Scalable Facility Location for Massive Graphs on Pregel-like Systems

Kiran Garimella
Aalto University
kiran.garimella@aalto.fi

Gianmarco De Francisci Morales
Aalto University
gdfm@acm.org

Aristides Gionis*
Aalto University
aristides.gionis@aalto.fi

Mauro Sozio†
Télécom ParisTech,
CNRS LTCI
sozio@telecom-paristech.fr

ABSTRACT

We propose a new scalable algorithm for the facility-location problem. We study the *graph setting*, where the cost of serving a client from a facility is represented by the shortest-path distance on a graph. This setting is applicable to various problems arising in the Web and social media, and allows to leverage the inherent sparsity of such graphs.

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 work builds upon previous results: a facility location algorithm for the PRAM model, a recent distance-sketching method for massive graphs, and a parallel algorithm to finding maximal independent sets. The main challenge is to adapt those building blocks to the distributed graph setting, while maintaining the approximation guarantee and limiting the amount of distributed communication. Extensive experimental results show that our algorithm scales gracefully to graphs with billions of edges, while, in terms of quality, being competitive with state-of-the-art sequential algorithms.

1. INTRODUCTION

Facility location is a classic combinatorial-optimization problem. It has been widely studied in *operations research* [31, 39] and *theoretical computer science* [1, 11, 30, 48], and it has many different applications, e.g., in data compression [9], grammar inference [23], information retrieval [49], and design of communication networks [43]. 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.

Modern 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, could also benefit from facility location as a general-purpose optimization mechanism. Examples of such applications include placing caches for content delivery on the Internet [24], placement of shopping centers on a road network, finding central nodes in a social network [7]. Similar objective functions arise in outbreak detection in networks [34].

We study the facility location problem in the graph setting, where the input consists of a graph with clients and facilities being represented by vertices while the cost of serving a client from a facility is represented by their shortest-path distance on the graph. Our goal is to develop efficient distributed algorithms for this problem setting.

Several sequential approximation algorithms [1, 11, 30, 48], as well as parallel/distributed algorithms [13, 4] have been devised for the facility location problem. Alas, adapting existing methods to the distributed graph setting is challenging, as the known algorithms present at least one of the following shortcomings: (i) they assume the input to be the full distance matrix between facilities and clients, which requires $\Omega(n^2)$ space and time to be materialized (where n is the number of vertices in the graph); (ii) they are sequential and assume that the input data reside in main memory.

In this paper, we present the first algorithm for the facility-location problem that addresses both issues outlined above. Our algorithm is designed for the Pregel model of computation [38]. In particular, we implement our algorithm on Giraph [12], thus adding facility location to the toolbox of optimization problems that can be solved for very large datasets on modern computer clusters.

Graph setting. 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 [46]. 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.

In many real-world problems the input for facility location can be represented as a graph $G = (V, E)$, where F and C are vertices of G. Typically, the number of edges $m = |E|$ is much smaller than $|F||C|$, 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 *induced shortest-path distance*. Hence, in this so-called *graph-setting*, we do not need to represent the full $|F| \times |C|$ set of distances. This setting poses non-trivial challenges, as such distances have to be computed

*This work was supported by a Yahoo! Faculty Research and Engagement Program (FREP) award.
†This work was partially funded by a Google Faculty Award.

CIKM'15, October 19–23, 2015, Melbourne, VIC, Australia
Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-3794-6/15/10 $15.00 DOI: `http://dx.doi.org/10.1145/2806416.2806508`.

throughout the execution of the algorithm. However, it is crucial to leverage the sparsity of a massive input graph.

Our approach is designed for this graph-based setting. We require that the time and space requirements of our algorithm be quasilinear functions of $|E|$. If the graph is sparse, as most real-world datasets are, this leads to a significantly more scalable algorithm. Thorup [48] has proposed the only previous (sequential) algorithm for the graph setting.

Pregel model. To cope with large problem sizes modern applications take advantage of distributed systems, such as MapReduce [19] and Hadoop, or of variants targeted to graph data, such as Pregel [38] and its open-source clones, Giraph [12] and GraphLab [36]. Such systems offer several advantages, among which, high scalability and a simple programming interface.

Our algorithm targets a parallel shared-nothing computing environment. While other parallel algorithms have been proposed in the literature, our approach is the first one to target modern 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.

Algorithm summary. Our approach has three phases: (*i*) *neighborhood sketching*, (*ii*) *facility opening*, and (*iii*) *facility selection*. All three phases are fully implemented in Giraph, and the code is open-source.[1]

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. Our sketch relies on the historic inverse probability (HIP) estimator, recently proposed by Cohen [15].

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 accomplish this task, we need to compute a maximal independent set (MIS) on the 2-hop graph of the open facilities. For this sub-problem, we design a distributed version of a recent greedy approximation algorithm [5].

Our main challenge is to adapt and combine previous results to the distributed graph setting, while maintaining the approximation guarantee of the algorithm, and limiting the amount of distributed communication. 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;
- at the same time, we do not compromise on accuracy – our algorithm provides an approximation guarantee similar to previous PRAM and sequential ones;
- our solution uses the sparse graph-based representation of the facility-location problem, which improves significantly the scalability of the method;
- our algorithm employs fundamental subproblems, all-distances sketch and maximal independent set, for which we 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.

The rest of the paper is organized as follows. In Section 2 we formally define our problem, and discuss the background techniques needed for our approach. Our algorithm, comprising of the three phases outlined above, is detailed in Section 3, while an experimental evaluation of the different components of our method is presented in Section 4. In Section 5 we place our work in the context of relevant research, while Section 6 is a short conclusion.

2. PRELIMINARIES

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 $f \in F$ and client $c \in C$, there is a cost $c(f)$ for *opening* the facility f, and a cost $d(c, f)$ for *serving* client c with facility f. The objective is to select a set of facilities $S \subseteq F$ to *open* in order to minimize the 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* open 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, 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. We assume that facility costs and edge weights be polynomial in $|V|$.

We focus on the graph setting of the facility-location problem so as to leverage the sparsity of real-world graphs, such as web graphs and social networks. This allows us to develop practical and scalable algorithms, with running time being quasilinear in the size of the input graph. Previous algorithms for our problem, are sequential or require all pairwise vertex distances which is not practical for large graphs.

We consider both directed and undirected graphs. For the case of undirected graphs our algorithm offers a provable approximation guarantee, while for the case of directed graphs the algorithm provides a practical heuristic. We focus on the case when $F = C = V$, although all our claims hold for the more general case when $F, C \subseteq V$.

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” [38]. 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.

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

Algorithm 1: Build ADS sequentially

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

Pregel also provides *aggregators*, a mechanism used for coordination and monitoring, as well as for computing global statistics. 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 $t+1$.

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. 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 3.5).

Approximate neighborhoods (ADS). The all-distances sketch (ADS) is a probabilistic data structure for approximating the neighborhood function of a graph [15]. 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. ADS-based techniques have been used to estimate efficiently graph properties, such as the distance distribution, effective diameter, and vertex similarities [6, 16].

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 simpler *min-hash* sketch [8, 14], which has been used for approximate distinct counting [14, 20, 21], and for similarity estimation [8, 14]. 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. The size of the ADS is $\mathcal{O}(k \log n)$.

Our algorithm relies heavily on a recently-proposed ADS structure, the historic inverse probability (HIP) estimator [15], 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 3).

Algorithm 2: 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  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              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 $\leftarrow$ bottomK(BKMH $\cup$ $u$)
8              OutMsgs $\leftarrow$ 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     sendMsgTo($e$, OutMsgs)
```

The sequential version of ADS is presented in Algorithm 1, and our adaptation for Giraph is shown in Algorithm 2. The algorithm works for both weighted and unweighted graphs. A cleanup operation is performed in line 6, which removes those entries for which the hash is not in the bottom-k min-hashes for a given distance (this may happen only for weighted graphs, when the vertices processed by ADS are discovered by a BFS and are not sorted by their distance from a given vertex). While the cleanup step can be time-consuming, we do not need to perform it in each superstep, but only periodically when the size of the ADS becomes too large. For unweighted graphs the cleanup is not needed.

3. ALGORITHM

As discussed earlier, our algorithm consists of three phases: (*i*) *neighborhood sketching*, (*ii*) *facility opening*, and (*iii*) *facility selection.* This section presents the main body of the algorithm: phases (*ii*) and (*iii*). In the pseudocode presented henceforward, `for` and `while` loops are meant to be *parallel* i.e., executed by all vertices in parallel.

3.1 PRAM algorithm for facility location

Our method is inspired by the algorithm of Blelloch and Tangwongsan [4], which is developed for the PRAM model. We adapt this algorithm to a Pregel-like platform, and also extend it to the graph setting, as discussed in Section 2.

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, H has F and C as vertices and an empty set of edges. 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 H. During the facility-opening phase, it is possible for a client to be connected to more than one facility. However, in the facility-selection phase, redundant facilities are closed and it is ensured 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 grows by a factor of $(1 + \epsilon)$, where ϵ is a parameter that provides an accuracy-efficiency trade-off.

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*. In particular, a facility f is opened when

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

For a newly opened facility f, all clients c within radius α from f are frozen, and the edges (c, f) are added in H. The facility-opening phase continues as long as there is at least one unopened facility and at least one unfrozen client.

At this point, as mentioned earlier, a client may be served by more than one facility. In the facility-selection phase, the algorithm closes 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 appropriately-defined graph $\overline{H}$: the open facilities are the vertices of $\overline{H}$ 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. 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.

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 $\gamma = \max_{c \in C} \gamma_c$. Blelloch and Tangwongsan [4] prove the following theorem on the quality of approximation.

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

3.2 Pregel-like algorithm for facility location

We now discuss how to adapt the algorithm of Blelloch and Tangwongsan to the graph setting discussed in Section 2, as well as in a Pregel-like platform, such as Apache Giraph, discussed in Section 2. The main challenges we need to tackle 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.

The first challenge, i.e., exploiting the sparsity of the graph, boils down to being able to check whether Equation (1) is satisfied so as to decide 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 2. Observe that 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

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

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

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

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 $a \in [(1 + \epsilon)^{-1}b, (1 + \epsilon)b]$.*

Our approximation is quantified with the following Lemma, whose proof is omitted from due to space limitations.

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:*

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

Algorithm 3: 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 $\epsilon$
Output: Subset of opened facilities $S$
$O \leftarrow \emptyset$ // opened facilities
$U \leftarrow C$ // Unfrozen clients
$\alpha \leftarrow \alpha_0 \leftarrow \frac{\gamma}{m^2}(1+\epsilon)$ // Initial ball radius
$q(f) \leftarrow 0$ for each $f \in F$
// next one is a sequential while
while $(O \neq F)$ and $(U \neq \emptyset)$ do
    $\alpha \leftarrow \alpha(1+\epsilon)$ // Increase ball radius
    $\alpha(c) \leftarrow \alpha$ for each $c \in U$
    $\overline{O} \leftarrow$ OpenFacilities$(G, F \setminus O, U, c(\cdot), \alpha, \alpha(\cdot), q(\cdot))$
    for $f \in \bar{O}$ do
        send($f, \alpha$,“FreezeClient”) // $f$ sends a
            "FreezeClient" message to all vertices
            within distance $\alpha$
    for $c \in U$ do
        if $c$ receives a “FreezeClient” message then
            $U \leftarrow U \setminus \{c\}$
    $O \leftarrow O \cup \overline{O}$
if $O = F$ and $U \neq \emptyset$ then
    for $c \in U$ do
        $\alpha(c) \leftarrow \arg\min_f d(c, f)$
$S = \mathrm{MIS}\overline{\mathrm{H}}(G, O, C, \alpha(\cdot))$ // Computing a MIS of $\overline{H}$
    without building $H$ nor $\overline{H}$
return $S$
```

Algorithm 4: 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 $\alpha$, radius for frozen clients and
       opened facilities $\alpha(\cdot)$, facility contribution from
       clients $q(\cdot)$
Output: Newly opened facilities $\overline{O}$
for $f \in D$ // For each unopened facility
do
    // use ADS Algorithm 2
    Compute $\hat{n}(d, f)$ for each $f \in F$ and $d \in R$
    if $\alpha = \alpha_0$ then
        Compute $t(f, \alpha)$ as in Equation (2)
    else
        Compute $t(f, \alpha)$ as in Equation (3)
    $q(f) \leftarrow q(f) + t(f, \alpha)$
    if $q(f) \geq c(f)$ then
        add $f$ in $\overline{O}$
        // $\alpha$'s allow not to materialize $H$ nor $\overline{H}$
        $\alpha(f) \leftarrow \alpha$
return $\overline{O}$
```

Pseudocode for our method is shown in Algorithm 3. It consists of two building blocks: an algorithm for deciding which facilities to open (Algorithm 4), and an algorithm for computing a maximal independent set of the graph $\overline{H}$ without explicitly building such a graph (Algorithm 5). 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.

3.3 Maximal independent set

Salihoglu and Widom [45] recently proposed an implementation of the classic Luby's algorithm [37] 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] and which works as follows.

Initially, all vertices are *active* and a unique ID is assigned randomly to each of them. Let $\pi(f)$ be the ID for facility f. Then, in parallel, 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 *inactive*. This process is iterated $O(\log^2 n)$ times. It can be shown that, with high probability, the selected vertices induce a maximal independent set.

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. 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 5.

In Section 4, we evaluate the proposed algorithms against the algorithm proposed by Salihoglu and Widom [45].

3.4 Approximation guarantee and running time

Combining Theorem 1, Lemma 3, and the fact that ADS provides an approximation to the values of $\hat{n}(f, d)$, we can show an approximation guarantee for our algorithm.

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

The parameter k is related to the bottom-k in ADS. Observe that we are able to derive the same approximation guarantees of Thorup [48], but in a distributed setting.

Algorithm 5: $\mathrm{MIS\overline{H}}(G, O, C, \alpha(\cdot))$

```
$S \leftarrow \emptyset, A \leftarrow O$
for $f \in A$ do
    $\pi(f) \leftarrow RAND([1, n^3])$
// next one is a sequential for
for $i = 1, \ldots, \lceil \log^2 n \rceil$ do
    for $f \in A$ do
        send$(f, (1+\epsilon)\alpha(f), (\pi(f), \alpha(f)))$
    for $c \in C$ do
        $\pi_{\min} = \min_{(\pi(f),\alpha(f)):\alpha(f)=\alpha(c)} \pi(f)$
        send$(c, (1+\epsilon)\alpha(c), \pi_{\min})$
    for $f \in A$ do
        if $\pi_{\min} = \pi(f)$ then
            $S \leftarrow S \cup \{f\}$
            $A \leftarrow A \setminus \{f\}$
            send$(f, (1+\epsilon)\alpha(f), \pi_{min})$
    for $c \in C$ do
        if $c$ receives $\pi_{\min}$, send$(c, (1+\epsilon)\alpha(c), \pi_{min})$
    for $f \in A$ do
        if $\pi_{\min} < \pi(f)$, $A \leftarrow A \setminus \{f\}$
return $S$
```

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 δ 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.

3.5 Implementation in Giraph

The facility-opening phase consists of two subroutines, *ball expansion* and *client freezing*, which are implemented by the vertices and masters compute functions. Initially, the algorithm expands the balls around the potential facilities in parallel. When one of the balls encompasses a large enough number of clients, the facility at the center of the ball opens. At this point, all clients within the ball freeze, via the `FreezeClients` subroutine. 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. We achieve this by using 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. 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.

4. EXPERIMENTS

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

Table 1: Datasets.

Name	$\lvert V\rvert$	$\lvert E\rvert$	Description
FF10K	10k	712k	Forest Fire random graphs
FF100K	100k	11M	
FF1M	1M	232M	
FF10M	10M	1.6B	
RMAT10K	2^{13}	3M	R-MAT random graphs
RMAT100K	2^{17}	5M	
RMAT1M	2^{20}	30M	
RMAT10M	2^{23}	500M	
ORKUT	3M	117M	Orkut social network
TUMBLR	10M	166M	Tumblr reblog network
USROAD	23M	58M	Complete road network of the USA
UK2005	39M	1.4B	Web graph of the .uk domain
FRND	65M	1.8B	Friendster social network

Q1: What is the performance of ADS and how does it affect the main algorithm? (Section 4.1)

Q2: How does our algorithm compare with state-of-the-art sequential ones, in terms of quality? (Section 4.2)

Q3: What is the scalability of our approach in terms of time and space? (Section 4.3)

Q4: How do the two implementations of MIS compare with each other? (Section 4.4)

Parameters. There are two parameters of interest in our approach: the parameter k of the bottom-k min-hash, which regulates the space-accuracy trade-off of ADS, and the parameter ϵ, which regulates the time-accuracy trade-off 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 using the Forest Fire (FF) model [33], where 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) [10] with parameters $a = 0.45$, $b = 0.15$, $c = 0.15$, and $d = 0.25$.[2] 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.

4.1 Evaluation of ADS

We perform experiments to assess the quality of the ADS estimates. To the best of our knowledge, we are the first to implement and test ADS on Giraph on a large scale. 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).

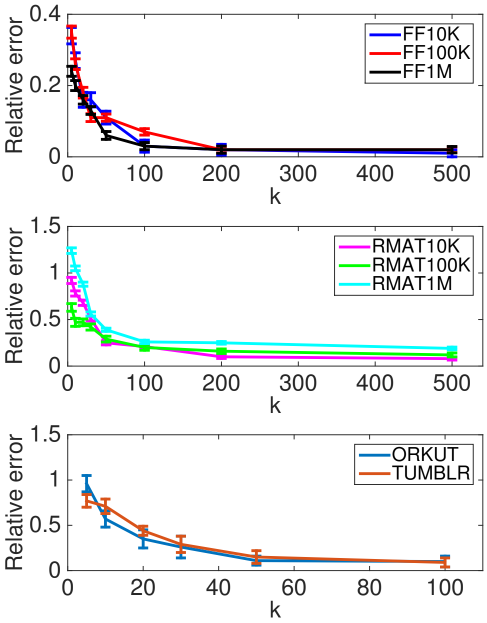

Figure 1: ADS relative error vs. k (unweighted graphs).

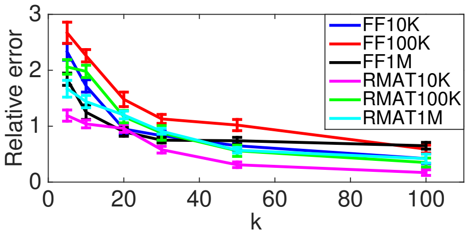

Figure 2: ADS relative error vs. k (weighted graphs).

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.

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

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

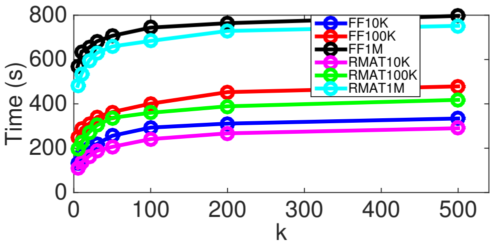

Figure 3: ADS time taken vs. k (unweighted graphs).

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

4.2 Facility-location algorithm

We evaluate the quality of the solutions produced by our distributed algorithm by comparing against a simple sequential baseline. We evaluate the performance and the running time of the algorithm as a function of ϵ.

Comparison with sequential algorithm: As a baseline we use the sequential approximation algorithm by Charikar and Guha [11], which achieves an approximation ratio of $(2.414+\epsilon)$ and has running time of $\tilde{\mathcal{O}}(n^2/\epsilon)$. The sequential algorithm assumes the availability of all-pairs shortest path distances, which is very expensive to compute, even for small graphs. Therefore, we perform our evaluation with graphs consisting of no more than 10 K 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 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): Figure 4 shows the time taken by our algorithm as a function of ϵ. We see that, as expected, our algorithm scales linearly with respect to the size of the graph,[3] and is faster for larger values of ϵ.

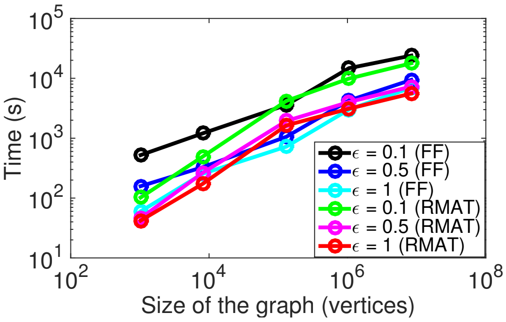

Figure 4: Time taken by the algorithm for different values of ϵ on several graphs.

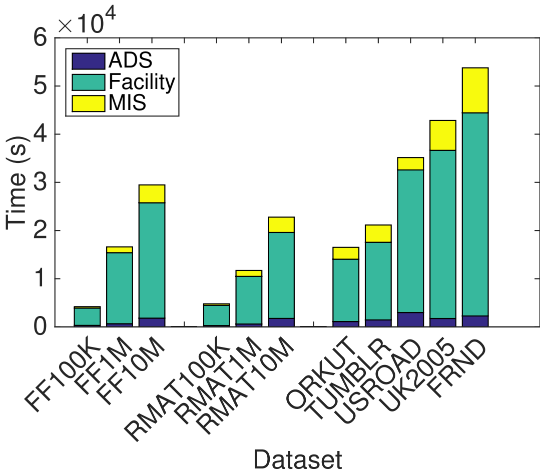

Figure 5: Time taken by each phase of the algorithm.

4.3 Scalability

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 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.

Next, we vary the size of the graph and the number of machines, keeping the number of vertices per machine constant and test the time taken by our algorithm. The results are presented in Figure 7. We can clearly see that the time taken almost remains constant, indicating the scalability of our approach to large graphs.

4.4 Luby's vs. parallel MIS

As discussed above, we implement two methods for finding the maximal independent set (MIS): Luby's [37], which was also implemented recently by Salihoglu and Widom [45], 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 implementation is at least 3 to 5 times faster than Luby's algorithm.

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

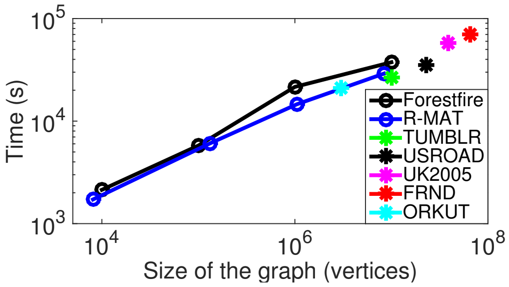

Figure 6: Total time taken by the algorithm.

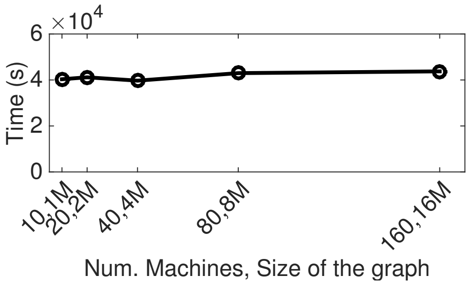

Figure 7: Total time taken by our algorithm for different number of machines for different sized graphs.

5. 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 $(1+\log|\mathcal{D}|)$ [29], while constant-factor approximation algorithms have also been introduced [1, 11]. The approximation algorithm with the best factor so far (1.488) is very close to the approximability lower bound (1.463) [35].

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 $\Omega(n^2)$ 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 [25, 41, 42].

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 [28].

Large-scale graph processing. MapReduce [19] 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 [18, 32, 40], building similarity graphs [3, 17], and finding densest subgraphs [2].

Table 3: Comparison of two implementations of MIS, 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

However, given the iterative nature of most graph algorithms, MapReduce is often not the most efficient solution. Pregel [38] 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, GraphLab [36], PowerGraph [26], GPS [44], and GraphX [27]. 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 [15] to estimate the number of vertices within certain distance from a given vertex. HIP is a cardinality estimator similar to HyperLogLog counters [22] and Flajolet-Martin counters [21]. 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.

6. 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 tool set 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.

7. REFERENCES

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

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

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

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

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

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

[7] U. Brandes and T. Erlebach. *Network analysis: methodological foundations*, volume 3418. Springer, 2005.

[8] A. Broder. On the resemblance and containment of documents. In *SEQUENCES*, 1997.

[9] 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*, 2000.

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

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

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

[13] F. Chudak, T. Erlebach, A. Panconesi, and M. Sozio. Primal-dual distributed algorithms for covering and facility location problems. *Unpublished Manuscript*, 2, 2005.

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

[15] E. Cohen. All-distances sketches, revisited: HIP estimators for massive graphs analysis. In *PODS*, 2014.

[16] 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*, 2013.

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

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

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

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

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

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

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

[24] B. Gavish. Topological design of telecommunication networks-local access design methods. *Annals of Operations Research*, 33(1):17–71, 1991.

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

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

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

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

[29] D. S. Hochbaum. Heuristics for the fixed cost median problem. *Mathematical Programming*, 22(1), 1982.

[30] K. Jain, M. Mahdian, E. Markakis, A. Saberi, and V. Vazirani. Greedy facility location algorithms analyzed using dual fitting with factor-revealing LP. *JACM*, 50(6), 2003.

[31] A. Kuehn and M. Hamburger. A heuristic program for locating warehouses. *Management science*, 9(4), 1963.

[32] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: A method for solving graph problems in MapReduce. In *SPAA*, 2011.

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

[34] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *KDD*, 2007.

[35] S. Li. A 1.488 approximation algorithm for the uncapacitated facility location problem. In *ICALP*, 2011.

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

[37] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. on Comp.*, 15(4), 1986.

[38] 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*, 2010.

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

[40] 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), 2013.

[41] T. Moscibroda and R. Wattenhofer. Facility location: distributed approximation. In *PODC*, 2005.

[42] S. Pandit and S. Pemmaraju. Return of the primal-dual: distributed metric facility location. In *PODC*, 2009.

[43] L. Qiu, V. Padmanabhan, and G. Voelker. On the placement of web server replicas. In *INFOCOM*, 2001.

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

[45] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 7(7), 2014.

[46] D. B. Shmoys. Approximation Algorithms for Facility Location Problems. *AACO*, 1913, 2000.

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

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

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