the parallel implementation of the QuickPoint algorithm. For example, considering the computation complexity of computing the intersection $Y_u \cap Y_w$ in the parallel implementation of QuickPoint: 1) without the Bloom filter, the computation complexity is at least $O(|Y_u| + |Y_w|)$ with all the members in both Y_u and Y_w are sorted in the same order; 2) with the Bloom filter, we reduce the computation complexity to $O(|Y_w|)$, as we can judge whether a member $y \in Y_w$ belongs to the Bloom filter of Y_u in a constant time.

Due to the false positive of the Bloom filter, the above scheme may determine a link $x \to y$ cross X_w to Y_w belong to $G(X_w, w, Y_w)$, although $G(X_w, w, Y_w)$ actually does not contain $x \to y$; or determine a link $x \to w$ ($x \in X_w$) or $w \to y$ ($y \in Y_w$) contained in $G(X_w, w, Y_w)$, although $G(X_w, w, Y_w)$ actually does not contain $x \to w$ or $w \to y$. For the former situation, it may affect the process of finding the densest sub-hub-structure of $G(X_w, w, Y_w)$. Such an influence is very slight because only a few false positive links $x \to y$ will appear in $G(X_w, w, Y_w)$. For the later situation, since $G(X_w, w, Y_w)$ actually does not contain links $x \to w$ or $w \to y$, there is no state updating for $x \to w$ or $w \to y$. The only influence is the extra messages between vertices, which can be controlled by adjusting the setting of the Bloom filter to achieve a very small false positive.

6 Dynamic Updates

It is noticeable that social graphs are dynamically evolving with incremental updates of user/link adding and removing [11], [17], [20], [45]. In this section, we consider such incremental updates of a social graph in our design. In the following, we first introduce how we deal with link adding and removing. We examine the marginal benefit of the saved communication cost after adding a new link. We then transform the cases of user adding and removing into a series of link adding and removing, respectively. For simplicity, we use the notation (H, L) to denote the link set of the piggyback assignment.

Algorithm 5 describes how we cope with the dynamic link adding. When a social link $u \to v$ is added, we compute the marginal benefit of assigning $u \to v$ as a push or pull link, respectively (lines 2-3). Specifically, the marginal benefit of assigning a push strategy to $u \to v$ is the total amount of saved communication cost of $u \to x$ ($u \to x \notin C$, $v \to x \in L$), where $u \to x$ can be piggybacked through the hub node v, and $u \to x$ does not exist in any other piggyback triangle structure. The case of assigning $u \to v$ as a pull link is similar. We then check whether $u \to v$ can be piggybacked (line 5): 1) If so, we assign $u \to v$ as push, pull, or piggyback strategy, whichever achieves greater marginal benefit (lines 6-9); 2) If not, we assign the push or pull strategy to $u \to v$ according to the greater marginal benefit.

Algorithm 5. Link Adding

```
1: void addLink (int u, int v){
2:    double mb1 = getMarginalBenefit(u, v, "pull");
3:    double mb2 = getMarginalBenefit(u, v, "push");
4:    double min_mb = mb1 < mb2 ? mb1 : mb2;
5:    if (u → v can be piggybacked){
6:       if (min_mb = 0){
7:          add u → v to C;
8:       }else{
9:          assignEdge(u, v, mb1, mb2);}
10:   }else{
11:      assignEdge(u, v, mb1, mb2);}}
12: void assignEdge (int u, int v, double mb1, double mb2){
13:   if (mb1 < mb2){
14:      add u → v to H;
15:   }else{
16:      add u → v to L;}}
```

Algorithm 6 describes the process of dynamic link removing in our design. The system copes with different kinds of links as follows. If $u \to v$ is a piggyback link (line 2), the system directly removes it from C (line 3). If $u \to v$ is a push link (line 4), it removes it from H (line 5) and re-

assign all the piggyback links $u \rightarrow v'$ ($v \rightarrow v' \in L$) in the hub node v (lines 6-9). Specifically, the algorithm removes $u \rightarrow v'$ and assigns it according to Algorithm 5. Similarly, if $u \rightarrow v$ is a pull link (line 10), the algorithm removes it from L (line 11) and re-assigns all the piggyback links $u' \rightarrow v$ ($u' \rightarrow u \in H$) following Algorithm 5 (lines 12-15).

Algorithm 6. Link Removing

```
1: void removeLink (int u, int v){
2:   if (u → v ∈ C){
3:     remove u → v from C;
4:   }else if (u → v ∈ H){
5:     remove u → v from H;
6:     for (each v' : v → v' ∈ L){
7:       if (u → v' ∈ C){
8:         remove u → v' from C;
9:         addLink(u, v');}}
10:  }else if (u → v ∈ L){
11:    remove u → v from L;
12:    for (each u' : u' → u ∈ H){
13:      if (u' → v ∈ C){
14:        remove u' → v from C;
15:        addLink(u', v');}}}
```

It is not difficult to see that the system dynamics of user adding/removing is equivalent to the process of link dynamics. When a new user u is appended in the social network, new social links are generated by u. We can add those social links one by one following Algorithm 5. Similarly, when u is removed from the social network, we remove all the links that connect u one by one according to Algorithm 6.

7 Experiment

In this section, we evaluate the performance of our design by comparing it with the state-of-the-art algorithms proposed by Gionis et al. [18]. We first present the experiment setups and the datasets used in the experiments. We then examine both the communication cost and the running time of our design by comparing with the performance of CHITCHAT and PARALLELNOSY schemes proposed by Gionis et al. [18]. At last, we further build an OSN prototype system to evaluate the actual traffic and throughput of our design.

7.1 Experiment Setups

In the experiments, we use large-scale traces collected from popular OSN systems including Twitter and Flickr, which is same as the traces used in [18]. The Twitter trace contains 54 million users and 1.9 billion links [9]. The Flickr trace includes 1.86 million users and 22 million links [10]. The large-scale traces are quite representative for real-world OSN systems. Since it is extremely difficult to obtain the real workloads (including the production rate and the consumption rate for each user) from the commercial OSN systems, we follow the methodology used by Gionis et al. [18] to synthetically generate the workloads. They generated the workloads information based on the observation of Huberman et al. [22] that a user with a higher out-degree tends to share new events more frequently while a user with a higher in-degree tends to read event streams more frequently. Accordingly, in the experiments, the production rate and the consumption rate of a user are modeled to be proportional to the logarithm of her out-degree and in-degree, respectively. The read/write ratio of the average consumption rate and the average production rate of users in the OSN graph is set to five, which is in agreement with the setting used by Gionis et al. [18].

We implement both the CHITCHAT [18] and PARALLELNOSY [18] algorithms and use them as the baseline schemes. The PARALLELNOSY algorithm is implemented using MapReduce and runs on a ten-node cluster of Intel Xeon servers each equipped with 16 2.13 GHz cores, 16 GB main memory, and a 1,000 Mbps Ethernet card. We implement our scheme on the same cluster with the execution environment of Apache Giraph [2], an open-source counterpart to Google's Pregel. Both the PARALLELNOSY algorithm and our scheme are implemented using JAVA. The CHITCHAT algorithm is implemented with C++ and executed one a single node because the CHITCHAT algorithm is a serial program proposed by Gionis et al. [18].

Since the computation cost of the baseline CHITCHAT algorithm is prohibitively high for large-scale social graphs, to conduct a comparable evaluation, we obtain a smaller graph for CHITCHAT using two sampling methods [19], [25], [33], [35], [40]: random-walk and breadth-first to restrict the number of edges of the sampling graph of different datasets within five million, following the same method used by Gionis et al. in the experiment for the baseline scheme [18]. The random walk sampling begins with a random node, and then keeps going from a node to its next random neighboring node until reaching five million new edges. The breadth-first sampling starts from a random node and explores all its neighboring nodes. Then, for each selected node, it explores their unexplored neighbors, and so on. With both of the sampling schemes, each algorithm runs five times in different sampling graphs and reports the average result.

We also implement the algorithms dealing with the incremental updates on the Flickr graph. We divide the Flickr graph into two parts: the static and dynamic graphs. The static graph contains 80 percent of the number of links in the Flickr graph. We obtain the piggyback assignment for the static graph according to QuickPoint. The dynamic graph is used for incremental updates. For simplicity, we consider the dynamics of link adding. Specifically, we divide the dynamic graph into ten equal parts. For each part, we compare the performance of incremental updates with that of the piggyback assignment re-computed by QuickPoint.

We further implement an online social network prototype system and measure the actual traffic and throughput to achieve a more realistic performance evaluation of our scheme. Fig. 2 shows the architecture of our prototype. Our prototype is deployed on the ten-node cluster. For simplicity, the memcache client stores the social graph and the piggyback assignment found by our QuickPoint scheme in main memory. The memcache client receives the requests of users' event sharing and event stream browsing operations, as well as sending the generated requests to the memcache servers, which keep user views in main memory. In our prototype, we distribute a users' view to a random memcache server selected by hashing the user's id. Each user view keeps an index including the references to the user's events. In our prototype, the operation of sharing event of a user is implemented through the memcache client pushing the user's event to all her friends' views after the memcache client receives the request of the user. Furthermore, the operation of browsing event stream can be divided into two

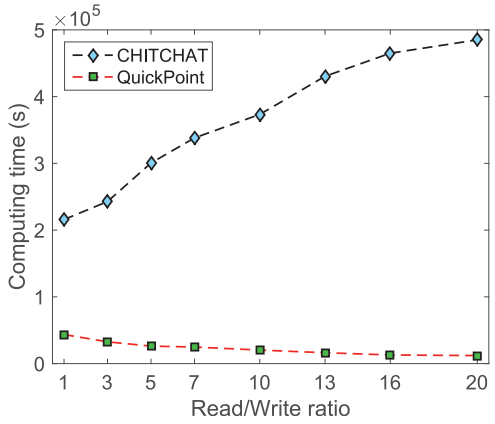


Fig. 4. Computing time on Flickr.

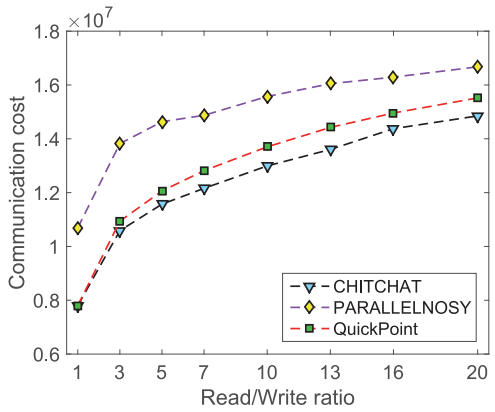


Fig. 5. Communication cost on Flickr.

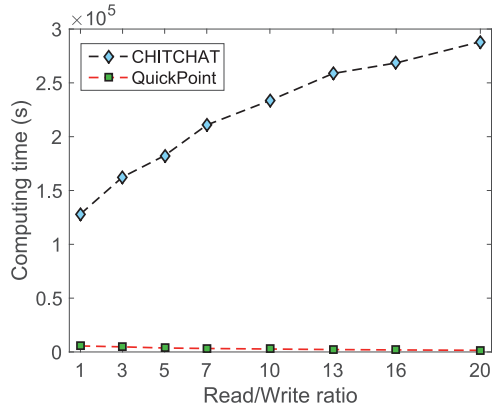


Fig. 6. Computing time on Twitter.

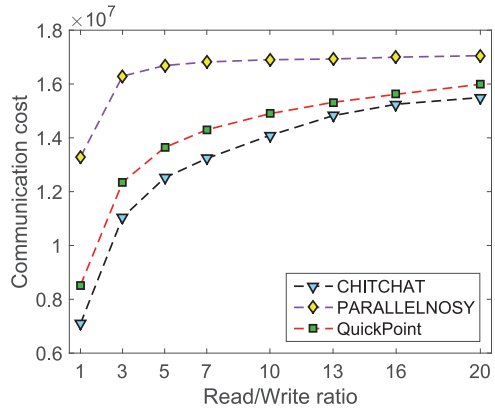


Fig. 7. Communication cost on Twitter.

step: 1) querying friends' views on different memcache servers to assemble the event stream; 2) fetching event data from the storage.

In our experiment, we mainly focus on the first step of assembling event stream. When a user generates an event, we insert the event as a (*userId, eventId, eventLength, timestamp*) tuple into the user view and store the context of the event into the storage. When a user browses the event stream, we fetch the recent events of each of her friends and return the ten latest events across all friends to the memcache client. We use TCP to transmit event data among memcache clients and servers. We consider the piggyback assignment of our QuickPoint and PARALLELNOSY on both the Flickr and Twitter breadth-first sampling graphs. We generate a workload with average read/write ratio set to five [18]. The workload is a sequence of user request of event sharing or event stream browsing operations.

Our design considers both the system performance for event stream dissemination and the efficiency of different algorithms. The system performance focuses on the communication cost for event stream dissemination over the achieved piggyback assignment [18], the actual traffic, and the throughput of our prototype system. The actual traffic is the size of total TCP packets generated in our prototype system for a given workload, and the actual throughput is the average request processing rate per-client in our prototype system.

A low communication cost and traffic are always desirable for sharing events and browsing event streams in OSN systems. A high throughput is always preferable for real-world OSN systems to achieve better user experience. We also examine the algorithm efficiency which is defined as the ratio of the percentage of the reduced communication cost and the running time. The running time has an important impact on the feasibility of the algorithm in practice. Long running time limits the applicability of the algorithm. In the experiment, we use the actual computing time, which summarizes the time from the beginning with all links unassigned to the end with all links assigned.

7.2 Results

Fig. 4 plots the computing time of different schemes on the Flickr random-walk sampling graph. We compare the computing time of our scheme with that of CHITCHAT. The results show that our scheme greatly reduces the computing time of CHITCHAT by 91.29 percent when the read/write ratio is set to five.

Fig. 5 shows the communication cost of different schemes on the Flickr random-walk sampling graph. We use the CHITCHAT algorithm as the baseline. The results show that PARALLELNOSY increases the communication cost of CHITCHAT by 26.49 percent, while the increment of our scheme is only 3.98 percent. The results show that our scheme greatly outperforms PARALLELNOSY by 84.96 percent.

Fig. 6 examines the computing time of our scheme compared to that of CHITCHAT on the Twitter random-walk sampling graph. The results show that our scheme greatly reduces the computing time of CHITCHAT by 97.98 percent.

Fig. 7 plots the communication cost of different schemes on the Twitter random-walk sampling graph. We use the CHITCHAT algorithm as the baseline. The results show that PARALLELNOSY increases the

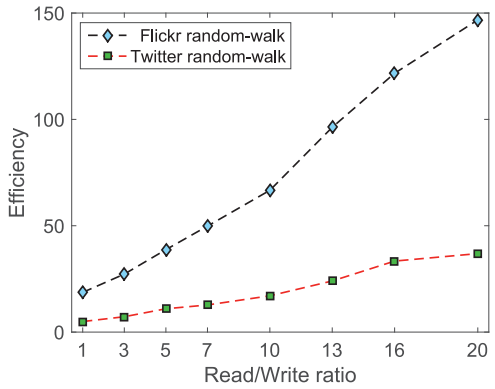


Fig. 8. Efficiency on social graphs.

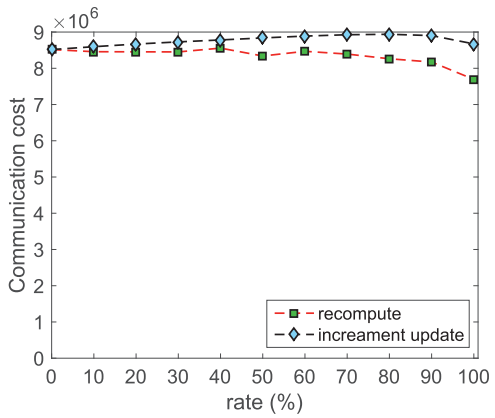


Fig. 9. Communication cost of incremental updates.

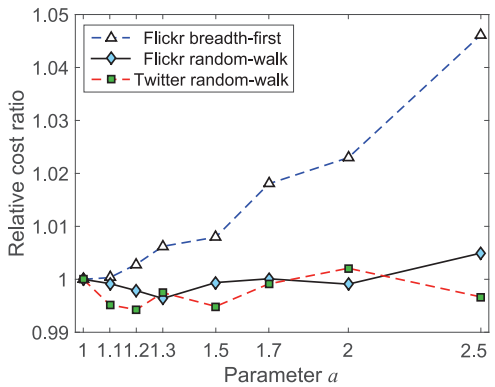


Fig. 10. Communication cost changing with a.

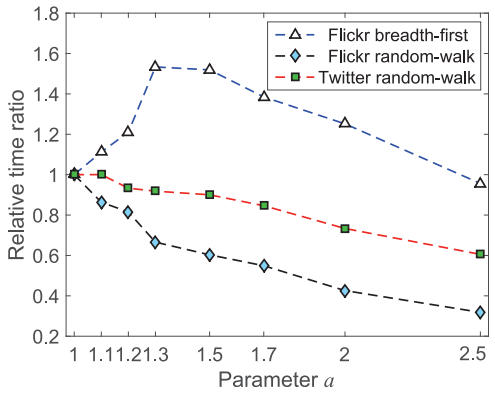


Fig. 11. Computing time changing with a.

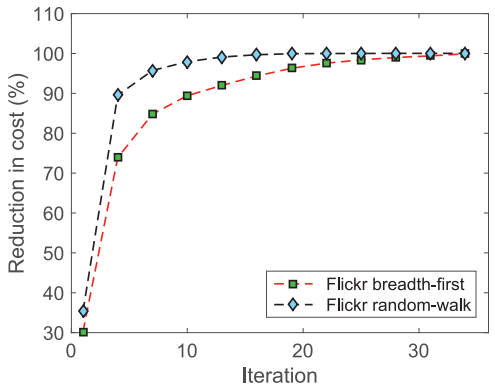


Fig. 12. Reduction in communication cost of QuickPoint.

communication cost of CHITCHAT by 33.35 percent, while the increment of our scheme is only 9.02 percent. The results show that our scheme greatly outperforms PARALLELNOSY by 72.95 percent.

Fig. 8 depicts the efficiency of our scheme compared to that of CHITCHAT on different graphs. The results show that our scheme significantly improves the efficiency of CHITCHAT by a factor of 11.0× on the Flickr random-walk sampling graph, and 38.8× on the Twitter random-walk sampling graph.

Fig. 9 depicts the communication cost of QuickPoint and incremental updates on the Flickr graph. We compare the communication cost of incremental updates with that of QuickPoint for ten equal parts of the dynamic graph. When all links in the dynamic graph are added, the results show that incremental updates slightly increases the communication cost of QuickPoint by 12.73 percent.

Fig. 10 shows how the communication cost of our scheme changes with the parameter a of QuickPoint on different graphs. To illustrate the change, we measure the communication cost ratio of a certain parameter a and the parameter $a = 1.0$. The results show that, with the increase of the parameter a, the communication cost of the three sampling graphs slightly increases or decreases within a percentage of 5. Thus, our scheme can guarantee the accuarcy of the communication cost even with a large parameter a.

Fig. 11 plots how the computing time of our scheme changes with the parameter a of QuickPoint on different graphs. To illustrate the change, we measure the computing time ratio of a certain parameter a and the parameter $a = 1.0$. The results show that, when the parameter a is set to 2.5, the computing time is decreased by 4.36 percent on the Flickr breadth-first sampling graph, by 68.13 percent on the Flickr random-walk sampling graph, and by 39.46 percent on the Twitter random-walk sampling graph. The results show that by choosing an appropriate parameter a, we can reduce the computing time of our scheme without sacrificing the communication cost.

Fig. 12 plots the reduction in the communication cost achieved in each iteration of the link assignment of the parallelized QuickPoint. We use the hybrid scheme as the baseline. We run the parallelized QuickPoint on the Flickr breadth-first and the Flickr rand-walk sampling graphs. The results show that the most reduction is achieved in the first few rounds of iterations. Until the 10th iteration, the communication cost reduction on the Flickr breadth-

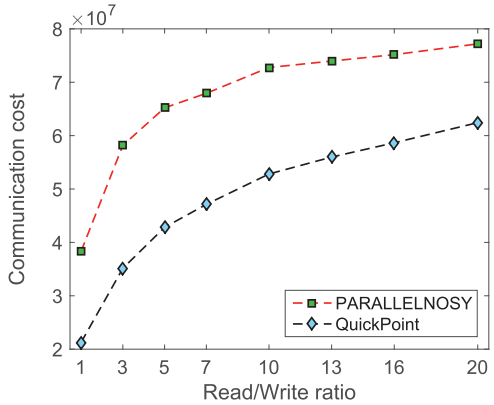


Fig. 13. Communication cost on Flickr.

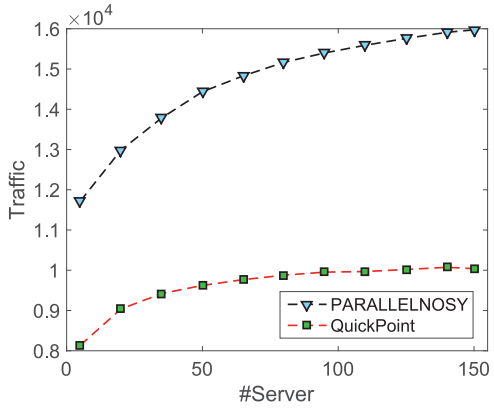


Fig. 14. Traffic on Flickr.

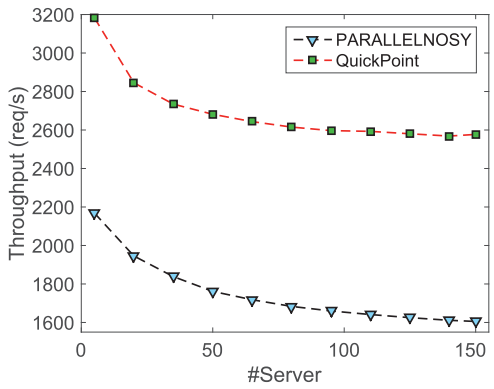


Fig. 15. Throughput on Flickr.

first and the Flickr rand-walk sampling graphs obtain an 86.12 and 97.91 percent reduction, respectively.

Fig. 13 shows the communication cost of different schemes on the Flickr graph. We compare the communication cost of our scheme with that of the PARALLELNOSY algorithm. The results show that the communication cost of our scheme is 4.28×10^7, while PARALLELNOSY incurs a communication cost of 6.52×10^7. The results show that our scheme reduces the communication cost of PARALLELNOSY by 34.35 percent.

Fig. 14 depicts the traffic of different schemes on the Flickr breadth-first sampling graph. We compare the traffic of our scheme with that of the PARALLELNOSY algorithm. The results show that the traffic increases with the increase of number of the memcache clients. This is because the piggybacking scheme works when the two end users of the

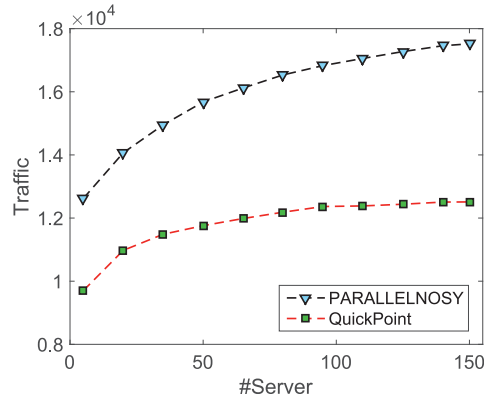


Fig. 16. Traffic on Twitter.

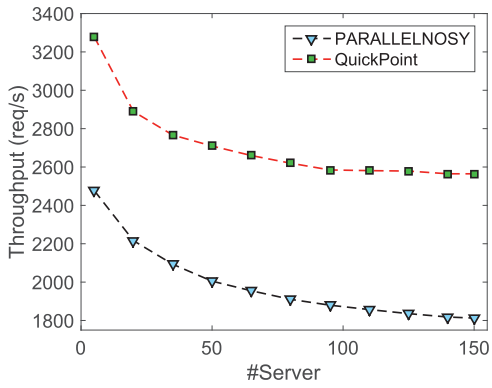


Fig. 17. Throughput on Twitter.

piggyback link are assigned to different memcache servers, and the two end users are more likely assigned to different memcache servers with more memcache servers. The results show that our scheme reduces the traffic of PARALLELNOSY by 37.15 percent.

Fig. 15 plots the throughput of different schemes on the Flickr breadth-first sampling graph. We compare the throughput of our scheme with that of the PARALLELNOSY algorithm. It is clear that the throughput decreases as the number of memcache servers increases. This is because the memcache clients are more likely to communicate with more memcache servers with the increase of the number of memcache servers. The results show that our scheme improves the throughput of PARALLELNOSY by 60.37 percent.

Fig. 16 depicts the traffic of different schemes on the Twitter breadth-first sampling graph. We compare the traffic of our scheme with that of the PARALLELNOSY algorithm. The results show that the traffic increases as the number of memcache servers increases. The results show that our scheme reduces the traffic of PARALLELNOSY by 28.59 percent.

Fig. 17 plots the throughput of different schemes on the Twitter breadth-first sampling graph. We use the PARALLELNOSY algorithm as the baseline scheme. The results show that the throughput of different schemes decreases as the number of memcache servers increases. The results show that our scheme improves the throughput of PARALLELNOSY by 41.50 percent.

8 Conclusion

In this work, we propose a novel algorithm QuickPoint for quickly finding the densest sub-hub-structures of users to

achieve efficient event stream dissemination in OSNs. We mathematically prove that the upper bound of the number of iterations of QuickPoint is $O(log_a n)(a > 1)$, and QuickPoint achieves a $2a$-approximation. We further implement QuickPoint in parallel on top of Pregel to obtain the piggyback assignment for a social graph. We also extend QuickPoint to support incremental updates in a dynamic social graph. Experimental results show that our scheme achieves a significant improvement in efficiency compared to existing schemes.

Acknowledgments

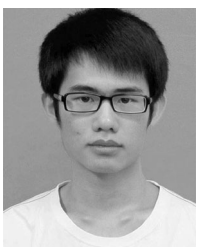
This research is supported in part by the National Key Research and Development Program of China under grant No.2016QY02D0302, NSFC under grants Nos. 61433019, 61422202, 61370233, and the Research Fund of Guangdong Province under grant No.2015B010131001. J. Liu's work is supported by an NSERC Discovery Grant and a MITACS grant. The preliminary results of this work were presented in IWQoS 2016.

References

[1] 2018. [Online]. Available: http://highscalability.com/blog/2012/2/13/tumblr-architecture-15-billion- page-views-a-month-and-harder.html
[2] 2018. [Online]. Available: http://giraph.apache.org/
[3] B. Bahmani, R. Kumar, and S. Vassilvitskii, "Densest subgraph in streaming and MapReduce," *Proc. VLDB Endowment*, vol. 5, no. 5, pp. 454–465, 2012.
[4] J. Bao, M. F. Mokbel, and C. Y. Chow, "GeoFeed: A location aware news feed system," in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 54–65.
[5] M. Bhuiyan and M. A. Hasan, "An iterative MapReduce based frequent subgraph mining algorithm," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 3, pp. 608–620, Mar. 2015.
[6] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1199–1214.
[7] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[8] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's distributed data store for the social graph," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 49–60.
[9] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi, "Measuring user influence in Twitter: The million follower fallacy," in *Proc. Int. Conf. Weblogs Social Media*, 2010, pp. 10–17.
[10] M. Cha, A. Mislove, and P. K. Gummadi, "A measurement-driven analysis of information propagation in the Flickr social network," in *Proc. 18th Int. Conf. World Wide Web*, 2009, pp. 721–730.
[11] Z. Chang, L. Zou, and F. Li, "Privacy preserving subgraph matching on large graphs in cloud," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 199–213.
[12] M. Charikar, "Greedy approximation algorithms for finding dense components in a graph," in *Proc. Approximation Algorithms Combinatorial Optimization*, 2000, pp. 84–95.
[13] H. Chen and H. Jin, "Efficient keyword searching in large-scale social network service," *IEEE Trans. Serv. Comput.*, vol. 11, no. 5, pp. 810–820, Sep./Oct. 2018.
[14] R. Chirkova, C. Li, and J. Li, "Answering queries using materialized views with minimum size," *The VLDB J.*, vol. 15, no. 3, pp. 191–210, 2006.
[15] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Opearting Syst. Des. Implementation*, 2004, pp. 10–10.
[16] Y. Dong, Y. Yang, J. Tang, Y. Yang, and N. V. Chawla, "Inferring user demographics and social strategies in mobile social networks," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2014, pp. 15–24.
[17] Z. Fan, B. Choi, J. Xu, and S. S. Bhowmick, "Asymmetric structure-preserving subgraph queries for large graphs," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 339–350.
[18] A. Gionis, F. Junqueira, V. Leroy, M. Serafini, and I. Weber, "Piggybacking on social networks," *Proc. VLDB Endowment*, vol. 6, no. 6, pp. 409–420, 2013.
[19] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou, "Walking in Facebook: A case study of unbiased sampling of OSNs," in *Proc. INFOCOM*, 2010, pp. 1–9.
[20] Y. Gu, C. Gao, L. Wang, and G. Yu, "Subgraph similarity maximal all-matching over a large uncertain graph," *World Wide Web*, vol. 19, no. 5, pp. 755–782, 2016.
[21] J. Hu, X. Wu, R. Cheng, S. Luo, and Y. Fang, "Querying minimal steiner maximum-connected subgraphs in large graphs," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage.*, 2016, pp. 1241–1250.
[22] B. A. Huberman, D. M. Romero, and F. Wu, "Social networks that matter: Twitter under the microscope," *First Monday*, vol. 14, no. 1, pp. 1–9, 2009.
[23] F. P. Junqueira, V. Leroy, M. Serafini, and A. Silberstein, "Shepherding social feed generation with sheep," in *Proc. 5th Workshop Social Netw. Syst.*, 2012, Art. no. 7.
[24] N. Kourtellis, G. D. F. Morales, and F. Bonchi, "Scalable online betweenness centrality in evolving graphs," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 1580–1581.
[25] J. Leskovec and C. Faloutsos, "Sampling from large graphs," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 631–636.
[26] K. Li, W. Lu, S. Bhagat, L. V. S. Lakshmanan, and C. Yu, "On social event organization," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2014, pp. 1206–1215.
[27] L. Li, J. He, M. Wang, and X. Wu, "Trust agent-based behavior induction in social networks," *IEEE Intell. Syst.*, vol. 31, no. 1, pp. 24–30, Jan./Feb. 2016.
[28] X. Lian, L. Chen, and G. Wang, "Quality-aware subgraph matching over inconsistent probabilistic graph databases," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 6, pp. 1560–1574, Jun. 2016.
[29] H. Liu, L. J. Latecki, and S. Yan, "Dense subgraph partition of positive hypergraphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 37, no. 3, pp. 541–554, Mar. 2015.
[30] X. Liu, Q. He, Y. Tian, W. Lee, J. McPherson, and J. Han, "Event-based social networks: Linking the online and offline social worlds," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2012, pp. 1032–1040.
[31] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[32] J. Mondal and A. Deshpande, "EAGr: Supporting continuous ego-centric aggregate queries over large dynamic graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 1335–1346.
[33] A. Nazi, Z. Zhou, S. Thirumuruganathan, N. Zhang, and G. Das, "Walk, not wait: Faster sampling over online social networks," *Proc. VLDB Endowment*, vol. 8, no. 6, pp. 678–689, 2015.
[34] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at Facebook," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.
[35] M. Papagelis, G. Das, and N. Koudas, "Sampling online social networks," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 3, pp. 662–676, Mar. 2013.
[36] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The little engine(s) that could: Scaling online social networks," in *Proc. ACM SIGCOMM Conf.*, 2010, pp. 375–386.
[37] A. Shalita, B. Karrer, I. Kabiljo, A. Sharma, A. Presta, A. Adcock, H. Kllapi, and M. Stumm, "Social hash: An assignment framework for optimizing distributed systems operations on social networks," in *Proc. 13th Usenix Conf. Netw. Syst. Des. Implementation*, 2016, pp. 455–468.
[38] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan, "Feeding frenzy: Selectively materializing users' event feeds," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 831–842.
[39] J. Tang, H. Tong, and M. Vazirgiannis, "Special focus on natural language processing and social computing," *Sci. China Inf. Sci.*, vol. 60, no. 11, pp. 110 100:1–110 100:2, 2017.
[40] J. Tang, C. Zhang, K. Cai, L. Zhang, and Z. Su, "Sampling representative users from large social networks," in *Proc. AAAI Conf. Artif. Intell.*, 2015, pp. 304–310.

[41] H. Wang, P. Zhang, X. Zhu, I. W. Tsang, L. Chen, C. Zhang, and X. Wu, "Incremental subgraph feature selection for graph classification," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 1, pp. 128–142, Jan. 2017.
[42] C. Wilson, B. Boe, A. Sala, K. P. N. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, 2009, pp. 205–218.
[43] S. Xu, S. Su, L. Xiong, X. Cheng, and K. Xiao, "Differentially private frequent subgraph mining," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 229–240.
[44] Y. Yang, X. Mao, J. Pei, and X. He, "Continuous influence maximization: What discounts should we offer to social network users?" in *Proc. Int. Conf. Manage. Data*, 2016, pp. 727–741.
[45] W. Zhang, X. Lin, Y. Zhang, K. Zhu, and G. Zhu, "Efficient probabilistic supergraph search," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 4, pp. 965–978, Apr. 2016.
[46] Y. Zhou, A. Salehi, and K. Aberer, "Scalable delivery of stream query results," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 49–60, 2009.
[47] B. Zong, R. Raghavendra, M. Srivatsa, X. Yan, A. K. Singh, and K. Lee, "Cloud service placement via subgraph matching," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 832–843.

Hai Jin received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST), in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering with HUST, in China. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked with The University of Hong Kong between 1998 and 2000, and as a visiting scholar with the University of Southern California between 1999 and 2000. He was awarded the Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of the National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of IEEE, a fellow of CCF, and a member of the ACM. He has coauthored 23 books and published more than 800 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.



Changfu Lin is currently working toward the PhD degree in the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), China. His research interests are in social network, and distributed stream processing systems. He is a student member of the IEEE.

Hanhua Chen received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), in 2010. He is currently a professor with the School of Computer Science and Technology, HUST, China. His research interests include distributed systems and BigData processing systems. He received the National Excellent Doctorial Dissertation Award of China in 2012. He is a member of the IEEE.

Jiangchuan Liu (S'01-M'03-SM'08-F'17) received the BEng degree (cum laude) from Tsinghua University, Beijing, China, in 1999, and the PhD degree from The Hong Kong University of Science and Technology, in 2003, both in computer science. He is a university professor with the School of Computing Science, Simon Fraser University, British Columbia, Canada. He is an NSERC E.W. R. Steacie Memorial fellow. He is an EMC-Endowed visiting chair professor of Tsinghua University, Beijing, China, and an adjunct professor of Tsinghua-Berkeley Shenzhen Institute. In the past he worked as an assistant professor at The Chinese University of Hong Kong and as a research fellow at Microsoft Research Asia. He is a co-recipient of the inaugural Test of Time Paper Award of IEEE INFOCOM (2015), ACM SIGMM TOMCCAP Nicolas D. Georganas Best Paper Award (2013), and ACM Multimedia Best Paper Award (2012). His research interests include multimedia systems and networks, cloud computing, social networking, online gaming, big data computing, RFID, and Internet of Things. He has served on the editorial boards of the *IEEE/ACM Transactions on Networking*, the *IEEE Transactions on Big Data*, the *IEEE Transactions on Multimedia*, the *IEEE Communications Surveys and Tutorials*, and the *IEEE Internet of Things Journal*. He is a steering committee member of the *IEEE Transactions on Mobile Computing* and steering committee chair of IEEE/ACM IWQoS (2015-2017). He is a fellow of the IEEE.