`GreediRIS`: Scalable influence maximization using distributed streaming maximum cover

Reet Barik [a], Wade Cappa [a], S.M. Ferdous [b], Marco Minutoli [b], Mahantesh Halappanavar [b,a], Ananth Kalyanaraman [a,b],*

[a] *Washington State University, 99164, WA, USA*
[b] *Pacific Northwest National Laboratory, Richland, 99354, WA, USA*

A B S T R A C T

Influence maximization—the problem of identifying a subset of k influential seeds (vertices) in a network—is a classical problem in network science with numerous applications. The problem is NP-hard, but there exist efficient polynomial time approximations. However, scaling these algorithms still remain a daunting task due to the complexities associated with steps involving stochastic sampling and large-scale aggregations. In this paper, we present a new parallel distributed approximation algorithm for influence maximization with provable approximation guarantees. Our approach, which we call `GreediRIS`, leverages the RANDGREEDI framework—a state-of-the-art approach for distributed submodular optimization—for solving a step that computes a maximum k cover. `GreediRIS` combines distributed and streaming models of computations, along with pruning techniques, to effectively address the communication bottlenecks of the algorithm. Experimental results on up to 512 nodes (32K cores) of the NERSC Perlmutter supercomputer show that `GreediRIS` can achieve good strong scaling performance, preserve quality, and significantly outperform the other state-of-the-art distributed implementations. For instance, on 512 nodes, the most performant variant of `GreediRIS` achieves geometric mean speedups of 28.99× and 36.35× for two different diffusion models, over a state-of-the-art parallel implementation. We also present a communication-optimized version of `GreediRIS` that further improves the speedups by two orders of magnitude.

1. Introduction

Given a large real-world graph (e.g., online social network), a fixed budget $k > 0$, and a stochastic diffusion model M, the influence maximization (henceforth, INFMAX) problem aims to identify k nodes (or "seeds") that when initially activated, are expected to maximize influence spread on the network under the model M. This "word of mouth" approach to spreading influence has made INFMAX rich in applications—e.g., for online viral marketing [1], network monitoring [2], controlling rumors in social networks [3], recommendation systems [4], and in understanding how contagions spread in a population [5,6].

INFMAX is NP-hard under classical diffusion models [5] such as *Independent Cascade* (IC) and *Linear Threshold* (LT). The seminal work by Kempe et al. [5] showed that the expected influence spread function is *monotone submodular*—which means the marginal gain of adding a new seed to the current solution set decreases as the set becomes larger. Using classical submodularity results [7], Kempe et al. provided a greedy $(1 − 1/e)$-approximation algorithm, which incrementally expands the seed set by selecting the next seed with the highest marginal gain in influence. Although the connection to submodularity provides efficient algorithms, approximating INFMAX still requires Monte Carlo simulations to generate samples that approximate the spread from the current seed set [5].

An alternative class of approaches, such as the IMM [8] and OPIM [9] algorithms, uses the notion of *reverse influence sampling* [10] to obtain $(1−1/e−\varepsilon)$-approximation, where $\varepsilon \in [0, 1]$ is a controllable parameter that models the sampling error. These approaches [11,8,9] use random sampling of the graph to "score" vertices based on how other vertices can reach them and use that information to identify seeds. These algorithms and related implementations have sufficiently demonstrated that the RIS-based approach is significantly faster in practice compared to the greedy hill climbing approach [5]—making the RIS-based approach the *de facto* choice to implement INFMAX.

The RIS based approaches consist of multiple rounds, each with two major phases: a) *Sampling*, and b) *Seed selection*. The sampling phase generates subgraph samples based on the diffusion model, and the seed selection phase solves a *maximum k-coverage* problem on the generated samples—i.e., identifying k vertex seeds that provide maximum coverage over the samples. Minutoli et al. [12,13] developed Ripples, the first parallel implementation for RIS-based IMM [8]. Subsequently, Tang et al. [14] developed DiIMM, which uses a similar distributed strategy to parallelize IMM. Both these approaches retain the approximation guarantees of their sequential predecessors. However, in both these methods, while the sampling step is easier to scale, the seed selection step is more challenging as it involves performing k global reductions—as observed later in the results.

Contributions: In this paper, we target the seed selection step to improve the scaling of RIS-based approaches. In particular, we propose a distributed streaming maximum-k-cover (abbreviated max-k-cover) to accelerate seed selection, using the distributed GREEDI [15,16] framework under a parallel setting. In this framework, a set of local machines computes partial greedy solutions of the local dataset which they transmit to a global machine, and the global machine applies an aggregation algorithm on the partial solutions received.

Distributed GREEDI avoids expensive reductions. However, we observe that employing the vanilla GREEDI to INFMAX, the global aggregation leads to a compute and communication bottleneck as the number of machines increases. To address this issue, we have designed two techniques: a) aggregation in the global machine asynchronously by adapting a recent *streaming* algorithm, and b) *truncating* elements in the local partial solutions to aggressively cut down on computing and communication for the global machine. Using the streaming algorithm in the global aggregation allows us to carry out the local and global seed selections in tandem, effectively masking the communication overheads. Both of these approaches come with theoretical approximation guarantees. In summary, the main contributions are:

- We present GreediRIS, a new distributed streaming algorithm for *RIS*-based INFMAX on distributed parallel platforms (§3).
- We present and implement GreediRIS-trunc, a truncation technique, to provide a knob to control communication overhead with a tradeoff in quality (§3.3.2).
- All our algorithms have provable worst-case approximation guarantees (§3.3.2).
- Experimental evaluation on up to 512 compute nodes (32K cores) of a supercomputer demonstrates that GreediRIS significantly outperforms the state-of-the-art distributed methods [12,14] (reported speedups with geometric mean of 28.99× and 36.35× for two popular diffusion models), while preserving output quality (geometric mean of reported quality change from state-of-the-art baseline is 2.72%).

Our work—the first of its kind to incorporate distributed parallel submodular optimization for INFMAX—is also generic enough to be extended to a broader base of application settings that use submodular optimization. While such adaptations need significant efforts in implementations, this paper represents a concrete demonstration for INFMAX.

2. Background and preliminaries

In this section, we review the necessary background for the INFMAX problem, including submodular optimization, max-k-cover, and the IMM algorithm which forms the sequential template for our method. Table 1 lists the key notations used in the paper.

Let $G = (V, E)$ be a graph, where V and E are the sets of vertices and edges, respectively, and $n = |V|$. The process of influence spread on G can be described as follows. Let $S \subseteq V$ denote a *seed set* of vertices which are already *activated*. Then, the *expected influence of S* (denoted

Table 1
Notation.

Symbol	Description
$G = (V, E)$	Input graph with n vertices in V
S	Seed set, $S \subseteq V$
k	Target number of output seed vertices
M	Diffusion process for influence spread over G
$\sigma(A)$	Expected influence of $A \subseteq V$ under M
OPT	Max influence spread of any set of k vertices
$N_a(u)$	Subset of activated neighbors of vertex u
$RRR(u)$	Random Reverse Reachable (RRR) set for u
θ	Target number of RRR sets (i.e., samples)
$\Re$	Set containing θ RRR samples
$\mathcal{R}(i)$	Subset of vertices in V present in RRR set i
$S(v)$	Covering subset for v: $S(v) = \{i | v \in \mathcal{R}(i)\}$
m	Number of machines (or processes)

by $\sigma(S)$) is the expected number of vertices that will be activated by S through a stochastic diffusion process, defined by a model M. Note that $0 < \sigma(S) \le n$.

Definition 2.1 (INFMAX). Given a graph $G = (V, E)$, a diffusion process M, and an integer k, the Influence Maximization problem finds a seed set $S \subseteq V$, where $|S| \le k$ maximizing $\sigma(S)$ under M.

Kempe et al. [5] showed that INFMAX is NP-hard under two practical diffusion models, namely, Independent Cascade (IC) and Linear Threshold (LT). In the IC model, at each diffusion step, every edge $\langle u, v \rangle$ has a fixed probability $p_u(v)$ for an active vertex u to activate a neighbor v. The process stops when no further activation is possible. The LT model describes group behavior where individuals (i.e., $v \in V$) have some inertia (modeled by a scalar τ_v) in adopting mass behavior. The strength of the relationship between two vertices $u, v \in V$ is captured by a probabilistic edge weight $w_{(u,v)}$, and the sum of the incoming weights on any vertex is assumed to be 1. A vertex v becomes active in LT if $\sum_{u \in N_a(v)} w_{u,v} \ge \tau_v$, where $N_a(v)$ is the subset of active neighbors of v.

Kempe et al. [5] also proved that the expected influence $\sigma(S)$ is a non-negative monotone submodular function. Submodular functions model diminishing returns of utilities. More formally:

Definition 2.2 (Submodular function). Let X be a finite set, the function $f : 2^X \to \mathbb{R}$ is submodular if and only if:

$$f(A \cup \{x\}) - f(A) \ge f(B \cup \{x\}) - f(B), \qquad (1)$$

where $A \subseteq B \subseteq X$ and $x \in X \setminus B$.

For INFMAX, where $X = V$, the left term represents the *marginal gain* in influence for adding a new vertex x to an existing seed set $A \subseteq V$. By the submodularity property, the net gain of adding a new vertex into the current seed set could only diminish as the seed set expands—suggesting a greedy approach toward seed set expansion. This seminal result led to the first greedy approximation algorithm for INFMAX [5]—using greedy hill climbing [7]. Estimating the actual influence spread is also shown to be #P-hard under both the IC and LT models [17,18]. The existing algorithms for INFMAX thus employ randomized sampling to estimate expected influence and introduce an additive sampling error in the approximation factor. The early efforts focused on simulation-based approaches such as Monte-Carlo simulation [5]. Subsequently, another class of approximation algorithms (albeit faster) was developed by exploiting *Reverse Influence Sampling (RIS)* that uses the notion of reverse reachability [10]. More specifically, this strategy is based on the construction of *random reverse reachable* set for vertices.

Definition 2.3 (Random Reverse Reachable (RRR) set). Let g be a random subgraph of G, obtained by removing edges randomly as dictated by

their respective edge probabilities. The *random reverse reachable* set for vertex $u \in V$ in g is given by: $RRR_g(u) = \{v | \exists \text{ a path from } v \text{ to } u \text{ in } g.\}$

We refer to each $RRR_g(u)$ as a *sample* generated for u. Note that each such sample is a subset of V. An alternative interpretation of $RRR_g(u)$ is—if a vertex v appears in $RRR_g(u)$, then v is a likely influencer of u. Consequently, higher the frequency of the vertex v appearing in such random reverse reachable sets, the larger is its expected influence in the input network.

2.1. The IMM algorithm

Tang et al. [8] proposed the IMM algorithm to implement the RIS approach [10,11]. The algorithm is summarized in Algorithm 1. The main intuition is to generate θ number of RRR samples, and then select a set of k vertices as the solution seed set such that they provide maximum coverage over the set of samples. It uses Martingale analysis and bootstrap techniques to estimate a lower bound of the sampling effort required to achieve OPT (the utility of the intractable optimal solution). Initially, using the analytical formula prescribed in [8], the algorithm estimates the number of RRR samples to compute, denoted by $\hat{\theta}$, using the values of $|V|$, ε and k (Estimate(.) in line 3). Next, the sampling procedure Sample(.) generates and populates those $\hat{\theta}$ RRR samples in $\mathfrak{R}$ (line 7). To generate such a sample:

1. a vertex $u \in V$ is chosen uniformly at random to be the source vertex.
2. given a source, a probabilistic BFS (edges are selected for traversal based on their assigned weights) on the reverse graph (destinations are vertices with incoming edges to the source) is used to populate the RRR sample.

The generated samples are then used to compute k seeds by the seed selection step SelectSeeds(.) as shown in the line 8. This is an instance of computing a max-k-cover on the universe of samples. Using this intermediate seed set, (provably an unbiased estimator of the unknown optimum (OPT)) a lower bound (LB) on OPT is computed, and compared against a certain fraction of $|V|$ (denoted by function CheckGoodness in Algorithm 1; details in [8]). If the condition isn't met, $\hat{\theta}$ is doubled and the process is repeated. The IMM algorithm conducts at most $\log |V|$ such rounds, which we will term as "martingale" rounds (lines 4–11) till the condition is met. Once a tight lower bound is computed, the final sample size (θ) is calculated using the LB, and the martingale loop is terminated. Finally, θ RRR samples are generated, and the set of k seeds are selected using those samples (lines 12–13).

The IMM algorithm is shown to be $(1 - 1/e - \varepsilon)$-approximate for INFMAX. Here, ε is the precision parameter dictating the sampling error. We restate a result from [8] here.

Theorem 2.1. *Given $\theta = \lambda^*/OPT$ and $\delta \in (0, 1)$, the Algorithm 1 returns an $(1 - 1/e - \varepsilon)$-approximate solution of INFMAX with probability at least $1 - \delta$.*

Here OPT is the optimal expected influence. The λ^* is a function of $n, \varepsilon, k, \delta$. We refer to [8,19] for the actual value of the function λ^*. The $(1 - 1/e)$ term in Theorem 2.1 is due to the standard greedy algorithm for the seed selection that computes a max-k-cover on the universe of θ samples. In fact, we can replace the greedy algorithm with any α-approximate algorithm, where $\alpha \leq (1 - 1/e)$.

Corollary 2.1. *Given $\theta = \lambda^*/OPT$, $\delta \in (0, 1)$ and α-approximate max-k-cover, the IMM algorithm returns an $(\alpha - \varepsilon)$-approximate solution of INFMAX with probability at least $1 - \delta$.*

The justification of the Corollary 2.1 is as follows. Since the sampling effort, $\theta = \lambda^*/OPT$ is estimated to guarantee an $(1 - 1/e)$-

---

Algorithm 1 IMM(G, k, ε) by [8].

Input: $G(V, E)$: graph; k: # of seeds; $\varepsilon \in [0, 1]$: sampling error.
Output: S: the seed set
1: {1. Estimate θ using martingale rounds}
2: $\mathfrak{R} \leftarrow \emptyset$
3: $\hat{\theta} \leftarrow \text{Estimate}(k, \varepsilon, |V|)$
4: **for** $x \in [1, \log |V|]$ **do**
5:     **if** $x > 1$ **then**
6:         $\hat{\theta} \leftarrow 2 \times \hat{\theta}$
7:     $\mathfrak{R} \leftarrow \text{Sample}(G, \hat{\theta} - |\mathfrak{R}|, \mathfrak{R})$
8:     $S \leftarrow \text{SelectSeeds}(G, k, \mathfrak{R})$
9:     **if** CheckGoodness($S, V, \& LB$) **then**
10:         $\theta \leftarrow f(k, \varepsilon, |V|, LB)$
11:         **break** {lower bound condition is met}
12: $\mathfrak{R} \leftarrow \text{Sample}(G, \theta)$ {2. Generate θ RRR samples}
13: $S \leftarrow \text{SelectSeeds}(G, k, \mathfrak{R})$ {3. Select final seeds}
14: **return** S

---

approximation for the max-k-cover, for any $\alpha \leq (1 - 1/e)$, the sampling effort is trivially sufficient.

The Sample(.) function is usually implemented as a probabilistic BFS originating from $\hat{\theta}$ randomly selected roots from V in any martingale round. All vertices visited in a probabilistic BFS become the RRR set for that sample. The SelectSeeds(.) procedure is greedy, and involves iteratively computing k vertices with a maximum coverage on the set of RRR samples as detailed in §3.2.

Among distributed parallel approaches for INFMAX, the state-of-the-art comprises of PREEMPT [20] and IMpart [21] for greedy hill climbing, and Ripples [12,13] and DiIMM [14] for parallel distributed IMM. As we focus on RIS-based approaches, we will review Ripples and DiIMM.

Prior work in parallel distributed IMM: Both Ripples and DiIMM approaches start by loading the entire graph on each machine, and efficiently distributing the generation of all (θ or $\hat{\theta}$) samples across m machines, i.e., $O(\frac{\theta}{m})$ samples per machine. However, seed selection involves significant data exchange to compute the updated marginal coverage for each vertex, making the step communication-intensive. In particular, since sampling is distributed, each machine can keep track of only the local sample coverage for each of the n vertices. Thus, to get the global coverage values, a global reduction is required each time a new seed is selected. The Ripples algorithm implements this using k global reductions (over an n-sized frequency vector). The updated global frequencies are used to pick the next best seed. In contrast, DiIMM computes these updates in a lazy fashion under a master-worker setting. Here, after performing a global reduction to select the first seed, the master processes the remaining vertices in a non-increasing order of their original coverage. If the next vertex selected is detected to have an outdated coverage, it is pushed back into the queue and the algorithm continues. Once it finds the next seed (one with the correct next best marginal coverage), it broadcast the information to all the machines so that they can update vertex coverages using their local sample sets. The master machine accumulates the changes via a global reduction for the next iteration of seed selection.

We can show that the DiIMM approach is algorithmically equivalent of performing k global reductions. While Ripples does this entirely in a distributed fashion, DiIMM computes this using a master-worker setting. In either case, seed selection dominates communication costs, as it entails a reduction of an $O(n)$-sized vector within each of the k iterations.

In our work, the distributed sampling step is similar to Ripples and DiIMM—as it is a step that is easily amenable for distributed parallelism. The major difference, however, stems in the way we perform seed selection. More specifically, we reformulate the seed selection step using the RANDGREEDI framework [16] so as to bring the state-of-the-art for distributed submodular optimization to parallelize influence maximization. The key advantages are: a) the k-steps of communication can be

reduced to just two steps (as will be explained next in §3.2); and b) streaming also can be used (along with a truncation scheme) to mask out communication overheads further (as will be explained in §3.3).

3. GreediRIS: a RANDGREEDI framework for distributed influence maximization

We exploit the submodular nature of the `max-k-cover` computation step in order to accelerate seed selection for INFMAX. The state-of-the-art distributed submodular maximization algorithm is the RANDGREEDI [16] framework. Adapting this framework, we present a new distributed memory parallel algorithm for influence maximization and refer to our approach GreediRIS.

We first provide a brief overview on GREEDI and RANDGREEDI (§3.1). Then we show how computing `max-k-cover` allows adapting RANDGREEDI for any *RIS*-based INFMAX algorithms (§3.2). We then discuss the limitations of this simple adaptation and propose a distributed streaming algorithm (§3.3) and a communication-reducing truncation technique (§3.3.2) to alleviate these limitations. Section 3.4 describes our overall parallel algorithm.

3.1. The GREEDI framework

The GREEDI framework represents approaches [16,22,23] for distributed submodular function optimization under budget constraints. These approaches initially partition the data set into different machines. Each machine, independently in parallel, executes the standard greedy algorithm [7] to compute its local solutions. These local solutions are then communicated to a global aggregation node, where they are merged. The greedy algorithm is then employed in the global machine on this set of merged local solutions. The final solution is the one that maximizes the function value from the set of local solutions and the global solution. This *two-stage* algorithm is shown to be more efficient [16] than earlier distributed algorithms. Although GREEDI performs well in practice, its approximation ratio depends inversely on the number of machines. RANDGREEDI [16] was proposed to improve the approximation guarantee of GREEDI. Through a probabilistic analysis, the authors showed that by partitioning the data uniformly at random, GREEDI could achieve a constant approximation guarantee. We restate the following theorem from [16] that proves the approximation ratio of RANDGREEDI.

Theorem 3.1. *Let the greedy method on the local machines be an α-approximate, and that the global algorithm be a β-approximate for maximizing a monotone submodular function subject to a cardinality constraint. Then, RANDGREEDI is (in expectation) $\frac{\alpha\beta}{\alpha+\beta}$-approximate for the same problem.*

While the local machines in RANDGREEDI need to execute the greedy algorithm locally, the global algorithm can use any approach.

3.2. Distributed IMM using RANDGREEDI

We first extend IMM into the RANDGREEDI framework. The focus on IMM is for expository reasons, as it a prototypical RIS approach. Our distributed approach also works for OPIM [9].

Recall from Table 1 that $\mathfrak{R}$ denotes the collection of sampled θ RRR sets, and $\mathcal{R}(i) \subseteq V$ denote the ith RRR set in this collection. Given $\mathfrak{R}$, V, and the target number k, the seed selection problem can be formulated as a `max-k-cover` computation as follows. The set $\mathfrak{R}$ represents the *universe* to be covered. From $\mathfrak{R}$, we generate a collection of *covering subsets*, S. For each node $v \in V$, let $S(v)$ be the subset of RRR sets (identified by their respective indices in $[0, \theta - 1]$) in which v appears—i.e., $S(v) = \{i \in [0, \theta-1] | v \in \mathcal{R}(i)\}$. It is now easy to see that the seed selection problem of IMM reduces to finding a `max-k-cover`, where the universe is

$\{0, \ldots, \theta - 1\}$ (i.e., $\mathfrak{R}$), and the collection of covering subsets is S. More formally, the `max-k-cover` problem here is to find a subset $S \subseteq V$ that is given by: $\arg \max_{S \subseteq V} C(S) = \left| \bigcup_{i \in S} S(i) \right|$, subject to $|S| \leq k$.

The function $C(S)$ can easily be shown to be non-negative monotone submodular. A greedy algorithm that repeatedly selects a covering subset from S with the largest marginal gain with respect to the current solution, achieves $(1 - 1/e)$ worst-case approximation guarantee [7] which is the best achievable unless P = NP [24]. In practice, faster variants of greedy exist, such as lazy greedy [25], threshold greedy [26,27], and stochastic greedy [15]. Lazy greedy [25], which exploits the monotone decrease in the marginal gain of a submodular function, is widely used and often performs much faster than the standard greedy in practice [27] while obtaining the same approximation guarantee. The submodularity property ensures that the marginal gain of an element with respect to the current solution can only decrease. Lazy greedy uses this observation and maintains a max heap of upper bounds on the marginal gains of each element. These upper bounds, updated lazily, are used to select the next element, potentially avoiding computations of marginal gains for a large fraction of the elements. Our implementation supports both standard and lazy greedy approaches. Algorithm 2 shows our lazy greedy `max-k-cover`.

Algorithm 2 Lazy-greedy-max-k-cover($\hat{\theta}, S, k$).

Input: $\hat{\theta}$: size of the universe, S: n covering subsets, k: an integer
Output: S: A subset of $[0, \ldots, n-1]$ of size at most k
1: Build a max heap Q using all n covering subsets S, and using the cardinality of each subset as the key
2: $S = \emptyset$
3: **while** ($|S| < k$) or Q is empty **do**
4:      v = Q.pop()
5:      marginal_gain = $C(S \cup \{v\}) - C(S)$
6:      **if** marginal_gain $\geq$ Q.top().key **then**
7:          $S = S \cup \{v\}$
8:      **else**
9:          Q.push($\langle v, $ marginal_gain$\rangle$)
10: **return** S

Next, we describe our distributed parallel version of Algorithm 1 using the RANDGREEDI framework. Let m to denote the number of machines (or processes or ranks, used interchangeably), and $p \in [0, m-1]$ denote an arbitrary machine. Given $G(V, E)$, k, and m, our distributed IMM algorithm conducts multiple martingale rounds just as the sequential version (Algorithm 1). Algorithm 3 shows the RANDGREEDI version of our distributed IMM for each round.

During the **sampling** phase (lines 1–4 of Algorithm 3), each machine independently generates $\hat{\theta}/m$ samples from G. This is achieved by selecting vertices in V uniformly at random and generating $RRR(v)$ for each of them. We use the Leap Frog method described in [12] to ensure consistent parallel pseudorandom generation across different values of m. At rank p, the parallel sampling phase produces $\mathfrak{R}_p$. Since $\hat{\theta}$ doubles with each martingale round, in our implementation we retain the previous batch of samples and simply add the second half (i.e., augment the previous round's set of $\frac{\hat{\theta}/m}{2}$ samples with a new set of $\frac{\hat{\theta}/m}{2}$ samples). The samples in $\mathfrak{R}_p$ are numbered from $[p \cdot \frac{\hat{\theta}}{m}, (p+1) \cdot \frac{\hat{\theta}}{m} - 1]$, so that each rank can claim a disjoint interval. Next, we generate a partitioning of the set of vertex ids $[0, n-1]$ uniformly at random across the m machines so that each rank p is assigned a distinct subset of $\approx n/m$ ids (line 5 of Algorithm 3). Let $V_p \subseteq [0, n-1]$ denote this subset assigned to rank p. Note that as this is a uniform random partitioning, and therefore each vertex is assigned only to one rank.

Next, we perform a *shuffle* communication such that the *ids* of all the RRR sets that contain a particular vertex u are gathered at the rank that is responsible for u. This is implemented using an MPI `all-to-allv` communication primitive (line 6 of Algorithm 3). Specifically, we first construct a set of partial covering subsets $S_p(u)$ at each rank p using the

---

Algorithm 3 RANDGREEDI-RIS-Round($G, k, \hat{\theta}, m$).

Input: $G(V, E)$: input graph, k: an integer, $\hat{\theta}$: number of samples for the current round, m: number of machines

Output: S: the seed set

1: **for** each machine $p \in [m]$ **in parallel do**
2:     $\Re_p \leftarrow$ Generate $\hat{\theta}/m$ random samples
3:     **for** each vertex $v \in V$ **do**
4:         $S_p(v) = \{j | v \in \mathcal{R}_p(j)\}$
5: Partition V uniformly at random into m vertex sets (V_p at p)
6: All-to-all: $S(u) = \bigcup_{p \in [m]} \{S_p(u) | u \in V_p\}$ at p
7: $S \leftarrow$ RANDGREEDI-max-k-cover($\hat{\theta}, S, k, \{V_p\}$)
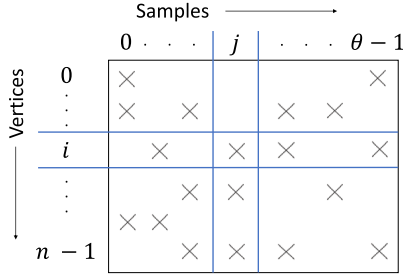8: **return** S

Fig. 1. Sampling visualized as a sparse matrix. The sampling phase populates the columns, and the shuffle phase distributes by rows.

RRR sets of $\Re_p$ (line 4 of Algorithm 3). These partial covering subsets are then scattered so that the lists corresponding to all vertices $u \in V_p$ are gathered at rank p as follows: $S(u) = \bigcup_{p \in [m]} \{S_p(u) | u \in V_p\}$ at p. At the end of this step, rank p contains the *complete* covering subset $S(u)$ for each $u \in V_p$. Fig. 1 illustrates this shuffle operation.

We are now ready to apply the RANDGREEDI framework to **compute seeds**. Algorithm 4 shows the function to compute the seed set using max-k-cover. As explained in §3.1, the RANDGREEDI framework involves two steps: computing local max-k-cover in a distributed fashion, and then merging those local solutions into a global (single) machine to compute a global max-k-cover. Subsequently, the global solution (S_g) is compared with the best of the local solutions (S_ℓ), and the best of the two is output as the final solution (S) for that martingale round. This is followed by a broadcast call to make the utility of the selected seed set available at all machines.

Algorithm 4 RANDGREEDI-max-k-cover($\theta, S, k, \{V_p\}$).

Input: θ: size of the universe, S: n covering subsets, k: an integer, $\{V_p\}$: uniform random set of partitions of V

Output: S: A subset of $V : [0, \dots, n-1]$ of size at most k

1: **for** each process $p \in [m]$ **in parallel do**
2:     $S_p \leftarrow$ Lazy-greedy-max-k-cover(θ, S_p, k) //local max-k-cover
3: Gather: $S' = \bigcup_p S(S_p)$
4: $S_g \leftarrow$ Lazy-greedy-max-k-cover(θ, S', k) //global max-k-cover
5: $S_\ell = \arg\max_{p \in [m]} \{C(S_p)\}$ //local best
6: $S = \arg\max\{C(S_g), C(S_\ell)\}$
7: **return** S

Limitations: A straightforward implementation of Algorithms 3 and 4 could lead to the global aggregation step becoming a bottleneck, as it has to wait until all the local machines complete their local max-k-cover computations. With increasing m, this issue is exacerbated by the global aggregation step receiving up to $m \cdot k$ local solutions. To test this hypothesis, we implemented and tested this template version. Table 2 shows the running times for the local and global max-k-cover steps as a function of m. The results show that as m is increased, the time to generate the local solutions decreases while the time to aggregate and produce the global solution increases. Also, the global aggregation step chooses the final k seeds from a significantly richer set of candidates,

Table 2
The runtimes in seconds for computing the local and global solutions of max-k-cover using the RANDGREEDI template. Livejournal ($|V| = 4.8M, |E| = 68.9M$) was used as the test input.

Time (in sec)	m: Number of machines				
	8	16	32	64	128
local max-k-cover	1.87	0.91	0.34	0.17	0.10
global max-k-cover	0.22	0.67	1.20	2.47	4.86

thereby increasing the amount of computations required to calculate the marginal gains to select each seed. This motivates the design of a streaming based implementation presented next in §3.3.

3.3. Distributed streaming IMM via RANDGREEDI

To overcome the above limitations of RANDGREEDI, we present two improvements. The first approach (§3.3.1) replaces the global aggregation algorithm with a streaming max-k-cover technique inside each round. The second approach (§3.3.2) reduces communication cost from the local max-k-cover solvers through a truncation technique.

3.3.1. Streaming computation in the global machine

The first approach to improve RANDGREEDI replaces the global aggregation algorithm with a *streaming* max-k-cover. The key idea is to immediately send the local seeds to the global machine, as they are generated, without waiting for the local machines to complete all their greedy selections. The global machine then runs a linear time one-pass streaming algorithm on the incoming data, thus offsetting the computational expense, and allowing masking of communication overhead. Streaming can therefore allow tandem executions of global and local solvers.

Implementing streaming necessitates that the global max-k-cover algorithm be changed to work with the incoming stream of local seeds (from different machines). The standard greedy max-k-cover algorithm (Algorithm 2) being offline, is not suitable here. However, with the RANDGREEDI framework, we are free to choose any algorithm for aggregation. As the max-k-cover problem is an instance of the submodular maximization with cardinality constraints, we can use one of the recent direct streaming algorithms designed for max-k-cover [28,29]. In this work, we employ the $(1/2 - \delta)$-approximate streaming algorithm (described next) also developed in [28] because it's nearly linear in runtime, and absence of any post-processing steps, thus generating the solution immediately after the streaming phase ends. The challenge here is also to show that the streaming algorithm provides good quality solution efficiently.

Algorithm 5 Streaming max-k-cover at the global receiver.

Input: S_m: streaming collection of subsets from local senders, k: an integer, u: upper bound on OPT, l: lower bound on OPT, δ

Output: set of seeds of size at most k

1: Create $B = \log_{1+\delta} \lfloor \frac{u}{l} \rfloor$ buckets
2: $C_b \leftarrow$ covering sets of a bucket b, initialized to empty
3: $S_b \leftarrow$ set of seed nodes at bucket b, initialized to empty
4: **for** $s \in S_m$ **do**
5:     **for all** $b \in [0, B-1]$ **in parallel do**
6:         **if** $|S_b| < k$ and $|s \setminus C_b| \geq \frac{(1+\delta)^b}{2k}$ **then**
7:             $C_b = C_b \cup \{s\}$
8:             $S_b = S_b \cup id(s)$
9: $b^* = \arg\max_b |C_b|$
10: **return** S_{b^*}

Let u and l be an upper and lower bound on optimum coverage, respectively. For a given $0 < \delta < 1/2$, we maintain $B = \log_{1+\delta} \lfloor \frac{u}{l} \rfloor$ buckets

indexed by the integers $[0, B - 1]$. Each bucket b has a lower value corresponding to the lower threshold limit representing the bucket, which is $(1 + \delta)^b$. When a streamed-in seed along with its covering subset s arrives, the algorithm computes the marginal gain of s with respect to current solution of b, S_b. If the bucket b holds a solution set of size smaller than k and the marginal gain is greater than $(1 + \delta)^b$, the vertex representing the subset s (denoted by id(s)) is inserted into the current solution, and the cover set (C_b) is updated; otherwise, s is discarded. This operation is repeated for all the buckets. Since the decision to recruit a seed into a bucket is independent across buckets, we use multithreaded parallelism to process the buckets. Once the streaming phase ends, we output the solution in the bucket (b^*) with maximum coverage. Algorithm 5 runs in $O(\theta B |S_m|/t)$ time (where t is the number of threads), and is shown to be $(1/2 - \delta)$-approximate of the optimal coverage [28].

Lemma 3.1. *The approximation ratio of* GreediRIS *with streaming aggregation is* $\frac{(1-1/e)(1/2-\delta)}{(1-1/e)+(1/2-\delta)} - \varepsilon$ *in expectation.*

Proof. Since the local machines run a greedy algorithm the approximation guarantee is $(1 - 1/e)$. The streaming algorithm is $(1/2 - \delta)$-approximate. The guarantee follows from Theorem 3.1 and Corollary 2.1. □

3.3.2. Communication reduction using truncation at the senders

In the RANDGREEDI framework for IMM, each machine can send up to k seeds, implying a total of $m \cdot k$ seeds arriving at the global machine (be it with or without streaming). Our second improvement to RANDGREEDI aims at reducing the volume of communication by restricting the local senders to send only a subset of the k seeds to the global machine. We refer to this approach as *truncated greedy*. Specifically, each local machine (sender) still computes all k local seeds. However, during streaming each sends only the top $\alpha \cdot k$ seeds (along with their local covering sets), where $0 < \alpha \leq 1$. We show that the approximation ratio for the truncated greedy is $(1 - e^{-\alpha})$ compared to the optimal solution of k subsets. Thus, the α value provides a trade-off between the scalability and approximation of the overall INFMAX algorithm.

Lemma 3.2. *The approximation guarantee of truncated greedy is* $1 - e^{-\alpha}$.

Proof. We modify the standard analysis of the greedy algorithm for max-k-cover [24]. Let OPT be the optimal coverage of the k elements, and x_i be the number of new elements covered by the truncated greedy algorithm in the i-th set it selects. We are interested to bound the coverage value of the truncated greedy, i.e., $\sum_{j=1}^{\alpha k} x_j$. Since the optimal algorithm uses k sets to cover OPT elements, at iteration $(i + 1)$ of the algorithm, there must be a set that covers at least $\frac{OPT - \sum_{j=1}^{i} x_i}{k}$ new elements. So, $x_{i+1} \geq \frac{OPT - \sum_{j=1}^{i} x_i}{k}$.

By a standard induction on the iteration, we can show that $\sum_{j=1}^{i+1} x_j \geq OPT - (1 - \frac{1}{k})^{i+1} OPT$. Plugging in $i + 1 = \alpha k$, we get $\sum_{j=1}^{i+1} x_j \geq OPT - (1 - \frac{1}{k})^{\alpha k} OPT \geq (1 - e^{-\alpha}) OPT$. □

We prove the following result using a similar technique as Lemma 3.1.

Lemma 3.3. *The approximation ratio of* GreediRIS *with truncated greedy in local machines and streaming aggregation is* $\frac{(1-e^{-\alpha})(1/2-\delta)}{(1-e^{-\alpha})+(1/2-\delta)} - \varepsilon$ *in exception. Here, $0 \leq \alpha \leq 1$ is the fraction of k local seeds communicated to the global machine.*

For $\alpha = 1$, the approximation ratio of GreediRIS-trunc is the same as GreediRIS. Lowering the value of α results in the senders communicating only the top α-fraction of their local k seeds which results in
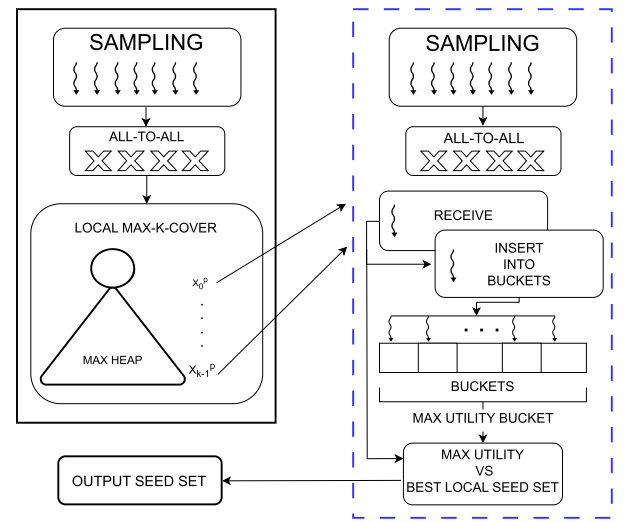

Fig. 2. Schematic illustration of our parallel GreediRIS approach inside one round.

a reduction of the approximation ratio from that of GreediRIS. For example, the drop in approximation ratio compared to GreediRIS when α is set at 0.75 and 0.5 is 1.86% and 4.95% respectively. The intuition behind why GreediRIS-trunc provides high quality solutions in practice (as seen later in §4.3) despite the reduced approximation guarantee, lies in the submodular property exhibited by the local seeds. For $0 \leq i \leq j \leq k$, the marginal utility of the i-th seed will always be greater than or equal to that of the j-th out of the k local seeds. This implies, stronger the submodular behavior of the underlying function, the greater is the benefit from truncation as the early choices are likely to provide significant gains. This also means that if a submodular function is close to a linear function, truncation might not be that useful.

Extension to other *RIS*-based INFMAX methods: Our GreediRIS approach also extends to OPIM [9], which is another RIS-based approach to INFMAX. Unlike IMM, OPIM delivers an instance-wise approximation guarantee in each computation round and is suitable for online processing of INFMAX. The instance-specific quality guarantees are achieved through partitioning the samples generated in each round (using the same Sample(.) subroutine as IMM) into two halves R_1 and R_2. The first is used to select an intermediate solution for that round (same mechanism as the SelectSeeds(.) step in IMM). The utility of this solution on R_2 is used as a validation score that is utilized by OPIM's version of the CheckGoodness subroutine to come up with the approximation guarantee for that round. Algorithmically, OPIM bears many similarities to IMM since it proceeds in rounds consisting of a sampling and seed selection phase and, as such, is also a suitable candidate for GreediRIS. We include experiments integrating OPIM with GreediRIS (§4.4).

3.4. Parallelization and implementation

Next, we present the parallelization details for the overall GreediRIS distributed streaming workflow. A schematic illustration of our GreediRIS workflow is shown in Fig. 2.

S1) Distributed sampling:
   The input graph $G = (V, E)$ is loaded on all machines. Using G, all machines generate $\approx \theta/m$ samples each. This step supports multithreaded parallelism within each machine.

S2) All to all:
   The list of vertex ids $[0, n - 1]$ is partitioned uniformly at random so that each rank p is assigned $\approx n/m$ ids, represented by V_p. Subsequently, using an MPI all-to-allv communication, we transport

all RRR set ids that contain any given vertex $v \in V_p$ to rank p. At the end of this step, each machine stores the RRR set ids corresponding to all vertex ids in its assigned partition (V_p). Recall the use of $S(u)$ for this covering set for each vertex id $u \in V_p$. Henceforth, we distinguish between a "sender" process and a "receiver" process. In our streaming implementation, there are $m - 1$ senders (ranks $[1, m - 1]$) and one receiver process (rank 0).

S3) Sender process:

A sender p uses the collection of its $\approx n/m$ covering sets to detect up to k local seeds. Let $[x_0^p, x_1^p, \dots x_{k-1}^p]$ denote these seeds. We use the lazy greedy `max-k-cover` algorithm (Algorithm 2) that internally maintains a max heap to identify the next best seed, starting with x_0^p. To support streaming, the sender process sends each seed x_i^p along with its covering subset to the receiver process as and when it is identified. This is implemented using a nonblocking send to minimize waiting at the sender process. The sender concludes when all of its k seeds are generated. Upon completion, it alerts the receiver of its termination.

For the truncated version (§3.3.2), when the truncated limit is reached no more sends happen from that sender. But the sender continues to generate all its k seeds locally. The idea is to enable comparison at the end with the global solution.

S4) Receiver process:

After participating in the initial sampling, the role of the receiver switches to: a) aggregating all the seeds sent by all senders; and b) performing the global `max-k-cover` using the multi-threaded streaming algorithm in Algorithm 5. Let t denote the number of threads at the receiver. Thread rank 0 is a dedicated *communicating thread* to listen to the senders using a *nonblocking receive*. The remaining $(t - 1)$ threads parallelize the insertion of seeds into the B buckets. We refer to these as *bucketing threads*. We assign $\lceil B/(t-1) \rceil$ threads per bucket. When a new seed x (along with its covering set $S(x)$) is received, the communicating thread places $\langle x, S(x) \rangle$ on a queue and atomically sets a flag corresponding to that index. For this purpose, the receiver maintains a shared array A of maximum size $m \cdot k$. Each bucketing thread monitors the *flag* corresponding to the next index yet to be read. When the flag is set, that means there is a new element x in that index, then all bucketing threads read x and insert them into their respective buckets independently. This allows for a lock-free (atomic) update. This also allows concurrent processing of the bucketing threads with the communicating thread. Finally, when all senders have alerted the receiver with termination, the receiver compares the best solution across the buckets with the best local solutions at the senders, and outputs the final seed set.

Runtime complexity: As for the runtime complexity, step S1 (sampling) takes $\mathcal{O}(\theta/m) \cdot \ell_s$, where ℓ_s is the average length of a sample. Given the same set of edge probabilities, this average is expected to be more for the IC model than for LT. Step S2 is an all-to-all communication step and can take $\mathcal{O}(\tau m + \mu \frac{n}{m}\theta)$ in the worst-case, where τ and μ are the network latency and reciprocal of bandwidth respectively. Note that the size of each covering set to be received is upper bounded by θ and in practice is likely to be less (depends on its n/m vertex subset). Also, even though the receiver process sends out the samples it generated, it will not receive any data (from the all-to-all) as all the local `max-k-cover` happen at the senders. The remaining part of the sender process (S3) is consumed in generating local seeds from its local (n/m) covering subsets. Using the greedy algorithm, this takes $\mathcal{O}(\frac{k\theta n}{m})$.

As for the receiver process (S4), the communication time is dominated by the nonblocking receive to collect all the streamed seeds, and the computation time by the time to insert into the local B buckets. However, considering masking and the parallel processing of buckets, we expect communication (or wait time for data to arrive) to dominate the total time—which is desirable to ensure the receiver is available for all senders. The u/l ratio of the number of buckets in the stream-

Table 3
Key details for our test inputs SNAP [31] and KONECT [32]). Avg. is the average out-degree and Max. is the maximum out-degree.

Input	#Vertices	#Edges	Avg.	Max.
Github	37,700	285,000	7.60	9,446
HepPh	34,546	421,578	24.41	846
DBLP	317,080	1,049,866	6.62	343
Pokec	1,632,803	30,622,564	37.51	20,518
LiveJournal	4,847,571	68,993,773	28.26	22,887
Orkut	3,072,441	117,184,899	76.28	33,313
Orkut-group	8,730,857	327,037,487	56.81	318,240
Wikipedia	13,593,032	437,217,424	22.56	5,576,228
Friendster	65,608,366	1,806,067,135	27.528	3615

ing algorithm (§3.3) can be shown to be k. This is because the optimal cover could be at most k times of the cover of a set with the maximum marginal gain. Using this, the runtime of the streaming computation in the receiver is $\mathcal{O}(mk\theta \cdot B) = \mathcal{O}\left(mk\theta \cdot (\log_{(1+\delta)} k)\right)$.

Memory complexity: Here we only discuss the memory complexity of the streaming algorithm. Each of the bucket needs to store the total cover of its partial solution. Since total cover could be at most θ, we require $\mathcal{O}(\theta)$ memory per bucket. Each bucket also needs to store the partial solution of size $\mathcal{O}(k)$. Putting these together, the total memory complexity of the streaming algorithm in the receiver is $\mathcal{O}\left((\theta + k) \cdot \log_{(1+\delta)} k\right)$. This is more space efficient than any offline algorithm that would require storing the complete incoming data, i.e., $\mathcal{O}(mk\theta)$ elements.

Software availability: `GreediRIS` is implemented using C/C++, MPI and OpenMP, and is open-sourced at the GreediRIS Github repository [30].

4. Experimental evaluation

4.1. Experimental setup

Platform: All experiments were conducted on the NERSC Perlmutter supercomputer, a HPE Cray EX system. For our experiments, we used the CPU nodes (machines), each of which consists of two AMD EPYC 7763 (Milan) CPUs (with 64 cores per CPU), 256 MB L3 cache, 512 GB of DDR4 memory, and a HPE Slingshot 11 NIC. Experiments were performed using up to $m = 512$ nodes (with a total of 32K cores). All runtimes are measured in terms of wall clock time and reported in seconds, and all distributed runs were executed by binding one MPI rank per node.

Input data: We use nine real world networks from SNAP [31] and KONECT [32] summarized in Table 3. The inputs cover a wide range in sizes, and application domains such as social networks, citation networks and web documents. Since edge probabilities are not available for these public networks, consistent with practice [12,13,33], we generated edge probabilities from a uniform random distribution between $[0, 0.1]$. Note that the weighted cascade (WC) model (i.e., edge weight from node u to v is given by $1/InDegree(v)$)—as has been used in [14]—is *not* an option as it has been shown [34] to *not* translate to more generic diffusion models.

Evaluation methodology: Our experiments include both IC and LT diffusion models. For related work comparison, we compared our new implementations against two state-of-the-art distributed IMM methods, namely Ripples [12] and DiIMM [14]. **Ripples** is available as an open source software [35] and we compared with the C/C++ and OpenMP + MPI distributed implementation. The **DiIMM** software however was

Table 4

Performance of `GreediRIS` implementations, Ripples [12] and our implementation of DiIMM [14]. All results reported were using $m = 512$ nodes and 32K cores in distributed memory. `GreediRIS`-trunc runs were performed using $\alpha = 0.125$. Bold values indicate the best entries for that input.

Diffusion: LT	Time (in Sec.)			
Input	Ripples [8]	DiIMM [14]	`GreediRIS`	`GreediRIS`-trunc
Github	108.3	114.7	1.8	**0.2**
HepPh	273.1	270.1	1.8	**0.5**
DBLP	365.0	357.7	3.7	**3**
Pokec	435.1	434.9	11.6	**9.5**
LiveJournal	482.1	478.0	22.0	**22.9**
Orkut	463.4	458.1	18.7	**18.2**
Orkut-group	549.4	564.6	81.7	**78.6**
wikipedia	537.0	528.4	**45.0**	46.6
friendster	948.2	994.9	746.1	**685.9**
Diffusion: IC	Time (in Sec.)			
Github	122.0	112.0	0.9	**0.3**
HepPh	175.0	185.5	1.5	**0.6**
DBLP	266.0	274.7	2.4	**1.1**
Pokec	96.0	101.4	77.4	**30.3**
LiveJournal	129.0	146.4	100.0	**64.3**
Orkut	74.5	88.2	32.5	**6.3**
Orkut-group	164.3	181.1	119.1	**60.6**
wikipedia	360.3	287.5	256.0	**222**
friendster	278.0	319.0	361.0	**211.2**

not available as of this writing.[1] Therefore, to enable a comparison, we implemented our own version of their method into the Ripples open source package [14]. Additionally, we also added the functionality of using OPIM [9] (instead of IMM) as the underlying *RIS*-based INFMAX strategy into `GreediRIS`.

For our proposed approach, `GreediRIS`, we tested two variants of our distributed algorithm:

- **`GreediRIS`**: uses the distributed streaming algorithm described in §3.3 and its parallelization in §3.4;
- **`GreediRIS`-trunc**: uses the truncated extension of `GreediRIS`, described in §3.3.2, with α fraction of the seeds communicated.

All runs not using OPIM were carried out for $k = 100$ and precision parameter $\varepsilon = 0.13$. For streaming, we chose $\delta = 0.077$—as this configuration set the number of buckets approximately equal to the number of available threads (63) at the global receiver. `GreediRIS`-trunc was run for values of $\alpha \in (0,1]$. For the experiments using OPIM, we set $k = 1000$, $\varepsilon = 0.01$, and adjusted $\delta = 0.0562$ to maintain the number of buckets at 63.

All implementations were compiled using (GCC 11.2.0; optimization flags `-O3` and `-mtune=native`), and MPI library Cray-mpich 8.1.24. For quality, we use the average number of node (vertex) activations over 5 simulations of the diffusion models (IC or LT) from the seed sets obtained by Ripples as the baseline, with the same for other implementations presented as a percentage change.

4.2. Comparative evaluation

Table 4 shows comparative results in terms of runtime performance. `GreediRIS`-trunc was the fastest for nearly all inputs, with `GreediRIS` coming a close second. For all inputs tested, both `GreediRIS` implementations significantly outperformed Ripples and DiIMM. For instance, for LT, the speedups of `GreediRIS`-trunc over Ripples ranged from 1.32× (for friendster) to 357.78× (for Github), with a geometric mean of 28.99× across all inputs. For IC, the corre-

---

[1] We were not able to find its public repository and the authors did not respond despite multiple requests.
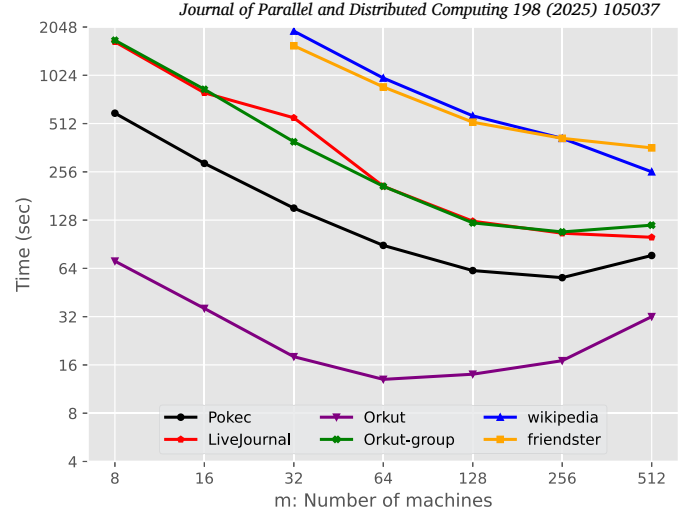

Fig. 3. *Strong scaling* performance of `GreediRIS` for different inputs, varying m up to 512 nodes for the IC model. All times are in seconds.

sponding speedups ranged from 1.38× (for friendster) to 526.13× (for Github), with a geometric mean of 36.35× across all inputs. The results also show that `GreediRIS` benefits in runtime savings from truncation. Overall, these runtime results uniformly show the effectiveness of our RANDGREEDI-based distributed streaming as well as truncation for the seed selection step. The variations of speedups with inputs and models used are to be expected due to the effects of graph topology and stochasticity of the process. Relative to IC, the performance benefits from `GreediRIS` under LT are more because it has been known to generate shallower BFS traversals (i.e., shorter RRR set sizes).

We also compared the quality of seeds generated by `GreediRIS` and `GreediRIS`-trunc, against the quality of seeds generated by Ripples. On average across all data sets, we observed that while using $m = 512$ nodes, the expected influence achieved by the seeds generated by `GreediRIS` and `GreediRIS`-trunc implementations was 2.72% away from the influence generated by Ripples. This is despite reduced worst-case approximations (as shown in the Lemmas of §3.3) relative to Ripples which is $(1 - 1/e - \varepsilon)$-approximate. For instance, our experimental settings for $\varepsilon = 0.13$ and $\delta = 0.077$ yield a worst-case approximation ratio of 0.123 in expectation for `GreediRIS` (compared to a 0.5 ratio for Ripples). These results show that despite being weaker in approximation guarantee, the practical quality of `GreediRIS` and `GreediRIS`-trunc is comparable to Ripples, while providing significant performance advantage.

4.3. Performance evaluation for `GreediRIS`

Next, we present a detailed parallel performance evaluation of our `GreediRIS` implementations. We first present the strong scaling results for our main `GreediRIS` implementation. Fig. 3 shows the strong scaling results. In the interest of space, we show results using the IC model. Results for smaller inputs (Github, HepPh, DBLP) that took less than 3 seconds, are omitted. On the remaining inputs, in general we see better scaling behavior as the input size increases—e.g., scaling on LiveJournal is near-linear until $m = 128$, while the further runtime reductions are continued to be achieved for up to $m = 256$ with Orkut-group, and up to $m = 512$ with Wikipedia and friendster. In general, we observed that `GreediRIS` is able to push the scaling to larger number of nodes, beyond what Ripples achieves. This is illustrated in Fig. 4 where we see the scaling benefits of using `GreediRIS` over Ripples for a representative input like Orkut-group. `GreediRIS`-trunc helps push this boundary even further with more results shown at the end of this Section.

Fig. 5 shows the detailed runtime breakdown for a representative input LiveJournal for the IC diffusion model. For sender time, we used the time for the longest running sender. As Fig. 5a corresponding to the total
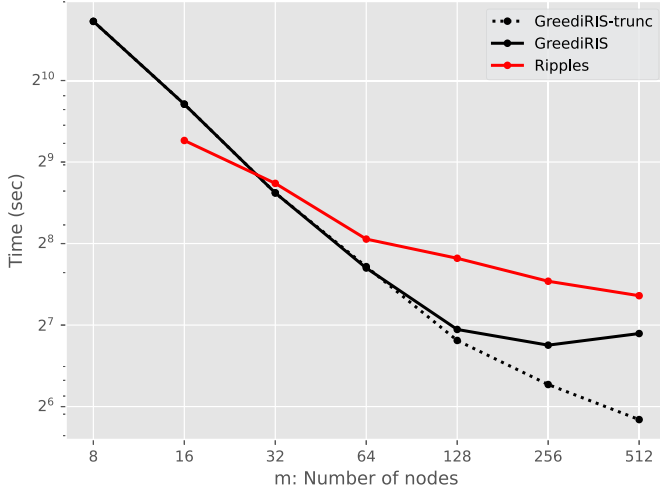
Fig. 4. Scaling of the total execution time for our methods `GreediRIS` and `GreediRIS-trunc` on up to 512 Perlmutter nodes for input Orkut-group. Also shown is the scaling behavior of the state-of-the-art tool Ripples.

breakdown shows, the total time is closer to the maximum of the sender and receiver times than the sum, and these two times are nearly comparable — suggesting the effectiveness of streaming. The sender times were evenly split between sampling and all-to-all. We see the sampling and all-to-all times scaling with m. The seed selection, however, starts to consume more time at the receiver for larger m (≥ 256) settings.

This is something the truncated version (`GreediRIS-trunc`) is better equipped to address, as we will see next.

We also examine the receiver process time closely for the same input in Fig. 5b, since the global seed selection using streaming is carried out at the receiver. Recall that at the receiver, thread rank 0 is the *communicating thread*, as it monitors the communication channel by doing a non-blocking receive, and when the next seed arrives, it pushes it to the local queue. All other $t-1$ threads (i.e., 63) are *bucketing threads*, handling insertions individually into a subset of $\lceil B/(t-1) \rceil$ buckets. The results show that the communicating thread spends most of its time on

the non-blocking receive, which implies high availability to the senders. The bucketing threads generally take significantly much less time, and their running times are subsumed within the communicating thread's time. Note that there are 63 threads to handle the buckets.

Evaluation of `GreediRIS-trunc`: In this section, we tested the framework's truncated variant, `GreediRIS-trunc` (§3.3.2) by varying the parameter α. Note that α is the fraction of seeds sent from each sender to the global receiver. Increasing the number of nodes leads to a proportional increase in the communication and computation performed by the global receiver.

The parameter α provides a way to cap this load on the receiver and extend scaling. This can be seen in Fig. 6(a), where the parallel runtime for `GreediRIS` starts to plateau for `GreediRIS` for $m \geq 256$, while for `GreediRIS-trunc` it continues to decrease (mainly due to decrease in α). While decreasing α results in a lower approximation guarantee (Lemma 3.3) and consequently degrade quality, our experiments showed this to be negligible (less than 0.36%) in practice for any given m.

4.4. Extension of `GreediRIS` to OPIM

In this section, we show OPIM results integrated to the `GreediRIS` framework. Consistent with the large scale experimental settings of [9], we set $k = 1000$, $\epsilon = 0.01$, and terminate when the number of generated samples exceeds 2^{20}. We set $\delta = 0.0562$ to ensure the number of buckets at the receiver is 63 (1 communicating thread vs. 63 bucketing threads). The number of nodes was set to $m = 512$.

For evaluation, we used the `GreediRIS-trunc` implementation that uses OPIM internally, and studied the seed-selection performance. Results are shown in Table 5. As can be observed, we are able to achieve significant reduction in time with increasing α, while maintaining the reported approximation guarantee (as reported by OPIM using the martingale based analysis in [9]).

5. Conclusions and future work

We presented `GreediRIS`, a new scalable distributed streaming algorithm and its parallel implementations for INFMAX. New ideas in-
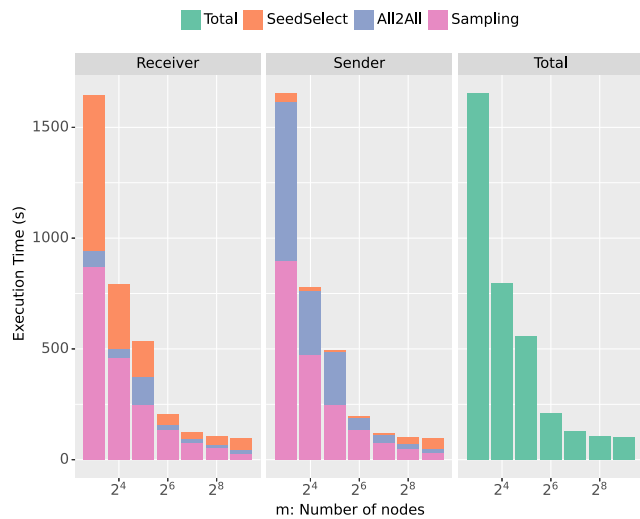
(a) LiveJournal: Total time breakdown
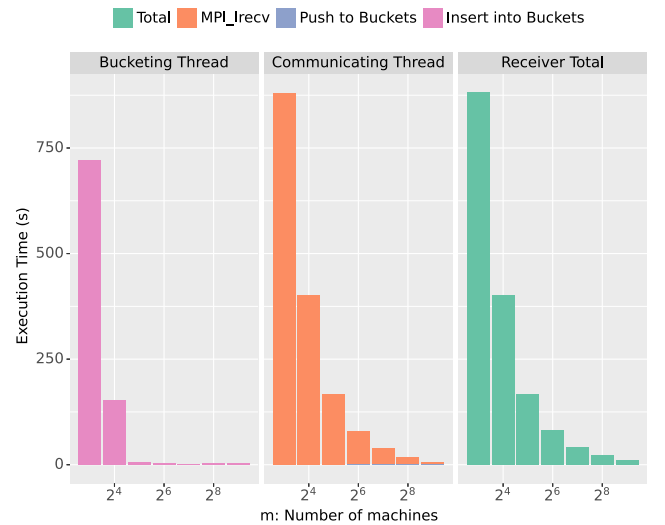
(b) LiveJournal: Receiver time breakdown

Fig. 5. *Parallel runtime breakdown* for input LiveJournal (Diffusion model IC) for `GreediRIS`: by the receiver, sender (longest running), and the total times. Note that in streaming, senders and the receiver run in parallel. The plot corresponding to the receiver shows the breakdown between its communicating thread and bucketing threads. Note that the majority of the SeedSelect time on the receiver is idle time, as senders participate in the all-to-all and then perform their local seed selections.
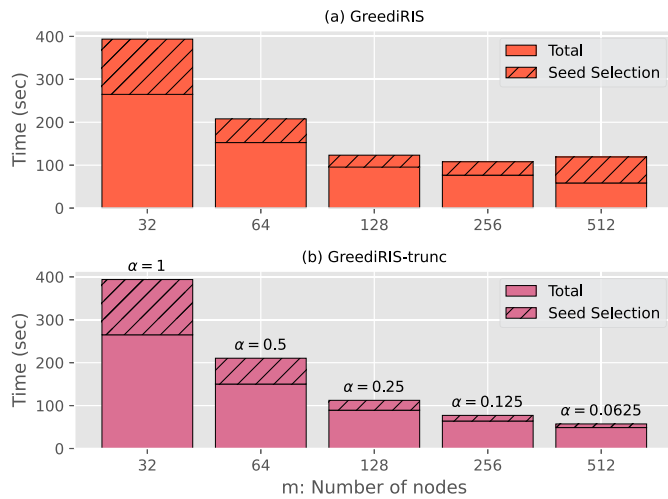
Fig. 6. Strong scaling plot of `GreediRIS` (top) and `GreediRIS-trunc` (bottom) to up to 512 nodes. The seed selection step is shown as the shaded region representing its fraction of the total runtime.

Table 5

Evaluation of `GreediRIS-trunc` using the OPIM RIS-strategy [9], and using parameters: $\{k = 1000, \varepsilon = 0.01, \text{target } \theta \approx 2^{20}, \delta = 0.0562\}$. All runs were performed on the friendster input on $m = 512$ nodes.

Truncation factor α:	1	0.5	0.25	0.125
Seed select time (sec):	381.42	200.59	99.30	95.43
OPIM approx. guarantee:	0.66	0.67	0.68	0.69

clude a) leveraging the RANDGREEDI framework for distributed submodular optimization for INFMAX, b) introducing streaming into distributed `max-k-cover` allowing efficient masking of communication overheads through overlapped computation; and c) truncation to further reduce communication burden. The experimental study demonstrated that `GreediRIS` significantly outperforms state-of-the-art distributed parallel implementations with comparable quality. The algorithms presented are generalizable to any other monotone submodular optimization problem. Future extensions and directions could include: i) enabling streaming during the sampling phase to mask all-to-all communication costs; ii) further optimizations to increase problem size reach to solve larger problems on smaller systems; iii) GPU acceleration for sampling; and iv) extension to other monotone submodular optimization problems.

CRediT authorship contribution statement

Reet Barik: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Conceptualization. **Wade Cappa:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology. **S.M. Ferdous:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization. **Marco Minutoli:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology. **Mahantesh Halappanavar:** Writing – review & editing, Writing – original draft, Software, Methodology, Funding acquisition. **Ananth Kalyanaraman:** Writing – review & editing, Writing – original draft, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

All data sets used are public and cited in the paper.

References

[1] P. Domingos, M. Richardson, Mining the network value of customers, in: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2001, pp. 57–66.

[2] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J.M. VanBriesen, N.S. Glance, Cost-effective outbreak detection in networks, in: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2007, pp. 420–429.

[3] C. Budak, D. Agrawal, A.E. Abbadi, Limiting the spread of misinformation in social networks, in: Proceedings of the 20th International Conference on World Wide Web, WWW, ACM, 2011, pp. 665–674.

[4] M. Ye, X. Liu, W. Lee, Exploring social influence for recommendation: a generative model approach, in: Proc. of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR, ACM, 2012, pp. 671–680.

[5] D. Kempe, J.M. Kleinberg, E. Tardos, Maximizing the spread of influence through a social network, in: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24–27, 2003, 2003, pp. 137–146.

[6] M. Azaouzi, W. Mnasri, L.B. Romdhane, New trends in influence maximization models, Comput. Sci. Rev. 40 (2021) 100393.

[7] G.L. Nemhauser, L.A. Wolsey, M.L. Fisher, An analysis of approximations for maximizing submodular set functions—I, Math. Program. 14 (1978) 265–294.

[8] Y. Tang, Y. Shi, X. Xiao, Influence maximization in near-linear time: a martingale approach, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 1539–1554.

[9] J. Tang, X. Tang, X. Xiao, J. Yuan, Online processing algorithms for influence maximization, in: Proceedings of the 2018 International Conference on Management of Data, 2018, pp. 991–1005.

[10] C. Borgs, M. Brautbar, J. Chayes, B. Lucier, Maximizing social influence in nearly optimal time, in: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2014, pp. 946–957.

[11] Y. Tang, X. Xiao, Y. Shi, Influence maximization: near-optimal time complexity meets practical efficiency, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 2014, pp. 75–86.

[12] M. Minutoli, M. Halappanavar, A. Kalyanaraman, A. Sathanur, R. Mcclure, J. McDermott, Fast and scalable implementations of influence maximization algorithms, in: 2019 IEEE International Conference on Cluster Computing, CLUSTER, IEEE, 2019, pp. 1–12.

[13] M. Minutoli, M. Drocco, M. Halappanavar, A. Tumeo, A. Kalyanaraman, cuRipples: influence maximization on multi-gpu systems, in: Proceedings of the 34th ACM International Conference on Supercomputing, 2020, pp. 1–11.

[14] J. Tang, Y. Zhu, X. Tang, K. Han, Distributed influence maximization for large-scale online social networks, in: 2022 IEEE 38th International Conference on Data Engineering, ICDE, IEEE, 2022, pp. 81–95.

[15] B. Mirzasoleiman, A. Badanidiyuru, A. Karbasi, J. Vondrák, A. Krause, Lazier than lazy greedy, in: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI Press, 2015, pp. 1812–1818.

[16] R. da Ponte Barbosa, A. Ene, H.L. Nguyen, J. Ward, The power of randomization: distributed submodular maximization on massive datasets, in: Proceedings of the 32nd International Conference on Machine Learning, ICML, vol. 37, 2015, pp. 1236–1244, JMLR.org.

[17] W. Chen, C. Wang, Y. Wang, Scalable influence maximization for prevalent viral marketing in large-scale social networks, in: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2010, pp. 1029–1038.

[18] W. Chen, Y. Yuan, L. Zhang, Scalable influence maximization in social networks under the linear threshold model, in: The Proc. of 10th IEEE International Conference on Data Mining, ICDM, IEEE Computer Society, 2010, pp. 88–97.

[19] W. Chen, An issue in the martingale analysis of the influence maximization algorithm imm, arXiv:1808.09363, 2018.

[20] M. Minutoli, P. Sambaturu, M. Halappanavar, A. Tumeo, A. Kalyananaraman, A. Vullikanti, PREEMPT: scalable epidemic interventions using submodular optimization on multi-gpu systems, in: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2020, pp. 1–15.

[21] R. Barik, M. Minutoli, M. Halappanavar, A. Kalyanaraman, Impart: a partitioning-based parallel approach to accelerate influence maximization, in: 2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics, HiPC, IEEE, 2022, pp. 125–134.

[22] B. Mirzasoleiman, A. Karbasi, R. Sarkar, A. Krause, Distributed submodular maximization: identifying representative elements in massive data, in: Proc. of the 27th Annual Conference on Neural Information Processing Systems, 2013, pp. 2049–2057.

[23] V.S. Mirrokni, M. Zadimoghaddam, Randomized composable core-sets for distributed submodular maximization, in: Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC, ACM, 2015, pp. 153–162.

[24] U. Feige, A threshold of ln n for approximating set cover, J. ACM 45 (1998) 634–652, https://doi.org/10.1145/285055.285059.

[25] M. Minoux, Accelerated greedy algorithms for maximizing submodular set functions, in: Proceedings of the 8th IFIP Conference on Optimization Techniques, Springer, 1977, pp. 234–243.

[26] A. Gupta, A. Roth, G. Schoenebeck, K. Talwar, Constrained non-monotone submodular maximization: offline and secretary algorithms, in: Proc. of the 6th International Workshop of Internet and Network Economics, WINE, in: Lecture Notes in Computer Science, vol. 6484, Springer, 2010, pp. 246–257.

[27] A. Badanidiyuru, J. Vondrák, Fast algorithms for maximizing submodular functions, in: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, SIAM, 2014, pp. 1497–1514.

[28] A. McGregor, H.T. Vu, Better streaming algorithms for the maximum coverage problem, Theory Comput. Syst. 63 (2019) 1595–1619, https://doi.org/10.1007/s00224-018-9878-x.

[29] M. Bateni, H. Esfandiari, V.S. Mirrokni, Almost optimal streaming algorithms for coverage problems, in: Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA, ACM, 2017, pp. 13–23.

[30] R. Barik, et al., GreediRIS: scalable influence maximization using distributed streaming maximum cover, https://github.com/ReetBarik/GreediRIS, 2023. (Accessed July 2023), Online.

[31] J. Leskovec, A. Krevl, SNAP datasets: Stanford large network dataset collection, http://snap.stanford.edu/data, 2014.

[32] J. Kunegis, Konect: the Koblenz network collection, in: Proceedings of the 22nd International Conference on World Wide Web, 2013, pp. 1343–1350.

[33] Y. Li, J. Fan, Y. Wang, K. Tan, Influence maximization on social graphs: a survey, IEEE Trans. Knowl. Data Eng. 30 (2018) 1852–1872, https://doi.org/10.1109/TKDE.2018.2807843.

[34] A. Arora, S. Galhotra, S. Ranu, Debunking the myths of influence maximization: an in-depth benchmarking study, in: Proceedings of the 2017 ACM International Conference on Management of Data, 2017, pp. 651–666.

[35] M. Minutoli, Ripples: a C++ library of parallel implementations for distributed influence maximization, https://github.com/pnnl/ripples/tree/master, 2019.

Reet Barik received his bachelor's degree in Computer Science and Engineering from the Maulana Abul Kalam Azad University of Technology, West Bengal, India, in 2017, and a M.S. degree in Computer Science from Washington State University, Pullman, WA, USA, in 2019 where he is currently a PhD candidate. His research area is at the intersection of parallel graph algorithms, combinatorial optimization, and high performance computing under the general umbrella of scalable data science.

Wade Cappa received his bachelor's degree in Computer Science from Washington State University in 2023. Since then, he has been working at Palantir Technologies on network-to-network data sharing and consistency systems. Wade started working on HPC applications while he was studying for his undergraduate degree, and has continued working on research in his free time. Since graduation, his research has focused on parallel and distributed algorithms and their applications.

Dr. SM Ferdous joined the Pacific Northwest National Laboratory as a Linus Pauling Postdoctoral fellow in June 2022 and is now a data scientist at PNNL. He received his Bachelor's and master's in computer science and engineering from Bangladesh University of Engineering and Technology (BUET), and PhD from Purdue University. His research interest is in combinatorial scientific computing, focusing on computing under various constraints such as memory limitations, dynamic data, or approximation algorithms.

Dr. Marco Minutoli is a data scientist at the Pacific Northwest National Laboratory. He received his Bachelor of Science in Computer and Telecommunication Engineering from University of Messina, Messina, Italy, in 2008, his Master of Science in Computer Engineering from Politecnico di Milano, Milan, Italy, in 2014, and his PhD in Computer Science from Washington State University, Pullman, WA, in 2021. His research focuses on the design of parallel graph algorithms for combinatorial scientific computing and on the definition of HW/SW co-design and High-Level Synthesis methodologies and their compilation and optimization pipelines for the generation of custom computing devices optimized for irregular applications.

Dr. Mahantesh Halappanavar is a chief data scientist at the Pacific Northwest National Laboratory, where he serves as the group lead of the Data Science and Machine Intelligence group. He also holds a joint appointment as adjunct faculty in computer science at the School of Electrical Engineering and Computer Science at Washington State University in Pullman. His research has spanned multiple technical foci and includes combinatorial scientific computing, parallel graph algorithms, artificial intelligence and machine learning, and the application of graph theory and game theory to solve problems in application domains, such as scientific computing, power grids, cybersecurity, and life sciences.

Dr. Ananth Kalyanaraman received his bachelor's degree from the Visvesvaraya National Institute of Technology, Nagpur, India, in 1998, and the M.S. and Ph.D. degrees from Iowa State University, Ames, IA, USA, in 2002 and 2006, respectively. He is a Professor and Boeing Centennial Chair of computer science, and the Interim Director for the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, USA. He also holds a joint appointment with Pacific Northwest National Laboratory, Richland, WA, USA. Ananth is the Director for the AgAID Institute. His research focuses on developing parallel algorithms and software for data-intensive problems originating in the areas of computational biology and graph-theoretic applications.