Scalable Facility Location for Massive Graphs on Pregel-like Systems

Kiran Garimella
Aalto University
kiran.garimella@aalto.fi

Gianmarco De Francisci Morales
Yahoo Labs, Barcelona
gdfm@acm.org

Aristides Gionis
Aalto University
aristides.gionis@aalto.fi

Mauro Sozio
Telecom Paristech
sozio@telecom-paristech.fr

ABSTRACT

We propose a new scalable algorithm for facility location. Facility location is a classic problem, where the goal is to select a subset of facilities to open, from a set of candidate facilities F, in order to serve a set of clients C. The objective is to minimize the total cost of opening facilities plus the cost of serving each client from the facility it is assigned to. In this work, we are interested in the *graph setting*, where the cost of serving a client from a facility is represented by the shortest-path distance on the graph. This setting allows to model natural problems arising in the Web and in social-media applications. It also allows to leverage the inherent sparsity of such graphs, as the input is much smaller than the full pairwise distances between all vertices.

To obtain truly scalable performance, we design a parallel algorithm that operates on clusters of shared-nothing machines. In particular, we target modern Pregel-like architectures, and we implement our algorithm on Apache Giraph. Our solution makes use of a recent result to build sketches for massive graphs, and of a fast parallel algorithm to find maximal independent sets, as building blocks. In so doing, we show how these problems can be solved on a Pregel-like architecture, and we investigate the properties of these algorithms. Extensive experimental results show that our algorithm scales gracefully to graphs with billions of edges, while obtaining values of the objective function that are competitive with a state-of-the-art sequential algorithm.

1. INTRODUCTION

Facility location is a classic combinatorial optimization problem. It has been widely studied in *operations research* [30, 37] and *theoretical computer science* [1, 10, 29, 48], and it has been applied in many information-management tasks, including clustering data streams [26], data compression [8], grammar inference [22], information retrieval [49], and design of communication networks [39, 42]. In the most basic setting of the problem, we are given a set of facilities F, a set of clients C, and costs $c(f)$ for *opening* a facility $f \in F$ and $d(c, f)$ for *serving* a client $c \in C$ with a facility $f \in F$. The goal is to select a subset of facilities to open so that all clients are served by an open facility and the total cost of opening the facilities plus serving the clients is minimized.

The facility-location problem is **NP**-hard, but a number of different approximation algorithms are known [1, 10, 29, 48], Each algorithm has different characteristics, but they all achieve an approximation guarantee that is a small constant, for example the algorithm of Jain et al. has an approximation guarantee of 1.61 [29]. The existing algorithms operate in the traditional sequential model and they assume that the data resides in main memory.

However, facility location is a general-purpose optimization problem that can be used in applications related to web graphs, large social networks, and other such massive-scale datasets, whose size may far exceed the memory of a single machine. Examples of such applications include placing caches for content delivery on the Internet, finding aggregators in information networks, and compressing social-media activity. Some of these application scenarios are discussed in Section 2.

To cope with large problem sizes, modern applications take advantage of large-scale distributed systems, such as MapReduce [18] and Hadoop, or of variants targeted to graph data, such as Pregel [36] and its open-source clones, Giraph [12] and GraphLab [34]. Such systems offer several advantages, among which higher potential for scalability and a simple programming interface that requires implementing only a small number of functions.

In this paper we present the first algorithm for the facility-location problem designed for a Pregel-like system. In particular, we implement our algorithm on Giraph, thus adding facility location to the toolbox of optimization problems that can be solved for very large datasets on modern computer clusters. Our work is inspired by, and follows, the large body of recent work that offer MapReduce-type solutions to important computational problems, such as linear programming [38], maximum cover [11], similarity join [3, 16], graph matching [17, 31, 38], counting triangles in graphs [47], and finding dense subgraphs [2].

Most works in the the area of theoretical computer science focus on the classical formulation of the facility-location problem, where the input consists of the full $|F| \times |C|$ set of distances [45]. Unfortunately, in this setting even algorithms with linear running time (in the size of input) are not practical when both $|F|$ and $|C|$ are large.

A more economical representation of the facility-location problem, which fits our applications of interest, assumes that

F and C are vertices of a graph $G = (V, E)$ (that is, $V = F \cup C$),[1] where the number of edges $m = |E|$ is very small compared to $|F| \times |C|$, i.e., the graph is sparse. A client $c \in C$ can be served by a facility $f \in F$ even if $(c, f) \notin E$, provided that there is a path in the graph from c to f, and the cost $d(c, f)$ is the shortest-path distance metric induced by the graph. We require that the running time and storage requirements of our algorithm be quasilinear functions of $|E|$. If the graph G is sparse, as most real-world graphs are, this leads to a significantly more scalable algorithm.

Our approach is designed for such a graph-based formulation of the facility-location problem. As mentioned above, most sequential algorithms for classical facility location cannot cope with this setting, as they require that the full distance matrix is provided as input, or, equivalently, that a constant-time distance oracle is available. A notable exception is the sequential algorithm by Thorup [48], which, like in our scheme, is able to take advantage of the underlying sparsity of the graph structure.

Our algorithm targets a parallel shared-nothing setting. While other parallel algorithms have been proposed in the literature, our approach is the first one to target modern computer clusters. In particular, Blelloch and Tangwongsan [4] proposed a parallel facility-location approximation algorithm for the PRAM model. Our work extends this parallel algorithm to the more scalable Pregel model.

There are three phases in our approach: (*i*) *neighborhood sketching*, (*ii*) *facility opening*, and (*iii*) *facility selection*. All three phases are fully implemented in Giraph, and the code is open-source and available on GitHub.[2]

The first phase builds an all-distances sketch (ADS) that estimates the neighborhood function of each vertex. This sketch is used in the second phase to decide when to open a facility. To do so, we use the historic inverse probability (HIP) estimator, recently proposed by Cohen [14].

The facility-opening phase expands balls around facilities in parallel. It decides which facilities to open depending on the number of clients that reside within the facility-centered balls. To estimate the number of clients inside the balls, we use the sketch created in the previous phase.

Finally, the facility-selection phase removes duplicate assignments of a client to more than one facility that might have been created due to the parallel nature of the algorithm. To do so, it computes a maximal independent set (MIS) on the 2-hop graph of the open facilities. For this sub-problem, we provide a parallel implementation of a recent greedy approximation algorithm [5].

Concretely, the contributions of this paper are as follows:

- we provide the first Pregel solution for the facility-location problem. Our algorithm, unlike previous sequential and PRAM ones, is deployable on clusters available in modern computing environments;

- our solution uses the sparse graph-based representation of the facility-location problem, which improves significantly the scalability of the method;

- our facility-location algorithm employs fundamental sub-problems, such as all-distances sketch (ADS) and max-

imal independent set (MIS), for which we also provide the first implementations in the Pregel model;

- we provide an extensive experimental evaluation that shows the scalability of our methods on very large datasets.

2. APPLICATION SCENARIOS

To further motivate the use of facility location in large-scale social-network analysis, we outline two application scenarios, both inspired by the Twitter micro-blogging platform, but applicable to other social-media systems as well.

Who to follow on Twitter? Deciding who to follow on Twitter poses a challenging trade-off: on the one hand we do not want to miss interesting news, on the other hand we want to avoid been bombarded with irrelevant information. One way to tackle this problem is to look for a small set of "aggregator" users who are likely to reproduce (e.g., retweet, reshare, or comment upon) most of the relevant information that we are interested in. Additionally, we would like to receive information in a timely manner, so we also seek to minimize the delay introduced by these aggregators.

The problem of deciding who to follow on Twitter can be formulated naturally as a facility-location problem. The set of clients in this instance of the problem consists of the set of relevant news items for a user u. The set of facilities consists of all the Twitter users that u can possibly follow.

Following user v corresponds to opening its facility, and it has a cost which is proportional to the amount of content that user v generates (as user u will have to skim over all that content). This cost can easily be expressed in terms of time. On the other hand, as user v is likely to produce content related to certain news items, following user v corresponds to serving some clients, which are the relevant news items that user v will reproduce. The cost of a news item (client) e served by a user (facility) v corresponds to the expected time that the item will take to reach and be reproduced by v. An estimate of the expected time required for an interesting news item to reach a certain user can be obtained by processing past logs of Twitter activity.

In summary, in this instance of the facility-location problem we are interested in selecting a set of users in order to minimize the time taken to read the content produced by these users plus the time required for interesting news items to be reproduced by these same users.

Network-based summarization of Twitter activity. This second application scenario addresses the problem of summarizing the overall activity in the Twitter network. The example has a somewhat similar flavor to the first one, but it is qualitatively and quantitatively different.

In this case, we are asked to represent the whole micro-blogging activity in the network, say all the topics discussed (or hashtags), together with the identity of the users who have mentioned those topics. A naïve way to represent such an activity dataset is to build the set $\{(u, t)\}$, where (u, t) indicates the fact that user u has mentioned topic t.

Consider now *compressing* the activity dataset $\{(u, t)\}$ by exploiting the underlying social characteristics of the network. In particular, as Twitter users influence their friends, it is likely that certain topics are confined to local neighborhoods of the social network. Based on this observation, we can represent the overall activity in the network in two parts: (*i*) a seed subset of users and all the topics they have mentioned; (*ii*) every other user who has mentioned a topic

---

[1] Note that we do not require F and C to be disjoint. In fact, in many cases of interest it is $V = F = C$.

[2] `https://github.com/gvrkiran/giraph-facility-location`

can be represented by a pointer (path) to their closest seed user who has mentioned the same topic.

We can formulate this problem as an instance of the facility-location problem by using the minimum description length (MDL) principle. Selecting seed users corresponds to opening facilities, and the cost of opening a facility corresponds to the total space required to describe the topics discussed by the corresponding user. Similarly, describing a topic for a non-seed user via a path to a seed user can be modeled as the service cost from a client to its closest facility. By using the MDL framework, both costs, opening a facility and service costs, can be neatly expressed in information bits, and the problem can be treated as a data compression problem.

This activity-summarization formulation can be seen from two different points of view: (i) as a data-compression task, per se; and (ii) as a data-clustering and data-reduction task, where the goal is to summarize the overall activity in the network by selecting the most central users, and ensuring that the rest of the activity is performed by users who are sufficiently close to the selected set of central users.

3. PRELIMINARIES

Before presenting our distributed algorithm for the facility-location problem, we briefly define the problem and specify the setting we are considering in this paper.

3.1 Problem definition

In the *metric uncapacitated facility-location problem*, we are given a set of facilities F and a set of clients C. For each facility $f \in F$ there is an associated cost $c(f)$ for *opening* that facility. Additionally, a distance function $d : C \times F \to \mathbb{R}_+$ is defined between the facilities and the clients. The distance satisfies the triangle inequality. The objective in the facility-location problem is to select a set of facilities $S \subseteq F$ to *open* in order to minimize the following objective function:

$$\sum_{f \in S} c(f) + \sum_{c \in C} d(c, S),$$

where $d(c, S)$ is the distance of client $c \in C$ to its *closest* opened facility, i.e.,

$$d(c, S) = \min_{f \in S} d(c, f).$$

In this paper, we are interested in the *graph setting* of the facility-location problem [48], where we are also given a weighted graph $G = (V, E, w)$, with $w : E \to \mathbb{R}_+$ a weight function on the edges of the graph. The sets of facilities and clients are subsets of the graph vertices ($F, C \subseteq V$). The distance between clients and facilities is given by the shortest-path distance on the weighted graph.

The motivation for focusing on the graph setting of the facility-location problem is to take advantage of the fact that many real graphs, such as web graphs and social networks, are sparse. The goal is to leverage the sparsity of such graphs in order to develop practical and scalable algorithms. Thus we aim for algorithms whose complexity is quasilinear with respect to the size of the underlying graph.

We note that algorithms for the classical formulation of the metric uncapacitated facility-location problem [4, 29] are not straightforward to adapt to the graph setting, as they require all pairwise vertex distances to be provided in input. This requirement is clearly not practical when the input graph is large.

In our work we consider both directed and undirected graphs. We develop an efficient algorithm with provable guarantees (on the quality of the solution delivered by the algorithm) for the undirected case, while the algorithm provides a practical heuristic for the case of directed graphs. We focus on the case when $F = C = V$, although all our claims hold for the more general case when $F, C \subseteq V$.

3.2 The Giraph platform

The algorithms presented in this paper are designed for the Giraph platform [12], an Apache implementation of the Pregel computational paradigm. Pregel is based on the Bulk Synchronous Parallel (BSP) computation model, and can be summarized by the motto "think like a vertex" [36]. At the beginning of the computation, the vertices of the graph are distributed across worker tasks running on different machines on a cluster. Computation proceeds as a sequence of iterations called supersteps. Algorithms are expressed in a vertex-centric fashion inside a `vertex.compute()` function, which gets called on each vertex exactly once in every superstep. The computation involves three activities: receiving messages from the previous superstep, updating the local value of the vertex, and sending messages to other vertices.

Pregel also provides *aggregators*, a mechanism for global communication and monitoring. Each vertex can write a value to an aggregator in superstep t, the system combines those values via a reduction operator, and the resulting value is made available to all vertices in superstep $t + 1$. Aggregators can be used for global statistics, e.g., to count the total number of edges by using a `sum` reduction. They can also be used for global coordination. For instance, a `min` or `max` aggregator applied to the vertex ID can be used to select a vertex to play a special role. Additionally, one branch of `vertex.compute()` can be executed until a boolean `and` aggregator determines that all vertices satisfy some condition.

Giraph adds an optional `master.compute()` to the Pregel model. This function performs centralized computation, and is executed by a single master task before each superstep. It is commonly used for performing serial computation, and for coordination in algorithms that are composed of multiple vertex-centric stages by using aggregators [44]. Aggregators written by workers are read by the master in the following superstep, while aggregators written by the master are read by workers in the same superstep. We employ this feature in our implementation (see Section 4.5).

3.3 Approximate neighborhoods (ADS)

The all-distances sketch (ADS) is a probabilistic data structure for approximating the neighborhood function of a graph [14]. Namely, ADS aims to answer the query *"how many vertices are within distance d from vertex v?"*. ADS maintains a logarithmic-size sketch for each vertex. In the sequential computational model, the total time to build the ADS is quasilinear in the number of graph edges. Once built, the ADS of a vertex can be used to estimate the number of vertices within some distance. It has also been used to estimate other properties of the graph, such as distance distribution, effective diameter, and vertex similarities [6, 15].

The ADS of a vertex v consists of a random sample of vertices. The probability that a vertex u is included in the sketch of vertex v decreases with the distance $d(u, v)$. The sketch contains not only the vertex u but also the distance $d(u, v)$. The ADS can be thought as an extension of the

Algorithm 1: Build ADS sequentially

Input: Graph $G(V, E)$
Output: ADS of G
1 for $v \in V$ **do**
2 $ADS(v) = \varnothing$
3 $BKMH(v) = \varnothing$ `// Bottom-k min-hash`
4 for $v \in V$ *and* $u \in \{V$ *sorted by* $d(v)\}$ **do**
     `// list vertices in incr. distance from` v
5   **if** $r(u) < \max_r(BKMH(v))$ **then**
         `// r(u) is the hash of` u
6 $ADS(v) \leftarrow ADS(v) \cup (u, d(v, u))$
7 $BKMH(v) \leftarrow \text{bottomK}(BKMH(v) \cup u)$
8 return ADS

Algorithm 2: Build ADS in Giraph. `Vertex.Compute()`

Input: vertex value v, edge values E, messages M
Output: updated vertex value v'
Data: ADS $= \varnothing$; BKMH $= \varnothing$
`// state variables are stored in the vertex` v
1 $\text{OutMsgs} = \varnothing$
2 for $m \in M$ **do**
     `// the message contains the entries of the`
     `// ADS of neighbors that were updated in`
     `// the previous super step`
3   **for** $(u, d) \in m.getEntries()$ **do**
         `// u is the vertex.id and d its distance`
4   **if** $r(u) < \max_r(BKMH)$ **then**
             `// if u has already reached v before,`
             `// it will not be considered again`
5 $ADS \leftarrow ADS \cup (u, d)$
6 $\text{CleanUp}(ADS(u), d)$ `// for each`
                 `distance, remove an entry from`
                 `ADS(u) if its hash is not in the`
                 `bottom-k for that distance`
7 $BKMH \leftarrow \text{bottomK}(BKMH \cup u)$
8 $\text{OutMsgs} \leftarrow \text{OutMsgs} \cup (u, d + e(u, v))$
             `// for unweighted graphs`
             `// e(u,v) is 1`
             `// for weighted graphs, its the weight`
             `// of the edge between u and v`
9 for $e \in E$ **do**
10 $\text{sendMsgTo}(e, \text{OutMsgs})$

simpler *min-hash* sketch [7, 13], which has been used for approximate distinct counting [13, 19, 20], and for similarity estimation [7, 13]. The ADS of v is simply the union of the min-hash sketches of all the sets of the ℓ closest vertices to v, for each possible value of ℓ. Min-hash sketches have a parameter k that controls the trade-off between size and accuracy: a larger k entails a better approximation at the expense of a larger sketch. Essentially, k controls the size of the sample. The size of the ADS is bounded by $k \log n$.

Our algorithm relies heavily on a recently-proposed ADS structure, the historic inverse probability (HIP) estimator [14], which extends significantly previous variants and offers novel estimation capabilities. In particular, HIP can be used to answer neighborhood queries for both *unweighted* and *weighted* graphs. It can also be used to answer *predicated* neighborhood queries, that is, to approximate the number of vertices in a neighborhood that satisfy a certain predicate on vertex attributes. We use this latter feature in to exclude already served clients from the estimation of the number of clients within a ball (see Section 4).

Pseudocode for the sequential version of ADS is presented in Algorithm 1, and for the Giraph version in Algorithm 2. Algorithm 2 works for both weighted and unweighted graphs. Line 6 of Algorithm 2 performs a cleanup of the ADS, which removes those entries for which the hash is not in the bottom-k min-hashes for a given distance. This condition happens because the vertices are processed by ADS in Giraph as discovered by a BFS (i.e., ignoring weights), rather than sorted by their distance. This cleanup operation can be time consuming, since it needs to sequentially access all the entries of the ADS. For this reason, we do not perform the cleanup in each superstep, instead, we do it only periodically, when the size of the ADS becomes too large. For unweighted graphs, the cleanup can be avoided altogether, since the order by which the vertices are presented to the ADS in Giraph corresponds to their distance from v. Therefore the bottom-k min-hash for a given distance d is always completed before the vertices at distance $d + 1$ are processed.

4. ALGORITHM

As discussed earlier, our algorithm consists of three phases: (*i*) *neighborhood sketching*, (*ii*) *facility opening*, and (*iii*) *facility selection* via maximum independent set (MIS). This section presents the main body of the algorithm; phases (*ii*) and (*iii*). We first describe the PRAM version of phase (*ii*), and our Pregel version. Then, we present our solution for

the MIS problem. Finally, we describe our implementation for Giraph. In the pseudocode presented below, `for` and `while` loops are meant to be *parallel* (i.e., executed by all vertices in parallel), unless otherwise specified.

4.1 PRAM algorithm for facility location

The distributed algorithm proposed in this paper is inspired by the algorithm by Blelloch and Tangwongsan [4], which is developed for the PRAM computational model. In our paper we adapt it to a Pregel-like platform, and we also extend it to the graph setting, as discussed in Section 3.1. For completeness, we provide a brief overview here.

The algorithm operates in two phases: facility opening, and facility selection. It starts with all facilities being *unopened* and all clients being *unfrozen*.

The algorithm maintains a graph H that represents the connections between clients and open facilities. Initially, the vertices of H consist of the set of facilities F and the set of clients C, while its set of edges is empty, that is, $H = (F \cup C, \emptyset)$. During the execution of the algorithm, if a client c is to be served by a facility f, the edge (c, f) is added in the graph H. In the first phase of the algorithm it is possible for a client c to be connected to more than one facility f in H. However, the facility-selection phase is a "clean up" phase where redundant facilities are closed so that each client is connected to exactly one facility.

In the facility-opening phase, each client tries to reach a facility by expanding a ball with radius α, in parallel. The expansion phase is iterative, and in each iteration the radius of the ball grows by a factor of $(1+\epsilon)$, where ϵ is a parameter that provides an accuracy-efficiency trade-off. Initially, the radius α is set to a sufficiently small value (details below).

The radius of the ball of a client c is denoted by $\alpha(c)$. If a client c is unfrozen, the radius of its corresponding ball is set to the current global value α, while if a client c gets frozen it does not increase the radius of its ball anymore. When a facility f is reached by a sufficiently large number of clients it is declared *open*. This number is proportional to the cost $c(f)$ of opening that facility. In particular, a facility f is opened if the following condition is satisfied:

$$\sum_{c \in C} \max\{0, (1+\epsilon)\alpha(c) - d(c,f)\} \geq c(f). \qquad (1)$$

For a newly opened facility f, all clients c within radius α from f are frozen, and the edges (c,f) are added to the graph H. The facility-opening phase continues by growing α in each iteration by a factor of $(1+\epsilon)$ as long as there is at least one unopened facility and at least one unfrozen client.

In the facility-selection phase, if all the facilities are open but some clients are not yet frozen, these unfrozen clients c are connected to their nearest facility, their radius is set to the distance from it, i.e., $\alpha(c) = \min_f d(c,f)$, and the graph H is updated accordingly.

At this point, a client may be served by more than one facility in H. The final step of the algorithm consists in closing the facilities that are not necessary, as their clients can be served by other nearby facilities. This step relies on computing a maximal independent set (MIS) in an appropriatelly-defined graph $\overline{H}$. In particular, $\overline{H}$ is the graph whose vertices are the open facilities and there is an edge between two facilities f_a, f_b if and only if there is a client c that is connected to both f_a and f_b in H. It is easy to see that a maximal independent set S in $\overline{H}$ has the property that each client c is connected to exactly one facility. Clients whose facility is not in S are assigned to the nearest open facility. Thus, the set of open facilities S returned by the algorithm is a maximal independent set on $\overline{H}$.

To complete the description of the algorithm, the initial ball radius is set to $\alpha_0 = \frac{\gamma}{m^2}(1+\epsilon)$, where $m = |F||C|$ and γ is defined as follows. For each client $c \in C$ we set

$$\gamma_c = \min_{f \in F}\{c(f) + d(c,f)\},$$

and then $\gamma = \max_{c \in C} \gamma_c$.

Pseudocode of the algorithm is shown in Algorithm 3. Blelloch and Tangwongsan [4] prove the following theorem on the quality of approximation.

Theorem 1 ([4]). *For any $\epsilon > 0$, Algorithm 3 has an approximation guarantee of $3 + \epsilon$, while the total number of parallel iterations is $\mathcal{O}(\frac{1}{\epsilon}\log(|F||C|))$.*

4.2 Pregel-like algorithm for facility location

We now discuss how to adapt Algorithm 3 to the graph setting discussed in Section 3.1, as well as in a Pregel-like platform, such as Apache Giraph, discussed in Section 3.2.

The main challenges we need to tackle for adapting the algorithm are the following:

- leverage the sparsity of the graph $G = (V, E)$ to avoid a quadratic blowup of distance computations between facilities and clients;

- compute efficiently, in a distributed manner, a maximal independent set on the graph $\overline{H}$. In particular, as the graphs H and $\overline{H}$ may be dense, it is desirable to compute a MIS of $\overline{H}$ without materializing H nor $\overline{H}$ explicitly.

Algorithm 3: PRAM algorithm for facility location [4]

Input: Facilities F, clients C, distance $d(\cdot, \cdot)$
facility opening cost $c(\cdot)$, accuracy parameter ϵ
Output: Subset of opened facilities S

1   $O \leftarrow \emptyset$ // Opened facilities
2   $U \leftarrow C$ // Unfrozen clients
3   $H \leftarrow (F \cup C, \emptyset)$ // Graph connecting F with C
4   $\alpha \leftarrow \frac{\gamma}{m^2}(1+\epsilon)$ // Initial ball radius
5   **while** $(O \neq F)$ *and* $(U \neq \emptyset)$ **do**
6    $\alpha \leftarrow \alpha(1+\epsilon)$ // Increase ball radius
    // For loops below executed in parallel
7   **for** $c \in U$ **do**
     // Set new radius for unfrozen clients
8     $\alpha(c) \leftarrow \alpha$
9   **for** $f \in D$ **do**
     // Open f if reached by many clients
10   If Eq. (1) is satisfied then add f to O
11   **for** $c \in U$ **do**
12   **if** $\exists f \in O$ *s.t.* $(1+\epsilon)\alpha(c) \geq d(f,c)$ **then**
      // if there is an opened facility
       nearby
13   Remove c from U // Freeze client
14   Add edge (c,f) in H // Update H graph
15 **if** $O = F$ *and* $U \neq \emptyset$ **then**
16   **for** $c \in U$ **do**
17     $f^* \leftarrow \arg\min_f d(c,f)$ // Nearest facility
18   Add edge (c, f^*) in H
19 $\overline{H} \leftarrow (F, E)$ where E contains edges (f_a, f_b) if there is $c \in C$ such that $(c, f_a), (c, f_b) \in E(H)$
20 $S \leftarrow \text{MIS}(\overline{H})$ // maximal independent set on $\overline{H}$
21 **return** S

The latter challenge is discussed in Section 4.3. Coping with the first challenge, boils down to been able to check, for each facility f, whether Equation (1) is satisfied, and thus deciding when to open a facility.

To this end, we rearrange the left-hand side of Equation (1), so as to be able to evaluate it by means of the ADS algorithm discussed in Section 3.3. Observe that in Algorithm 3 the α's take values in the range

$$R = \{\alpha_0, (1+\epsilon)\alpha_0, (1+\epsilon)^2\alpha_0, \dots\}.$$

For every facility f, let $N(f, d)$ be the number of clients within distance d from f, while let $n(f, d)$ be the number of clients whose distance from f is in the range $(d/(1+\epsilon), d]$. Suppose that all clients within distance $\alpha \in R$ from facility f are unfrozen. In this case, we know that for all these unfrozen clients $\alpha(c) = \alpha$, so we can rewrite the left-hand side of Equation (1) as follows:

$$\sum_{c \in C | d(c,f) \leq \alpha} \max\{0, (1+\epsilon)\alpha(c) - d(c,f)\} =$$

$$\sum_{d \in R | d \leq \alpha} n(f, d) \cdot \max\{0, (1+\epsilon)\alpha - d\},$$

where we replace $\alpha(c)$'s with α and rearrange the terms of the summation by grouping terms with the same value.

If some clients within distance α from f are frozen, the former claim might not hold anymore, and we need a more

sophisticated solution. Our goal is then to maintain an approximation of the the left-hand side of Equation (1) incrementally. Let $q(f)$ denote the current approximation computed by our algorithm. Also, for each facility f, let $\hat{N}(f,d)$ be the number of *unfrozen* clients within distance d from f, while let $\hat{n}(f,d)$ be the number of *unfrozen* clients at distance in the range $(d/(1+\epsilon),d]$. At each iteration of the ball-expansion phase, we add a term $t(f,\alpha)$ to $q(f)$. This term accounts for the increase in contribution to $q(f)$ due to the newly-reached unfrozen clients, while subtracting excess contribution due to previous iterations.

The increase in contribution $t(f,\alpha)$ is defined as

$$\sum_{d \in R \mid d \leq \alpha} \hat{n}(f,d) \cdot \max\{0, (1+\epsilon)\alpha - d\}, \qquad (2)$$

if $\alpha = \alpha_0$ (no excess contribution to be subtracted), and

$$\sum_{d \in R \mid d \leq \alpha} \hat{n}(f,d) \cdot (\max\{0, (1+\epsilon)\alpha - d\} - \max\{0, \alpha - d\}), \quad (3)$$

otherwise. The term $t(f,\alpha)$ is added to $q(f)$ in each iteration of the algorithm for the current value of radius α.

The terms $\hat{N}(f,d)$ can be computed efficiently in a distributed fashion by employing the ADS. Given that $\hat{n}(f,d) = \hat{N}(f,d) - \hat{N}(f,d/(1+\epsilon))$, it follows that also the left-hand side of Equation (1) can be computed efficiently in a distributed fashion. To show the validity of our approximation we need the following definition.

Definition 2. *Given real numbers $a, b, \epsilon > 0$, we say that a approximates b with accuracy ϵ, and write $a \approx_\epsilon b$, if $a \in [(1+\epsilon)^{-1}b, (1+\epsilon)b]$.*

Lemma 3. *Given $\epsilon > 0$, consider the quantity $q(f)$ computed as described above, for $f \in F$. Let α be the ball radius at the current step of the algorithm. The following holds:*

$$q(f) \approx_\epsilon \sum_{c \in C} \max\{0, (1+\epsilon)\alpha(c) - d(c,f)\}.$$

Proof. The proof is by induction on the steps of the algorithm. In the first step of the algorithm where $\alpha = \alpha_0$, all clients are unfrozen, so all $\alpha(c)$ are equal to α. Therefore, it follows that

$$\begin{aligned} q(f) &= \sum_{d \in R \mid d \leq \alpha} \hat{n}(f,d)\left((1+\epsilon)\alpha - d\right) \\ &\approx_\epsilon \sum_{c \in C} \max\{0, (1+\epsilon)\alpha(c) - d(c,f)\}. \end{aligned}$$

Now suppose the lemma holds at the $(k-1)$-th step of the algorithm. At step k, $q(f)$ must include the quantities $\max\{0, (1+\epsilon)\alpha(c) - d(c,f)\}$ for each unfrozen client c at step k. If a client c is unfrozen at step k, then the contribution of c to $q(f)$ at step $(k-1)$ is $\max\{0, \alpha(c) - d(c,f)\}$ (by inductive hypothesis). Hence, at step k it suffices to add

$$\max\{0, (1+\epsilon)\alpha(c) - d(c,f)\} - \max\{0, \alpha(c) - d(c,f)\}$$

to $q(f)$, for each unfrozen client c at step k. The lemma follows from simple algebraic manipulations, taking into account the fact that $\alpha(c) = \alpha$ for each unfrozen client. $\qquad\square$

Pseudocode is shown in Algorithm 4. It consists of two main building blocks: an algorithm for deciding which facilities should be opened (Algorithm 5), and an algorithm

---

Algorithm 4: Pregel-like algorithm for facility location (graph setting)

> **Input**: Graph $G = (V,E,d)$, facilities $F \subseteq V$, clients $C \subseteq V$, facility opening cost $c(\cdot)$, accuracy ϵ
> **Output**: Subset of opened facilities S

1   $O \leftarrow \emptyset$ // opened facilities
2   $U \leftarrow C$ // Unfrozen clients
3   $\alpha \leftarrow \alpha_0 \leftarrow \frac{\gamma}{m^2}(1+\epsilon)$ // Initial ball radius
4   $q(f) \leftarrow 0$ for each $f \in F$
    // next one is a sequential while
5   **while** $(O \neq F)$ *and* $(U \neq \emptyset)$ **do**
6     $\alpha \leftarrow \alpha(1+\epsilon)$ // Increase ball radius
7     $\alpha(c) \leftarrow \alpha$ for each $c \in U$
8     $\overline{O} \leftarrow \text{OpenFacilities}(G, F \setminus O, U, c(\cdot), \alpha, \alpha(\cdot), q(\cdot))$
9   **for** $f \in \overline{O}$ **do**
10       $\text{send}(f, \alpha, \text{``FreezeClient''})$ // f sends a
          clients a "FreezeClient" message to
          all vertices within distance α
11   **for** $c \in U$ **do**
12   **if** *c receives a "FreezeClient" message* **then**
13         $U \leftarrow U \setminus \{c\}$
14     $O \leftarrow O \cup \overline{O}$
15   **if** $O = F$ *and* $U \neq \emptyset$ **then**
16   **for** $c \in U$ **do**
17       $\alpha(c) \leftarrow \arg\min_f d(c,f)$
18   $S = \text{MIS}\overline{H}(G, O, C, \alpha(\cdot))$ // Computing a MIS of $\overline{H}$
    without building H nor $\overline{H}$
19   **return** S

---

for computing a maximal independent set of the graph $\overline{H}$ without explicitly building such a graph (Algorithm 6). The pseudocode for distributing messages in the graph (denoted by the `send` procedure) is omitted for brevity.

During the execution of the algorithm, for each open facility f we let $\alpha(f)$ be the value of α when f is opened. Observe that there is an edge (c,f) in H only if (1) $\alpha(c) = \alpha(f)$, (2) c is within distance $(1+\epsilon)\alpha(c)$ from f, and (3) f is open. Therefore, storing the values for $\alpha(f)$ and $\alpha(c)$ allows us not to materialize $\overline{H}$, which might be very costly.

4.3 Maximal independent set

Salihoglu and Widom [44] recently proposed an implementation of the classic Luby's algorithm [35] for computing the MIS in a Pregel-like system such as Giraph. In our approach, we need to compute a MIS of $\overline{H}$ which is essentially the graph H^2 after removing all unopened facilities (and their edges) from H^2. As we do not materialize H nor $\overline{H}$, even computing the degree of a vertex in $\overline{H}$ (which is needed in Luby's algorithm) might require to exchange a large number of messages. Therefore, we resort to another algorithm developed by Blelloch et al. [5] which works as follows.

Initially, all vertices are *active* and a unique ID is assigned randomly to each of them. This operation can be done in one parallel step by letting each of the n vertices pick an integer in the range $[1, n^3]$, uniformly at random. Then, with high probability, the vertex IDs are unique, The ID of facility f is denoted by $\pi(f)$. Then, in every parallel step, each active vertex v checks whether its ID is the minimum among its neighbors. If this is the case, v is included in the maximal independent set and all its neighbors become *inac-*

Algorithm 5: OpenFacilities($G, D, U, c(\cdot), \alpha, \alpha(\cdot), q(\cdot)$)

Input: Graph $G = (V, E)$, unopened facilities D, unfrozen clients U, facility opening cost $c(\cdot)$, current radius α, radius for frozen clients and opened facilities $\alpha(\cdot)$, facility contribution from clients $q(\cdot)$

Output: Newly opened facilities $\overline{O}$

1 **for** $f \in D$ // For each unopened facility
2 **do**
 // use ADS Algorithm 2
3 Compute $\hat{n}(d, f)$ for each $f \in F$ and $d \in R$
4 **if** $\alpha = \alpha_0$ **then**
5   Compute $t(f, \alpha)$ as in Equation (2)
6 **else**
7   Compute $t(f, \alpha)$ as in Equation (3)
8 $q(f) \leftarrow q(f) + t(f, \alpha)$
9 **if** $q(f) \geq c(f)$ **then**
10   add f in $\overline{O}$
  // α's allow not to materialize H nor $\overline{H}$
11   $\alpha(f) \leftarrow \alpha$
12 **return** $\overline{O}$

Algorithm 6: MIS$\overline{H}$($G, O, C, \alpha(\cdot)$)

1 $S \leftarrow \emptyset, A \leftarrow O$
2 **for** $f \in A$ **do**
3 $\pi(f) \leftarrow RAND([1, n^3])$
 // next one is a sequential for
4 **for** $i = 1, \ldots, \lceil \log^2 n \rceil$ **do**
5 **for** $f \in A$ **do**
6   send($f, (1 + \epsilon)\alpha(f), (\pi(f), \alpha(f))$)
7 **for** $c \in C$ **do**
8   $\pi_{\min} = \min_{(\pi(f), \alpha(f)):\alpha(f)=\alpha(c)} \pi(f)$
  send($c, (1 + \epsilon)\alpha(c), \pi_{\min}$)
9 **for** $f \in A$ **do**
10   **if** $\pi_{\min} = \pi(f)$ **then**
11    $S \leftarrow S \cup \{f\}$
12    $A \leftarrow A \setminus \{f\}$
13   send($f, (1 + \epsilon)\alpha(f), \pi_{min}$)
14 **for** $c \in C$ **do**
15   **if** c receives $\pi_{\min}$, send($c, (1 + \epsilon)\alpha(c), \pi_{min}$)
16 **for** $f \in A$ **do**
17   **if** $\pi_{\min} < \pi(f)$, $A \leftarrow A \setminus \{f\}$
18 **return** S

---

tive. This procedure is iterated $O(\log^2 n)$ times, after which it can be shown that the selected vertices induce a maximal independent set in the input graph, with high probability.

As we do not materialize H nor $\overline{H}$, we need to slightly modify the algorithm by Blelloch et al. [5]. Recall that there is an edge (c, f) in $E(H)$ only if 1) $\alpha(c) = \alpha(f)$, 2) c is within distance $(1 + \epsilon)\alpha(c)$ from f, and 3) f is open. Moreover, there is an edge (f_a, f_b) in $E(\overline{H})$ if there exist $c \in C$ such that (c, f_a) and $(c, f_b) \in E(H)$. After determining its ID $\pi(f)$, each facility f sends a message $(\pi(f), \alpha(f))$ to all vertices within distance $(1 + \epsilon)\alpha(f)$ from f. Each client c collects all messages $(\pi(f), \alpha(f))$, and retains only the pairs $(\pi(f), \alpha(f))$ corresponding to the facilities f that c is connected to, i.e., $\alpha(f) = \alpha(c)$. Then, each client computes the minimum ID $\pi_{\min}$ among all the facilities it is connected to, and sends back a message containing $\pi_{\min}$ to all such facilities. Each facility f is included in the maximal independent set if an only if $\pi_{\min} = \pi(f)$, in which case it sends $\pi_{\min}$ to all neighboring facilities (in $\overline{H}$) so that they are removed from the set of active vertices. The last step is performed by letting each facility f send $\pi_{\min}$ to all clients c within distance $(1 + \epsilon)\alpha(f)$, which in turn deliver such message to all facilities within distance $(1 + \epsilon)\alpha(c)$. For pseudocode see Algorithm 6.

In Section 5, we evaluate the algorithms for computing a MIS proposed by Blelloch et al. [5] as well as the algorithm proposed by Salihoglu and Widom [44].

4.4 Approximation guarantee and running time

As a consequence of Theorem 1 and Lemma 3, as well as from the fact that the ADS provides an approximation to the values of $\hat{n}(f, d)$, we are able to show a guarantee on the quality of the solution computed by our algorithm.

Theorem 4. *For any $\epsilon > 0$ and any integer $k \geq 1$, Algorithm 4 has an approximation guarantee of $3 + o(1) + \epsilon$. The total number of parallel iterations is $\mathcal{O}(\frac{D}{\epsilon} \log^2(n))$, while the total number of messages exchanged by vertices is $\mathcal{O}(m)$, with each message requiring $\mathcal{O}(k \log n)$ bits.*

The $o(1)$ term, which denotes a function that goes to zero as n becomes large, is due to the ADS scheme, while k is the ADS bottom-k parameter. Observe that we are able to derive the same approximation guarantees of Thorup [48], but in a distributed setting.

Theorem 4 holds only for undirected graphs. For directed graphs our guarantee does not hold, even though our algorithm can be adapted in a straightforward manner. In our experiments we have used directed graphs as well, and the performance of the algorithm is equally good.

With respect to the running time, the overall number of supersteps required by the algorithm is proportional to the diameter D of the graph. This follows from the hop-by-hop communication between clients and facilities. Thus, as typically real-world graphs have small diameter, we expect our algorithm to terminate in a small number of supersteps.

4.5 Implementation in Giraph

The facility-opening phase consists of two main subroutines, which are implemented by the vertices and masters compute functions: *ball expansion* and *client freezing*. Initially, the algorithm expands the balls around the potential facilities in parallel, one superstep at a time. When one of the balls encompasses a large enough number of clients, the facility at the center of the ball opens. At this point, the `FreezeClients` subroutine gets called to freeze all the clients within the ball. The algorithm then resumes expanding the balls in parallel until another facility opens. This phase terminates when no unfrozen client remains, condition monitored by the master via a `sum` aggregator. By the end of the algorithm, vertices are either open facilities, or frozen clients with at least one facility serving them. Clients may have multiple facilities serving them as a result of concurrent openings or intersecting balls.

We now describe in detail the implementation of this phase of the algorithm in Giraph. The communication and coordination between the two main subroutines is particularly interesting. In Giraph, the communication between master

and workers happens via aggregators.

How to "call a subroutine"? While expanding the balls, we use a boolean aggregator called `SwitchState` to monitor if any facility was opened in the current superstep. Every vertex can write a boolean value to this aggregator, and the master can read the boolean `and` of all the values in its next superstep. The value of the aggregator is computed efficiently in parallel via a tree-like reduction. If `SwitchState` is true, the master writes to another aggregator `State` that represents the current function being computed. By setting the value of `State` to `FreezeClients`, the master can communicate to the vertices to switch their computation, effectively mimicking a subroutine call.

How does `FreezeClients` work? The vertices execute different subroutines by switching on the `State` aggregator. When `FreezeClients` is executed, each facility opened in the last superstep sends a "FreezeClient" message to all the clients within the current radius of the ball. This message contains the ID of the facility, and the distance it needs to reach, i.e., the radius. Each client that receives this message gets activated modifies its state to frozen by the facility whose ID is in the message, and propagates the message to its own neighbors, as explained next. When a vertex deactivates, it writes true on the `SwitchState` aggregator. When all the vertices terminate and deactivate, the master's `SwitchState` aggregator (which is a boolean `and`) becomes true. The master can then resume the `OpenFacilities` routine by writing on the `State` aggregator.

How to send a message to all vertices at distance d? In Giraph, messages are usually propagated along the graph, hop-by-hop. A vertex v that wants to send a message $\mathcal{M}$ to all veritces within distance d, sends to each neighbor u a message containing $\mathcal{M}$, as well as, the remaining distance $d - d(v, u)$, if such a distance is larger than zero. The message is then in turn propagated by u to its neighbors if the remaining distance is larger than zero. If a vertex receives multiple copies of the same message $\mathcal{M}$ it propagates only the one with maximum remaining distance. This subroutine takes several supersteps to complete, proportional to the distance to reach, and sends a number of messages proportional to the number of edges within distance d.

How to estimate the number of unfrozen clients? We employ ADS to estimate $N(f_i, d)$, i.e., the number of unfrozen clients within distance d from f_i. To do so, we use the *predicated* query feature of ADS. Given that ADS is composed by a sample of the vertices in a graph (for each possible distance), we can obtain an unbiased sample of a subset of the vertices that satisfy a predicate simply by filtering the ADS with such predicate. That is, we can apply the condition *a posteriori*, after having built the ADS.

However, there is another issue to solve in our setting. The predicate we want to compute (unfrozen) is dynamic, as clients are frozen continuously while the algorithm is running. Therefore, we implement this predicate by maintaining explicitly the set of frozen clients. Whenever a client is frozen, it writes its own ID to a custom aggregator which computes the set union of all the values written in it. At the next superstep, each facility has access to this set, and can use it to filter the ADS for the following query. Notice that, even though this set can grow quite large, it can be approximated by using a bloom filter at the cost of decreased accuracy in the estimate. However, our experiments are not

affected by this issue, so for simplicity we do not explore the use of bloom filters, and defer its study to a later work.

5. EXPERIMENTS

We perform extensive experiments to test our approach on several datasets by using a shared Giraph cluster containing up to 500 machines. We design our experiments so as to answer the following questions:

Q1: What is the performance of ADS and how does it affect the main algorithm? (Section 5.1)

Q2: How does our algorithm compare with state-of-the-art sequential ones, in terms of quality? (Section 5.2)

Q3: What is the scalability of our approach in terms of time and space? (Section 5.3)

Q4: How do the two implementations of MIS compare with each other? (Section 5.4)

Parameters. There are two parameters of interest in our approach: the parameter k of the bottom-k min-hash, which regulates the space-accuracy tradeoff of ADS, and the parameter ϵ, which regulates the time-accuracy tradeoff in the facility-location algorithm.

Datasets. Table 1 summarizes the datasets used in our experiments. We use both synthetic and real-world datasets. We use two types of synthetic datasets and create instances with exponentially increasing sizes to test the scalability of our approach. We choose graph-generation models that resemble real-world graphs. The first type of synthetic graphs is generated by using the Forest Fire (FF) model [32] with the following parameter values: the forward burning probability is set equal to 0.3 and the backward equal to 0.4. The second type of synthetic graphs uses the recursive matrix model (RMAT) [9] with parameters $a = 0.45$, $b = 0.15$, $c = 0.15$, and $d = 0.25$.[7] This model can only generate graphs with a number of vertices that is a power of 2. For weighted graphs, we assign weights between 1 and 100, uniformly at random.

5.1 Evaluation of ADS

As described in Section 3.3, we approximate neighborhood sizes by using the ADS data structure. To the best of our knowledge, we are the first to implement and test ADS on

---

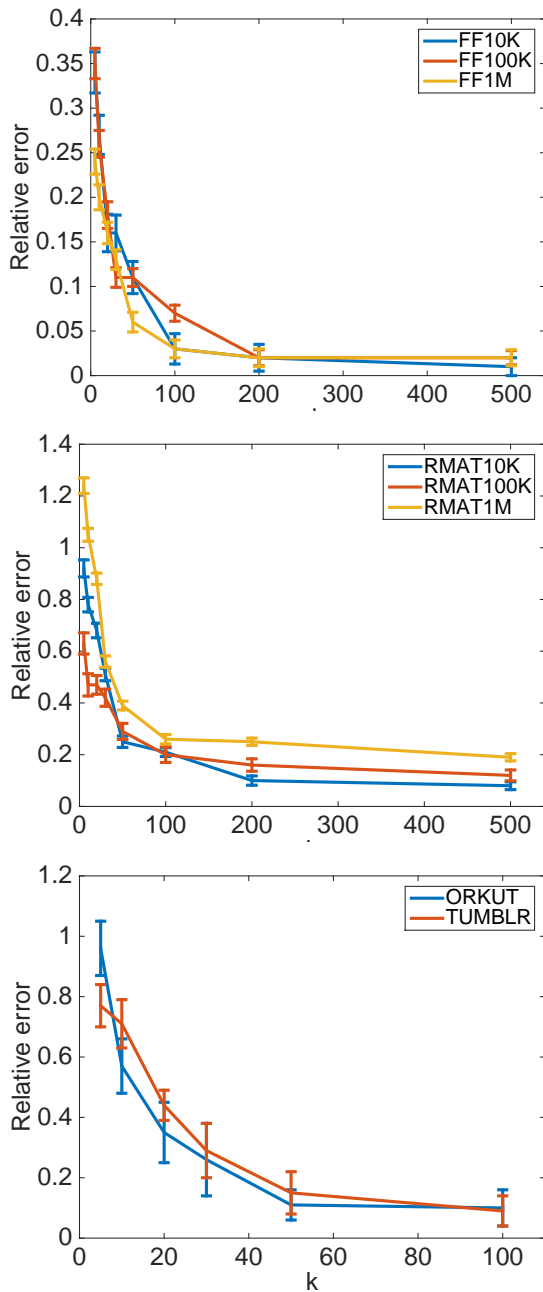[7]For both models, FF and RMAT, we use the default parameters that the data generators come with.

Table 1: Datasets.

Name	$|V|$	$|E|$	Description
FF10K	10k	712k	
FF100K	100k	11M	Forest Fire random graphs
FF1M	1M	232M	
FF10M	10M	1.6B	
RMAT10K	2^{13}	3M	
RMAT100K	2^{17}	5M	R-MAT random graphs
RMAT1M	2^{20}	30M	
RMAT10M	2^{23}	500M	
ORKUT	3M	117M	Orkut social network[3]
TUMBLR	10M	166M	Tumblr reblog network[4]
UK2005	39M	1.4B	Web graph of the .uk domain[5]
FRND	65M	1.8B	Friendster social network[6]

Figure 1: ADS relative error vs. k (unweighted graphs).

Figure 2: ADS relative error vs. k (weighted graphs).

Figure 3: ADS time taken vs. k (unweighted graphs).

Giraph on large scale. In this section, we perform experiments to validate our choice and assess the quality of the ADS estimates. First, we evaluate the quality of ADS approximation by comparing against exact neighborhood sizes. Then, we experiment with the time taken for computing the ADS as a function of k.

Accuracy vs. k: To evaluate the accuracy of the estimates produced by ADS, we need to compute exact neighborhood sizes. Since such computation is infeasible for large graphs, we compute exact neighborhood sizes on a sample of vertices. For each neighborhood distance (from 1 to 20 for unweighted graphs, and from 100 to 2000, at increments of 100, for weighted graphs), we sample 100 random vertices
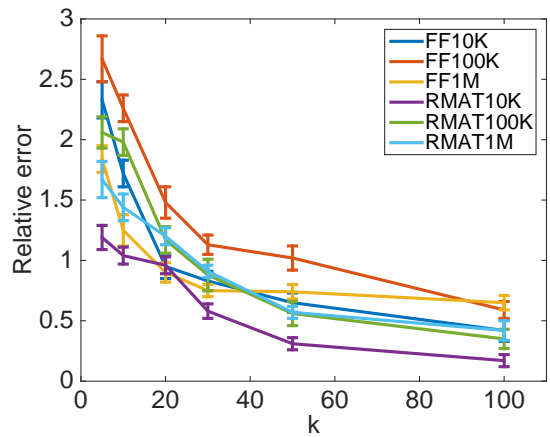
and compute their exact neighborhood sizes. For each sampled vertex and each distance, we compute the relative error as $|S_E - S_{\text{ADS}}|/S_E$, where S_E is the exact neighborhood size and S_{ADS} the ADS estimate. Relative error averages and variances across the 2000 samples are reported in Figure 1 (unweighted) and Figure 2 (weighted).

We can see that the estimates are of high quality. In most cases, even for small values of k, the average relative error is less than 50%. The variance is also small.

Note that in Figures 1 and 2 we do not report accuracy for the largest of our datasets. The reason is that computation of exact neighborhood sizes becomes a bottleneck.

Time vs. k: Next, we measure the time taken to compute the ADS as a function of k, as shown in Figure 3. We clearly see that even for graphs with 1 million vertices and k as large as 500, the algorithm finishes in less than 800 seconds.

Space requirements: Since increasing the value of k does not increase the time taken by the algoritm, one would assume that we could use a very high value of k in order to improve the quality of approximation of ADS. The bottleneck, though, is the size of the ADS, which is proportional to k. Thus, increasing k increases the memory requirements of ADS. In our setting, we have a limitation of 3.5 GB of memory per machine, which makes it infeasible to store an ADS for large graphs with very large values of k (say, $n >$ 10m and $k > 200$). Recall, however, that the size of the ADS

Table 2: Relative cost of the Giraph algorithm against the sequential one ($k = 200$).

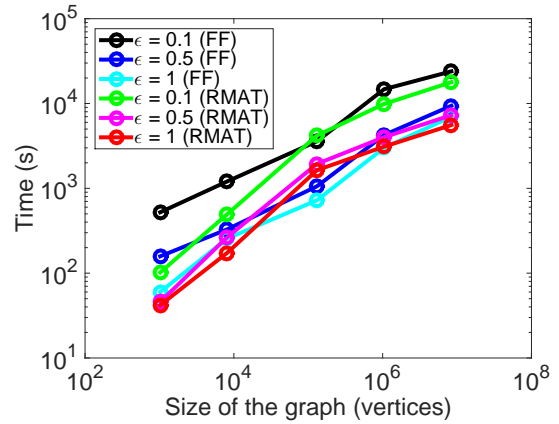| Type	$|V|$	$|E|$	$\epsilon = 0.01$	$\epsilon = 0.1$	$\epsilon = 1$
FF	1k	11k	1.21	1.46	2.56
FF	2k	25k	1.15	1.60	2.45
FF	3k	60k	1.07	1.75	2.47
FF	4k	67k	1.08	1.48	2.16
FF	5k	121k	1.05	1.5	2.13
FF	6k	206k	1.01	1.41	2.02
FF	7k	268k	1.03	1.33	1.67
FF	8k	380k	1.03	1.25	1.55
FF	9k	520k	1.01	1.18	1.41
FF	10k	712k	1.02	1.09	1.43
RMAT	2^{10}	100k	1.08	1.55	1.88
RMAT	2^{11}	200k	1.06	1.36	1.7
RMAT	2^{12}	500k	1.05	1.35	1.73
RMAT	2^{13}	800k	1.05	1.24	1.44
RMAT	2^{14}	1000k	1.02	1.14	1.39

Figure 4: Time taken by the algorithm for different values of ϵ on several graphs.

Figure 5: Time taken by each phase of the algorithm.

is proportional to $nk \log n$, thus, the value of k can increase linearly with the number of available machines.

5.2 Facility-location algorithm

In this section, we evaluate the quality of the Giraph implementation of the algorithm presented in Section 4. We first compare our algorithm against a simple sequential baseline, in terms of the cost function, for different values of the accuracy parameter ϵ. We then evaluate the performance and the running time of the algorithm as a function of ϵ.

Comparison with sequential algorithm: We use the sequential approximation algorithm by Charikar and Guha [10] as a baseline. The algorithm is a simple local-search method that achieves an approximation ratio of $(2.414 + \epsilon)$ and has running time of $\tilde{\mathcal{O}}(n^2/\epsilon)$.

Note that the sequential algorithm assumes the availability of all-pairs shortest path distances, which is computationally very expensive, even for small graphs. Therefore, we perform our evaluation with graphs consisting of no more than 10 000 vertices.

Table 2 shows the results of the comparison in terms of *relative cost*, which is defined as the cost of the sequential algorithm divided by the cost of our algorithm, for different values of ϵ. A smaller value means that our algorithm is competitive with the baseline. We can see that, for small graphs, our algorithm performs quite well, even for large values of ϵ.

Cost vs. accuracy (ϵ): Table 2 shows the relative cost of our algorithm (compared again to the sequential algorithm) with respect to the accuracy. As expected, we get better solutions for smaller values of ϵ, but the solution does not get much worse even for large values of ϵ.

Running time vs. accuracy (ϵ): We also measure the time taken by our algorithm as a function of ϵ. Figure 4 shows these results. From the figure we see that, as expected, our algorithm scales linearly with respect to the size of the graph,[8] and is faster for larger values of ϵ.

5.3 Scalability

---

[8] A linear fit gives R^2 values between 0.8 and 0.9.

Next, we examine the scalability of the different phases of our algorithm. Recall that the three phases of our algorithm are (i) ADS computation (pre-processing), (ii) facility-location algorithm, and (iii) MIS computation (post-processing). Figure 5 presents the time taken, for different datasets, broken down by phase. The total running time, for various graph sizes, is also shown in Figure 6.

For the results shown in Figure 5, we have used $k = 20$, $\epsilon = 0.1$, and 200 machines. Since the running time on a distributed cluster depends on various factors, such as the current load of the machines, we repeat all experiments three times and report the median running time.

5.4 Luby's vs. parallel MIS

As discussed above, we implement two methods for finding the maximal independent set (MIS): Luby's classic one [35], which was also implemented recently by Salihoglu and Widom [44], and a recent algorithm by Blelloch et al. [5].

We compare the two methods in terms of total time taken and number of supersteps needed to converge. Table 3 shows the results. We see that our parallel MIS algorithm is at least 3 to 5 times faster than Luby's algorithm.
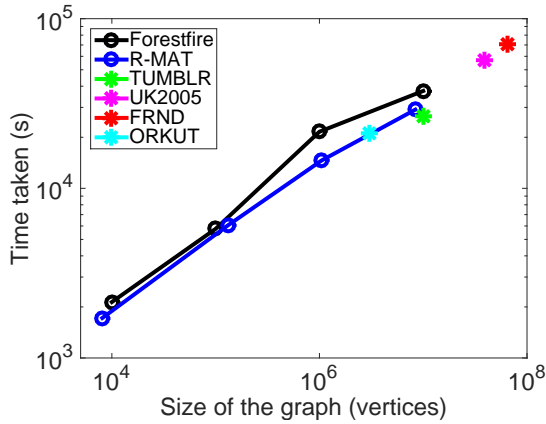
Figure 6: Total time taken by the algorithm.

Table 3: Comparison of two implementations of MIS in Giraph, in terms of supersteps and time taken (median over three runs).

Graph	Supersteps		Time (s)	
	Luby's	MIS	Luby's	MIS
FF10K	750	29	730	104
FF100K	3473	85	1869	296
FF1M	6119	325	6155	1205
FF10M	20 154	1613	11 744	3711
RMAT10K	645	17	616	87
RMAT100K	3200	73	1576	319
RMAT1M	5832	285	4109	1232
RMAT10M	17 557	1533	9492	3181

6. RELATED WORK

Facility location. Facility location is a classic optimization problem. The traditional formulation (metric uncapacitated facility location) is **NP**-hard, and so are many of its variants. Existing algorithms rely on techniques such as LP rounding, local search, primal dual, and greedy. The greedy heuristic obtains a solution with an approximation guarantee of $(1 + \log |\mathcal{D}|)$ [28], while constant-factor approximation algorithms have also been introduced [1, 10]. The approximation algorithm with the best factor so far (1.488) is very close to the approximability lower bound (1.463) [33].

Differently from most previous work, the input to our algorithm is a sparse graph representing potential facilities and clients and their distances, rather than the full bipartite graph of distances between facilities and clients. Note that building the full bipartite graph requires computing all-pairs of distances and implies an $\mathcal{O}(n^2)$ algorithm. Thorup [48] considers a setting similar to ours, and provides a fast sequential algorithm $\tilde{\mathcal{O}}(n + m)$.

Blelloch and Tangwongsan [4] propose a parallel approximation algorithm for facility location in the PRAM model. In this work, we extend the former algorithm to work in a more realistic shared-nothing Pregel-like model. Other parallel algorithms have also been proposed [23, 40, 41].

Applications. Facility location is a flexible model that has been applied successfully in many domains, such as city planning, telecommunications, electronics, and others. For an overview of applications, please refer to the textbook of Hamacher and Drezner [27]. Furthermore, applications of facility location in social-network analysis are provided in our motivation scenarios in Section 2.

Large-scale graph processing. MapReduce [18] is one of the most popular paradigms used for mining massive datasets. Many algorithms have been proposed for various graph problems, such as counting triangles [47], matching [17, 31], and finding densest subgraphs [2].

However, given the iterative nature of most graph algorithms, MapReduce is often not the most efficient solution. Pregel [36] is large-scale graph processing platform that supports a vertex-centric programming paradigm and uses the bulk synchronous parallel (BSP) model of computation. Giraph [12] is an open-source clone of Pregel. It is the platform that we use in this work. Other distributed systems for graph processing have recently been proposed, for instance, Signal/Collect [46], GraphLab/PowerGraph [34, 24], GPS [43], and GraphX [25]. Most of the APIs of these system follow the gather-apply-scatter (GAS) paradigm, which can be readily used to express our algorithm. However, the BSP model is still used due to its simplicity and ease of use.

Algorithms. Our work takes advantage of a number of successful algorithmic techniques. We use the all-distance-sketches (ADS) and the historic inverse probability (HIP) estimator by Cohen [14] to estimate the number of vertices within certain distance from a given vertex. HIP is a cardinality estimator similar to HyperLogLog counters [21] and Flajolet-Martin counters [20]. HyperANF [6] is a related algorithm that approximates the global neighborhood function of the graph by using HyperLogLog counters, but it is not directly usable in our case as we need separate neighborhood functions for each vertex.

7. CONCLUSIONS

We have shown how to tackle the facility-location problem at scale by using Pregel-like systems. In particular, we addressed the graph setting of the problem, which allows to represent the input in sparse format as a graph. We leveraged graph sparsity to tackle problem instances whose size is much larger than previously possible.

Our algorithm is composed by three phases: (i) *neighborhood sketching*, (ii) *facility opening*, and (iii) *facility selection*. We implemented all three phases in Giraph, and published the code as open-source software. For the first phase, we showed how to use ADS with HIP, a recent graph-sketching technique. We adapted an existing PRAM algorithm with approximation guarantees for the second phase. Finally, for the third phase we proposed a new Giraph algorithm for the maximal independent set (MIS), which is much faster than the previous state-of-the-art. Our approach was able to scale to graphs with millions of vertices and billions of edges, thus adding facility location to the toolset of algorithms available for large-scale problems.

This work opens up several new research questions. From the point of view of the practitioner, this algorithm enables to solve large-scale facility-location problems, thus is a candidate for real-world applications in Web and social-network analysis. A more general question is whether better algorithms exist for the setting we consider. Also, it would be interesting to know whether there are any primitives that the system could offer to develop better algorithms.

8. REFERENCES

[1] K. Aardal, D. Shmoys, and E. Tardos. Approximation algorithms for facility location problems. In *STOC*, pages 265–274, 1997.

[2] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest Subgraph in Streaming and MapReduce. *PVLDB*, 5(5): 454–465, 2012.

[3] R. Baraglia, G. De Francisci Morales, and C. Lucchese. Document similarity self-join with MapReduce. In *ICDM*, pages 731–736, 2010.

[4] G. E. Blelloch and K. Tangwongsan. Parallel approximation algorithms for facility-location problems. In *SPAA*, page 315, 2010.

[5] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *SPAA*, pages 308–317, 2012.

[6] P. Boldi, M. Rosa, and S. Vigna. HyperANF: Approximating the neighbourhood function of very large graphs on a budget. In *WWW*, pages 625–634, 2011.

[7] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences*, pages 21–29, 1997.

[8] A. L. Buchsbaum, D. F. Caldwell, K. W. Church, G. S. Fowler, and S. Muthukrishnan. Engineering the compression of massive tables: an experimental approach. In *SODA*, pages 175–184, 2000.

[9] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.

[10] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and *k*-median problems. In *FOCS*, pages 378–388, 1999.

[11] F. Chierichetti, R. Kumar, and A. Tomkins. Max-cover in map-reduce. In *WWW*, pages 231–240, 2010.

[12] A. Ching and C. Kunz. Giraph: Large-scale graph processing infrastructure on Hadoop. *Hadoop Summit*, 2011.

[13] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[14] E. Cohen. All-distances sketches, revisited: HIP estimators for massive graphs analysis. In *PODS*, pages 88–99, 2014.

[15] E. Cohen, D. Delling, F. Fuchs, A. V. Goldberg, M. Goldszmidt, and R. F. Werneck. Scalable similarity estimation in social networks: Closeness, node labels, and random edge length. In *COSN*, pages 131–142, 2013.

[16] G. De Francisci Morales, C. Lucchese, and R. Baraglia. Scaling Out All Pairs Similarity Search with MapReduce. In *LSDS-IR*, 2010.

[17] G. De Francisci Morales, A. Gionis, and M. Sozio. Social Content Matching in MapReduce. *PVLDB*, 4(7):460–469, 2011.

[18] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, page 10, 2004.

[19] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *ESA*, 2003.

[20] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[21] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA*, pages 127–146, 2007.

[22] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: a system for extracting document type descriptors from XML documents. *ACM SIGMOD Record*, 29(2):165–176, 2000.

[23] J. Gehweiler, C. Lammersen, and C. Sohler. A distributed $O(1)$-approximation algorithm for the uniform facility location problem. In *SPAA*, pages 237–243, 2006.

[24] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, pages 17–30, 2012.

[25] J. Gonzalez, R. Xin, and A. Dave. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[26] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *FOCS*, 2000.

[27] H. W. Hamacher and Z. Drezner. *Facility location: applications and theory*. Springer, 2002.

[28] D. S. Hochbaum. Heuristics for the fixed cost median problem. *Mathematical Programming*, 22(1):148–162, 1982.

[29] K. Jain, M. Mahdian, E. Markakis, A. Saberi, and V. Vazirani. Greedy facility location algorithms analyzed using dual fitting with factor-revealing LP. *Journal of the ACM*, 50(6):795–824, 2003.

[30] A. Kuehn and M. Hamburger. A heuristic program for locating warehouses. *Management science*, 9(4):643–666, 1963.

[31] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: A method for solving graph problems in mapreduce. In *SPAA*, pages 85–94, 2011.

[32] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 1 (1):2, 2007.

[33] S. Li. A 1.488 approximation algorithm for the uncapacitated facility location problem. In *ICALP*, pages 77–88, 2011.

[34] Y. Low, D. Bickson, and J. Gonzalez. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.

[35] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15 (4):1036–1053, 1986.

[36] 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 *SIGMOD*, pages 135–145, 2010.

[37] A. Manne. Plant location under economies-of-scale-decentralization and computation. *Management Science*, 11(2):213–235, 1964.

[38] F. M. Manshadi, B. Awerbuch, R. Gemula, R. Khandekar, J. Mestre, and M. Sozio. A distributed algorithm for large-scale generalized matching. *PVLDB*, 6(9):613–624, 2013.

[39] A. Mirzaian. Lagrangian relaxation for the star-star concentrator location problem: Approximation algorithm and bounds. *Networks*, 15(1):1–20, 1985.

[40] T. Moscibroda and R. Wattenhofer. Facility location: distributed approximation. In *PODC*, pages 108–117, 2005.

[41] S. Pandit and S. Pemmaraju. Return of the primal-dual: distributed metric facility location. In *PODC*, pages 180–189, 2009.

[42] L. Qiu, V. Padmanabhan, and G. Voelker. On the placement of web server replicas. In *INFOCOM*, pages 1587–1596, 2001.

[43] S. Salihoglu and J. Widom. GPS: A graph processing system. In *SSDBM*, page 22, 2013.

[44] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 7(7):577–588, 2014.

[45] D. B. Shmoys. Approximation Algorithms for Facility Location Problems. *AACO*, 1913:27–32, 2001.

[46] P. Stutz, A. Bernstein, and W. Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web. In *ISWC*, pages 764–780, 2010.

[47] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

[48] M. Thorup. Quick k-median, k-center, and facility location for sparse graphs. In *ICALP*, pages 249–260, 2001.

[49] G. Zuccon, L. Azzopardi, D. Zhang, and J. Wang. Top-*k* retrieval using facility-location analysis. In *ECIR*, pages 305–316, 2012.