Fast and Space-Efficient Parallel Algorithms for Influence Maximization

Letong Wang
UC Riverside
lwang323@ucr.edu

Xiangyun Ding
UC Riverside
xding047@ucr.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

ABSTRACT

Influence Maximization (IM) is a crucial problem in data science. The goal is to find a fixed-size set of highly influential *seed* vertices on a network to maximize the influence spread along the edges. While IM is NP-hard on commonly used diffusion models, a greedy algorithm can achieve $(1-1/e)$-approximation by repeatedly selecting the vertex with the highest *marginal gain* in influence as the seed. However, we observe two performance issues in the existing work that prevent them from scaling to today's large-scale graphs: space-inefficient memorization to estimate marginal gain, and time-inefficient seed selection process due to a lack of parallelism.

This paper significantly improves the scalability of IM using two key techniques. The first is a *sketch-compression* technique for the independent cascading model on undirected graphs. It allows combining the simulation and sketching approaches to achieve a time-space tradeoff. The second technique includes new data structures for parallel seed selection. Using our new approaches, we implemented *PaC-IM*: Parallel and Compressed IM.

We compare *PaC-IM* with state-of-the-art parallel IM systems on a 96-core machine with 1.5TB memory. *PaC-IM* can process the ClueWeb graph with 978M vertices and 75B edges in about 2 hours. On average, across all tested graphs, our uncompressed version is 5–18× faster and about 1.4× more space-efficient than existing parallel IM systems. Using compression further saves 3.8× space with only 70% overhead in time on average.

PVLDB Reference Format:
Letong Wang, Xiangyun Ding, Yan Gu, and Yihan Sun. Fast and Space-Efficient Parallel Algorithms for Influence Maximization. PVLDB, 17(3): 400 - 413, 2023.
doi:10.14778/3632093.3632104

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ucrparlay/Influence-Maximization.

1 INTRODUCTION

Influence Maximization (IM) is a crucial problem in data science. The goal is to find a fixed-size set of highly influential *seed* vertices on a network to maximize the spread of influence along the edges. For example, in viral marketing, the company may choose to send free samples to a small set of users in the hope of triggering a large cascade of further adoptions through the "word-of-mouth" effects.

Proceedings of the VLDB Endowment, Vol. 17, No. 3 ISSN 2150-8097.
doi:10.14778/3632093.3632104

Given a graph $G=(V,E)$ and a stochastic *diffusion model* to specify how influence spreads along edges, we use $n=|V|$, $m=|E|$, and $\sigma(S)$ to denote the expected influence spread on G using the seed set $S \subseteq V$. The IM problem aims to find a seed set S with size k to maximize $\sigma(S)$. Given its importance, IM is widely studied, and we refer the audience to a list of surveys [4, 6, 88] that reviews the numerous applications and a few hundred papers on this topic.

Among various diffusion models, Independent Cascade (IC) [32] (defined in Sec. 2) is one of the earliest and most widely used. In IC, only seeds are ***active*** initially. In each timestamp, each vertex v that is newly activated in the last timestamp will activate its neighbors u with a probability p_{vu}. Although IM is NP-hard on IC [42], the monotone and submodular properties of IC allow for a greedy algorithm with $(1-1/e)$-approximation [42]. Given the current seed set S, the greedy algorithm selects the next seed as the vertex with the highest *marginal gain*, i.e., $\arg\max_{v\in V}\{\sigma(S\cup\{v\})-\sigma(S)\}$. Due to the theoretical guarantee, this greedy strategy generally gives better solution quality than other heuristics [51]. However, the challenge lies in estimating the influence $\sigma(S)$ of a seed set S. Early work uses Monte-Carlo (MC) experiments by averaging R' rounds of influence diffusion simulation [42, 49], but the solution quality relies on a high value of R' (usually around 10^4). Later work uses ***sketch-based*** approaches [16, 20–22, 59, 74, 75] to avoid MC experiments. Such algorithms pre-store R ***sketches***. Each sketch is a sampled graph—an edge (v,u) is chosen with probability p_{vu}. When estimating $\sigma(S)$, the sampled graphs are used as the results of the MC experiments of influence diffusion. In an existing study [21], using $R\approx 200$ sketches achieves a similar solution quality to $R'=10^4$ MC experiments, greatly improving efficiency. The sketches can either be the sampled graphs and/or *memoizing* more information from the sampled graphs to accelerate influence computation, such as connectivity [20, 31] or strong connectivity [59].

While numerous sketch-based solutions have been developed, we observed great challenges in scaling them to today's large-scale graphs. In a benchmark paper [2] on nine state-of-the-art (SOTA) sequential IM solutions, none of them can process the Friendster (FT) graph [47] with 65M vertices and 3.6B edges due to timeout (more than 40 hours) or out-of-memory. Even the recent parallel algorithms [31, 55, 56, 63] need more than half an hour to process FT on a 96-core machine (See Table 4). Two major challenges exist to scale sketch-based approaches to billion-scale graphs. The first is the *space*. Storing each sketch usually needs per-vertex information. This indicates $O(Rn)$ space, which is expensive on large graphs (empirically, R is a few hundred). The second is *insufficient parallelism*. Many SOTA IM solutions use the *CELF* [49] optimization for seed selection (see details below), which is inherently sequential.

This paper takes a significant step to ***improve the scalability of sketch-based IM solutions*** and tests the algorithms on ***real-world billion-scale graphs***. We propose two techniques to improve both

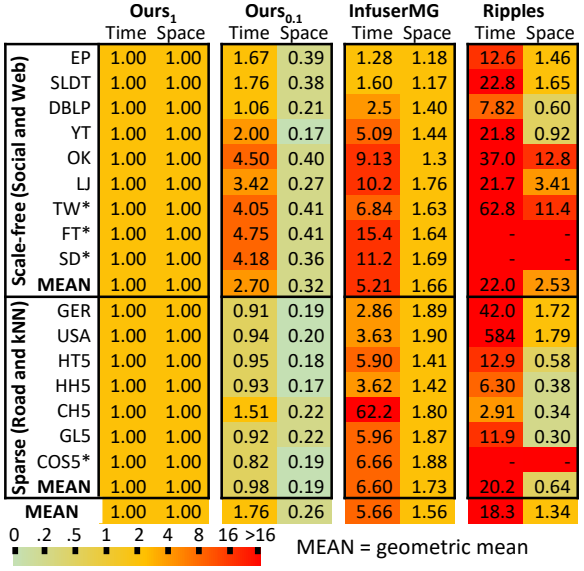

Figure 1: Heatmap of relative running time and space usage, normalized to Ours$_1$. Ours$_1$: *PaC-IM* with no compression. Ours$_{0.1}$: *PaC-IM* with 10× sketch compression. *InfuserMG* [31] and *Ripples* [56]: existing parallel IM systems. Lower/green is better. The graph information is in Tab. 3. The running times are in Tab. 4. ∗: graphs with more than a billion edges.

space and time. The first is a ***sketch compression*** technique for the *IC model on undirected graphs*, which limits the auxiliary space by a user-defined capacity to reduce space usage. Our second technique is ***parallel data structures*** for seed selection to reduce running time, which works on general graphs and any diffusion model with submodularity. Combining the new ideas, we implemented ***PaC-IM*: Parallel and Compressed IM**. We show a heatmap in Fig. 1 to overview our results. On the aforementioned FT graph, *PaC-IM* only uses 128 seconds without compression (using 2.5× auxiliary space on top of the input graph), or 609 seconds when limiting auxiliary space in 0.45× input size, using a 96-core machine. *PaC-IM* is at least 15× faster than existing parallel solutions while using much less space and achieving the same solution quality (see Tab. 4). Below, we overview the key contributions of this paper.

Our first contribution is a **compression scheme for sketches** on undirected graphs and the IC model, which allows for user-defined compression ratios (details in Sec. 3). Similar to existing work [20, 31], *PaC-IM* memoizes connected components (CC) of the sketches but *avoids the $O(Rn)$ space to store per-vertex information.* Our idea is a combination (and thus a tradeoff) of memoization and simulation. The idea is to memoize the CC information only for *centers* $C \subseteq V$, where $|C| = \alpha n$, and $\alpha \in [0, 1]$ is a user-defined parameter. A local simulation will retrieve the influence information of a non-center vertex. Theoretically, we show that we can limit the auxiliary space by a factor of α by increasing the time by a factor of $O(1/\alpha)$. Experimentally, such tradeoff is studied in Fig. 7.

Our second contribution is **two new parallel data structures for seed selection**. Recall that many SOTA IM solutions [20, 21, 31, 43, 56] use the CELF optimization [49] (details in Sec. 2) for seed selection. In a nutshell, CELF is an iterative approach that lazily evaluates the marginal gain of vertices in seed selection, one at a time. While laziness reduces the number of vertices to evaluate, CELF is inherently sequential. We proposed two novel solutions that achieve high parallelism for CELF. The challenge is evaluating more vertices in parallel while avoiding "unpromising" vertices as in CELF. Our first solution is a binary search tree (BST) called *P-tree* [10, 12, 73] (Sec. 4.1). We highlight its *theoretical efficiency* (Thm. 4.1 to 4.3). Our second solution is referred to as *Win-Tree* (Sec. 4.2), which has lower space usage, leading to slightly better overall performance. The two solutions work on both directed and undirected graphs and any diffusion model with submodularity. They are potentially extendable to other optimization problems with submodular objective functions (see discussions in Sec. 7).

We experimentally study the performance of *PaC-IM* and compare it with SOTA parallel IM systems. We tested 17 graphs, five of which have over a billion edges. Besides social networks, we also tested other real-world graphs, including web graphs, road graphs, and k-NN graphs. One can consider IM on such graphs as studying the influence diffusion among webpages and geologically or geometrically close objects. On all tested graphs, *PaC-IM* achieves the best running time and space usage while guaranteeing comparable or better solution quality to all baselines. Compared to the best baseline, *PaC-IM* with no compression is 5.6× faster and is 1.5× more space-efficient on average (geometric mean across tested graphs), and is 3.2× faster and uses 6× less space using compression with $\alpha = 0.1$. Due to space- and time-efficiency, *PaC-IM* is the only system to process the largest graph ClueWeb [54] with 978M vertices and 75B edges. We believe *PaC-IM* is the first IM solution that scales to tens of billions of edges and close to a billion vertices.

For page limit, we provide the full version of this paper [77] with full proofs and more experimental evaluations.

2 PRELIMINARIES

For graph $G = (V, E)$, we use $n = |V|$ and $m = |E|$. Since our sketch compression is designed for undirected graphs and IC model, throughout the paper, we assume G is undirected and consider the IC model unless otherwise specified. A ***connected component (CC)*** is a maximal subset in V s.t. every two vertices in it are connected by a path. In a max-priority-queue, we use ***top*** to refer to the element with the largest key, and use function ***pop*** to find and remove the top element. $\tilde{O}(f(n))$ denotes $O(f(n) \cdot \text{polylog}(n))$.

Computational Model. We use the fork-join parallelism [13, 23], and the work-span analysis [15, 36, 38]. We assume a set of threads that access a shared memory. A thread can `fork` two child software threads to work in parallel. When both children complete, the parent process continues. A parallel for-loop can be simulated by recursive `fork`s in logarithmic levels. The ***work*** of an algorithm is the total number of instructions, and the ***span*** is the length of the longest sequence of dependent instructions. We can execute the computation using a randomized work-stealing scheduler [3, 15] in practice. We use *atomic* operation WriteMax(t, v_{new}) to write value v_{new} at the memory location t if v_{new} is larger than the current value in t. We use compare-and-swap to implement WriteMax.

The Influence Maximization (IM) Problem

Given a graph $G = (V, E)$, an influence diffusion model M specifies how influence spreads from a set of current ***active*** vertices to ***activate*** more vertices in V. Given a ***seed*** set $S \subseteq V$, we use $\sigma_{G,M}(S)$

Algorithm 1: Sketch-based IM algorithm

Notations: $G = (V, E)$: the input graph. k: the number of seed vertices. R: the number of sampled graphs
Output: S: a set of K seeds that maximizes influence on G
Notes: $\Phi_{1..R}$: R sketches computed from R sampled graphs

// Step 1: Sketch construction

ParallelForEach $r \leftarrow 1 \dots R$ **do**
  $\Phi_r \leftarrow \textsc{Sketch}(r)$ *// Compute the r-th sketch*

// Step 2: Seed selection using CELF

$S \leftarrow \emptyset$
while $|S| < k$ **do**
  $s^* \leftarrow \textsc{NextSeed}(S, \Phi_{1..R})$ *// Find* $\arg\max_{v \in V} \textsc{Marginal}(S, v, \Phi_{1..R})$
  $\textsc{MarkSeed}(s^*, \Phi_{1..R}, S)$ *// Mark s^* as a seed in the sketches*
  $S \leftarrow S \cup \{s^*\}$
return S

Algorithm 2: Sequential Seed Selection with CELF

Notes: Q: max-priority-queue on all vertices $v \in V$ with key $\bar{\Delta}[v]$
Initially $\bar{\Delta}[v] = \textsc{Marginal}(\emptyset, v, \Phi_{1..R})$

Function $\textsc{NextSeed}(S, \Phi_{1..R})$ *// S: current seed set; $\Phi_{1..R}$: R sketches*
  Repeat
    $s^* \leftarrow Q.\textsc{Pop}()$ *// Pop: find and remove the top*
    $\bar{\Delta}[s^*] \leftarrow \textsc{Marginal}(S, s^*, \Phi_{1..R})$
    if $\bar{\Delta}[s^*] > \bar{\Delta}[Q.\textsc{Top}()]$ **then return** s^*
    else $Q.\textsc{Insert}(s^*)$ *// insert s^* back with new score*

Table 1: Notations in the paper.

$G = (V, E)$: The input graph.	k : number of seeds.
$\sigma(S)$ or $\sigma_{G,M}(S)$: The influence spread of seed set $S \subseteq V$ on graph G and diffusion model M.	
$\Delta(v \mid S)$: The ***true score*** (marginal gain) of v on top of S. $\Delta(v \mid S) = \sigma(S \cup \{v\}) - \sigma(S)$. We omit S and use $\Delta(v)$ with clear context.	
$\bar{\Delta}[v]$: The ***stale score*** (lazily-evaluated marginal gain) of v stored in an array. It is an upper bound of $\Delta(v \mid S)$ for the current seed set S.	
$\Phi_{1..R}$: the sketches. Formally defined in Sec. 3.	
ρ and α : $\rho = \alpha n$ is the number of centers.	

Function names:

$\textsc{Sketch}(G, r)$: Compute the r-th sketch from graph G
$\textsc{Marginal}(S, v, \Phi_{1..R})$: The marginal gain of vertex v given the current seed set S estimated from R sketches $\Phi_{1..R}$
$\textsc{NextSeed}(S, \Phi_{1..R})$: Greedily determine the next seed based on sketches $\Phi_{1..R}$ given the current seed set S.

to denote the expected number of vertices that S can activate (including S) under diffusion model M on graph G. The IM problem is to find $S^* \subseteq V$ with size k, s.t. S^* maximizes the influence spread function $\sigma_{G,M}$. With clear context, we omit M and G, and use $\sigma(\cdot)$. Several propagation models have been proposed, including the Independent Cascade (IC) model [32], Linear Threshold (LT) model [34, 67], and more [18, 42, 52, 65]. Since our sketch compression focuses on the IC model, we briefly introduce it here. In the IC model, influence spreads in rounds. Initially, only the seed vertices are active. In round i, each vertex u that was newly activated in round $i-1$ attempts to spread the influence via all incident edges e, and activate the other endpoint v with probability p_e.

Kempe et al. [42] proved that IM under the IC model is NP-hard, and that the influence spread function has the following properties: for every $X, Y \subseteq V$ where $X \subseteq Y$, and $v \in V \setminus Y$, we have:

$$\textbf{Monotonicity:} \quad \sigma(Y \cup \{v\}) \geq \sigma(Y) \tag{1}$$

$$\textbf{Submodularity:} \quad \sigma(X \cup \{v\}) - \sigma(X) \geq \sigma(Y \cup \{v\}) - \sigma(Y) \tag{2}$$

These two properties allow the following **greedy algorithm** (later referred to as *GeneralGreedy*) to give a $(1 - 1/e)$-approximation. The algorithm starts with $S = \emptyset$ and repeatedly adds the vertex with the highest ***marginal gain*** to S, until $|S| = k$. The marginal gain $\Delta(v \mid S)$ of a vertex v given the current seed set S is defined as:

$$\Delta(v \mid S) = \sigma(S \cup \{v\}) - \sigma(S) \tag{3}$$

With clear context, we omit S and use $\Delta(v)$, and also call it the ***score*** or the ***true score*** of v. We call the process to compute the true score of a vertex an ***evaluation***. To estimate $\sigma(\cdot)$, early solutions average R' rounds of Monte Carlo (MC) experiments of influence diffusion simulation. However, on real-world graphs, this approach requires a large value of R' to converge, which can be expensive.

Sketch-Based Algorithms. Sketch-based algorithms are proposed to accelerate $\Delta(v)$ evaluations. Instead of running independent MC experiments for each evaluation, sketch-based algorithms statically sample R graphs to reflect MC experiments and consistently simulate the results on them. Using sketches allows for much faster convergence, making the number of needed simulations (i.e., sketches) R smaller than that in MC experiments. Hence, sketch-based algorithms are widely studied [16, 20–22, 59, 74, 75]. We summarize sketch-based algorithms in two steps (see Alg. 1): *sketch construction* and *seed selection*. Next, we introduce both steps with optimizations in previous work. We summarize some related work in Tab. 2, and review more in Sec. 6. Some notations are given in Tab. 1.

Sketch Construction. In the earliest sketch-based algorithm *StaticGreedy*, R sampled graphs [21] are explicitly stored as sketches. In the IC model, the r-th sketch corresponds to a sampled graph $G'_r = (V, E'_r)$, where $E'_r \subseteq E$, such that each edge $e \in E$ is sampled with probability p_e, meaning a successful activation. An evaluation will average the number of reachable vertices on all sampled graphs from the seed set S. A later paper *Infuser* [31] proposed the ***fusion*** optimization, which uses hash functions to avoid explicitly storing the sampled graphs G'_r. They sample an edge e in a sketch G'_r with a random number generated from seed $\langle e, r \rangle$, such that whether an edge is selected in a certain sketch is always deterministic, and a sampled graph G'_r can be fully reconstructed from the sketch id r. We also use this idea in our sketch compression algorithm.

Many existing algorithms also use ***memoization*** to avoid influence spread simulation on sketches. On undirected graphs and the IC model, the *MixGreedy* paper [20] first observed that a vertex v's influence on a sketch is all vertices in the same connected component (CC) as v, but they only used this idea to select the first seed. Later, *InfuserMG* adopts this idea to select all seeds and memoizes the CC information of each sampled graph as the sketch. A vertex v's score is then the average of the (inactivated) CC sizes on all R sketches, which can be obtained in $O(R)$ cost. This approach avoids simulation but leads to $O(Rn)$ space that is expensive for large graphs. Sec. 3 presents how our sketch compression approach reduces this high space usage.

Table 2: Existing approaches and our new one. *MixGreedy* uses different approaches to select the first seed and the other seeds, so we list them separately. "#vertices per seed": number of vertices to visit in all re-evaluations involved to find a seed. For *StaticGreedy*, we assume the *fusion* optimization in [31] to avoid explicitly storing sampled graphs. n: number of vertices. n_c: number of re-evaluations needed in CELF. T: the average number of reachable vertices in a simulation (or a sketch). Empirically, T is large, and $n_c \ll n$. To achieve similar quality, $R \ll R'$.

	Randomization	Compute Influence	Select Seed	#vertics per seed	Space	Parallel
***GeneralGreedy* [42]**	R' Monte Carlo experiments	Simulation	Evaluate all	$O(nR'T)$	$O(n)$	no
***MixGreedy* [20] 1st Seed**	Fixed R sampled graphs	Memoization	Evaluate all	$O(nR)$	$O(n)$	no
***MixGreedy* [20] Others**	R' Monte Carlo experiments	Simulation	CELF	$O(n_cR'T)$	$O(n)$	no
***StaticGreedy* [21]**	Fixed R sampled graphs	Simulation	CELF	$O(n_cRT)$	$O(n)$	no
***InfuserMG* [31]**	Fixed R sampled graphs	Memoization	CELF	$O(n_cR)$	$O(nR)$	yes
***PaC-IM* (this work)**	Fixed R sampled graphs	Simulation + Memoization	CELF	$O(n_cR \cdot \min(T, 1/\alpha))$	$O((1+\alpha R)n)$	yes

Algorithm 3: Our sketch algorithm with compression

Global Variables: $G = (V, E)$: input graph; R: number of sketches
$C = \{c_1, c_2, \ldots, c_\rho\} \subseteq V$: randomly selected *centers*. $\rho = |C|$
Notes: A sketch Φ_r is a triple $\langle r, size[1..\rho], label[1..\rho]\rangle$, defined at the beginning of Sec. 3.

1 **Function** SKETCH(r) *// r: sketch id*
2 Compute the connected components of graph $G' = (V, E')$, where $E = \{(u,v) \mid (u,v) \in E, \text{SAMPLE}(u, v, r) = true\}$
3 **ParallelForEach** $c_i \in C$ **do**
4 $label[i] \leftarrow \min_j \{c_j$ is in the same CC as $c_i\}$
5 **ParallelForEach** $c_i \in C$ **do**
6 **if** $label[i] = i$ **then** $size[i] \leftarrow$ the CC size of center c_i
7 **return** $\langle r, label[1..\rho], size[1..\rho]\rangle$

// sample an edge e with probability p_e for sketch r
8 **Function** SAMPLE(e, r) *// e: edge identifier; r: sketch identifier*
9 $p \leftarrow random(e, r)$ *// Generate $p \in [0,1]$ from random seed e, r*
10 **return** $p \le p_e$

// δ: marginal influence of v on sketch Φ_r. l: label of v's CC in sketch Φ_r if v is connected to any center; otherwise $l = -1$. S: the current seed set.
11 **Function** $\langle \delta, l\rangle = \text{GETCENTER}(\Phi_r, v, S)$
12 Start BFS from v on a sampled subgraph of G, where an edge $e \in E$ exists if SAMPLE(e, r) = *true*. Count the #reached-vertices as n'
13 **if** a center $c_i \in C$ is encountered during BFS **then**
14 $l \leftarrow \Phi_r.label[i]$ *// Find the label of the center*
15 **return** $\langle \Phi_r.size[l], l\rangle$ *// the CC size and label of the center*
16 **if** any $v' \in S$ has been visited by BFS **then return** $\langle 0, -1\rangle$
17 **else return** $\langle n', -1\rangle$ *// influence is #visited vertices in BFS*

// Marginal gain of v given seed set S on sketches $\Phi_{1..R}$
18 **Function** MARGINAL($S, v, \Phi_{1..R}$)
19 **ParallelForEach** $r \leftarrow 1..R$ **do**
20 $\langle \delta_r, \cdot\rangle \leftarrow \text{GETCENTER}(\Phi_r, v, S)$
21 **return** $(\sum_{r=1}^{R} \delta_r)/R$ *// the sum can be computed in parallel*
22 **Function** MARKSEED($v, \Phi_{1..r}, S$)
23 **ParallelForEach** $r \leftarrow 1..R$ **do** *// For each sketch Φ_r*
24 $\langle \delta, l\rangle \leftarrow \text{GETCENTER}(\Phi_r, v, S)$
25 **if** $l > 0$ **then** $\Phi_r.size[l] \leftarrow 0$ *// Clear the corresponding CC size*

Seed Selection with CELF. A useful optimization for the greedy algorithm is CELF [49], which avoids evaluating all vertices in NEXTSEED. CELF uses *lazy* evaluation for submodular functions and evaluates a vertex only if it becomes a "promising" candidate for the next seed. We show the CELF seed selection algorithm in Alg. 2. CELF uses a priority queue Q to maintain all vertices with their scores as the key. Due to submodularity, $\Delta(v)$ is non-increasing with the expansion of the seed set S. With lazy evaluation, the scores in Q may be stale but are always upper bounds of the true scores. We call this lazily-evaluated score the ***stale score*** of v and denote it as $\bar{\Delta}[v]$ stored in an array. To select the next seed, CELF keeps popping the top element v from Q, re-evaluating its true score $\Delta(v)$, storing $\Delta(v)$ to $\bar{\Delta}[v]$, and inserting it back unless $\Delta(v)$ is greater than the current largest value in Q (Line 3-6). In this case, we can set v as the next seed without more evaluations, as the true scores of other vertices can not exceed their values in Q. Although CELF can reduce the number of evaluations, it is essentially sequential and evaluates vertices one by one. In Sec. 4, we present our new data structures that allow for parallel evaluations in CELF.

3 SPACE-EFFICIENT SKETCHES

This section presents our new technique to construct compressed sketches in parallel for the IC model on undirected graphs. In this setting, as discussed in Sec. 2, a vertex v can activate all vertices in the same CC on a certain sketch. Thus, *memoizing* per-vertex CC information in sketches [31] can accelerate the influence evaluation but requires $O(Rn)$ space, which does not scale to large input graphs. Alternatively, one can avoid memoization and run a *simulation* by traversing the sampled graph to find the CC when needed. This requires no auxiliary space but can take significant time.

Our approach combines the benefit of both, using bounded-size auxiliary space while allowing for efficiency. Our key idea is to only *partially memoize* CC information in each sketch, and retrieve this information by *partial simulations*. In *PaC-IM*, we only store the CC information for $\rho = \alpha \cdot n$ ***center*** vertices $C \subseteq V$, where $\alpha \in [0,1]$ is a user-defined parameter. Each sketch Φ_r is a triple $\langle r, label[\cdot], size[\cdot]\rangle$ corresponding to an implicit sampled graph G'_r from G, where each edge $e = (u,v)$ is retained with probability p_e.

- r: the sketch id. Similar to the fusion idea mentioned in Sec. 2, the sketch id fully represents the sampled graph.
- $label[1..\rho]$: the CC label of center c_i on this sketch, which is the smallest center id j where c_j is in the same CC as c_i.
- $size[1..\rho]$: if $label[i] = i$ (i.e., i represents the label of its CC), $size[i]$ is the influence of center c_i on this sketch. It is initially the CC size of c_i and becomes 0 when any vertex in this CC is selected as a seed.

Algorithm Overview. We first present the high-level idea of our sketch compression algorithm. We present our algorithm in Alg. 3 and an illustration in Fig. 2. As mentioned, we select $\rho = \alpha n$ center vertices uniformly at random. We only store the CC information (label and size) for the centers in sketches. We use a global flag array to indicate if a vertex is a center, and thus the total space is $O((1+\alpha R)n)$. To retrieve the CC size of a vertex v on sketch

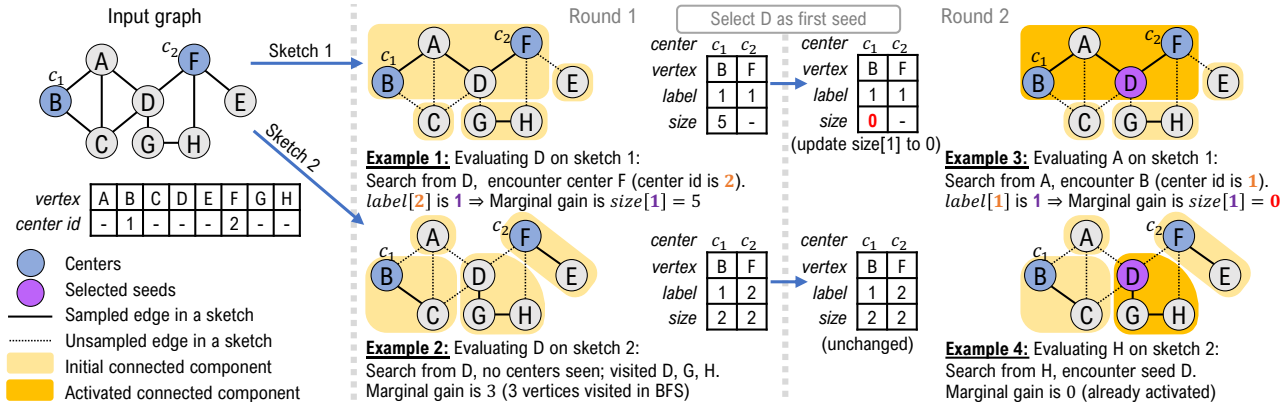

Figure 2: An example of our sketch compression on a graph with 8 vertices (B and F as centers) and $R = 2$ sampled graphs.

Φ_r, we start a breadth-first search (BFS) from v. If any center c_i is encountered, v should activate the same set of vertices as c_i on this sketch. As such, we can stop searching and use c_i's influence (CC size) as the influence for v. If v is not connected to any center, the CC containing v is likely small, and the BFS can visit all of them quickly. In either case, the number of visited vertices in BFS can be bounded. In Thm. 3.1, we show that compressing the auxiliary space by a factor of α roughly increases the evaluation time by a factor of $O(1/\alpha)$. By controlling the number of centers, we can achieve a tradeoff between the evaluation time and space usage.

Next, we elaborate on the three functions in Alg. 1: SKETCH(G, r), which constructs the r-th sketch from G, MARGINAL$(S, v, \Phi_{1..R})$, which computes the score (marginal gain) of a vertex v on top of S using sketches $\Phi_{1..R}$, and MARKSEED$(s^*, \Phi_{1..R})$, which adjusts the sketches $\Phi_{1..R}$ when s^* is selected as a seed.

Sketch Construction (SKETCH(G, r)). Recall that we maintain CC information for $\rho = \alpha n$ centers $C = \{c_1, c_2, \ldots, c_\rho\}$ in sketches. To construct a sketch Φ_r, we first compute the CC information of the sampled graph G'_r, which can be performed by any parallel connectivity algorithm [26]. We store the CC information for all centers in two arrays. $\Phi_r.label[i]$ records the label of CC of the i-th center. For multiple centers in the same CC, we simply use the smallest CC id as the label for all of them to represent this CC, so all centers find the CC information by referring to their label. For a center c_i, If i is the label of its CC, we use $\Phi_r.size[i]$ to record the size of this CC. An example of these arrays is given in Fig. 2.

Computing the Marginal Gain (MARGINAL$(S, v, \Phi_{1..R})$). Given the sketches $\Phi_{1..R}$ and the current seed set S, the function MARGINAL$(S, v, \Phi_{1..R})$ computes the marginal gain of a vertex v by averaging the marginal gains of v on all sketches. We use a helper function $\langle \delta, l \rangle$ = GETCENTER(Φ_r, v, S), which returns δ as the marginal gain of v on sketch Φ_r, and l as the label of centers connected to v ($l = -1$ if no center is connected to v). This function will run a breadth-first search (BFS) from v on G'_r (i.e., only using edges $e \in E$ s.t. SAMPLE(e, r) is *true*). If any center is encountered during this BFS (Examples 1 and 3 in Fig. 2), then the influence of v is the same as c_i on this sketch. The information of c_i is retrieved by its label $l = \Phi_r.label[i]$, and thus $\delta = \Phi_r.size[l]$. The influence δ is either the size of the CC containing v when no vertices in this CC are seeds (Example 1 in Fig. 2), or 0 otherwise, as is updated in MARKSEED (Example 3 in Fig. 2). Otherwise, if the BFS terminates without visiting any centers, it will return -1 for the label l. The influence δ is either the number of vertices n' visited during the BFS, which is also the size of CC containing v (Example 2 in Fig. 2), or $\delta = 0$ (Line 16) if any seed is visited during BFS (Example 4). Using the GETCENTER function, the marginal influence of v on sketch Φ_r can be obtained as the first return value δ of GETCENTER(Φ_r, v, S).

Marking a Seed (MARKSEED$(s^*, \Phi_{1..R})$). The function updates the sketches when $s^* \in V$ is selected as a seed. For each sketch Φ_r, the CC label of s^* is the second return value of GETCENTER(Φ_r, s^*, S). If the label is not -1, we set $\Phi[r].size[l]$ as 0—since s^* is selected, all other vertices in this CC will get no marginal gain on this sketch.

Our approach allows for a tradeoff between space and query time in MARGINAL(): a smaller ρ (fewer centers) means less space but a higher evaluation cost, as it may take longer to find a center. *PaC-IM* unifies and is a hybrid of *StaticGreedy* and *InfuserMG*. Theoretically, using $\alpha = 1$, our sketch is equivalent to *InfuserMG* where the CC information for all vertices on all sketches are memoized; using $\alpha = 0$, our sketch is equivalent to *StaticGreedy* with no memoization, and evaluations are done by traversing the sampled graph. In practice, *PaC-IM* is much faster than *StaticGreedy* and *InfuserMG* even when with no compression due to better parallelism. We summarize the theoretical guarantees in Tab. 2, and state them in Thm. 3.1. We give the proof in the full version of this paper.

THEOREM 3.1. *PaC-IM with parameter α requires $O((1 + \alpha R)n)$ space to maintain R sketches, and visits $O(R \cdot \min(1/\alpha, T))$ vertices to re-evaluate the marginal gain of one vertex v, where T is the average CC size of v on all sketches.*

4 PARALLEL SEED SELECTION

We now present the parallel seed selection process in *PaC-IM*. We call each iteration in seed selection (selecting one seed) a ***round***. Recall that prior solutions use the CELF optimization (see Tab. 2 and Sec. 2), which maintains (possibly stale) scores of all vertices in a priority queue Q and updates them lazily. In each round, CELF pops the vertex with the highest (stale) score from Q and re-evaluates it. The process terminates when a newly evaluated score is higher than all scores in Q; otherwise, the new score will be inserted back into Q. For simplicity, we assume no tie between scores. In

Algorithm 4: Seed Selection based on *P-tree*

Maintains: A parallel binary search tree T for all $\bar{\Delta}[v]$.

1 **Function** NextSeed($S, \Phi_{1..R}$)
2 $s^* \leftarrow \perp$ *// The best seed so far*
3 $j \leftarrow 0$
4 **repeat**
  // Extract (remove & output) the top 2^j elements in T into array B_j
5 $B_j[1..2^j] \leftarrow T$.SplitAndRemove(2^j)
6 **ParallelForEach** $v \in B_j$ **do** *// Get the true score for each $v \in B_j$*
7 $\bar{\Delta}[v] \leftarrow$ Marginal($S, v, \Phi_{1..R}$)
8 $t \leftarrow \arg\max_{v \in B_j} \bar{\Delta}[v]$ *// can be computed in parallel*
9 **if** $(s^* = \perp)$ or $(\bar{\Delta}[t] > \bar{\Delta}[s^*])$ **then** $s^* \leftarrow t$
10 $j \leftarrow j + 1$
11 **until** $\bar{\Delta}[s^*] > T$.Max() *// $\bar{\Delta}[s^*]$ is better than the top in T*
12 T.BatchInsert($\bigcup_{j'=0}^{j-1} B_{j'} \setminus \{s^*\}$)
13 **return** s^*

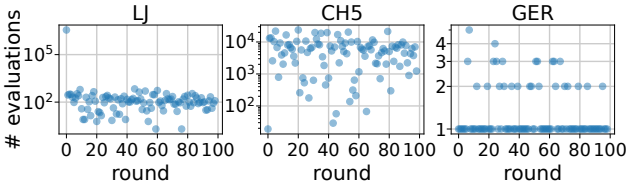

Figure 3: The number of evaluations by CELF in each round. A point (x, y) means that CELF does y re-evaluations in the x-th round.

the cases when ties exist, we break the tie by vertex id. Let $F_i = \{v \mid \bar{\Delta}_{i-1}[v] \geq \Delta^*\}$ be the set of vertices re-evaluated by CELF in round i, where $\bar{\Delta}_{i-1}[v]$ is the stale score of v after round $i-1$, and Δ^* is the maximum true score in round i (i.e., the score of the chosen seed in round i). We experimentally study the distribution of $|F_i|$ on all graphs and show three representative graphs in Fig. 3. Except for road networks (e.g., GER in Fig. 3), most graphs may require a large number of re-evaluations in certain rounds. However, CELF is inherently sequential. Some existing parallel implementations (e.g., [31, 56]) only parallelize the evaluation function Marginal (Line 4 in Alg. 2), but leave the CELF process sequential, and perform all $|F_i|$ evaluations one by one. When F_i is large (as in social and web networks), the sequential CELF results in low parallelism.

Prior Work on Parallel Priority Queue. As a fundamental data type, parallel priority queues are widely studied [9, 12, 27, 66, 72, 73, 82]. However, as far as we know, all these algorithms/interfaces require knowing the batch of operations (e.g., the threshold to extract keys or the number of keys to extract) *ahead of time*. This is not true in CELF—the set F_i is only known during the execution. Thus, we need different approaches to tackle this challenge.

Overview of Our Approaches. We first formalize the interface of the parallel priority queue needed in Alg. 1. The data structure maintains an array $\bar{\Delta}[\cdot]$ of (stale) scores for all vertices. It is allowed to call Marginal function to re-evaluate and obtain the true score of any vertex. The interface needs to support NextSeed function, which returns the vertex id with the highest true score.

We present two parallel data structures to maintain the scores. The first one is based on a parallel binary search tree (BST) called ***P-tree*** [10, 12, 24, 73]. We prove that using *P-tree*s, our approach has work (number of evaluations) asymptotically the same as CELF, while is highly parallel: the selection of the i-th seed finishes in $\log |F_i|$ iterations of evaluations (each iteration evaluates multiple vertices in parallel), instead of $|F_i|$ iterations (one vertex per iteration) in CELF. We also propose a new data structure ***Win-Tree***, based on parallel winning trees. *Win-Tree* does not maintain the total order of scores, and is simpler and potentially more practical than *P-tree*s. We introduce the *P-tree*-based approach in Sec. 4.1 and *Win-Tree*-based approach in Sec. 4.2, and compare their performance in Sec. 5.3. *These two approaches are **independent of the sketching algorithm** and apply to seed selection on all submodular diffusion models* (not necessarily the IC model and/or on undirected graphs). Note that most IM diffusion models are submodular (e.g., IC, Linear Threshold (LT), Triggering (TR [42]), and more [42, 76, 85]).

4.1 Parallel Priority Queue Based on *P-tree*

Our first approach to parallelizing CELF is to maintain the total (decreasing) order of the scores of all vertices, using a parallel binary search tree (BST) called *P-tree* [10, 12]. We will use two functions on a *P-tree* T: 1) T.SplitAndRemove(k), which extracts (removes and outputs to an array) the first k tree nodes (k largest scores) from T, and 2) T.BatchInsert(B), which inserts a set of keys B to T. Both algorithms are parallel with a polylogarithmic span.

Since *P-tree* maintains the total order of all vertices, we can extract a batch of vertices with top (stale) scores and evaluate them in parallel. The key challenge is a similar stop condition as CELF to avoid evaluating too many vertices, since we are unaware of the number of "useful" vertices, i.e., $|F_i|$, ahead of time. Our idea is to use *prefix doubling* [14, 25, 35, 37, 68, 69, 78] to achieve work efficiency and high parallelism. The pseudocode is given in Alg. 4 with an illustration in Fig. 4. To find the next seed, the *P-tree* T starts with the stale scores $\bar{\Delta}[\cdot]$ from the previous round. We then extract the top (largest stale score) nodes in batches of size $1, 2, 4, 8, \ldots$ from T (Line 5). Within each batch B_j, we re-evaluate all vertices in parallel (Line 7). These new scores are used to update the current best seed s^*. The loop terminates when the score of s^* is better than the best score in T (Line 11). Finally, we select s^* as the seed and insert the rest of the new true scores $\bigcup B_j \setminus \{s^*\}$ back to T.

Due to prefix doubling, each seed selection finishes in at most $O(\log n)$ rounds. Note that our approach evaluates more vertices than CELF, but due to the stop condition (Line 11), the extra work is bounded by a constant factor (proved in Thm. 4.2).

Theorem 4.1 (*P-tree* Correctness). *Alg. 4 always selects the next seed with the largest marginal gain, i.e.,* $\Delta(s^*) = \max_{v \in V}\{\Delta(v)\}$.

Due to the space limit, we defer the proof to the full version.

Theorem 4.2 (*P-tree* Efficiency). *Alg. 4 has the total number of evaluations at most twice that of CELF.*

Proof. Recall that F_i is the set of evaluated vertices by CELF when selecting the i-th seed. Let F'_i be the set of evaluated vertices by *P-tree*s in Alg. 4. We first show a simple case—if both CELF and Alg. 4 start with the same stale scores $\bar{\Delta}_{i-1}[\cdot]$, then $|F'_i| \leq 2|F_i|$. Let us reorder vertices in V as $v_1, v_2, \ldots, v_n$ by the decreasing order of their stale score $\bar{\Delta}_{i-1}[\cdot]$. Assume v_l is the last vertex evaluated by CELF, so $|F_i| = l$. v_l must be in the last batch in *P-tree*. Assume

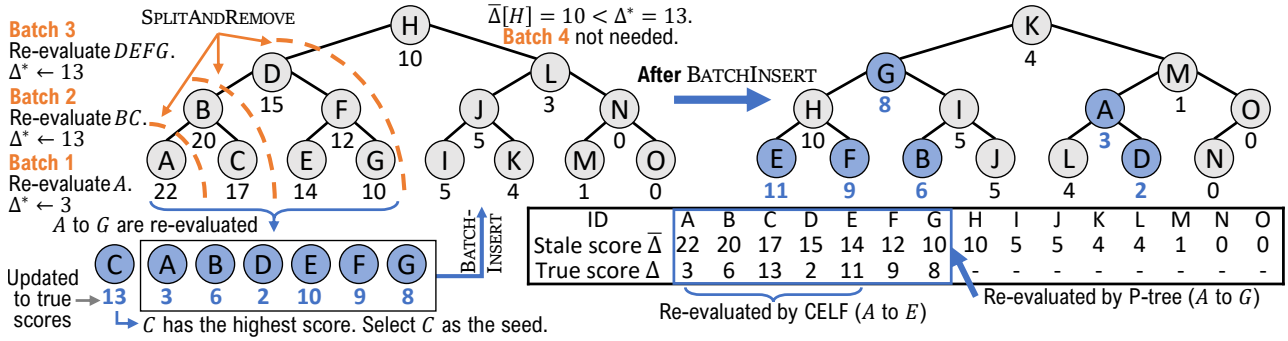

Figure 4: Example of *P-tree*-Based Seed Selection. The letters in the tree nodes represent vertices, and the numbers below them are their stale scores. *P-tree* maintains decreasing order of the (stale) scores. By prefix-doubling, we extract batches of 1, 2, 4 vertices and evaluate each batch in parallel. After batch 3, the highest true score (13) is higher than the current best in the tree (10), and the algorithm stops. We will select the node with the highest true score and insert the rest back to the tree with their new score. *P-tree* may evaluate more vertices than CELF, but the extra work can be bounded (Thm. 4.2).

the last batch is batch j with 2^j vertices. This indicates that all $2^j - 1$ vertices in the previous $j-1$ batches are before v_l. Therefore $2^j - 1 < l = |F_i|$, and $|F_i'| = 2^{j+1} - 1$, which proves $|F_i'| \le 2|F_i|$.

We now consider the general case. We first focus on a specific seed selection round i. Due to different sets of vertices evaluated in each round, at the beginning of round i, CELF and *P-tree* may not see exactly the same stale scores. We denote the stale score *at the beginning of round i* in CELF as $\bar{\Delta}_{CELF}[\cdot]$ and that for *P-tree* as $\bar{\Delta}_{BST}[\cdot]$. We reorder vertices by the decreasing order of $\bar{\Delta}_{CELF}[\cdot]$ as $v_1, v_2, \ldots, v_n$, and similarly for $\bar{\Delta}_{BST}[\cdot]$ as $u_1, u_2, \ldots, u_n$. Denote Δ^* as the highest true score in round i. Let v_{x_i} be the last vertex in $v_{1..n}$ such that $\bar{\Delta}_{CELF}[v_{x_i}] \ge \Delta^*$, and u_{y_i} the last vertex in $u_{1..n}$ such that $\bar{\Delta}_{BST}[u_{y_i}] \ge \Delta^*$. Namely, x_i is the rank of Δ^* in $v_{1..n}$, and y_i is the rank of Δ^* in $u_{1..n}$. By definition, F_i is exactly the first x_i vertices in sequence v, and thus $x_i = |F_i|$; F_i' contains all vertices smaller than u_{y_i} and possibly some more in the same batch with u_{y_i}, so $|F_i'| \le 2y_i$. In the simple case discussed above, where we assume $\bar{\Delta}_{CELF}[\cdot] = \bar{\Delta}_{BST}[\cdot]$, we always have $x_i = y_i$.

We will use amortized analysis to show that $\sum y_i \le \sum x_i$, which further indicates $\sum |F_i'| \le 2 \cdot \sum |F_i|$. We can partition $u_{1..y_i}$ into two categories: U_1 as the intersection of $u_{1..y_i}$ and $v_{1..x_i}$, and U_2 as the rest. Note that for $t \in U_2$, $\bar{\Delta}_{BST}[t] \ge \Delta^* > \bar{\Delta}_{CELF}[t]$. This happens iff. t is evaluated by CELF in a previous round but not by *P-tree*. Namely, there exists a round j, s.t. $t \in F_j$ but $t \notin \bigcup_{x=j}^{i-1} F_x'$. Let $v^*_{1..n}$ and $u^*_{1..n}$ be the u and v sequences in round j, respectively. Note that $F_j = v^*_{1..x_j}$, and $u^*_{1..y_j} \subseteq F_j'$. Since $t \in F_j$, then vertex t was counted in x_j. Since $t \notin F_j'$, t is not counted in any y_j. Therefore, we can save a token for such t when it is evaluated by CELF in round j but not by *P-tree*, such that when later t is counted in $u_{1..y_i}$ in round i, we will use the token to count t for free. Note that $t \notin (F_j' \cup F_{j+1}' \cup \ldots \cup F_{i-1}')$, for the same reason, t is not counted in any $y_{j'}$ for $j \le j' < i$, so the token must still be available in round i. In summary, all vertices in U_1 are counted in x_i, and all vertices in U_2 can be counted by the saved tokens (charged to some previous x_j). Therefore, using amortized analysis, we have $\sum y_i \le \sum x_i$.

Recall that $x_i = |F_i|$, and $|F_i'| \le 2y_i$. Since $\sum y_i \le \sum x_i$, we proved that $\sum |F_i'| \le \sum 2y_i \le 2 \sum x_i = 2 \sum |F_i|$. □

Theorem 4.3 (*P-tree* Cost Bound). *Given the same sketches $\Phi_{1..R}$, our seed selection based on P-tree will select the same seed set as CELF with $O(n \log n + W_{CELF})$ work and $\tilde{O}(kD_\Delta)$ span, where k is the number of seeds, W_{CELF} is the work (time complexity) by CELF, and D_Δ is the span to evaluate one vertex.*

Thm. 4.3 can be proved by Thm. 4.2 and the cost bound of *P-trees*. For the page limit, we present the proof in the full version.

4.2 Parallel Priority Queue: *Win-Tree*

While *P-tree* provides theoretical efficiency for seed selection, it maintains the total order of all vertices, which is not needed in priority queues and may cause performance overhead. Also, *P-tree* explicitly maintains the parent-child pointers, which causes additional space usage. We now propose a more practical data structure based on a winning tree that overcomes these two challenges, although it does not have the same bounds as in Thm. 4.3.

A classic *winning tree* (aka. tournament tree) is a complete binary tree with n leaf nodes and $n-1$ interior nodes. The data are stored in the leaves. Each interior node records the larger key of its two children. Since a winning tree is a complete binary tree, it can be stored *implicitly* in an array $T[1..2n-1]$, which consumes smaller space. In our case, each tree node stores a vertex id (noted as $t.id$). The key of a node t is the (stale) score of this vertex, i.e., $\bar{\Delta}[t.id]$. Each interior node stores the id of its children with a larger score. As a result, the vertex at the root has the highest (stale) score.

To support CELF efficiently, we use the internal nodes of *Win-Tree* to prune the search process. Suppose the best true score evaluated so far is Δ^*, then if we see a subtree root t with a stale score smaller than Δ^*, we can skip the *entire subtree*. This is because all nodes in this subtree must have smaller stale scores than $t.id$, which indicates even smaller true scores. Although this idea is simple, we must also carefully maintain the *Win-Tree* structure, with the newly evaluated true scores. We presented our algorithm in Alg. 5 with an illustration in Fig. 5, and elaborate on more details below.

For NextSeed, we keep a global variable Δ^* as the largest true score obtained so far, initialized to 0. The algorithm calls the FindMax$(t, \ldots)$ routine starting from the root, which explores the subtree rooted at t. We first check if the score at t is stale: if t's id is the same as its parent, the true score has been re-evaluated at its parent (Line 2). Based on the node's status, we discuss three cases. First, if the score is stale and is lower than Δ^*, as discussed above,

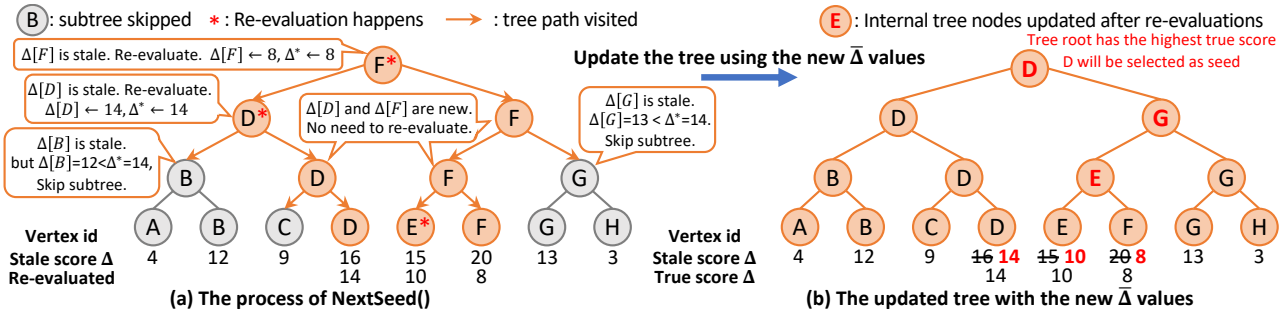

Figure 5: Seed selection based on *Win-Tree*. Each leaf stores a vertex id, and each internal node stores the vertex in its subtree with the highest (stale) score. (a) An example of finding the maximum score. For illustration purposes, we assume the parallel threads work at the same speed and all tree nodes on the same level are processed in parallel (in reality the threads run asynchronously in fork-join parallelism). Therefore, the subtree at G will see $\Delta^* = 14$ updated by D, and this subtree will be skipped. (b) Updating the internal nodes with the new $\bar{\Delta}[\cdot]$ values. Finally, the root of the tree has the highest true score.

Algorithm 5: Seed Selection based on *Win-Tree*

Maintains: Global variable Δ^*: the highest true score evaluated so far
A winning tree T with n leaf nodes each storing a record.
For a tree node $t \in T$, we use the following notations:
$t.id$: the id of the vertex stored in this node
$t.parent$ / $t.left$ / $t.right$: the parent/left child/right child of node t
The *Win-Tree* is a max-priority-queue based on (stale) score $\bar{\Delta}[t.id]$ for each vertex.

1 **Function** FindMax(tree node t, current seed set S, sketches $\Phi_{1..R}$)
2 **if** $t.id = t.parent.id$ **then** $stale \leftarrow false$ *// evaluated by parent*
3 **else** $stale \leftarrow true$
// Skip a subtree if the max is stale and is smaller than the current best score; no re-evaluation needed for the entire subtree
4 **if** $stale = true$ and $\bar{\Delta}[t.id] < \Delta^*$ **then return**
5 **if** $stale = true$ **then** *// Current value is stale, re-evaluation needed*
6 $\bar{\Delta}[t.id] \leftarrow$ MarginalGain$(S, t.id, \Phi_{1..R})$ *// Re-evaluate*
7 WriteMax$(\Delta^*, \bar{\Delta}[t.id])$ *// update the best score so far*
8 **if** t is a leaf **then return**
9 **In Parallel:**
10 FindMax$(t.left, S, \Phi_{1..R})$
11 FindMax$(t.right, S, \Phi_{1..R})$
// compare two branches and reset max
12 **if** $\bar{\Delta}[t.left.id] > \bar{\Delta}[t.right.id]$ **then** $t.id \leftarrow t.left.id$
13 **else** $t.id \leftarrow t.right.id$
14 **Function** NextSeed$(S, \Phi_{1..R})$
15 $\Delta^* \leftarrow 0$
16 FindMax$(T.root, S, \Phi_{1..R})$
17 **return** $T.root.id$

we can skip the entire subtree and terminate the function (Line 4). Second, if the score is stale but is higher than Δ^*, we have to re-evaluate the vertex $t.id$, since it may be a candidate for the seed (Line 6). We then use the atomic operation WriteMax to update Δ^* by this true score if it is better. The third case is when the score is not stale. Although no evaluation is needed on $t.id$, we should still explore the subtrees, since with the newly evaluated score, the subtree structure (i.e., the ids of the internal nodes) may change. Hence, in cases 2 and 3, we recursively explore the two subtrees in parallel (Lines 9 to 11). After the recursive calls, we set the vertex id as one of its two children with a higher score (Lines 12 to 13).

Theorem 4.4 (*Win-Tree* Correctness). *Alg. 5 always selects the next seed with the largest marginal gain.*

We present the proof in the full version of this paper. Unlike *P-trees*, we cannot prove strong bounds for the number of re-evaluations in *Win-Tree*—since the parallel threads are highly asynchronous, the progress of updating Δ^* and pruning the search cannot be guaranteed. However, we expect *Win-Tree* to be more practical than *P-tree* for a few reasons. First, *Win-Tree* is a complete binary and can be maintained in an array, which requires smaller space (no need to store metadata such as pointers in *P-trees*). Second, the *P-tree* algorithm requires $O(\log n)$ batches and synchronizing all threads between batches. Such synchronization may result in scheduling overhead, while the *Win-Tree* algorithm is highly asynchronous. Most importantly, *Win-Tree* does not maintain the total order, and the construction time is $O(n)$ instead of $O(n \log n)$. In Sec. 5.3, we experimentally verify that although *Win-Tree* incurs more re-evaluations than *P-trees*, it is faster in most tests.

5 EXPERIMENTS

Setup. We implemented *PaC-IM* in C++. We run our experiments on a 96-core (192 hyperthreads) machine with four Intel Xeon Gold 6252 CPUs and 1.5 TB of main memory. We use `numactl -i all` in experiments with more than one thread to spread the memory pages across CPUs in a round-robin fashion. We run each test four times and report the average of the last three runs.

We tested 17 graphs with information shown in Tab. 3. We include real-world graphs with a wide range of sizes and distributions, including **five billion-scale or larger graphs**. In addition to the commonly used benchmarks of social networks, we also include web graphs, road networks, and k-NN graphs (each vertex is a multi-dimensional data point connecting to its k-nearest neighbors [81]). Solving IM on such graphs simulates the influence spread between websites (web graphs), geologically connected objects (road networks), and geometrically close objects (k-NN graphs). Based on graph patterns, we call social and web graphs ***scale-free*** graphs, and the rest ***sparse*** graphs. We symmetrize the directed graphs to make them undirected. *PaC-IM* (with compression) is the only tested system that can process the largest graph ClueWeb [54] with 978M vertices and 74B edges. Even with 1.5TB memory, *PaC-IM* can process Clueweb only when $\alpha \leq 0.25$ (4× or more compression ratio in sketches), which shows the necessity of compression.

Table 3: Graph Information. Influence: the maximum influence spread of 100 seeds selected by *PaC-IM* ($R = 256$), *InfuserMG* ($R = 256$), and *Ripples* ($\epsilon = 0.5$), rounded to integers. Most influence values are evaluated by 20000 simulations. The underline numbers on large graphs use 2000 simulations. "-": unable to evaluate within 50 hours using 2000 simulations.

		$\|V\|$	$\|E\|$	**Influence**	**Notes**
Social	**EP**	0.08M	0.81M	5332	Epinions1 [64]
	SLDT	0.08M	0.94M	6342	Slashdot [50]
	DBLP	0.32M	2.10M	1057	DBLP [84]
	YT	1.14M	5.98M	29614	com-Youtube [84]
	OK	3.07M	234M	1460000	com-orkut [84]
	LJ	4.85M	85.7M	376701	soc-LiveJournal1 [5]
	TW	41.7M	2.40B	11776629	Twitter [47]
	FT	65.6M	3.61B	19198744	Friendster [84]
Web	**SD**	89.2M	3.88B	15559737	sd_arc [54]
	CW	978M	74.7B	-	ClueWeb [54]
Road	**GER**	12.3M	32.3M	384	Germany [1]
	USA	23.9M	57.7M	370	RoadUSA [1]
k-NN	**HT5**	2.05M	13.0M	1018	HT [28, 81], k=5
	HH5	2.05M	13.0M	2827	Household [28, 81], k=5
	CH5	4.21M	29.7M	355065	CHEM [30, 81], k=5
	GL5	24.9M	157M	11632	GeoLife [81, 86], k=5
	COS5	321M	1.96B	4753	Cosmo50 [48, 81], k=5

We select $k = 100$ seeds in all tests. We use the IC model with constant propagation probability p in the same graph. For scale-free networks, we use $p = 0.02$, similar to previous work [20, 22, 31, 43]. For sparse graphs, we set $p = 0.2$ since the average vertex degrees are mostly within 5. We also tested two other edge probability distributions similar to previous papers [31, 56]. Among different distributions, we observed similar relative performance among the tested algorithms. Thus, we provide the result using fixed $p = 0.02$ or 0.2 here. Full results on other distributions can be found in the full version [77]. When comparing the ***average*** numbers across multiple graphs, we use the ***geometric mean***.

Software Libraries. We use ParlayLib [11] for fork-join parallelism and some parallel primitives (e.g., sorting). We use the *P-tree* implementation from the PAM library [71, 73], and the UniteRemCAS implementation from ConnectIt [26] for parallel connectivity.

Tested Algorithms. We tested both *P-tree* and *Win-Tree* for seed selection. In most cases, *Win-Tree* is more efficient in both time and space, so use *Win-Tree* as the default option in *PaC-IM*. We present more results comparing the two options in Sec. 5.3.

We compare with three existing parallel IM systems: *InfuserMG* [31], *No-Singles* [63] and *Ripples* [55, 56], and call them ***baselines***. As introduced in Sec. 2, *InfuserMG* uses a similar sketch-based approach as *PaC-IM* but does not support compression or parallel CELF. *No-Singles* and *Ripples* both use Reverse Influence Sampling [16]. In our tests, *Ripples* is always better than *No-Singles* in time and space, so we only report the results of *Ripples*. We have also tested some sequential algorithms, such as PMC [59] and IMM [74], but their running times are not competitive to the parallel implementations. We observe that *InfuserMG* and *Ripples* have *scalability issues* when the number of threads increases (see examples in Fig. 6). Hence, we report *their shortest time among all the tested numbers of threads*.

Each algorithm has a parameter that controls the solution quality: R for *InfuserMG* and *PaC-IM* and $\epsilon \in (0, 0.5]$ for *Ripples*. The solution quality increases with larger R or smaller ϵ. When comparing running time, we guarantee that *PaC-IM always gives a better solution* than the baselines. For *InfuserMG* and *PaC-IM*, we set the number of sketches $R = 256$. *PaC-IM* with $R = 256$ is on average more than 99% of the quality when using $R = 2^{15}$, which is consistent with the observation in the *StaticGreedy* paper [21] (see our full version for quality analysis). For *Ripples*, smaller ϵ means better accuracy but more time. We tested ϵ in $[0.13, 0.5]$ as in their paper. *PaC-IM* under $R = 256$ yields about the same solution quality as $\epsilon = 0.13$ (the best tested setting in their paper). When reporting time, we use $\epsilon = 0.5$, which gives the fastest running time, and the quality is still reasonably high (at least 93% of our best influence).

We observe that on sparse graphs, the influence spread of *InfuserMG* is only 38–92% of the best achieved by *PaC-IM* and *Ripples*. Although theoretically, *PaC-IM* and *InfuserMG* should give the same output, *InfuserMG* uses many optimizations that sacrifice solution quality. We tried to increase R and various other attempts in *InfuserMG*, but they did not improve the solution quality. Therefore, we keep the same value $R = 256$ for *PaC-IM* and *InfuserMG*.

5.1 Overall Time and Space

Tab. 4 shows the running time, memory usage, and normalized influence spread of all systems. Ours_1 and $\text{Ours}_{0.1}$ are *PaC-IM* with $\alpha = 1$ (no compression) and $\alpha = 0.1$ (10× sketch compression), respectively. To illustrate the relative performance, we present a heatmap in Fig. 1, where all the numbers (time and space) are normalized to Ours_1. $\text{Ours}_{0.1}$ is the *only algorithm that can process the largest graph CW* [54]. With similar quality, Ours_1 is *faster than all baselines on all graphs*, and $\text{Ours}_{0.1}$ is just slower than *InfuserMG* on the two smallest graphs. $\text{Ours}_{0.1}$ has the smallest space usage on *all* graphs. The advantage of *PaC-IM* is more significant on larger graphs, both in time and space.

Running Time. *PaC-IM* is significantly faster than the baselines on almost all graphs. On average, Ours_1 is 5.7× faster than *InfuserMG* and 18× faster than *Ripples*. $\text{Ours}_{0.1}$ is slightly slower than Ours_1, but is still 3.2× faster than *InfuserMG* and 10× faster than *Ripples*.

In general, the compression in *PaC-IM* saves space by trading off more time. When $\alpha = 1$, the CC sizes for all vertices are stored in the sketches, and a re-evaluation only needs a constant time per sketch. When $\alpha = 0.1$, each query involves a search to either find a center, or visit all connected vertices, which roughly costs $O(1/\alpha)$ on each sketch. Indeed, on all scale-free graphs, $\text{Ours}_{0.1}$ takes a longer time than Ours_1. Interestingly, on most sparse graphs, $\text{Ours}_{0.1}$ can be faster than Ours_1. This is because seed selection only takes a small fraction of the total running time (except for CH5), so the slow-down in seed selection is negligible for the overall performance. Meanwhile, avoiding storing $O(Rn)$ connectivity sizes reduces memory footprint and makes the sketching step slightly faster, which overall speeds up the running time.

Memory Usage. We show space usage of *PaC-IM* and baselines in Fig. 1 and 7 and Tab. 4. We also show the size of representing the graph in standard Compressed Sparse Row (CSR) format as a reference in Tab. 4 and Fig. 7, which roughly indicates the space to store the input graph. CSR uses 8 bytes for each vertex and each edge. Using $\alpha = 0.1$, *PaC-IM* uses the least memory on all

Table 4: Running time, memory usage, and influence spread (normalized to the maximum) of all tested systems on a machine with 96 cores (192 hyperthreads). Relative influence is the influence spread normalized to the maximum influence spread among Ours ($R = 256$), *InfuserMG* ($R = 256$), and *Ripples* ($\epsilon = 0.5$). "-": out of memory (1.5 TB) or time limit (3 hours). Ours_1 is our implementation with *Win-Tree* without compression. $\text{Ours}_{0.1}$ is our implementation with $\alpha = 0.1$ (10× compression for sketches). *InfuserMG* [31] and *Ripples* [55, 56] are baselines. We report the *best time* of *InfuserMG* and *Ripples* by varied core counts (the scalability issue of *InfuserMG* and *Ripples* are shown in Fig. 6). CSR is the memory used to store the graph in CSR format (see more in Sec. 5.1). The bold numbers are the fastest time/smallest memory among all implementations on each graph. The underlined numbers in memory usage are the smallest memory among systems that do not use compression (Ours_1, *Ripples* and *InfuserMG*). A heatmap of the full result is in Fig. 1.

		Relative Influence			Total Running Time (second)				Memory Usage (GB)				
		Ours	*InfuserMG*	*Ripples*	Ours_1	$\text{Ours}_{0.1}$	*InfuserMG*	*Ripples*	CSR	Ours_1	$\text{Ours}_{0.1}$	*InfuserMG*	*Ripples*
Social	**EP**	100%	99.9%	98.9%	**0.29**	0.49	0.38	3.70	0.01	0.14	**0.06**	0.17	0.21
	SLDT	100%	99.8%	99.1%	**0.30**	0.53	0.48	6.88	0.01	0.15	**0.05**	0.17	0.24
	DBLP	100%	99.0%	98.3%	**0.35**	0.37	0.86	2.71	0.02	0.48	**0.10**	0.67	0.29
	YT	100%	99.9%	98.2%	**1.22**	2.44	6.20	26.6	0.05	1.63	**0.28**	2.36	1.50
	OK	100%	100%	99.8%	**8.79**	39.6	80.3	325	1.77	6.13	**2.44**	8.07	78.3
	LJ	100%	99.9%	99.4%	**6.00**	20.5	61.3	130	0.68	5.98	**1.63**	10.5	20.4
	TW	100%	100%	-	**93.4**	378	639	5863	18.2	63.0	**25.9**	103	718
	FT	100%	100%	-	**128**	609	1973	-	27.4	97.9	**39.9**	161	-
Web	**SD**	100%	97.1%	-	**150**	627	1684	-	29.6	125	**44.8**	211	-
	CW	100%	-	-	-	9776	-	-	564	-	738	-	-
Road	**GER**	100%	70.4%	94.4%	9.35	**8.53**	26.7	392	0.33	13.3	**2.58**	25.1	22.8
	USA	100%	74.7%	92.8%	14.6	**13.7**	53.1	8534	0.61	25.8	**5.05**	49.0	46.1
k-NN	**HT5**	100%	85.7%	94.8%	0.72	**0.68**	2.73	9.26	0.11	1.37	**0.24**	1.93	0.79
	HH5	100%	79.5%	97.6%	2.28	**2.11**	8.26	14.4	0.11	3.00	**0.51**	4.26	1.14
	CH5	100%	92.3%	97.8%	**3.52**	5.32	124	10.2	0.25	4.87	**1.05**	8.77	1.64
	GL5	100%	76.9%	98.9%	19.5	**17.9**	116	232	1.36	27.6	**6.02**	51.6	8.18
	COS5	100%	37.7%	-	348	**284**	2319	-	17.0	355	**66.1**	666	-

graphs. Even without compression, our algorithm uses less memory than the baselines on 10 out of 16 graphs. Note that although the compression rate for sketches is 10× in $\text{Ours}_{0.1}$, the total space also includes the input graph and the data structure for seed selection. Therefore, we cannot directly achieve a 10× improvement in space. In most cases, the total memory usage is about 5× smaller.

Summary. Overall, *PaC-IM* has better performance than the baselines in both time and space. We note that the space usage of *Ripples* can be (up to 3.4×) better than Ours_1 on certain graphs (but still worse than $\text{Ours}_{0.1}$), but in these cases the running time is also much longer (by 2× to 583×). On scale-free graphs, $\text{Ours}_{0.1}$ is 2.5× slower than Ours_1 on average, but uses 3× less space. On sparse graphs, $\text{Ours}_{0.1}$ is almost always better in both time and space.

5.2 Scalability

We study the scalability of all systems. We present the performance on six representative graphs in Fig. 6 with varying core counts P. We separate the time for sketch construction (the "sketch time") and seed selection (the "selection time") to study the two components independently. For *PaC-IM*, both sketch and selection time decrease with more cores and achieve almost linear speedup. In contrast, *InfuserMG* and *Ripples* get the best speedup with 8–16 cores on many graphs, and perform worse with more cores. For both *InfuserMG* and *Ripples*, the sketch time scales better than the selection time. This is because both algorithms use well-parallelized algorithms to construct sketches. For example, *InfuserMG* uses a standard coloring [60, 70] idea for parallel connectivity. However, in seed selection, the baseline algorithms can have longer running times with more threads used. This is because they only use parallelism within one evaluation, and perform all evaluations one by one. For both baselines, a single evaluation does not cause much computation. In this case, the overhead of scheduling the parallel threads can be more expensive than the computation (and can even dominate the cost), which increases with the number of threads.

The scalability curves indicate a major performance gain of *PaC-IM* over baselines is from ***better scalability***. On SLDT, LJ, and SD, although the sequential running time of *InfuserMG* is better than *PaC-IM*, *PaC-IM* achieves better performance when more than 8 cores are used. The advantage is more significant with more cores.

The scalability curves also *indicate the necessity of our parallel CELF*. For $\text{Ours}_{0.1}$, the sequential selection time takes a large portion of the total running time, so parallelizing seed selection is crucial for improving the overall performance. Even for Ours_1, the selection time on 1 core is still much longer than the total parallel running time. This means that *if* this step was not well-parallelized, it would become the performance bottleneck when P is large. Using our new techniques from Sec. 3 and 4, both steps scale well.

5.3 Analysis of the Proposed Techniques

Compression. We evaluate the time and space usage of different compression ratios by controlling the parameter α, and present the results in Fig. 7. The gray dashed line represents the CSR size of each graph, which is the space to store the input. Compression always reduces memory usage, but may affect running time differently. Memory usage always decreases with the value α decreases. As mentioned previously, the actual compression rate can be lower than $1/\alpha$, since our compression only controls the memory for storing sketches, and there are other space usage in the algorithm.

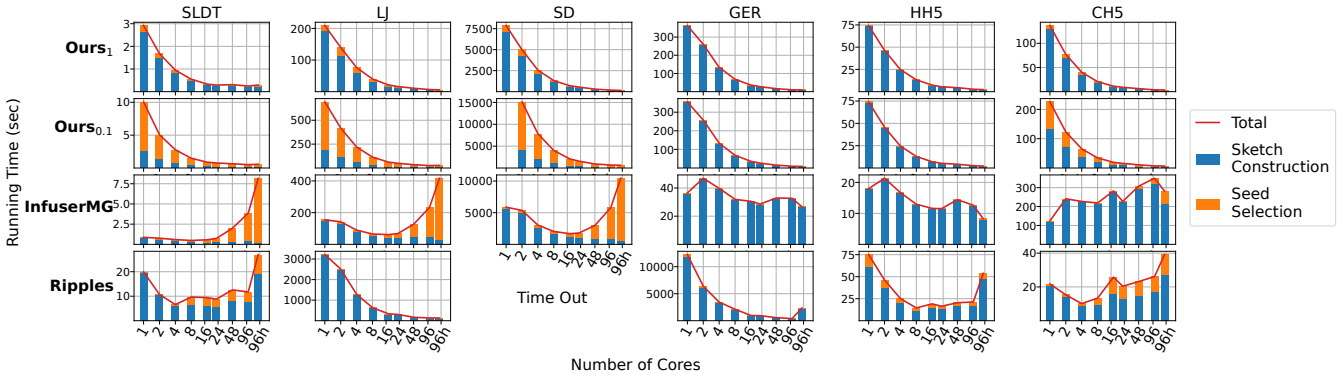

Figure 6: Running time using different core counts for different IM algorithms. In each plot, the x-axis shows core counts (96h means 96 cores with hyperthreading) and the y-axis is running time in seconds.

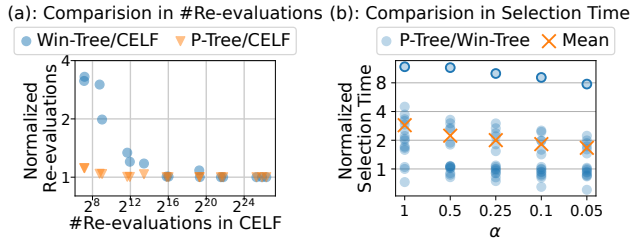

Figure 8: Compare *P-tree* and *Win-Tree* in the number of re-evaluations and selection time. (a): Let $\#CELF$, $\#P\text{-}tree$ and $\#Win\text{-}Tree$ represent the total number of re-evaluations needed in the corresponding method. Each point represents a tested graph. Each blue circle is a data point $(\#CELF, \#Win\text{-}Tree/\#CELF)$. Each orange triangle is a data point $(\#CELF, \#P\text{-}tree/\#CELF)$. **(b)**: Each point represents a tested graph under different values of α. The y-axis is *P-tree* selection time/*Win-Tree* selection time. The orange cross is the average value under each α.

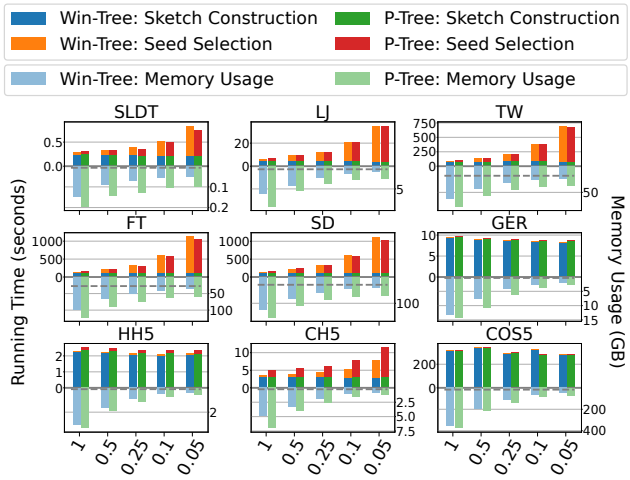

Figure 7: Running time and memory with different values of α. The x-axis represents the compression rate. The growing up y-axis represents the running time (in seconds) of the sketching and selecting process. The growing down y-axis represents the total memory usage (in GB). The gray horizontal line represents the CSR size of each graph, which is the basic memory we need to load the graph.

Roughly speaking, using $\alpha = 0.05$ can save space on graphs by up to 8× and shrink the space very close to the input graph size.

Compression affects the running time differently for sketch construction and seed selection. Smaller α indicates longer running time in seed selection, but may improve the sketch time slightly due to the reduced memory footprint. The effect of compression on total running time depends on the ratio of sketching and selection time, where scale-free and sparse graphs exhibit different patterns. Scale-free graphs usually have one or several large CCs in the sampled graphs since they are dense, while sparse graphs usually have many small CCs. This can also be seen by the overall influence in Tab. 3: even though we use a smaller p on scale-free networks, the total influence is still much larger than the sparse graphs. On scale-free graphs, due to large CCs on sampled graphs, selecting any seed in a large CC may significantly lower the score of other vertices, leading to much more total evaluations than sparse graphs. As a result, a smaller α causes a clear time increase for seed selection on scale-free networks because each evaluation becomes slower while only having a small impact on most sparse graphs.

We provide the total number of evaluations on each graph as a reference in the full version [77].

***P-tree* vs. *Win-Tree*.** We now study both data structures for seed selection. Our goal is to achieve high parallelism without causing much overhead in work. *P-tree* has the theoretical guarantee that the total number of evaluations is no more than twice that in CELF. To evaluate the work overhead, we compare the number of evaluations for both data structures to CELF in Fig. 8(a). For each graph, we count the number of evaluations done by CELF as x and that by *P-tree* and *Win-Tree* as y_1 and y_2, respectively. We plot all $(x, y_1/x)$ as blue dots and all $(x, y_2/x)$ as orange triangles. The number of evaluations done by *P-tree*s is very close to that by CELF (1.03× on average), while *Win-Tree*s require slightly more (1.7× on average). The result is consistent with our theoretical analysis.

As mentioned in Sec. 4.2, we expect *Win-Tree*s to perform better in practice due to various reasons. To study this, we plotted Fig. 8(b) as a comparison for selection time between *P-tree* and *Win-Tree* under different compression ratios. Each data point represents a graph, and the value is the ratio of selection time between *P-tree* and *Win-Tree*. The average ratios are always larger than 1, indicating

better performance for *Win-Tree*, but the advantage decreases as α becomes smaller (higher compression). This is because when α is large, the evaluation is fast, and the major time is on the tree operations, where *Win-Tree*s is more advantageous due to the reasons mentioned in Sec. 4.2. With higher compression (small α), the evaluation becomes more expensive. Since the *P-tree* evaluates fewer vertices, the overall selection time is more likely to be better. To further understand this, we focus on the five topmost points (circled) in Fig. 8(b): those data points are from COS5, where *P-tree* is 8–12× slower than *Win-Tree*. For COS5, the number of vertices n is large, but only hundreds of vertices are evaluated in total. Thus, the seed selection time is dominated by constructing the data structures, i.e., $O(n \log n)$ work for *P-tree* to maintain total ordering, and $O(n)$ for *Win-Tree* that is a lot faster. Since the number of evaluations is small, most of the vertices in the tree are never touched, and thus, maintaining their order wastes the work.

In Fig. 7, we also compare the time and space between *P-tree* and *Win-Tree*. Similar to the discussions above, when α is small, *Win-Tree* is almost always faster than *P-tree*, but *P-tree* may perform better when α is large. However, the advantage is quite small since the number of evaluations of *Win-Tree* is still close to *P-tree* (see Fig. 8(a)). *Win-Tree* also uses smaller memory than *P-tree*. As discussed in Sec. 4.2, this is because *P-tree* needs to explicitly maintain tree pointers and balancing criteria, while each *Win-Tree* node only needs to store the vertex id ($2n$ integers in total).

Based on these observations, *PaC-IM* always uses *Win-Tree* by default, but also provides the interface for users to choose *P-tree*s.

6 RELATED WORK

Influence Maximization (IM). IM has been widely studied for decades with a list of excellent surveys [4, 6, 51, 62, 88] that review the applications and papers on this topic. IM is also of high relevance of the data management community, and many excellent papers published recently regarding benchmarking [2, 58], new algorithms [39], new propagation models and applications [8, 40, 76, 83], and interdisciplinary extensions [85, 89].

A survey paper [51] roughly categorizes IM algorithms into three methodologies: simulation-based, proxy-based, and sketch-based. Among them, (Monte Carlo) simulation-based approaches (e.g., [33, 42, 49, 80, 87, 90]) are the most general and apply to most settings (i.e., graph types and diffusion models); however, they do not take advantage of the specific settings, so generally, their performance is limited. Proxy-based approaches (e.g., [19, 41, 43, 44, 53, 61]) use simpler algorithms/problems (e.g., PageRank or shortest-paths) to solve IM. While they can be fast in practice, their solution quality has no theoretical guarantees. Sketch-based solutions, as mentioned in Sec. 1 and 2, generally have good performance and theoretical guarantees, but only apply to specific diffusion models. Our sketch compression focuses on the IC model. We note that the parallel data structures in Sec. 4 are general to submodular diffusion models such as linear threshold [42], and more [76, 85].

Sketch-based algorithms can further be categorized into forward (influence) sketches and reverse (reachable) sketches [51]. As the names suggest, forward sketches record the influence that each vertex can propagate to in sampled graphs. Most algorithms [20–22, 31, 42, 59] mentioned in this paper, including *PaC-IM*, use forward sketches. Reverse sketches find a sample of vertices T and keep the sets of vertices that can reach them. Many IM algorithms (e.g., [16, 74, 75, 79]), including *No-Singles* [63] and *Ripples* [55, 56] that we compared to, use reverse sketches. These algorithms can trade off (lower) solution quality for (better) performance/space, by adjusting the size of T. This paper focuses on forward sketches because our compression technique is designed for them. However, the parallel data structures we proposed for seed selection, *P-tree* and *Win-Tree*, are independent of forward sketches. Applying *P-tree* and *Win-Tree* to reverse-reachable sketches can be interesting for future work.

Space-Efficient Connectivity. We are aware of a few algorithms that can compute graph connectivity using $o(n)$ space [7, 17, 29, 45], which share a similar motivation with *PaC-IM*. However, *PaC-IM* does not require computing connectivity in $o(n)$ space, but only requires storing it in $O(n)$ space. Hence, the goal here is essentially different, although these approaches are inspiring.

7 CONCLUSION AND DISCUSSIONS

This paper addresses the scalability issues in existing IM systems by novel techniques including sketch compression and parallel CELF. Our sketch compression (Sec. 3) applies to the IC model and undirected graphs, and avoids the $O(Rn)$ space usage in SOTA systems, which allows *PaC-IM* to run on much larger graphs without sacrificing much performance. To the best of our knowledge, our new data structures (Sec. 4) are the first parallel version of CELF seed selection, which is general to any submodular diffusion models.

In addition to new algorithms, our techniques are carefully analyzed (Thm. 3.1 and 4.3) and have good theoretical guarantees regarding work, parallelism, and space. These analyses not only lead to good practical performance but also help to understand how the techniques interplay. The techniques are also experimentally verified in Fig. 6 to 8. The theory and our careful implementation also lead to stable speedup with increasing core counts (see Fig. 6).

Limitations, Generalizations, and Future Work. Our paper discussed two techniques: sketch compression and parallel CELF. Sketch compression uses the idea of memoization. As in previous work [20, 31], this approach only applies to IC models on undirected graphs, so that each edge can be sampled in advance regardless of the actual influence propagation direction. One future direction is to study similar approaches for sketch compression on directed graphs, but it may require a different technique, such as a compressed representation for strongly connected components.

For parallel CELF, we believe that our approaches based on *P-tree* and *Win-Tree* can be generalized to other settings. Within the scope of IM, this technique can also be combined with reverse (reachable) sketches, as long as the diffusion models are submodular. Moreover, CELF is a general greedy approach to accelerate optimization problems with submodular objective functions [46, 49, 57]. Therefore, our approaches can potentially provide parallelism for these problems. We leave these extensions as future work.

One limitation of *Win-Tree* is that it does not have strong bounds as *P-tree*s. An interesting future work is to derive a strong bound regarding the number of evaluations. In the worst case, *Win-Tree* may require $O(n)$ evaluations due to the asynchrony of the threads. However, such a bad case is very unlikely in practice, and *Win-Tree* has demonstrated good performance in our experiments. Giving a tighter bound under some practical assumptions may be interesting.

REFERENCES

[1] 2010. OpenStreetMap © OpenStreetMap contributors. https://www.openstreetmap.org/.

[2] Akhil Arora, Sainyam Galhotra, and Sayan Ranu. 2017. Debunking the myths of influence maximization: An in-depth benchmarking study. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 651–666.

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (01 Apr 2001).

[4] M. Athanassoulis, A. Ailamaki, S. Chen, P. B. Gibbons, and R. Stoica. 2010. Flash in a DBMS: Where and How? *IEEE Data Engineering Bulletin* 33, 4 (2010), 28–34. Special issue on data management using modern storage hardware.

[5] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 44–54.

[6] Suman Banerjee, Mamata Jenamani, and Dilip Kumar Pratihar. 2020. A survey on influence maximization in a social network. *Knowledge and Information Systems* 62, 9 (2020), 3417–3455.

[7] Naama Ben-David, Guy E. Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. Implicit Decomposition for Write-Efficient Connectivity Algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[8] Song Bian, Qintian Guo, Sibo Wang, and Jeffrey Xu Yu. 2020. Efficient algorithms for budgeted influence maximization on massive social networks. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1498–1510.

[9] Timo Bingmann, Thomas Keh, and Peter Sanders. 2015. A bulk-parallel priority queue in external memory with STXXL. In *International Symposium on Experimental Algorithms (SEA)*. Springer, 28–40.

[10] Guy Blelloch, Daniel Ferizovic, and Yihan Sun. 2022. Joinable Parallel Balanced Binary Trees. *ACM Transactions on Parallel Computing (TOPC)* 9, 2 (2022), 1–41.

[11] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.

[12] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[13] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102.

[14] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2020. Parallelism in Randomized Incremental Algorithms. *J. ACM* 67, 5 (2020), 1–27.

[15] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (1999), 720–748.

[16] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Brendan Lucier. 2014. Maximizing social influence in nearly optimal time. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 946–957.

[17] Andrei Z Broder, Anna R Karlin, Prabhakar Raghavan, and Eli Upfal. 1994. Trading space for time in undirected s-t connectivity. *SIAM J. Comput.* 23, 2 (1994), 324–334.

[18] Wei Chen, Wei Lu, and Ning Zhang. 2012. Time-critical influence maximization in social networks with time-delayed diffusion process. In *AAAI Conference on Artificial Intelligence*, Vol. 26. 591–598.

[19] Wei Chen, Chi Wang, and Yajun Wang. 2010. 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*. 1029–1038.

[20] Wei Chen, Yajun Wang, and Siyu Yang. 2009. Efficient influence maximization in social networks. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 199–208.

[21] Suqi Cheng, Huawei Shen, Junming Huang, Guoqing Zhang, and Xueqi Cheng. 2013. Staticgreedy: solving the scalability-accuracy dilemma in influence maximization. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 509–518.

[22] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F Werneck. 2014. Sketch-based influence maximization and computation: Scaling up with guarantees. In *Proceedings of the 23rd ACM international conference on conference on information and knowledge management*. 629–638.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd edition)*. MIT Press.

[24] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-trees: Supporting Parallel and Compressed Purely-Functional Collections. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[25] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)* 8, 1 (2021), 1–70.

[26] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. ConnectIt: a framework for static and incremental parallel graph connectivity algorithms. *Proceedings of the VLDB Endowment (PVLDB)* 14, 4 (2020), 653–667.

[27] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 184–197.

[28] Dheeru Dua and Casey Graf. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/.

[29] Jeff Edmonds, Chung Keung Poon, and Dimitris Achlioptas. 1999. Tight lower bounds for st-connectivity on the NNJAG model. *SIAM J. Comput.* 28, 6 (1999), 2257–2284.

[30] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical* 215 (2015), 618–629.

[31] Gökhan Göktürk and Kamer Kaya. 2020. Boosting parallel influence-maximization kernels for undirected networks with fusing and vectorization. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2020), 1001–1013.

[32] Jacob Goldenberg, Barak Libai, and Eitan Muller. 2001. Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Marketing letters* 12 (2001), 211–223.

[33] Amit Goyal, Wei Lu, and Laks VS Lakshmanan. 2011. Celf++ optimizing the greedy algorithm for influence maximization in social networks. In *Proceedings of the 20th international conference companion on World wide web*. 47–48.

[34] Mark Granovetter. 1978. Threshold models of collective behavior. *American journal of sociology* 83, 6 (1978), 1420–1443.

[35] Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. 2023. Parallel Longest Increasing Subsequence and van Emde Boas Trees. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[36] Yan Gu, Zachary Napier, and Yihan Sun. 2022. Analysis of Work-Stealing and Parallel Cache Complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 46–60.

[37] Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. 2022. Parallel Cover Trees and their Applications. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 259–272.

[38] Yan Gu, Omar Obeya, and Julian Shun. 2021. Parallel In-Place Algorithms: Theory and Practice. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 114–128.

[39] Qintian Guo, Sibo Wang, Zhewei Wei, and Ming Chen. 2020. Influence maximization revisited: Efficient reverse reachable set generation with bound tightened. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2167–2181.

[40] Shixun Huang, Wenqing Lin, Zhifeng Bao, and Jiachen Sun. 2022. Influence maximization in real-world closed social networks. *Proceedings of the VLDB Endowment* 16, 2 (2022), 180–192.

[41] Kyomin Jung, Wooram Heo, and Wei Chen. 2012. Irie: Scalable and robust influence maximization in social networks. In *2012 IEEE 12th international conference on data mining*. IEEE, 918–923.

[42] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the spread of influence through a social network. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 137–146.

[43] Jinha Kim, Seung-Keol Kim, and Hwanjo Yu. 2013. Scalable and parallelizable processing of influence maximization for large-scale social networks?. In *2013 IEEE 29th international conference on data engineering (ICDE)*. IEEE, 266–277.

[44] Masahiro Kimura and Kazumi Saito. 2006. Tractable models for information diffusion in social networks. In *Knowledge Discovery in Databases: PKDD 2006: 10th European Conference on Principles and Practice of Knowledge Discovery in Databases Berlin, Germany, September 18-22, 2006 Proceedings 10*. Springer, 259–271.

[45] Adrian Kosowski. 2013. Faster walks in graphs: a $\tilde{O}(n^2)$ time-space trade-off for undirected st connectivity. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1873–1883.

[46] Andreas Krause and Daniel Golovin. 2014. Submodular function maximization. *Tractability* 3 (2014), 71–104.

[47] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *International World Wide Web Conference (WWW)*. 591–600.

[48] YongChul Kwon, Dylan Nunley, Jeffrey P Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. 2010. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *International Conference on Scientific and Statistical Database Management*. Springer, 132–150.

[49] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. 2007. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 420–429.

[50] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[51] Yuchen Li, Ju Fan, Yanhao Wang, and Kian-Lee Tan. 2018. Influence maximization on social graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 30, 10 (2018), 1852–1872.

[52] Bo Liu, Gao Cong, Dong Xu, and Yifeng Zeng. 2012. Time constrained influence maximization in social networks. In *International Symposium on Algorithms and Computation (ISAAC)*. IEEE, 439–448.

[53] Qi Liu, Biao Xiang, Enhong Chen, Hui Xiong, Fangshuang Tang, and Jeffrey Xu Yu. 2014. Influence maximization over large-scale social networks: A bounded linear approach. In *Proceedings of the 23rd ACM international conference on conference on information and knowledge management*. 171–180.

[54] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. 2014. Web Data Commons — Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph.

[55] Marco Minutoli, Maurizio Drocco, Mahantesh Halappanavar, Antonino Tumeo, and Ananth Kalyanaraman. 2020. cuRipples: Influence maximization on multi-GPU systems. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 1–11.

[56] Marco Minutoli, Mahantesh Halappanavar, Ananth Kalyanaraman, Arun Sathanur, Ryan Mcclure, and Jason McDermott. 2019. Fast and scalable implementations of influence maximization algorithms. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–12.

[57] Baharan Mirzasoleiman, Ashwinkumar Badanidiyuru, Amin Karbasi, Jan Vondrák, and Andreas Krause. 2015. Lazier than lazy greedy. In *AAAI Conference on Artificial Intelligence*, Vol. 29.

[58] Naoto Ohsaka. 2020. The solution distribution of influence maximization: A high-level experimental study on three algorithmic approaches. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2151–2166.

[59] Naoto Ohsaka, Takuya Akiba, Yuichi Yoshida, and Ken-ichi Kawarabayashi. 2014. Fast and accurate influence maximization on large networks with pruned monte-carlo simulations. In *AAAI Conference on Artificial Intelligence*.

[60] Simona Mihaela Orzan. 2004. On distributed verification and verified distribution. (2004).

[61] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[62] Sancheng Peng, Yongmei Zhou, Lihong Cao, Shui Yu, Jianwei Niu, and Weijia Jia. 2018. Influence analysis in social networks: A survey. *Journal of Network and Computer Applications* 106 (2018), 17–32.

[63] Diana Popova, Naoto Ohsaka, Ken-ichi Kawarabayashi, and Alex Thomo. 2018. Nosingles: a space-efficient algorithm for influence maximization. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*. 1–12.

[64] Matthew Richardson, Rakesh Agrawal, and Pedro Domingos. 2003. Trust management for the semantic web. In *The Semantic Web-ISWC 2003: Second International Semantic Web Conference, Sanibel Island, FL, USA, October 20-23, 2003. Proceedings 2*. Springer, 351–368.

[65] Manuel Gomez Rodriguez, David Balduzzi, and Bernhard Schölkopf. 2011. Uncovering the temporal dynamics of diffusion networks. (2011), 561–568.

[66] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. [n.d.]. *Sequential and Parallel Algorithms and Data Structures*. Springer.

[67] Thomas C Schelling. 2006. *Micromotives and macrobehavior*. WW Norton & Company.

[68] Zheqi Shen, Zijin Wan, Yan Gu, and Yihan Sun. 2022. Many Sequential Iterative Algorithms Can Be Parallel and (Nearly) Work-efficient. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[69] Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B Gibbons. 2015. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 431–448.

[70] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 550–559.

[71] Yihan Sun and Guy Blelloch. 2019. Implementing Parallel and Concurrent Tree Structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 447–450.

[72] Yihan Sun and Guy E Blelloch. 2019. Parallel Range, Segment and Rectangle Queries with Augmented Maps. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*. 159–173.

[73] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[74] Youze Tang, Yanchen Shi, and Xiaokui Xiao. 2015. Influence maximization in near-linear time: A martingale approach. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 1539–1554.

[75] Youze Tang, Xiaokui Xiao, and Yanchen Shi. 2014. Influence maximization: Near-optimal time complexity meets practical efficiency. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 75–86.

[76] Dimitris Tsaras, George Trimponias, Lefteris Ntaflos, and Dimitris Papadias. 2021. Collective influence maximization for multiple competing products with an awareness-to-influence model. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1124–1136.

[77] Letong Wang, Xiangyun Ding, Yan Gu, and Yihan Sun. 2023. Fast and Space-Efficient Parallel Algorithms for Influence Maximization. arXiv:2311.07554 [cs.DS]

[78] Letong Wang, Xiaojun Dong, Yan Gu, and Yihan Sun. 2023. Parallel Strong Connectivity Based on Faster Reachability. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[79] Xiaoyang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Chen Chen. 2016. Bring order into the samples: A novel scalable method for influence maximization. *IEEE Transactions on Knowledge and Data Engineering* 29, 2 (2016), 243–256.

[80] Yu Wang, Gao Cong, Guojie Song, and Kunqing Xie. 2010. Community-based greedy algorithm for mining top-k influential nodes in mobile social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1039–1048.

[81] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2021. GeoGraph: A Framework for Graph Processing on Geometric Data. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 38–46.

[82] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. 2021. A Parallel Batch-Dynamic Data Structure for the Closest Pair Problem. In *ACM Symposium on Computational Geometry (SoCG)*.

[83] Jiadong Xie. 2022. Hindering Influence Diffusion of Community. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2518–2520.

[84] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[85] Wentao Zhang, Zhi Yang, Yexin Wang, Yu Shen, Yang Li, Liang Wang, and Bin Cui. 2021. GRAIN: improving data efficiency of gra ph neural networks via diversified in fluence maximization. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2473–2482.

[86] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning transportation mode from raw gps data for geographic applications on the web. In *International World Wide Web Conference (WWW)*. 247–256.

[87] Chuan Zhou, Peng Zhang, Wenyu Zang, and Li Guo. 2015. On the upper bounds of spread for greedy algorithms in social network influence maximization. *IEEE Transactions on Knowledge and Data Engineering* 27, 10 (2015), 2770–2783.

[88] Fan Zhou, Xovee Xu, Goce Trajcevski, and Kunpeng Zhang. 2021. A survey of information cascade analysis: Models, predictions, and recent advances. *ACM Computing Surveys (CSUR)* 54, 2 (2021), 1–36.

[89] Yuqing Zhu, Jing Tang, Xueyan Tang, and Lei Chen. 2021. Analysis of influence contribution in social advertising. *Proceedings of the VLDB Endowment* 15, 2 (2021), 348–360.

[90] Honglei Zhuang, Yihan Sun, Jie Tang, Zhang Jialin, and Xiaoming Sun. 2013. Influence Maximization in Dynamic Social Networks. In *IEEE International Conference on Data Mining (ICDM)*.