Efficient GPU-Accelerated Reverse Sampling for Adaptive Minimum Cost Seed Selection

Ben Trovato
Institute for Clarity in Documentation
Dublin, Ireland
trovato@corporation.com

Lars Thørväld
The Thørväld Group
Hekla, Iceland
larst@affiliation.org

Valerie Béranger
Inria Paris-Rocquencourt
Rocquencourt, France
vb@rocquencourt.com

Jörg von Ärbach
University of Tübingen
Tübingen, Germany
jaerbach@uni-tuebingen.edu
myprivate@email.com
second@affiliation.mail

Wang Xiu Ying
Zhe Zuo
East China Normal University
Shanghai, China
firstname.lastname@ecnu.edu.cn

Donald Fauntleroy Duck
Scientific Writing Academy
Duckburg, Calisota
Donald's Second Affiliation
City, country
donald@swa.edu

ABSTRACT

Minimum-cost seed selection (MCSS) is a fundamental problem in social advertising, which aims to influence a target number of users η, by inviting some influential users (*seeds*) to join the campaign at the minimum cost. Different from traditional approaches, existing work selects seeds in an adaptive way, and apply multi-root reverse reachable sets (mRR-sets) for influence estimation. Due to the varying graph structure in adaptive seeding, the mRR-sets need to be regenerated from scratch, incurring substantial computational overhead. Moreover, existing algorithms for AMCSS are all single-threaded, neglecting the prevalence of GPUs, which contain thousands of threads for parallel computing.

In this paper, we propose `MERIT`, a novel framework for solving the AMCSS problem. We introduce a segment-based parallel method for mRR-set generation, a kernel for efficient batch seed selection, and a theoretically grounded, unbiased mRR-set update mechanism that allows reuse across adaptive rounds instead of regenerating them from scratch. To further amplify computational efficiency, we propose several parallel optimization, harnessing the capabilities of exploit GPU resources. Extensive experiments on large-scale social networks, conducted under two widely adopted diffusion models, show that `MERIT` usually achieves an order of magnitude speedup over existing parallel adaptations and consistently delivers seed sets with lower or comparable total cost.

PVLDB Reference Format:
Ben Trovato, Lars Thørväld, Valerie Béranger, Jörg von Ärbach, Wang Xiu Ying, Zhe Zuo, and Donald Fauntleroy Duck. Efficient GPU-Accelerated Reverse Sampling for Adaptive Minimum Cost Seed Selection. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/gongyguo/MERIT.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

As of October 2025, over 5.6 billion users, more than two thirds of the population on Earth, create and consume information on social networks [7], producing a social advertising market valued at over 136 billion dollars [3]. The objectives of advertising (such as revenue attainment, lead generation) often can be converted to the number of influenced users by dividing a conversion rate [32]. Thus, to attain the objectives, advertisers hope to influence a target number of users η, called influence threshold. By exploiting the spontaneous interaction between users, advertisers only need to employ some influential users (called *seeds*) to promote their products. Then, a large influence cascade can be triggered. As one may see, the crux lies in how to select the seeds such that η is reached while the cost to employ the seeds is minimized.

The above problem is formulated as the minimum cost seed selection (MCSS) problem [10, 15, 18]. Early studies [13, 15, 18, 23] only aim to reach η in expectation. Actually, an expectation is an average over all possible samples. Thus, there must be a number of diffusion cases that fail to reach η, and the failure originates from their non-adaptive seed selection mechanism, where all seeds are selected before any actual diffusion starts, leaving the diffusion process totally random and uncontrollable. To address this, Tang et al. [29] study MCSS under an adaptive setting: seeds are selected iteratively in batches, observing actual diffusion after each batch until η users are influenced, and propose ASTI, the first practical adaptive algorithm for AMCSS, which achieves an approximation ratio of $(\ln \eta + 1)^2/[(1-1/e)(1-\epsilon)]$ and estimates influence spread using multi-root reverse reachable sets (mRR-sets).

However, ASTI 's practicality is limited. First, since activated nodes are removed across rounds, previously generated mRR-sets may become invalid due to the inclusion of deleted nodes. As a result, ASTI regenerate mRR-sets from scratch in every round, leading to substantial computational overhead. Additionally, ASTI generates mRR-sets in a single-threaded manner, requiring nearly two days to reach the targeted η on the largest dataset in our experiments. Although we implement ASTI in parallel on the CPU, it still requires more than half a day. In contrast, modern graph algorithms

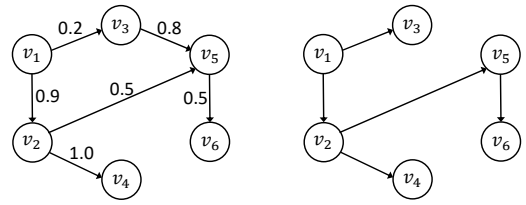
increasingly exploit GPUs to accelerate by leveraging massive parallelism. For example, GPU-based frameworks have been developed for influence maximization [24, 25, 27], PageRank [28], graph pattern mining [19, 26, 38], and clique counting [36, 37]. Failing to leverage such hardware resources leaves substantial parallel performance potential untapped. However, efficiently implementing the AMCSS algorithm on GPUs is challenging. Nodes within each mRR-set are supposed to be stored contiguously for traversal when estimating node influence. Maintaining dynamic arrays on GPUs can be expensive, so all mRR-sets are often stored in a preallocated flat array, which may complicate memory management. Similarly, updating mRR-sets may require preserving previously stored nodes while maintaining a contiguous memory layout, which could introduce additional overhead. In addition, mRR-set construction relies on graph traversal, which may result in load imbalance across threads due to the skewed degree distribution of real-world graphs.

In this paper, we propose an efficient GPU framework for AMCSS that solves the above challenges and guarantees achieving the influence threshold η under heterogeneous user costs. The framework enables GPU-accelerated mRR-set generation, seed selection, and mRR-set updates across adaptive rounds, and we prove that MERIT achieves an approximation ratio linear in $\ln \eta$. Specifically, we develop a parallel mRR-set generation method in which each block independently generates mRR-sets within its assigned segment of memory, and we further optimize node probing in mRR-sets using a warp-centric technique. We also deploy a dedicated kernel to accelerate seed selection within each adaptive round. Additionally, we improve efficiency by updating existing mRR-sets correctly. However, such updates are non-trivial, as mRR-sets must remain consistent with the current residual graph. To resolve this, we theoretically invent an unbiased mRR-set update strategy and leverage the circular segment layout to devise an optimized GPU kernel with lower update overhead and no additional memory allocation. We further improve efficiency by incorporating workload balancing across segments. We evaluate MERIT under two widely adopted diffusion models, comparing it against state-of-the-art methods, including their optimized parallel implementations on both CPUs and GPUs. Experiments show that MERIT usually achieves 10× speedup than parallel adaptations of existing methods, and up to 68.9× on a million-scale Flickr graph. Moreover, MERIT consistently produces seed sets that satisfy the theoretical approximation guarantees and incurs lower total cost compared to existing methods sometimes. Our key contributions are summarized as follows.

- We propose a GPU framework MERIT for AMCSS problem that achieves an improved theoretical approximation ratio and always attains η under heterogeneous user costs.
- We design a new global mRR-set storage on GPU with segmentation and circularity to accelerate mRR-set generation and update.
- We develop an efficient mRR-set update strategy with correctness guarantees and practical optimizations.
- We introduce several parallel optimizations specifically designed to improve the efficiency of MERIT.
- Extensive experiments demonstrate that MERIT achieves an order of magnitude speedup over existing methods, and produces seed sets with comparable or even lower total cost.

2 PRELIMINARIES

The social network is modeled as a directed graph $G = (V, E)$, where V is the set of users with $|V| = n$, and E is the set of edges with $|E| = m$. Each edge $e = (u, v) \in E$ captures the social relationship from u to v. Technically, we say u is an in-neighbor of v and v is an out-neighbor of u. The set of in-neighbors (resp. out-neighbors) of u is denoted as $N_{\text{in}}(u)$ (resp. $N_{\text{out}}(u)$). Each edge e is associated with a propagation probability $p(e) = p(u, v) \in [0, 1]$, and each node $u \in V$ demands a cost $c(u) \in (0, 1]$[1] to become a seed. Figure 1(a) shows a social graph G with six nodes and directed edges. The number on each edge indicates the propagation probability.



(a) Social network G (b) Realization ϕ

Figure 1: A social graph and one possible realization.

2.1 The Influence Diffusion Model

Given a seed set S, we depict the influence diffusion from S by the widely applied *independent cascade* (IC) model and *linear threshold* (LT) model. The diffusion process in both models expands in discrete steps. In step 0, only the seeds in S are activated, while all other nodes remain inactive. For each node u activated at step t, it can try to activate its out-neighbors in step $t + 1$ for only one chance. After that, u cannot probe its neighbors anymore. The main difference between the IC and LT models lies in how an active node u activates its out-neighbor v.

- In the IC model, u successfully activates v with probability $p(u, v)$.
- In the LT model, each node v has a threshold $\lambda_v \in U(0, 1)$, and the probability $p(u, v)$ is treated as a weight. When u probes v, v becomes active if the total weight from its active in-neighbors is no less than λ_v. That is, $\sum_{u \in N_{\text{in}}(v)} \mathbb{I}_{\{u \text{ is active}\}} p(u, v) \geq \lambda_v$.

$\mathbb{I}_{\{\cdot\}}$ denotes the indicator function. This diffusion process continues until no node is further activated in some step. The total number of nodes activated by the seed set S is denoted by $I(S)$.

Note that $I(S)$ is a random variable, due to the randomness in the diffusion process. To derive a sample for any $I(S)$, we introduce the concept of *realization*, which samples all the random variables in the diffusion models.

- For the IC model, a realization ϕ is obtained by marking each edge $e = (u, v)$ as *live* with probability $p(e)$ or *blocked* otherwise. User u can activate v only when $e = (u, v)$ is *live*.
- For the LT model, a realization ϕ is derived by sampling each λ_v from $[0, 1]$ uniformly at random. User v becomes active when $\sum_{u \in N_{\text{in}}(v)} \mathbb{I}_{\{u \text{ is active}\}} p(u, v)$ is no less than the sampled value.

Let Ω be the set of all possible realizations, and Φ be a random realization sampled from the above process. Consider an arbitrary realization $\phi \in \Omega$, which happens with probability p_ϕ. The number

[1] Note that node costs are assumed to be positive, and any positive cost can be normalized to be in $(0, 1]$ by dividing the largest cost.

of nodes activated by the seed set S can be denoted as $I_\phi(S)$, which is a deterministic value since ϕ eliminates all the randomness in the diffusion process. Accordingly, the expected influence spread of S can be expressed as $\mathbb{E}_\Phi[I(S)] = \sum_{\phi\in\Omega} I_\phi(S) \cdot p_\phi$. For instance, Figure 1(b) is one possible realization of the graph G, corresponding to a specific ϕ generated by the stochastic diffusion process. Diffusing v_2 over this realization, the resulting influence spread is $I(\{v_2\}) = 2$, since v_2 activates itself and v_4.

2.2 Problem Definition

We consider the seed selection process under the adaptive setting, where seeds are selected in multiple rounds. In each round i, we first select a batch of seeds S_i from the graph. Next, we trigger a diffusion process from S_i, where some edges and nodes will be probed with their randomness eliminated. As a result, some nodes A_i (including S_i) will be activated. Then, the influence threshold to be reached in the next round $i+1$ becomes $\eta_{i+1} = \eta_i - |A_i|$. Note that nodes in A_i do not need to be considered in subsequent rounds, since they are already activated and are not allowed to probe their neighbors again according to the diffusion models. Thus, we remove them and their incident edges to derive the residual graph $G_{i+1} = (V_{i+1}, E_{i+1})$ for the next round of seeding, where $|V_{i+1}| = n_{i+1}$ and $|E_{i+1}| = m_{i+1}$. Especially, we denote $G_1 = G, V_1 = V, E_1 = E$, and $\eta_1 = \eta$. This process goes on until at least η users are activated. The strategy for selecting the seeds in each round is called a "policy" π, which returns the seeds based on the residual graph.

Assume the sampling results of the edges/nodes correspond with a realization $\phi \in \Omega$. The set of seeds selected by π under ϕ is denoted as $S(\pi, \phi)$. Similarly, the cost spent by π to select seeds under ϕ is denoted as $C(\pi, \phi) = c(S(\pi, \phi)) = \sum_{v\in S(\pi,\phi)} c(v)$. Then, we can formally define the AMCSS problem in Definition 2.2. The NP-hardness of AMCSS is direct, as it reduces to the adaptive seed minimization problem when $c(v) = 1$ for all $v \in V$, which was shown to be NP-hard in [29].

Definition 2.1 (The AMCSS Problem). Given a graph $G = (V, E)$ with propagation probability $p(e)$ for each $e \in E$, cost $c(v)$ for each $v \in V$, and an influence threshold η, AMCSS seeks a policy π^* that minimizes the expected cost of selected seeds while ensuring the influence spread is at least η on any realization $\phi \in \Omega$. That is,

$$\pi^* = \underset{\pi}{\operatorname{argmin}}\ \mathbb{E}_\Phi[C(\pi, \Phi)] \quad \text{s.t. } |I(S(\pi, \phi))| \geq \eta, \forall\, \phi \in \Omega.$$

Note that the aim of AMCSS is only reaching η, and any influence beyond η is unnecessary. Thus, we consider the truncated influence $\Gamma_\phi(S) = \min\{I_\phi(S), \eta\}$ for S in each possible realization ϕ [29]. In each round i, previous seeds $\mathcal{S}_{i-1} = \bigcup_{j=1}^{i-1} S_j$ have activated $I_\phi(\mathcal{S}_{i-1})$ nodes. To identify the nodes solely activated by S, we define the marginal truncated influence as $\Gamma_\phi(S|\mathcal{S}_{i-1}) = \Gamma_\phi(S \cup \mathcal{S}_{i-1}) - \Gamma_\phi(\mathcal{S}_{i-1})$. Especially, when no previous seed is selected (e.g., in the first round $\mathcal{S}_0 = \emptyset$), we have $\Gamma_\phi(S) = \Gamma_\phi(S|\mathcal{S}_{i-1})$. Considering all the realizations, we can derive the expected version $\mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})] = \sum_\phi p_\phi \Gamma_\phi(S|\mathcal{S}_{i-1})$ and $\mathbb{E}[\Gamma(S)] = \sum_\phi p_\phi \Gamma_\phi(S)$.

2.3 Existing Solutions

The work of [11, 12] takes the head in studying the AMCSS problem, where a greedy policy is designed to select the seed with the maximum influence-to-cost ratio in each round, providing an approximation ratio of $(\ln\eta + 1)^2$. However, their policy needs to know the exact truncated influence of nodes, which is #P-hard to compute [5]. To overcome this drawback, Tang et al. propose the *multi-root Reverse Reachable* set (mRR-set) to efficiently estimate the truncated influence [29].

In some round i, we generate an mRR-set from the residual graph G_i by three steps. (1) Uniformly sample $k_i = n_i/\eta_i$ nodes from V_i as the roots. (2) Perform a stochastic breadth-first search (BFS) from the roots along the reversed direction of edges, where the probability of visiting u from v is $p(u, v)$. (3) Take the set of visited nodes including the roots as an mRR-set. Note that k_i is not always an integer. Thus, we respect k_i in expectation, by sampling $\lceil k_i \rceil$ roots with probability $k_i - \lfloor k_i \rfloor$ and $\lfloor k_i \rfloor$ roots otherwise. Moreover, we say a set of nodes S *covers* an mRR-set R if $S \cap R \neq \emptyset$, and the *coverage* $\Lambda(S)$ of S is the number of mRR-sets covered by S. Given θ mRR-sets, we estimate $\mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})]$ by $\hat{\Gamma}(S|\mathcal{S}_{i-1}) = \eta_i\Lambda(S)/\theta$. It is proven in [29] that $\hat{\Gamma}(S|\mathcal{S}_{i-1})$ can estimate $\mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})]$ with bounded error:

$$(1 - 1/e)\mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})] \leq \mathbb{E}[\hat{\Gamma}(S|\mathcal{S}_{i-1})] \leq \mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})],$$

where the expectation in $\mathbb{E}[\hat{\Gamma}(S|\mathcal{S}_{i-1})]$ is taken w.r.t.the randomness in mRR-sets generation.

On this basis, Tang et al. [29] propose the ASTI algorithm. Specifically, in each round, ASTI first generates a collection of mRR-sets $\mathcal{R}$ and selects the seed u with the largest coverage. Then, ASTI verifies whether u approximates the optimal seed with a factor of $(1 - 1/e)(1 - \epsilon)$, where ϵ is the estimation error of the mRR-sets. It is proven that ASTI achieves an approximation ratio of $(\ln\eta + 1)^2/[(1 - 1/e)(1 - \epsilon)]$.

ASTI marks a milestone in solving the AMCSS problem. However, in each round, the mRR-sets need to be generated from scratch in a single thread, consuming substantial running time. Thus, it is necessary to devise practical and efficient algorithms.

2.4 GPU Architecture

GPUs are a powerful hardware in parallel computation. Typically, GPUs comprise multiple Streaming Multiprocessors (SMs), each capable of executing hundreds of threads concurrently. For NVIDIA GPUs, this massive parallelism can be exploited through the CUDA platform, organized in a hierarchical way as follows.

Kernels and Threads. In CUDA, the main computation tasks are encapsulated as *kernels*, which are functions executed in parallel on the GPU. When a kernel is launched, a user-specified number of threads are instantiated, which are organized as a *grid*. The grid partitions these threads into blocks of equal sizes. A block contains tens of warps. And a warp typically consists of 32 threads, which are the basic unit of scheduling and execute in lockstep on an SM. Like many CUDA applications, we employ a one-dimensional grid and block configuration for simplicity, with the grid and block sizes being *gridDim.x*/*blockDim.x*, respectively.

Memory Hierarchy. GPUs feature a layered memory hierarchy that balances storage capacity and access latency. Specifically, the global memory provides a large capacity but exhibits high access latency, which can significantly degrade performance for fine-grained

and irregular data exchanges. Additionally, dynamic memory allocation inside kernels is supported through the device heap, which is a dedicated region carved out from the global memory. Allocations from the device heap often incur substantial overhead, mainly due to the costs of allocation and deallocation, as well as fragmented and unpredictable memory layouts. The shared memory, by contrast, offers low-latency access and enables efficient cooperation among threads within the same thread block, although its capacity is limited on a per-block basis. Each thread further benefits from the efficiency of private registers, and intra-warp communication benefits from the design of warp-level primitives.

Atomic Operations. CUDA provides atomic operations (*e.g.*, atomicAdd, atomicSub, atomicOr) to ensure thread-safe variable updates in both shared and global memory. These operations are indivisible during their execution, with no interruption from other threads, and return the value before modification. However, when multiple atomic operations attempt to update the same variable concurrently, GPUs serialize these operations, causing waiting and reduced parallel efficiency. This overhead is higher for global memory due to its higher access latency.

3 THE MERIT FRAMEWORK

To leverage the power of GPUs, we design a new framework called MERIT to efficiently solve AMCSS. Generally, the MERIT framework includes three GPU-accelerated core modules as follows.

- `Generate`. When the number of mRR-sets is not enough, new mRR-sets are generated in parallel across multiple GPU blocks.
- `Select`. Given a set of mRR-sets, seeds are greedily selected in batches according to their coverage-to-cost ratio, with the selection parallelized on the GPU for efficiency.
- `Update`. We do not generate mRR-sets from scratch in each round, but update ones from the previous round in parallel on the GPU.

The application of the three modules in MERIT is depicted in Algorithm 1. Given the graph G and influence target η, MERIT selects seeds based on mRR-sets, which are stored in the array mrr. An auxiliary array $offset$ is applied to record the start position of each mRR-set in mrr, and an additional array $flag$ is applied to signify whether an mRR-set is covered by seeds. The coverage of each node is stored in the array cnt, which keeps track of the number of occurrences across all mRR-sets, initialized to 0.

In each round, MERIT aims to select a qualified batch of b seeds which approximates the optimal seeds within a ratio of $\rho_b(1-1/e)(1-\epsilon)$, where $\rho_b = (1-1/b)^b$. In general, the seeds are selected in a "trial-and-error" manner. The maximum number of mRR-sets used in this process is $\theta_{\max}$ (Line 5), which when reached can guarantee the $\rho_b(1-1/e)(1-\epsilon)$ approximation. Starting from θ mRR-sets, the maximum number of trials is simply H (Line 6). To begin with, we first update the mRR-sets from the previous round in parallel (Line 9), by applying Algorithm 5 named `Update` in Section 3.3. With the updated mRR-sets, we try to select the seeds S_i with Algorithm 4 called `Select` in Section 3.2 (Line 11). Since seed selection may fail, we first copy the coverage array cnt to cnt' for seed selection, and set θ_i to twice θ_{i-1} (Line 10) for generating additional mRR-sets if the selection fails. Next, we verify whether S_i is qualified. If S_i is unqualified, in which case `Select` returns an empty set, we supply the mRR-sets in parallel by calling the

Algorithm 1: MERIT

Input: Graph G, target η, estimation error ϵ, batch size b, cost $c(\cdot)$.
Output: seed set S.

1 $i=1, G_i=G, \eta_i=\eta, \rho_b \leftarrow 1-(1-1/b)^b, \theta_{i-1}=0, S \leftarrow \emptyset$;
2 $mrr \leftarrow \emptyset$; $offset \leftarrow \emptyset$; $flag \leftarrow \emptyset$; $cnt[u] \leftarrow 0, \forall u \in V$;
3 **while** $\eta_i > 0$ **do**
4 $\quad \delta \leftarrow \frac{1}{n_i}$; $\hat{\epsilon} \leftarrow (\epsilon-\delta)(1-\delta)$; $\rho_b \leftarrow 1-(1-1/b)^b$;
5 $\quad \theta_{\max} \leftarrow 2n_i\left(\sqrt{\ln\frac{6}{\delta}}+\sqrt{\left(\ln\binom{n_i}{b}+\ln\frac{6}{\delta}\right)/\rho_b}\right)\Big/(\hat{\epsilon}^2 b)$;
6 $\quad \theta \leftarrow \theta_{\max}\cdot\hat{\epsilon}^2 b/n_i$, $H \leftarrow \lceil \log_2(\theta_{\max}/\theta)\rceil+1$, $\theta_i \leftarrow \theta$;
7 $\quad a_1 \leftarrow \ln(3H/\delta)+\ln\binom{n_i}{b}$; $a_2 \leftarrow \ln(3H/\delta)$;
8 $\quad$ **if** $\theta_{i-1} > 0$ **then**
9 $\quad\quad$ **Parallel** `Update`$(G_i, cnt, mrr, offset, \theta_{i-1}, flag)$;
10 $\quad\quad cnt'[\cdot] \leftarrow cnt[\cdot]$; $\theta_i \leftarrow 2\theta_{i-1}$;
11 $\quad\quad S_i \leftarrow$ **Parallel** `Select`$(b, cnt', mrr, offset, \theta_{i-1}, flag)$;
12 $\quad\quad$ **if** $S_i \neq \emptyset$ **then** break;
13 $\quad$ **while** $\theta_i < \theta_{\max}$ **do**
14 $\quad\quad$ **Parallel** `Generate`$(G_i, cnt, mrr, offset, \theta_{i-1}, \theta_i)$;
15 $\quad\quad cnt'[\cdot] \leftarrow cnt[\cdot]$; $\theta_{i-1}=\theta_i$; $\theta_i \leftarrow 2\theta_i$;
16 $\quad\quad S_i \leftarrow$ **Parallel** `Select`$(b, cnt', mrr, offset, \theta_i, flag)$;
17 $\quad\quad$ **if** $S_i \neq \emptyset$ **then** break;
18 $\quad cnt[\cdot] \leftarrow cnt'[\cdot]$; $S \leftarrow S \cup S_i$;
19 $\quad$ Diffuse from S_i, and obtain the activated nodes A_i;
20 $\quad G_{i+1} \leftarrow G_i \backslash A_i$; $\eta_{i+1} \leftarrow \eta_i - |A_i|$; $i = i+1$;
21 $\quad$ **Parallel** `Discover`$(cnt, mrr, offset, \theta_{i-1}, flag, A_i)$;

`Generate` kernel (Line 14), depicted in Algorithm 2. Then, we repeat the above process to select seeds and verify the seed quality until S_i is non-empty or $\theta_{\max}$ mRR-sets are used.

When a qualified batch of seeds is selected, we renew the coverage of nodes cnt by the modified one cnt' (Line 18). Next, we trigger the influence diffusion from the selected seeds S_i (Line 19), and obtain the activated nodes A_i. Then, we delete the activated nodes to derive the residual graph G_{i+1} for the next round of seed selection (Line 20). Note that the mRR-sets may be *polluted* by containing nodes in A_i that are absent in G_{i+1}. To identify polluted mRR-sets and facilitate the update process, we apply the `Discover` kernel in Algorithm 3 to mark the polluted mRR-sets (Line 21).

The above process goes on until η nodes are observed to be activated (Line 20). The performance guarantee of MERIT is presented in Theorem 3.1, where π^o is the optimal policy under the same cost in each round as MERIT.

THEOREM 3.1. *The cost of* MERIT *reaching* η *approximates that of* π^o *in the following form*

$$c(\texttt{MERIT}) \leq \frac{2\ln\eta}{(1-\epsilon)(1-1/e)\rho_b}\mathbb{E}[c(\pi^o)] + b + 1.$$

[Due to space limitations, we defer the detailed proof to our report [2]. The proof sketch is as follows.]

3.1 mRR-set Generation

To accelerate mRR-sets generation in parallel on a GPU, we exploit the hierarchical execution model of CUDA to achieve two-level parallelism. Specifically, we parallelize not only the generation of mRR-sets in multiple blocks, but also the reverse BFS process. A direct way to realize this idea is to extend [27], which parallelizes

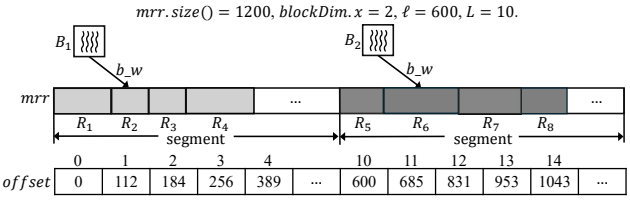


Figure 2: Example of mRR-set generation.

the generation of single-root RR-sets, to our case of multiple roots. In the resultant baseline called GAMCSS, each block generates mRR-sets in parallel as many as possible, until the total quantity reaches the required number in each round. All the mRR-sets are stored contiguously in the global memory like [27]. GAMCSS suffers from two major drawbacks. (1) The size of an mRR-set is unknown before its generation completes. Thus, GAMCSS can not reserve memory for it to allow concurrent writing. Instead, GAMCSS has to store it in the device heap before copying it to the global memory, incurring additional runtime and memory. (2) Some blocks may finish generating mRR-sets and begin writing to the memory at the same time. Then, a global write pointer is required to determine the writing position. However, updating this pointer necessitates additional atomic operations, which introduce additional overhead.

To address the above limitations, we propose partitioning the array storing mRR-sets (*i.e.*, *mrr*) into non-overlapping consecutive segments, each containing an equal number of entries ℓ. Each segment (*e.g.*, the j-th one) is dedicated to storing the mRR-sets from the j-th block. In this way, each block can independently generate mRR-sets within its assigned segment, eliminating the need for inter-block communication. Moreover, no extra temporary storage is required, as each block can write directly to its designated segment. With equal-sized segments, when θ mRR-sets need to be generated, we equally distribute the task across thread blocks, each generating $\bar{\theta} = \theta / gridDim.x$ mRR-sets. Often, $\bar{\theta}$ is not an integer. In this case, we require the first $\theta \% gridDim.x$ blocks to generate $\lceil \bar{\theta} \rceil$ mRR-sets, and the remaining blocks to generate $\lfloor \bar{\theta} \rfloor$ mRR-sets. Even if the mRR-sets are incrementally generated over rounds with a total number of θ, the above arguments still hold, and the number of mRR-sets in a block can be derived as $\theta / gridDim.x + \mathbb{I}_{\{\theta \% gridDim.x > block.id\}}$.

Since we need to traverse *mrr* when computing node coverage, we adopt an array $offset$ to record the start position of each mRR-set in *mrr*. Note that the end position of an mRR-set is the start position of the next mRR-set. Thus, we do not need to maintain the end positions of mRR-sets. However, the number of mRR-sets needed in each round may vary, and if the positions of mRR-sets across different segments are stored contiguously in $offset$ array, any change in the total number of mRR-sets would shift subsequent start positions, requiring reconstruction of $offset$. To avoid the reconstruction, we stipulate a maximum number of mRR-sets $L-1$ that each segment can hold. Then, we can fix the length of the array $offset$ by $L * gridDim.x$, where the number of blocks $gridDim.x$ is also the number of segments since they are in a one-to-one correspondence. That is, for each segment, $L-1$ entries in $offset$ are assigned to record the start positions of at most $L-1$ mRR-sets, while the L-th entry indicates the end position of the last mRR-set in the segment. For example, let us consider the r-th mRR-set ($0 \le r \le L-1$) in a segment, its start position of in *mrr* is $offset[r + block.id * L]$.

Example 3.2. Figure 2 illustrates the segmented memory layout of the global array *mrr* and its corresponding $offset$ structure under a two-block configuration for the mRR-set generation process. In this example, the size of *mrr* is 1200. The global *mrr* array is evenly partitioned into two equal-sized segments, each with capacity $\ell = 600$, exclusively assigned to one thread block (B_1 and B_2). Within each segment, mRR-sets (*e.g.*, R_1–R_4 in the first segment and R_5–R_8 in the second) are stored contiguously. The write pointer b_w of each block indicates the current insertion position within its segment, ensuring that memory accesses remain local to the assigned region. The auxiliary array $offset$ records the starting position of each mRR-set in *mrr*, and the end position of an mRR-set is implicitly determined by the start position of the next mRR-set. The numbers shown above the $offset$ array denote its index positions, as each block reserves $L = 10$ entries in the $offset$ array.

Algorithm 2 presents the pseudocode of the kernel function for generating mRR-sets. Given θ_{i-1} existing mRR-sets, we aim to expand the collection to θ_i mRR-sets. In Lines 1-2, we calculate the number of existing mRR-sets r_1 in each block, and require each block to reach the same number of mRR-sets r_2. In Line 3, we set the write position b_r to the start position of the (r_1+1)-th mRR-set (i.e., the end of the r_1-th mRR-set), and initialize the read position b_r to coincide with b_w. Then, each block iteratively generates new mRR-sets until there are r_2 mRR-sets. For each new mRR-set, the block parallelly marks all residual nodes in G_i as unvisited, samples k_i roots from V_i, directly writes roots to the block's segment, and updates their coverage count (Lines 6-9). We also make the segment support circular placement of visited nodes v within the segment, which will be detailed in Section 3.3. Then, a reverse BFS process is triggered from the k_i roots by invoking the Procedure `Probe` in Line 10. Upon completion of the BFS, the block increases its mRR-set number by 1, and records the next write position b_w as the beginning of the next mRR-set in the $offset$ array (Lines 11-12).

Algorithm 2: `Generate`

Input: Graph G_i, coverage counter cnt, mRR-set mrr, mRR-set $offset$, segment length ℓ, maximum #mRR-sets per block L, #previous mRR-sets θ_{i-1}, and #required mRR-sets θ_i.
Output: cnt, mrr, $offset$.

1 $r_1 \leftarrow \theta_{i-1}/gridDim.x + (\theta_{i-1} \,\%\, gridDim.x) > block.id$;
2 $r_2 \leftarrow \theta_i/gridDim.x + (\theta_i \,\%\, gridDim.x) > block.id$;
3 $b\_w \leftarrow offset[r_1 + block.id * L]$; $b\_r \leftarrow b\_w$;
4 **while** $r_1 < r_2$ **do**
5 $\forall u \in V_i, visited[u] \leftarrow false$;
6 **parallel for** $i = thread.id; i < k_i; i \mathrel{+}= blockDim.x$ **do**
7 Uniformly sample $u \in V_i$ s.t. $visited[u] = flase$;
8 $mrr[(b\_w + i) \,\%\, \ell + block.id * \ell] \leftarrow u$;
9 $visited[u] \leftarrow true$; atomicAdd($cnt[u]$, 1);
10 __syncthreads();
11 `Probe`($G_i, k_i, \ell, b\_r, b\_w, mrr, visited, cnt$);
12 $r_1 \leftarrow r_1 + 1$;
13 $offset[r_1 + block.id * L] \leftarrow b\_w$; $b\_r \leftarrow b\_w$;

Procedure Probe(G, k, ℓ, b_r, b_w, mrr, $visited$, cnt)

```
1  b_w ← b_r + k; head ← 0; tail ← k;
2  while head < tail do
3      if thread.id % 32 = 0 then
4          idx ← atomicAdd(head, 1); work ← head < tail;
5      __syncwarp();
6      Broadcast idx and work to all threads in the same warp;
7      if !work then break;
8      u ← mrr[(b_r + idx) % ℓ + block.id * ℓ];
9      parallel for each v ∈ N_in(u) & visited[v] = flase do
10         if p(v, u) meets the condition in IC or LT then
11             mrr[atomicAdd(b_w, 1) % ℓ + block.id * ℓ] = v;
12             visited[v] ← true;
13             atomicAdd(cnt[v], 1), atomicAdd(tail, 1);
14     __syncwarp();
```

The Procedure `Probe` details how the threads in a block collaboratively perform a multi-source stochastic BFS. Before probing, the read pointer b_r is positioned at the start of the current mRR-set, while the write position b_w is modified to the end of the k_i roots. Further, we apply a new variable *head* to help indicate the next unexplored node. The other new variable *tail* tracks the total number of nodes in the mRR-set. During probing, we adopt a warp-centric probing strategy in which warps process unprobed frontier nodes in a work-conserving manner, with threads within each warp cooperatively traversing the in-neighbors of a frontier. This approach reduces branch divergence and maximizes thread utilization, particularly under high skewness in node degrees, where naive thread-level parallelism, which processes one frontier node at a time, would otherwise lead to workload imbalance and poor utilization of threads. Specifically, we designate the first thread in each warp to fetch the index of a frontier node and determine whether unprobed frontiers remain. The obtained index (*i.e.*, idx) and the corresponding validity indicator (*i.e.*, $work$) are then broadcast to all threads in the warp, which cooperatively probe the unvisited in-neighbors of u in parallel. For node $v \in N_{\text{in}}(u)$, it will be appended to the mRR-set if the following condition is satisfied.

- For the IC model, the edge probability $p(v, u)$ exceeds a random value drawn from $[0, 1]$.
- For the LT model, the cumulative weights from its active in-neighbors surpasses its threshold $\lambda_v \in [0, 1]$.

Accordingly, the coverage $cnt[v]$ as well as the number of nodes in the mRR-set is atomically increased by 1. The above process goes on until all the nodes are probed, *i.e.*, $head = tail$.

3.2 Seed Selection

In each round i, given a collection of mRR-sets, we greedily select a batch of b seeds by iteratively selecting the node with the maximum coverage-to-cost ratio. Once a node u becomes a seed, some mRR-sets (e.g., R) become covered by u. Accordingly, the number of mRR-sets that can be covered by other nodes (i.e., the coverage $cnt[\cdot]$) should be decreased. However, recomputing their coverage from scratch by traversing the remaining mRR-sets is computationally expensive. Instead, we decrement the coverage of nodes that occur in the mRR-sets already covered by u. This process is referred to as *affected* mRR-sets discovery. To efficiently conduct this process, we devise the `Discover` kernel, which flags affected mRR-sets and updates the node coverage counter.

As shown in Algorithm 3, `Discover` receives as input the node coverage cnt, the mRR-sets mrr along with the array $offset$, an array $flag$ indicating whether an mRR-set is affected, and the candidate seed set. Each thread block first obtains the number of existing mRR-sets stored in its segment (Line 1), and then iterates over all of the mRR-sets. For each mRR-set, a block first retrieves its start and end positions from the $offset$ array, then computes the size of the mRR-set in the segment (Lines 3–5). If the flag of the mRR-set is false, indicating no candidate seeds have been identified yet, threads within the block scan this mRR-set in parallel to check whether it contains any candidate nodes. If a candidate seed is spotted, the flag of this mRR-set is set to true, and the mRR-set becomes affected (Lines 9-11), meaning it will not contribute to coverage calculations during subsequent seed selection. Concurrently, for each node u in the affected mRR-set, the corresponding coverage count is atomically decremented in parallel (Lines 14-16).

In this way, we incorporate the kernel `Discover` into the batch seeding algorithm `Select`, which initializes an empty seed set S_i and sets the mRR-set flag array to false for all mRR-sets, indicating no mRR-sets are affected. A copy of the coverage counter cnt' is also maintained, allowing the counter cnt to be restored if the candidate seed batch does not meet the approximation guarantee. Then, we progressively select nodes with the highest coverage-to-cost ratio until the batch size is reached. During the selection of each candidate seed, the `Discover` kernel is invoked to identify the mRR-sets that contain the candidate node, mark them as affected by setting their flags to true, and atomically decrement the coverage counters of nodes within those affected mRR-sets.

Algorithm 3: `Discover`

Input: Coverage counter cnt, mRR-set mrr, offset $offset$, flag array $flag$, segment length ℓ, maximum #mRR-sets per block L, #previous mRR-sets θ_{i-1}, candidate seed set A.
Output: cnt, $flag$.

```
1  r ← θ_{i-1}/gridDim.x + (θ_{i-1} % gridDim.x) > block.id; r_1 ← 0;
2  while r_1 < r do
3      start ← offset[r_1 + block.id * L];
4      end ← offset[r_1 + 1 + block.id * L];
5      len ← (end > start) ? (end − start) : ℓ − (start − end);
6      affected ← false;
7      if flags[r_1 + block.id * L] = false then
8          parallel for i = thread.id; i < len; i += blockDim.x do
9              if mrr[(start + i) % ℓ + block.id * ℓ] ∈ A then
10                 flag[r_1 + block.id * L] ← true;
11                 affected ← true;
12             if affected then break;
13         __syncthreads();
14     if affected = true then
15         parallel for i = thread.id; i < len; i += blockDim.x do
16             u ← mrr[(start + i) % ℓ + block.id * ℓ];
17             atomicSub(cnt[u], 1);
18         __syncthreads();
19     r_1 ← r_1 + 1 ;
```

Algorithm 4: `Select`

Input: Batch size b, coverage counter cnt, mRR-set mrr, offset array $offset$, flag array $flag$, #previous mRR-sets θ_{i-1}.
Output: $S_i, \Lambda(S_i), cnt$.

$S_i \leftarrow \emptyset; \Lambda(S_i) \leftarrow 0; flag[\cdot] \leftarrow false;$
while $|S_i| < b$ **do**
  $u \leftarrow \text{argmax}_{v \in V_i} \frac{cnt[v]}{c(v)}$ by **parallel** reduction on GPU;
  $S_i \leftarrow S_i \cup \{u\}, \Lambda(S_i) \leftarrow \Lambda(S_i) + cnt[u];$
  Parallel `Discover`$(cnt, mrr, offset, \theta_{i-1}, flag, \{u\})$;
$\Lambda^l(S_i) \leftarrow \left(\sqrt{\Lambda(S_i) + 2a_1/9} - \sqrt{a_1/2}\right)^2 - a_1/18;$
$\Lambda^u(S_i^o) \leftarrow \left(\sqrt{\Lambda(S_i)/\rho_b + a_2/2} + \sqrt{a_2/2}\right)^2;$
if $\Lambda^l(S_i) > \rho_b(1-\hat{\epsilon})\Lambda^u(S_i^o)$ **then** return S_i;
else return $\emptyset$;

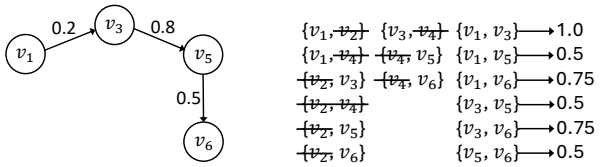
3.3 mRR-set Update

To reduce the computational overhead, we aim to reuse mRR-sets from the previous round rather than regenerate them. Recap that in each round i, we observe the actual diffusion triggered by the seed set S_{i-1}, and obtain newly activated nodes A_{i-1}. An mRR-set is said to be *clean* if it does not contain any activated nodes; otherwise, it is *polluted*. Additionally, in the current round, the number of toots in each mRR-set is required to be $k_i = n_i/\eta_i = (n_{i-1} - |A_{i-1}|)/(\eta_{i-1} - |A_{i-1}|)$, which is monotonically increasing across rounds (*i.e.*, $k_i \geq k_{i-1}$), since $|A_{i-1}| \geq 1$ and $n_i \geq \eta_i$. Thus, not only polluted mRR-sets but also clean mRR-sets need to be updated to make them equivalent to those directly generated from G_i.

3.3.1 Update Clean mRR-set. Let R be a random clean mRR-set in the previous round. The roots of R follow the same distribution as roots sampled from the residual graph G_i. Moreover, since R does not contain any activated nodes, the reverse BFS results remain valid on G_i. This observation motivates an update strategy that preserves R and extends it with additional roots. The correctness of this strategy is formalized in Lemma 3.3.

Lemma 3.3. *If the previous mRR-set R with k_{i-1} roots is clean, the nodes in R can be preserved. Additionally, by appending $k_i - k_{i-1}$ new roots sampled uniformly at random from V_i and probing from these additional roots, the resultant mRR-set is equivalent to one regenerated from scratch from all these k_i roots.*

Detailed proof of Lemma 3.3 is provided in our technical report [2]. This lemma underpins an efficient update strategy for clean mRR-sets, allowing them to be updated without regeneration from scratch. However, some sampled roots may already exist in the mRR-set, and we refer to these kinds of roots as *promoted* roots, since they were not initial roots in this mRR-set. We have proven that promoted roots will not change the composite of the mRR-set in Lemma 3.3. However, since sampled roots are directly written to the mRR-set storage, a promoted root leads to duplicate entries in the mRR-set. Such duplicates must be explicitly removed to ensure the correct computation of node coverage. Eliminating such duplicates on GPUs is inefficient, as it requires additional scans over variable-length mRR-sets, followed by conditional handling, such as marking or compaction of duplicated entries, which introduces warp divergence and extra overhead. To resolve this issue,



(a) Residual graph G_2 (b) Root sets

Figure 3: The residual graph and root sets.

we deliberately split the task of adding $k_i - k_{i-1}$ roots into two subprocesses as shown in Lemma 3.4.

Lemma 3.4. *For a random mRR-set R with k_{i-1} roots K, if it is clean, we can select $k_\Delta = (k_i - k_{i-1}) \cdot \frac{n_i - |R|}{n_i - k_{i-1}}$ roots from $V_i \backslash R$ and the rest $k_\nabla = k_i - k_{i-1} - k_\Delta$ from $R \backslash K$ uniformly at random. The resultant root set has the same distribution as one directly generated from V_i.*

Lemma 3.4 guarantees that the additional roots remain uniformly distributed, matching the distribution obtained by sampling directly from the new residual graph when updating a clean mRR-set. The first subset contains k_Δ new roots that are not present in the previous mRR-set and therefore trigger reverse BFS on G_i to discover additional nodes. The second subset contains k_∇ promoted roots, whose reverse neighborhoods have been explored and thus do not require repeated BFS. By separating roots in this way, we avoid duplicate entries and expensive membership checks in mrr, resulting in a more efficient and GPU-friendly update procedure.

3.3.2 Update Polluted mRR-set. If an mRR-set contains activated nodes, the BFS results from the previous roots become incorrect in the current residual graph G_i. However, discarding the whole mRR-set and regenerating it from scratch would induce a non-uniform distribution over root sets, as demonstrated in Example 3.5.

Example 3.5. Assume the algorithm starts with $\eta = 4$ on graph G under the IC model in Figure 1(a). We sample sufficient mRR-sets, each with $\lceil \frac{n}{\eta} \rceil = 2$ roots. Figure 3(b) enumerates $\binom{6}{2} = 15$ possible root sets, each with uniform probability. In the first round, v_2 is selected, and activates v_2 and v_6 under the realization in Figure 1(b). In the next round, we proceed on the residual graph G_2 in Figure 3(a) with $\eta_2 = 2$, requiring mRR-sets with root count $\lceil \frac{n_2}{\eta_2} \rceil = 2$ again. During the update, if all polluted mRR-sets are regenerated by uniformly selecting roots from the remaining nodes, the root distribution becomes biased. For example, $\frac{3}{5}$ mRR-sets with root set including v_2 or v_6 are regenerated. Other mRR-sets, such as the one with root set $\{v_1, v_3\}$, can never include v_2. Thus, $\frac{11}{15}$ of the total mRR-sets are regenerated. As a result, the final probability of sampling with root set $\{v_1, v_3\}$ becomes $\frac{1}{15} + \frac{11}{15} \cdot \frac{1}{6} = \frac{17}{90}$, deviating from the uniform probability $\frac{1}{6}$, demonstrating sampling bias.

Instead, following Lemma 3.6, we keep available roots in previous polluted mRR-sets together with additional sampled roots to reprobe over G_i, which ensures the root sets among all mRR-sets are correctly distributed over G_i. Its proof is deferred to the report.

Lemma 3.6. *Let K be the set of roots of a random mRR-set R from round $i-1$. We write $K = K_1 \cup K_2$, where K_1 is not activated while K_2 is. Then, in round i, we delete K_2 from K and select $|K_2| + k_i - k_{i-1}$ nodes from $V_i \backslash K_1$ uniformly at random. The distribution of the new root set is the same as the one generated from V_i directly.*

Example 3.7. To further illustrate the correctness of the root set distribution, we revisit Example 3.5. We preserve all previous clean mRR-sets and retain available roots in polluted mRR-sets. $\frac{2}{15}$ of the mRR-sets already contain root v_1 or v_3, with a probability $\frac{1}{3}$ of forming a root set $\{v_1, v_3\}$. $\frac{1}{15}$ of mRR-sets with root $\{v_2, v_4\}$ are regenerated. Additionally, $\frac{1}{15}$ of the mRR-sets with root set $\{v_1, v_3\}$ are kept. Thus, the total probability of sampling root set $\{v_1, v_3\}$ is $\frac{2}{15} \cdot 2 \cdot \frac{1}{3} + \frac{1}{15} \cdot \frac{1}{6} + \frac{1}{15} = \frac{1}{6}$, confirming the unbiased root sampling.

3.3.3 Update mRR-sets on a Segment. Now we move to update mRR-sets that are generated as described in Section 3.1, where mRR-sets produced by each thread block are stored contiguously within the associated segment. With the theoretical foundation for the mRR-set update strategy established, the mRR-sets generated in the previous round can be updated without exhaustive regeneration. Clean mRR-sets are reused directly, adding additional roots only when they trigger new BFS, while polluted mRR-sets are reprobed from non-activated roots together with newly sampled roots. The mRR-sets are stored contiguously within dedicated segments and each thread block independently updates the mRR-sets it generated. Consider a segment D with d mRR-sets. A naive method is to sequentially process the d mRR-sets from the first to the last in D. However, since the size of each updated mRR-set is unknown in advance, temporary GPU memory is needed to hold the revised content. When the update for the j-th mRR-set is complete, we need to move backward/forward all subsequent $d - j$ to accommodate the new content. Summing over j, we see that the mRR-sets are moved for $\Theta(d^2)$ times, leading to substantial cumulative runtime overhead.

To address the above issues, we devised a new memory layout tailored to our segmented memory design. Consider a hypothetical scenario where the capacity of the segments is infinite, *i.e.*, $\ell \to \infty$. In this case, the last mRR-set can always be updated in place since sufficient free space is available. All other mRR-sets awaiting update can be written sequentially after the last updated mRR-set, thereby eliminating the need for shifting subsequent mRR-sets in the collection. However, in practice, ℓ is constrained as GPU memory is limited. To preserve the advantage of the infinite scenario, we treat the segments as *circular*. That is, the end of each segment is logically linked to its beginning, forming a continuous loop. Under this circular design, the updated content of some mRR-set may exceed the end of the segment. Then, we continue writing from the start of the segment. Note that the first mRR-set in the segment has been updated, so its previous content is invalid and can be overwritten safely. With the circular design, the index of the inserted node in *mrr* keeps increasing. When a node u needs to be written, the physical index of u in *mrr* (denoted as *idx*) may exceed $\ell - 1 + block.id * \ell$, while actually u will be inserted to the front of the segment with index $block.id * \ell$. To achieve this, we take the remainder of *idx* modulo ℓ as the index in the segment. Further adding the beginning index of the segment $block.id * \ell$, we derive the writing position of u, which is $idx\%\ell + block.id * \ell$.

With the above idea, we design an efficient kernel for updating mRR-sets using segment circularity. We first update the last mRR-set in place, which becomes the first after the update. The update then proceeds sequentially from the first to the penultimate mRR-set, which are written after the newly updated mRR-set (i.e., at the tail of the new mRR-set collection). For clean mRR-sets, all elements are copied to the tail, whereas for polluted mRR-sets, only the feasible roots are copied. Since the mRR-set is moved only when it is processed, yielding an overall time complexity of $\Theta(d)$ within a segment. New content is then written into the remaining free space of the segment in a circular manner. For correctness, the end of the mRR-set being updated must not exceed the start of the next mRR-set awaiting update. This guarantees that the content of the pending mRR-sets remains intact and can be accessed correctly. When an overwrite risk is detected, we can fall back to regenerating mRR-sets with a reduced number of segments, which helps mitigate skewness of mRR collection within a segment, especially when the number of mRR-sets is relatively small, or using a larger mRR array if available.

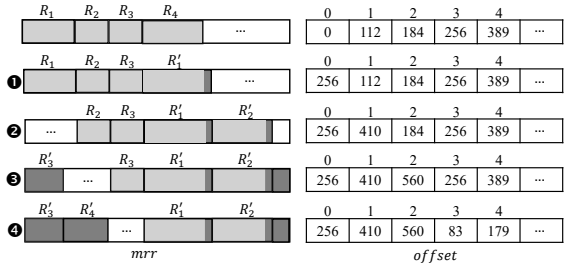


Figure 4: Example of mRR-set update.

Example 3.8. We explain the workflow of our update process in Figure 4(b). When updating R_1–R_4 generated in a block, we first process R_4 (step ❶), since it is the last mRR-set in memory and free space starts after it. R_4 is clean, we retain its elements and write newly visited nodes from its end, producing the first updated mRR-set R'_1. We then update from R_1 to R_3 in steps ❷-❹. Clean mRR-set R_1 is moved to the end of the last updated mRR-set R'_1 and extended with new nodes, while the polluted ones (*i.e.*, R_2 and R_3) are regenerated and written directly from the end of last updated mRR-set. Upon reaching the end of the current segment, the write position wraps around to the segment's beginning, following the circularity design. In this way, our update method only moves the previous R_1 once and does not incur additional memory use.

Based on the theoretical correctness of our update strategy and the practical implementation optimizations, Algorithm 5 outlines the update process within a segment. Each block first determines the current number of mRR-sets r within its segment. We then initialize the index r_1, pointing to the last existing mRR-set from the previous round, and r_2, tracking the order for the revising mRR-set in the current round. The update kernel starts by handling the last existing mRR-set, recording its start and end positions. This mRR-set is processed in place, and thus the writing position b_w is set to this position (Lines 3-4). To update an mRR-set, we first mark all nodes in V_i as unvisited and record the initial writing position as b_r, while also obtaining the mRR-set's size (Lines 6–7). If the last existing mRR-set is clean, indicated by a false flag (Line 8), we retain all its original nodes (Lines 10–12) and sample additional k_Δ roots in parallel according to Lemma 3.4 (Lines 13–16). We then perform reverse BFS from k_Δ roots, appending all newly visited nodes directly after copying nodes of the previous mRR-set. Moreover, we randomly choose and move k_∇ promoted roots,

and reposition k_Δ roots in Line 18. This ensures that the first k_i nodes are always roots in the current round, enabling correct root identification when updating these mRR-sets later.

If the last previous mRR-set is polluted, as shown in Lines 20-25, we collect all available roots in the previous mRR-set and sample additional roots to reach k_i. A reverse BFS is then performed using all of these roots in Line 26. The updated last mRR-set will become the new first mRR-set in the current round. Subsequent waiting mRR-sets are updated sequentially. Before each update, we retrieve the original start and end positions of the mRR-set waiting to be updated (Lines 27-28). The start position of each updated mRR-set is stored in the offset array according to its new order (Line 29). For later mRR-sets, clean mRR-sets are first moved to the current tail, additional roots are added, and visited nodes in reverse BFS are appended directly. Polluted mRR-sets are regenerated from previous non-activated roots together with the newly sampled roots. After all mRR-sets in the segment have been updated, the tail position of the last updated mRR-set is stored as the end offset of the mRR-set collection in this block (Line 30).

3.3.4 Update Workload Balance. In the `Update` algorithm, we retain the same block-to-segment mapping as in mRR-set generation, *i.e.*, each thread block is responsible for updating the segment it originally produced. However, due to skewness in the number of polluted mRR-sets across segments, this static assignment may lead to workload imbalance. To mitigate this workload imbalance, we further propose a lightweight global load-balancing strategy. Specifically, we compute the number of polluted mRR-sets in each segment in parallel using the $flag$ array, and create an index array of segment IDs (from 0 to $gridDim.x - 1$). The index array is then sorted in descending order based on the number of polluted mRR-sets in each segment and partitioned into β groups, such that each group contains segments with similar numbers of polluted mRR-sets. The `Update` kernel processes mRR-sets in these groups over β rounds, with explicit block-level synchronization between rounds. In each round, the number of active blocks is reduced by a factor of β, allowing the number of threads per block to be increased proportionally to fully utilize GPU resources. This grouping strategy reduces workload disparity among active blocks, leading to more balanced execution across them and improved overall efficiency, without requiring complex dynamic task stealing.

Algorithm 5: `Update`

Input: Graph G_i, coverage counter cnt, mRR-set mrr, mRR-set offset array $offset$, segment length ℓ, maximum #mRR-sets per block L, #existing mRR-sets θ_{i-1}, flag array $flag$.
Output: cnt, mrr, $offset$.

1 $r \leftarrow \theta_{i-1}/gridDim.x + (\theta_{i-1} \,\%\, gridDim.x) > block.id$;
2 $r_1 \leftarrow r-1; r_2 \leftarrow 0; \ell_b \leftarrow block.id * \ell$;
3 $start \leftarrow offset[r_1 + block.id * L]; b\_w \leftarrow start$;
4 $end \leftarrow offset[r_1 + 1 + block.id * L]$;
5 **while** $r_2 < r$ **do**
6 $visited[u] \leftarrow false (\forall u \in V_i); b\_r \leftarrow b\_w$;
7 $len \leftarrow (end > start)\ ?\ (end - start) : \ell - (start - end)$;
8 **if** $flag[r_1 + block.id * L] = false$ **then** `// clean`
9 Obtain k_Δ and k_∇ *w.r.t.* Lemma 3.4;
10 **parallel for** $i = thread.id; i < len; i$ += $blockDim.x$ **do**
11 $u \leftarrow mrr[(start + i) \,\%\, \ell + \ell_b]; visited[u] \leftarrow true$;
12 $mrr[(b\_w + i) \,\%\, \ell + \ell_b] \leftarrow u$;
13 **parallel for** $i = thread.id; i < k_\Delta; i$ += $blockDim.x$ **do**
14 Uniformly sample $u \in V_i$ s.t. $visited[u] = flase$;
15 $mrr[(b\_w + len + i) \,\%\, \ell + \ell_b] = u$;
16 $visited[u] \leftarrow true$; atomicAdd($cnt[u]$, 1);
17 __syncthreads();
18 `Probe`($G_i, k_\Delta, \ell, b\_r + len, b\_w, mrr, visited, cnt$);
19 **parallel** swap new roots behind the previous k_{i-1} roots;
20 **else** `// polluted`
21 **parallel for** $i = thread.id; i < k_i; i$ += $blockDim.x$ **do**
22 **if** $i < k_{i-1}$ **then** $u \leftarrow mrr[(start + i) \,\%\, \ell + \ell_b]$;
23 **if** $u \notin V_i$ *or* $i \geq k_{i-1}$ **then**
24 Uniformly sample $u \in V_i$ s.t. $visited[u] = flase$;
25 $visited[u] \leftarrow true$; atomicAdd($cnt[u]$, 1);
26 $mrr[(b\_w + i) \,\%\, \ell + \ell_b] \leftarrow u$;
27 __syncthreads();
28 `Probe`($G_i, k_i, \ell, b\_r, b\_w, mrr, visited, cnt$);
29 $r_1 \leftarrow (r_1 + 1) \,\%\, r; start \leftarrow offset[r_1 + block.id * L]$;
30 $end \leftarrow offset[r_1 + 1 + block.id * L]$;
31 $offset[r_2 + block.id * L] \leftarrow b\_r; r_2 \leftarrow r_2 + 1$;
32 $offset[r_2 + block.id * L] \leftarrow b\_w$;

4 RELATED WORK

Minimum Cost Seed Selection (MCSS). The MCSS problem aims to find a seed set that minimizes the cost to achieve the influence spread threshold η. First formulated in [23], the problem shares the submodularity and monotonicity properties with the classic Influence Maximization (IM). Existing approaches to MCSS can be divided into two categories. The first category focuses on reaching η only in expectation [10, 13, 15, 23], where selected seeds can only guarantee the coverage meets η. However, the actual influence spread of the selected seed often falls significantly short of the target η. Therefore, these methods are not feasible for the AMCSS problem, as we require that the selected nodes must influence at least η nodes in the network in every realization. The other category addresses this practical limitation by providing probabilistic guarantees [10, 39], ensuring that with a predefined probability p that the influence achieved by the selected seeds reaches η in any given realization. Thus, we can set a high probability value to increase the likelihood that the selected seeds consistently meet the target influence threshold in practice.

Adaptive Seed Minimization (ASM). The adaptive setting in IM and MCSS receives considerable attention in recent years [11, 12, 14, 16, 20, 29, 33, 34]. In the adaptive setting, nodes are seeded in batches at different rounds, and after each batch is seeded, an actual diffusion process is allowed to unfold, with the outcomes observed before deciding the next batch of seeds. Golovin and Krause [11] pioneered the extension of the classic notions of submodularity and monotonicity to the adaptive setting, thereby establishing an important theoretical foundation. However, their work did not provide a practical algorithm tailored to the ASM problem. Tang et al. [29] first introduced truncated influence, recognizing that influence beyond a threshold η provides no additional value, and

Table 1: Datasets information. ($K = 10^3, M = 10^6$)

Name	n	m	Avg. deg	Type
DBLP	317K	1.05M	6.62	undirected
YouTube	1.13M	2.99M	5.27	undirected
Flickr	2.30M	33.1M	14.4	directed
LiveJournal	4.85M	68.5M	14.1	directed

they devised mRR-sets to estimate marginal truncated influence targeted for ASM. However, ASTI assumes uniform seed costs across all nodes, which significantly restricts its applicability, since real-world influence campaigns typically involve heterogeneous costs. Moreover, ASTI is a single-threaded CPU-based implementation and regenerates the entire collection of mRR-sets in each round.

Parallel RR-set Approaches. Several methods have been proposed to solve the IM problem in parallel [24, 25, 27] using the Reverse Reachable set (RR-set) technique, which uses only one root. While the RR-set generation phase can be adapted for mRR-set generation in AMCSS, these approaches face notable efficiency challenges. Ripples and cuRipples [24, 25] utilize thread-level parallelization for RR-set generation. However, these methods suffer from workload imbalance due to skewed RR-set sizes and suboptimal thread utilization when the number of RR-sets is fewer than the available threads. Additionally, each thread requires its own auxiliary arrays (*e.g.*, for tracking visited nodes), limiting scalability to large graphs. These issues remain when the approach is extended to mRR-set generation in adaptive rounds. For the block-level parallelization approach [27], its implementation necessitates temporary storage and frequent atomic operations to manage RR-sets in global memory. The implementation also degrades performance and introduces further memory constraints.

5 EXPERIMENTS

We evaluate the performance of our proposed method against state-of-the-art baselines for AMCSS, including ASTI and its parallel implementations on both CPUs and GPUs, as well as a representative non-adaptive method SCORE. All experiments are conducted on a Linux server equipped with an Intel Xeon Gold 6226R 2.90 GHz CPU with 16 physical cores, 384 GB RAM, and an NVIDIA RTX 3090 GPU with 24 GB of memory. Notably, the recommended retail prices of the Xeon Gold 6226R CPU and the RTX 3090 GPU are comparable, at approximately $1,517 and $1,499, respectively.

5.1 Experimental Setup

Datasets. We adopt four datasets used in previous work [10, 16, 20, 29, 35], with detailed statistics in Table 1. These datasets capture different types of social networks in the real world and are available from [22]. The largest dataset in our experiments is LiveJournal, which comprises over 4.85 million nodes and approximately 68.5 million edges, and it is also the largest dataset tested in [29] which proposes the ASTI algorithm.

Baselines. We evaluate five algorithms to compare with our proposed method `MERIT`. The first baseline, ASTI, is the original single-threaded CPU implementation for AMCSS under uniform seed costs. We extend it to select seeds with the largest influence-to-cost ratios in each round, which substantially benefits ASTI and makes its total cost comparable to that of `MERIT`. PASTI-4 and PASTI-16 are two CPU-based parallel variants built on ASTI, utilizing 4 threads and 16 threads, respectively. We equip PASTI with OpenMP [6] to parallelize all applicable components. Since PASTI requires non-trivial algorithmic modifications from ASTI, we provide a detailed implementation in the technical report [2]. GAMCSS is a GPU-based adaptation of ASTI, retaining its core logic but with an mRR-set generation strategy inspired by the related work [27]. SCORE is the state-of-the-art non-adaptive MCSS method that provides a probabilistic guarantee to ensure selected seeds that reach the target influence threshold η in any realization. We configure SCORE with a high confidence ($p = 0.99$) to align with the stopping criterion of other adaptive methods which reach η with probability 1.0.

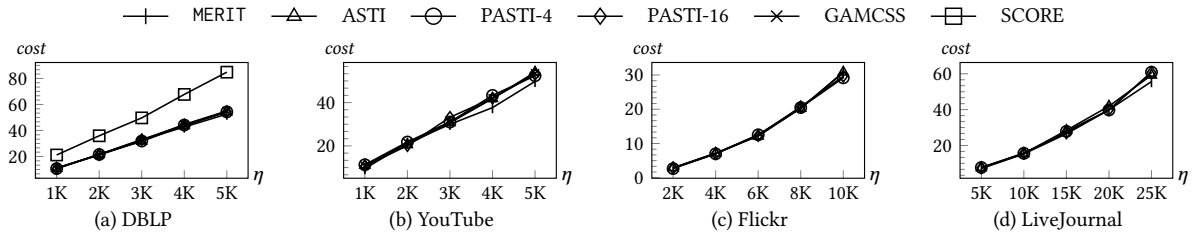
Parameters. We follow the experimental setting adopted in [10, 16, 20, 29]. The estimation error of mRR-sets ϵ is set to 0.5, and the failure probability is $\delta = 1/n_i$ in each round. The diffusion probability of edge $p(u, v)$ is defined as $1/|N_{\text{in}}(v)|$. We also predefined the user cost as $c(u) = c_0 + 0.01 * |N_{\text{out}}(u)|$, following the setting in [4, 8, 14, 17], where c_0 is set to 0.01 to avoid zero-cost nodes. We evaluate all methods under both the IC model and the LT model. For each diffusion model, we randomly generate 10 realizations to test the adaptive algorithms and report the average results.

- Under the IC model, we select 4 nodes per batch on DBLP for all adaptive methods, while select 16 nodes per batch on the three larger datasets to ensure that the single-threaded baseline ASTI can finish within two days. The influence threshold η ranges from 1K to 5K on DBLP and YouTube, from 2K to 10K on Flickr, and from 5K to 25K on LiveJournal, reflecting their different scales.
- Under the LT model, we set the batch size $b = 4$, and vary the influence threshold η ranges adapted to the dataset size. Specifically, the ranges of η vary from 1K to 3K on DBLP and YouTube, from 2K to 6K on Flickr, and from 2.5K to 12.5K on LiveJournal.

GPU Configuration. For the GPU-based algorithms, each thread block is launched with $N_t = 32$ threads, and we employ the maximum possible number of blocks N_b during mRR-set generation to maximize occupancy. During mRR-set updates, we adopt the load-balanced strategy presented in Section **??**. The segments are partitioned into β workload-balanced groups, which are processed sequentially. In each round, we activate N_b/β blocks, with each block consisting of $N_t * \beta$ threads to fully utilize GPU resources. We set $\beta = 8$ for all datasets under both IC and LT models.

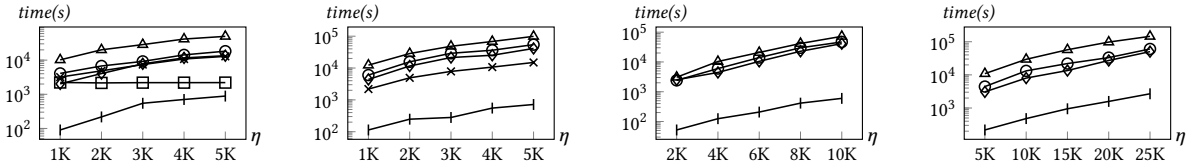
5.2 Results under the IC model

Cost Minimization. Figure 5 reports the seed costs incurred by each method to achieve η under the IC model. We observe that all adaptive methods consistently achieve comparable seed cost, and are lower cost than of the non-adaptive method SCORE on the DBLP dataset. The higher cost of SCORE is due to its inability to observe intermediate results during seeding, which requires it to select more nodes to ensure reaching η. Moreover, SCORE runs out of memory on the other three larger datasets, since tremendous graph sketches need to be generated to provide a probabilistic guarantee, and the sketches are especially large in large datasets. We also notice that `MERIT` achieves a slightly lower cost than other adaptive methods, especially on YouTube and LiveJournal datasets. This is because,



(a) DBLP (b) YouTube (c) Flickr (d) LiveJournal

Figure 5: Seed cost of algorithms under the IC model.



(a) DBLP (b) YouTube (c) Flickr (d) LiveJournal

Figure 6: Running time in seconds of each algorithm under the IC model.

when updating mRR-sets in adaptive rounds, MERIT deliberately retains all previous mRR-sets. However the number of mRR-sets required is typically decreasing over rounds. By doing so, we could utilize more mRR-sets in certain rounds compared to regenerating. This leads to higher-quality seed selection and also improve the seeding decisions in subsequent rounds.

Time Efficiency. We report the running time of all evaluated methods across different influence threshold targets η in Figure 6. The results clearly show that MERIT consistently runs faster than the five baselines. For all methods, the running time generally increases with a larger threshold η because higher targets usually require more rounds to select sufficient seeds. Compared to the strongest baseline GAMCSS which is based on GPU, MERIT delivers speedups of up to 33.9× and 24.2× on DBLP and YouTube, respectively. However, GAMCSS fails to execute on the larger Flickr and LiveJournal graphs due to OOM. The reason is two-fold. First, GAMCSS frequently allocates dynamic memory, causing severe memory fragmentation. Second, GAMCSS temporarily stores the mRR-sets being generated in global memory, doubling the memory demand in the worst case. MERIT method avoids these problems through pre-allocated fixed memory segments, where thread blocks do not require any duplicated storage for mRR-set generation. On the two largest datasets, MERIT demonstrates substantial performance advantages over the strongest CPU baseline PASTI-16. Specifically, MERIT runs 68.9× faster than PASTI-16 on Flickr when η = 10K. On LiveJournal, MERIT consistently achieves more than 10× speedup over PASTI-16 across all tested thresholds. Moreover, MERIT achieves an average 8.8× speedup over the non-adaptive method SCORE, which selects all seeds in a single batch.

5.3 Results under the LT model

Cost Minimization. Figure 7 reports the cost of the selected seeds under the LT model for each evaluated algorithm. The results exhibit trends similar to those observed in Figure 5. Across different influence thresholds η, all adaptive methods incur comparable costs on each dataset. Notably, the non-adaptive method SCORE requires substantially higher cost than the adaptive approaches on DBLP while runs out of memory on larger datasets. Overall, MERIT maintains comparable or lower cost than all baselines, and its advantage becomes more pronounced as η increases across all datasets. This improvement stems from the fact that larger η typically demands more seeding rounds, enabling MERIT to reuse and update mRR-sets from previous rounds rather than regenerating them from scratch. As a result, MERIT effectively leverages a larger set of updated mRR-sets, leading to more accurate estimation of truncated influence.

Time Efficiency. Figure 8 reports the running time under the LT model. The results show that MERIT consistently outperforms all baseline methods across all evaluated datasets. Note that SCORE and GAMCSS fail to produce results for the largest two datasets, due to the same reason discussed in Section 5.2. For all influence targets η, MERIT achieves more than 10× speedup over the strongest competitor. In particular, MERIT attains speedups up to 19.2× on DBLP, 22.3× on YouTube, 14.6× on Flickr, and 15.7× on LiveJournal compared to the fastest baseline. These substantial performance gains arise from our carefully designed GPU optimizations, including independent inter-block mRR-set generation and the efficient selective reuse of unaffected mRR-sets, which together minimize redundant computation and fully exploit GPU parallelism.

5.4 Experimental Analysis

Vary β. We conduct an ablation study to evaluate the performance of the update kernel and the impact of the group-based update strategy by varying the parameter β under both the IC and the LT model. We set $\beta \in \{1, 2, 4, 8, 16\}$ and execute our MERIT method under different influence thresholds. When $\beta = 1$, we do not consider update workload divergence and directly update the previously generated mRR-sets within the designated segment. For $\beta \geq 2$, we enable the update workload balance approach as described in Section **??**. Figure 9 studies the effect of β under the IC model. We observe that the running time usually decreases as β increases from 1 to 8. Specifically, the runtime is reduced by up to 1.3×, 1.4×, 1.4×, 1.3× across the datasets when β increases from 1 to 8. This improvement is achieved because the mRR-sets are effectively updated by blocks with similar workloads. However, as β increases from 8 to 16, the running time starts to increase, because the overhead of excessive

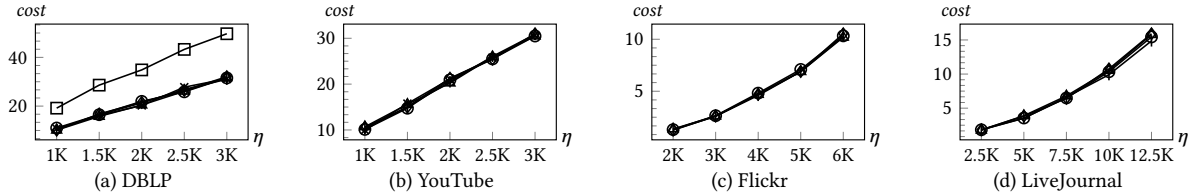


Figure 7: Seed cost of algorithms under the LT model.

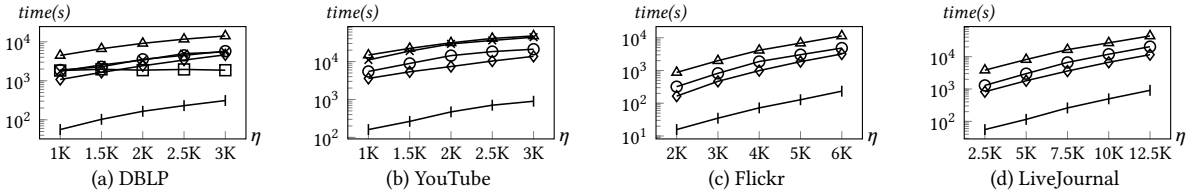


Figure 8: Running time in seconds of each algorithm under the LT model.

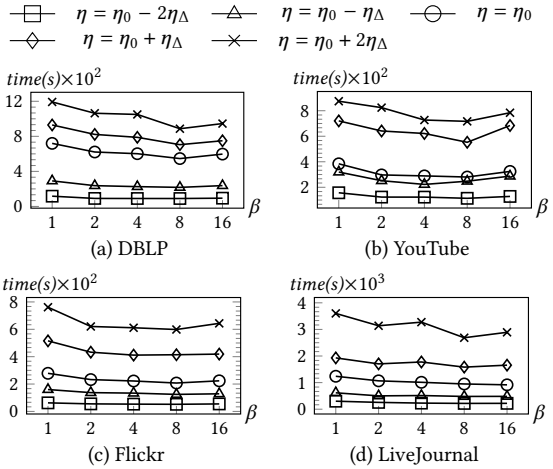


Figure 9: Vary β under the IC model (η_0/η_Δ = 3K/1K, 6K/2K, 15K/5K on DBLP+YouTube, Flickr, LiveJournal).

grouping and block-level synchronization in each round outweighs the marginal load-balancing benefits. We also provide the effect of β under the LT model in the report, and the trends observed are generally consistent with those under the IC model.

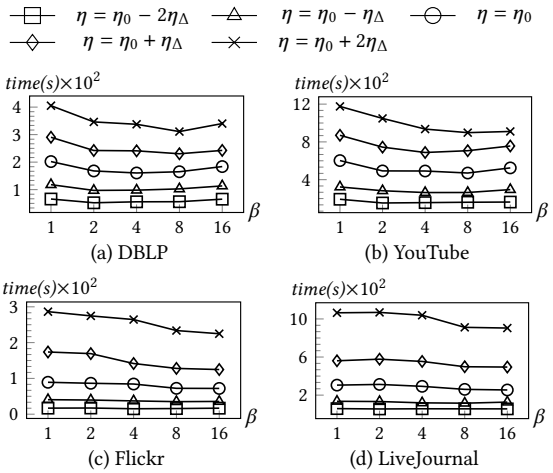


Figure 10: Vary β under the LT model (η_0/η_Δ = 2K/0.5K, 4K/1K, 7.5K/2.5K on DBLP+YouTube, Flickr, LiveJournal).

Time Breakdown. To evaluate the contribution of our proposed techniques, including mRR-set creation, mRR-set update, and others (including seed selection), we decompose the running time of MERIT into these three main components and compare them against a variant where MERIT reduces to solely regenerating the required mRR-sets within the segment in parallel. MERIT is marked with ⋆. Figures 11 and 12 report the results under both the LT and IC models. The results reveal that mRR-set generation dominates the runtime of PASTI-16, as it regenerates mRR-sets from scratch in every round. In contrast, MERIT spends a large portion of execution time on mRR-set updates, sometimes exceeding 90%. This difference in runtime composition reflects that MERIT avoids regenerating mRR-sets from scratch in each round and instead efficiently reuses existing mRR-sets. Overall, MERIT consistently achieves substantial speedups over PASTI-16, reaching up to 41.3× speedup when η = 10K under the IC model on the Flickr dataset. We also provide results on DBLP and YouTube in the report. In general, the results are similar to those on two larger datasets, and MERIT consistently outperforms PASTI-16 in efficiency, validating the effectiveness of our proposed mRR-set generation and update mechanisms.

GPU Profiling. Figure 13 compares detailed performance metrics outputted by the NVIDIA Nsight Compute profiler [1] for two GPU-based methods MERIT and GAMCSS. Specifically, we present the profiling results under the IC model with η = 1K, and focus on the DBLP and YouTube datasets as GAMCSS runs out of GPU memory on two larger datasets. We can observe that MERIT attains 1.8× and 2.1× higher occupancy than GAMCSS. MERIT also requires fewer atomic operations as we independently generate and update mRR-sets within segments. Moreover, MERIT achieves higher instructions executed per cycle and SM throughput than GAMCSS, demonstrating the superior efficiency of MERIT in utilizing the GPU's resources, owing to our algorithmic and implementation optimizations.

6 CONCLUSION

This paper presents MERIT, an efficient GPU-based algorithm for the adaptive minimum-cost seed selection problem. Our contributions include theoretical algorithm design and practical algorithmic designs for mRR-set generation, discovery, and update, as well as GPU optimizations. Extensive experiments demonstrate that MERIT outperforms existing baselines with a large margin and scales to million-node social networks. As future work, we plan to extend MERIT to distributed or multi-GPU settings to solve AMCSS.

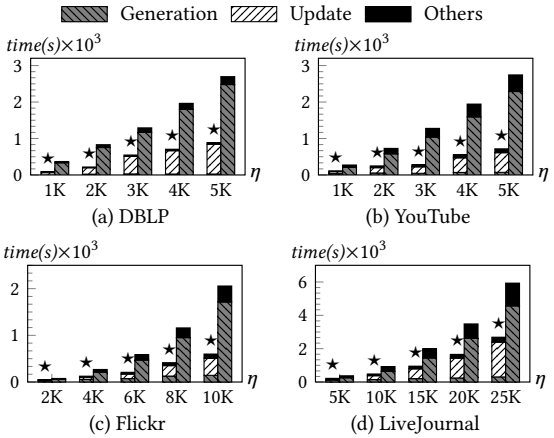


Figure 11: Runtime breakdown under the IC model.

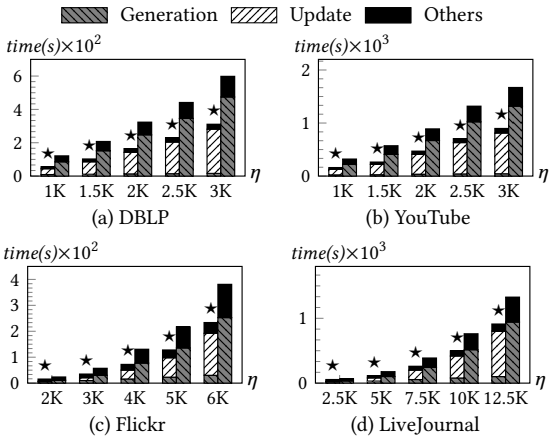


Figure 12: Runtime breakdown under the LT model.

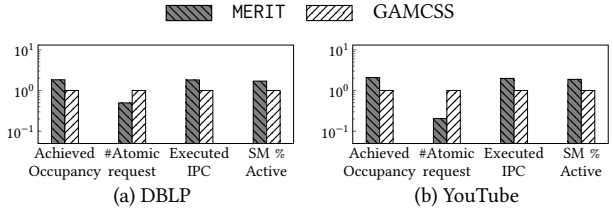


Figure 13: GPU profiling analysis relative to GAMCSS.

REFERENCES

[1] 2023. INSIGHT COMPUTE COMMAND LINE INTERFACE. https://docs.nvidia.com/nsight-compute/pdf/NsightComputeCli.pdf.

[2] 2026. Technical Report. https://github.com/gongyguo/MERIT/blob/main/MERIT_Technical_Report.pdf.

[3] 360iResearch. 2026. Social Media Advertising Market – Global Forecast 2026-2032. (2026), 1–187.

[4] Song Bian, Qintian Guo, Sibo Wang, and Jeffrey Xu Yu. 2020. Efficient Algorithms for Budgeted Influence Maximization on Massive Social Networks. *VLDB* 13, 9 (2020), 1498–1510.

[5] 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.

[6] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[7] Datareportal. 2025. Global social media statistics. https://datareportal.com/social-media-users.

[8] Artem Dogtiev. 2023. Influencer Marketing Costs. https://www.businessofapps.com/marketplace/influencer-marketing/research/influencer-marketing-costs/.

[9] Hossein Esfandiari, Amin Karbasi, and Vahab S. Mirrokni. 2021. Adaptivity in Adaptive Submodularity. In *COLT*, Vol. 134. 1823–1846.

[10] Chen Feng, Xingguang Chen, Qintian Guo, Fangyuan Zhang, and Sibo Wang. 2024. Efficient Approximation Algorithms for Minimum Cost Seed Selection with Probabilistic Coverage Guarantee. *Proc. ACM Manag. Data* 2, 4 (2024), 1–26.

[11] Daniel Golovin and Andreas Krause. 2011. Adaptive submodularity: Theory and applications in active learning and stochastic optimization. *J. Artif. Intell. Res.* 42 (2011), 427–486.

[12] Daniel Golovin and Andreas Krause. 2017. Adaptive Submodularity: Theory and Applications in Active Learning and Stochastic Optimization. arXiv:1003.3967 [cs.LG] https://arxiv.org/abs/1003.3967

[13] Amit Goyal, Francesco Bonchi, Laks VS Lakshmanan, and Suresh Venkatasubramanian. 2013. On minimizing budget and time in influence propagation over social networks. *Social Netw. Anal. Mining* 3 (2013), 179–192.

[14] Qintian Guo, Chen Feng, Fangyuan Zhang, and Sibo Wang. 2023. Efficient Algorithm for Budgeted Adaptive Influence Maximization: An Incremental RR-set Update Approach. *Proc. ACM Manag. Data* 1, 3 (2023), 1–26.

[15] Kai Han, Yuntian He, Keke Huang, Xiaokui Xiao, Shaojie Tang, Jingxin Xu, and Liusheng Huang. 2019. Best bang for the buck: Cost-effective seed selection for online social networks. 32, 12 (2019), 2297–2309.

[16] Kai Han, Keke Huang, Xiaokui Xiao, Jing Tang, Aixin Sun, and Xueyan Tang. 2018. Efficient algorithms for adaptive influence maximization. *VLDB* 11, 9 (2018), 1029–1040.

[17] Joe Hitchcock. 2025. Influencer Pricing: The Cost of Influencers in 2025. https://www.shopify.com/hk-en/blog/influencer-pricing.

[18] Wenjing Hong, Chao Qian, and Ke Tang. 2020. Efficient minimum cost seed selection with theoretical guarantees for competitive influence maximization. *IEEE transactions on cybernetics* 51, 12 (2020), 6091–6104.

[19] Lin Hu, Yinnian Lin, Lei Zou, and M Tamer Özsu. 2025. A graph pattern mining framework for large graphs on GPU. *The VLDB Journal* 34, 1 (2025), 6.

[20] Keke Huang, Jing Tang, Kai Han, Xiaokui Xiao, Wei Chen, Aixin Sun, Xueyan Tang, and Andrew Lim. 2020. Efficient approximation algorithms for adaptive influence maximization. *The VLDB Journal* 29, 6 (2020), 1385–1406.

[21] Andreas Krause and Carlos Guestrin. 2005. *A note on the budgeted maximization of submodular functions.* Carnegie Mellon University. Center for Automated Learning and Discovery .

[22] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[23] Cheng Long and Raymond Chi-Wing Wong. 2011. Minimizing seed set for viral marketing. In *2011 IEEE 11th International Conference on Data Mining*. IEEE, 427–436.

[24] Marco Minutoli, Maurizio Drocco, Mahantesh Halappanavar, Antonino Tumeo, and Ananth Kalyanaraman. 2020. cuRipples: Influence maximization on multi-GPU systems. In *Proceedings of the 34th ACM international conference on supercomputing*. 1–11.

[25] 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.

[26] Linshan Qiu, Lu Chen, Hailiang Jie, Xiangyu Ke, Yunjun Gao, Yang Liu, and Zetao Zhang. 2024. GPU-Accelerated Batch-Dynamic Subgraph Matching. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3204–3216.

[27] Soheil Shahrouz, Saber Salehkaleybar, and Matin Hashemi. 2021. gim: Gpu accelerated ris-based influence maximization algorithm. *IEEE Transactions on Parallel and Distributed Systems* 32, 10 (2021), 2386–2399.

[28] Jieming Shi, Renchi Yang, Tianyuan Jin, Xiaokui Xiao, and Yin Yang. 2019. Real-time top-k personalized pagerank over large graphs on gpus. *Proceedings of the VLDB Endowment* 13, 1 (2019), 15–28.

[29] Jing Tang, Keke Huang, Xiaokui Xiao, Laks VS Lakshmanan, Xueyan Tang, Aixin Sun, and Andrew Lim. 2019. Efficient approximation algorithms for adaptive seed minimization. In *SIGMOD*. 1096–1113.

[30] Jing Tang, Xueyan Tang, Xiaokui Xiao, and Junsong Yuan. 2018. Online Processing Algorithms for Influence Maximization. In *SIGMOD*. 991–1005.

[31] Youze Tang, Yanchen Shi, and Xiaokui Xiao. 2015. Influence maximization in near-linear time: A martingale approach. In *SIGMOD*. 1539–1554.

[32] Ritika Tiwari. 2025. Mastering Social Media Conversion: How to Track and Best Practices. https://www.socialinsider.io/blog/social-media-conversion/.

[33] Guangmo Tong, Weili Wu, Shaojie Tang, and Ding-Zhu Du. 2016. Adaptive influence maximization in dynamic social networks. *IEEE/ACM Transactions on Networking* 25, 1 (2016), 112–125.

[34] Sharan Vaswani and Laks VS Lakshmanan. 2016. Adaptive influence maximization in social networks: Why commit when you can adapt? *arXiv preprint arXiv:1604.08171* (2016).

[35] Letong Wang, Xiangyun Ding, Yan Gu, and Yihan Sun. 2023. Fast and Space-Efficient Parallel Algorithms for Influence Maximization. *VLDB* 17, 3 (2023), 400–413.

[36] Yifei Xia, Feng Zhang, Qingyu Xu, Mingde Zhang, Zhiming Yao, Lv Lu, Xiaoyong Du, Dong Deng, Bingsheng He, and Siqi Ma. 2024. GPU-based butterfly counting. *The VLDB Journal* 33, 5 (2024), 1543–1567.

[37] Qingyu Xu, Feng Zhang, Zhiming Yao, Lv Lu, Xiaoyong Du, Dong Deng, and Bingsheng He. 2022. Efficient load-balanced butterfly counting on GPU. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2450–2462.

[38] Yichao Yuan, Haojie Ye, Sanketh Vedula, Wynn Kaza, and Nishil Talati. 2023. Everest: Gpu-accelerated system for mining temporal motifs. *arXiv preprint arXiv:2310.02800* (2023).

[39] Peng Zhang, Wei Chen, Xiaoming Sun, Yajun Wang, and Jialin Zhang. 2014. Minimizing seed set selection with probabilistic coverage guarantee in a social network. In *KDD*. 1306–1315.

7 NOTATIONS

Table 2: Frequently used notations.

Notation	Description
$G = (V, E)$	the social network with users V and edges E
$G_i = (V_i, E_i)$	the residual graph used for the i-th round
n, m	the number of users and edges, respectively
$N_{\text{in}}(u), N_{\text{out}}(u)$	in-neighbors and out-neighbors of u, resp.
$p(e),\ p(u, v)$	diffusion probability of edge e or from u to v
$\mathcal{R}$	a collection of mRR-sets
θ	the number of mRR-sets
ϵ	the estimation error of $\mathcal{R}$ on $\Gamma(S)$
η	the influence threshold
η_i	the remaining influence to be achieved in G_i
S	the set of seed users
$c(v)$	the cost of selecting a seed v
$I(S)$	a random influence spread of S
ϕ, Φ	a specified realization and a random realization
$\Gamma_i(S)$	the truncated influence of S in G_i
$\hat{\Gamma}(S)$	the estimated truncated influence of S
π	an adaptive seed selection policy
k_i	the number of roots in an mRR-set at round i

8 MISSING PROOFS

8.1 Concentration Inequalities

Let Y_i be an indicator random variable, which takes value 1 if the i−th randomly generated mRR-set is covered by the seed set S, and takes value 0 otherwise. On this basis, we present Lemma 8.1, which is proven in [30].

Lemma 8.1. *Let $\bar{Y} = \frac{1}{\theta}\sum_{i=1}^{\theta} Y_i$ denote the average of the indicator random variables. For any $\lambda \geq 0$, we have*

$$\Pr\left[\mathbb{E}[\bar{Y}]\cdot\theta < \left(\sqrt{\bar{Y}\cdot\theta + \frac{2\lambda}{9}} - \sqrt{\frac{\lambda}{2}}\right)^2 - \frac{\lambda}{18}\right] \leq e^{-\lambda},$$

$$\Pr\left[\mathbb{E}[\bar{Y}]\cdot\theta > \left(\sqrt{\bar{Y}\cdot\theta + \frac{\lambda}{2}} + \sqrt{\frac{\lambda}{2}}\right)^2\right] \leq e^{-\lambda}.$$

The derivation of Theorem 3.1 relies on Lemma 8.2 and Lemma 8.3.

Lemma 8.2. *Given a budget B and a seed set S selected by* MERIT*, if $c(S) = B$ (i.e., the budget is used up exactly), then the coverage of S approximates the seed set S^o with the largest coverage by $\Lambda(S) \geq \rho_b\Lambda(S^o)$.*

Lemma 8.3. *In each round, the seed set S returned by* MERIT *approximates the optimal seed set S^o that takes the same cost as S in the following form*

$$\mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})] \geq \rho_b(1-1/e)(1-\epsilon)\mathbb{E}[\Gamma(S^o|\mathcal{S}_{i-1})].$$

To prove the approximation ratio of MERIT, we consider a dual problem of AMCSS, called truncated influence maximization (TIM). Specifically, we are given some budget B, and select b seeds in each round. The influence considered is truncated by η. In this way, we connect TIM and AMCSS. The aim is to maximize the truncated influence.

For solution to TIM, we adopt the same seed selection process as MERIT in each round. Nevertheless, the stopping condition is not reaching η, but the budget is used up. For the last round of seed selection, the remaining budget B' may be not enough (i.e., $B' < c(S)$). In this case, we select S with probability $B'/c(S)$, to respect the budget constraint in expectation.

Let the policy described above be π^d. To derive a general performance guarantee for π^d, we consider a general budget l, which can be larger than B. Given the approximation ratio of $\rho_b(1-1/e)(1-\epsilon)$ in each round of seed selection, we can derive the approximation ratio of π^d in Theorem 8.4 by applying Theorem 38 in [12], which only relies on the adaptive submodularity and adaptive monotonicity of $\Gamma(\cdot)$ which are proven in [29]. That is, we have $\mathbb{E}[\Gamma(\pi^d)] \geq (1-e^{-\alpha l/B})\mathbb{E}[\Gamma(\pi)]$, where $\alpha = \rho_b(1-1/e)(1-\epsilon)$ and π is an arbitrary policy for TIM.

Theorem 8.4. *Consider any user cost $c : V \to \mathbb{N}$ and any seed selection accuracy $0 \leq \alpha \leq 1$ in each round. For any policy π under budget B, we have*

$$\mathbb{E}[\Gamma(\pi^d)] \geq (1-e^{-\alpha l/B})\mathbb{E}[\Gamma(\pi)].$$

8.2 Proof of Lemma 8.2

By previous work [31], we know that $\Lambda(\cdot)$ is a submodular set function, since it is actually the objective function of the set cover problem. Let S^* be the node set with the largest coverage. Note that S^* is different from S^o, since the definitions of their optimality are different. And we have $\Lambda(S^*) \geq \Lambda(S^o)$. By Lemma 3 of [21], we have

$$\Lambda(S) \geq \left[1 - \prod_{j=1}^{b}\left(1 - \frac{c(s_j)}{B}\right)\right]\Lambda(S^*), \tag{1}$$

where s_j is the j-th seed selected into S. Note that the estimation error in Lemma 3 of [21] is set to 0, since the coverage of any node set in a collection of mRR-sets can be accurately counted.

By the arguments in the proof of Theorem 1 of [21], the approximation ratio in Ineq. 1 takes the minimum value when $c(s_j) = B/b, \forall j$. Thus, we have

$$\Lambda(S) \geq \left[1 - \prod_{j=1}^{b}\left(1 - \frac{1}{b}\right)\right]\Lambda(S^*) = \rho_b\Lambda(S^*) \geq \rho_b\Lambda(S^o).$$

8.3 Proof of Lemma 8.3

The proof idea is similar to the proof in [29]. We first clarify two concepts during the seed selection in MERIT, which involves multiple trials to select a seed. In each trial, MERIT *proposes* a candidate seed set, based on the coverages of nodes. If the candidate seed set is verified to be unqualified, MERIT discards this candidate set, and doubles the mRR-sets to select a new candidate set. Otherwise, MERIT *returns* the selected seed set as the final seed set.

MERIT contains two stopping conditions. First, the number of mRR-sets reaches $\theta_{\max}$; second, the candidate seed set is verified to be legitimate. We next show that, in any case, the seed set returned by MERIT is approximates the optimal seed set by at least $\rho_b(1-1/e)(1-\epsilon)$.

For the first case $\theta \geq \theta_{\max}$, following similar derivation from [29, 30], we can see that $\mathbb{E}[\hat{\Gamma}(S|\mathcal{S}_{i-1})] \geq (1-\hat{\epsilon})\rho_b\mathbb{E}[\hat{\Gamma}(S^o|\mathcal{S}_{i-1})]$ holds with probability $1-\delta/3$. Further, we have

$$\mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})] \geq \mathbb{E}[\hat{\Gamma}(S|\mathcal{S}_{i-1})]$$

$$\geq (1-\hat{\epsilon})\rho_b\mathbb{E}[\hat{\Gamma}(S^o|\mathcal{S}_{i-1})] \geq (1-\hat{\epsilon})\rho_b(1-1/e)\mathbb{E}[\Gamma(S^o|\mathcal{S}_{i-1})],$$

where the first and last inequalities are due to Ineq. (2.3).

For the second case, we consider the following two events for any seed set $S \subseteq V$,

$$E_1(S) : \mathbb{E}[\Lambda(S)] \geq \left(\sqrt{\Lambda(S) + \frac{2a_1}{9}} - \sqrt{\frac{a_1}{2}}\right)^2 - \frac{a_1}{18},$$

$$E_2(S) : \mathbb{E}[\Lambda(S)] \leq \left(\sqrt{\Lambda(S) + \frac{a_2}{2}} + \sqrt{\frac{a_2}{2}}\right)^2.$$

Then, by Lemma 8.1 and the definitions of a_1 and a_2, we have

$$\Pr[\neg E_1(S)] \leq \delta/3HC_{n_i}^b,$$
$$\Pr[\neg E_2(S)] \leq \delta/3H.$$

Note that the seed set S returned by MERIT is random, and can be any node set with size b from V. Thus, we apply the total probability and derive

$$\begin{aligned}\Pr\left[\mathbb{E}[\Lambda(S) < \Lambda^l(S)]\right] &= \Pr[\neg E_1(S)] \\ &= \sum_{S' \subseteq V_i} \Pr[\neg E_1(S')] \cdot \Pr[S' = S] \\ &\leq \frac{\delta}{3H} \sum_{S' \subseteq V_i} \Pr[S = S'] \\ &= \delta/3H.\end{aligned} \tag{2}$$

For the optimal seed set S^o, the event $E_2(S^o)$ directly implies

$$E_2'(S^o) : \mathbb{E}[\Lambda(S^o)] \leq \left(\sqrt{\Lambda(S)/\rho_b + \frac{a_2}{2}} + \sqrt{\frac{a_2}{2}}\right)^2 = \Lambda^u(S^o),$$

since $\Lambda(S) \geq \rho_b \Lambda(S^o)$ by Lemma 8.2 and the r.s.h. of E_2 is increasing with $\Lambda(S^o)$. Note that S^o is unknown but certain. Thus, we do not need to apply the total probability and can write

$$\Pr[\mathbb{E}[\Lambda(S^o)] > \Lambda^u(S^o)] = \Pr[\neg E_2'(S^o)] \leq \Pr[\neg E_2(S^o)] \leq \delta/3H. \tag{3}$$

Combining the above analysis for S and S^o, we can see that, with probability $1 - 2\delta/3H$, the events in Ineqs. (2) and (3) do not hold, which lead to $\mathbb{E}[\Lambda(S)] \geq \Lambda^l(S)$ and $\mathbb{E}[\Lambda(S^o)] \leq \Lambda^u(S^o)$. Further, when MERIT returns S in the second case, we can derive

$$\begin{aligned}&\mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})] \geq \mathbb{E}[\hat{\Gamma}(S|\mathcal{S}_{i-1})] = \mathbb{E}[\Lambda(S)]n_i/\theta \geq \Lambda^l(S)n_i/\theta \\ \geq&(1-\hat{\epsilon})\rho_b\Lambda^u(S^o)n_i/\theta \geq (1-\hat{\epsilon})\rho_b\mathbb{E}[\Lambda(S^o)]n_i/\theta \\ =&(1-\hat{\epsilon})\rho_b\mathbb{E}[\hat{\Gamma}(S^o|\mathcal{S}_{i-1})] \geq (1-\hat{\epsilon})\rho_b(1-1/e)\mathbb{E}[\Gamma(S^o|\mathcal{S}_{i-1})],\end{aligned} \tag{4}$$

where the first and last inequalities are due to Ineq. (2.3). The second and penultimate equalities are due to the definition of $\hat{\Gamma}(\cdot)$. The third and antepenult inequalities are due to Ineqs. (2) and (3). The middle fourth inequality is due to the stopping condition of MERIT. When η is reached, at most H rounds of seed selection are carried out. The probability that Ineq. (4) holds in any round is at least $1 - (2\delta/3H) \times H = 1 - 2\delta/3$.

Although Ineq. (4) fails with very small probability, we still take it into account to derive the performance guarantee of MERIT in expectation, which is taken w.r.t. the randomness ω in seed selection. That is

$$\begin{aligned}&\mathbb{E}_\omega[\mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})]] \\ \geq&(1-\hat{\epsilon})(1-\delta)\rho_b(1-1/e)\mathbb{E}[\Gamma(S^o|\mathcal{S}_{i-1})] \\ \geq&(1-\epsilon)\rho_b(1-1/e)\mathbb{E}[\Gamma(S^o|\mathcal{S}_{i-1})].\end{aligned}$$

This completes the proof.

8.4 Proof of Theorem 3.1

The proof idea is similar to [9]. We prove this theorem by showing that when we spend $(\ln\eta + 1)$ times more cost than the optimal policy, the influence threshold η can be reached with high probability.

Let π^o be the optimal policy. The expected cost of π^o is $\mathbb{E}[c(\pi^o)]$. Next, we apply Theorem 8.4 by letting $B = \mathbb{E}[c(\pi^o)$ and $l = \frac{2\ln\eta}{(1-\epsilon)(1-1/e)\rho_b}\mathbb{E}[c(\pi^o)$, i.e., the ratio times optimal cost. Recall the influence obtained by π^d in Theorem 8.4 and let the arbitrary policy be π^o, we see that

$$\mathbb{E}[\Gamma(\pi^d)] \geq \left(1 - e^{-\frac{\rho_b(1-1/e)(1-\epsilon)l}{\mathbb{E}[c(\pi^o)]}}\right)\mathbb{E}[\Gamma(\pi^o)].$$

Plugging the value of l, we derive that

$$\mathbb{E}[\Gamma(\pi^d)] \geq \left(1 - e^{-2\ln\eta}\right)\mathbb{E}[\Gamma(\pi^o)] = \eta - 1/\eta.$$

Note that $\mathbb{E}[\Gamma(\pi^d)]$ is an expected influence, which can be expressed as

$$\begin{aligned}\eta - 1/\eta \leq \mathbb{E}[\Gamma(\pi^d)] &= \sum_{x=1} x \Pr[\Gamma(\pi^d = x)] \\ \leq&\eta \Pr[\Gamma(\pi^d) = \eta] + (\eta - 1)\Pr[\Gamma(\pi^d) < \eta],\end{aligned}$$

which leads to $\Pr[\Gamma(\pi^d) = \eta] \geq 1 - 1/\eta$.

Note that the policy π^d designed for TIM selects seed sets in the same as MERIT in AMCSS, and the influence function is also truncated. Thus, the influence reached by π^d is the same as MERIT, before the last round of seed selection, where the seed set in π^d is selected probabilistically. To achieve at least the same influence as π^d, we only need to ask MERIT to select the seed set in the last round of π^d with certainty. Then, the cost spent by MERIT in the last round is also certain, resulting at most b more cost than π^d. That is, when the cost used by MERIT is $l + b$, we have $\Pr[\Gamma(\text{MERIT}) = \eta] \geq 1 - 1/\eta$.

On the other hand, with probability at most $1/\eta$, the influence of MERIT is smaller than η. In this case, at most η users are selected, resulting in a cost of η.

Summarizing the above analysis, we can derive the expected cost MERIT needed to reach η

$$\begin{aligned}c(\text{MERIT}) &\leq (l + b)(1 - 1/\eta) + \eta \cdot 1/\eta \leq l + b + 1 \\ &= \frac{2\ln\eta}{(1-\epsilon)(1-1/e)\rho_b}\mathbb{E}[c(\pi^o)] + b + 1.\end{aligned}$$

This completes the proof.

8.5 Proof of Lemma 3.6

For ease of notation, we write $\Delta = k_i - k_{i-1}$. Consider an arbitrary set K_i of k_i nodes in V_i. When generating a fresh mRR-set, the probability of K being selected as the roots is $1/C_{n_i}^{k_i}$, which is uniformly distributed. To establish the lemma, we only need to prove that the root set extended from the previous round is also uniformly distributed, corresponding with K_i with probability $1/C_{n_i}^{k_i}$.

Let $K_{i-1} = K_1 \cup K_2$ ($K_1 \cap K_2 = \emptyset$), where K_1 is the set of inactivated roots preserved from V_{i-1}, and K_2 is the set of roots activated and deleted in the last round. According to Alg. 5, to obtain k_i roots, we need to select $\Delta' = \Delta + |K_2|$ roots from V_i. Combining the Δ'

roots and the preserved roots K_1, we obtain the new root set K_i'. The probability that K_i' corresponds with K_i is

$$\begin{aligned}\Pr[K_i' = K_i] &= \frac{C_{k_i}^{|K_1|}}{C_{n_i}^{|K_1|} C_{n_i-|K_1|}^{\Delta'}} \\ &= \frac{k_i!}{|K_1|!(k_i-|K_1|)!} \cdot \frac{|K_1|!(n_i-|K_1|)!\Delta'!(n_i-|K_1|-\Delta')!)}{n_i!(n_i-|K_1|)!} \\ &= \frac{k_i!(n_i-k_i)}{n_i!} = 1/C_{n_i}^{k_i},\end{aligned}$$

where the second equality is due to $\Delta' + |K_1| = k_i$.

As can be seen, after applying Alg. 5 to update the roots of an mRR-set, the distribution of the roots is the same as directly selecting nodes from the new residual graph, both uniformly distributed.

8.6 Proof of Lemma 3.3

Note that the root distribution of an unpolluted mRR-set R is the same as selecting the roots in the new residual graph. Let the original root set of R be K. The probing result that generates R can be represented as a tree structure T. The fact that R is not polluted indicates that the frontier of the reverse from K (i.e., the leaves) did not probe activated nodes or the activation failed. For each probe, the probability of its success is not affected by whether it is probed in G_{i-1} or G_i. Thus, the probability of reaching the nodes in R is the same as reaching them in G_i. Similarly, for the nodes in V_i that were probed but failed, the probability is also the same. For the failed probes w.r.t. nodes in $V_{i-1} \backslash V_i$, since they are not in V_i, they will not be in R as well. Thus, the composite of R will not change in a new generation in G_i.

When new roots are selected, two cases arise. For the roots that were in R, we argue that the composite of R will not change. For each node $u \in R$, there must be a path from one of the root (e.g. r) to u. When some node v in the path is elevated to be a root. The path from r to v will not change, but the nodes after v (denoted as $O = \{o_1, o_2, \cdots, o_l\}$, where l is the length) may be traversed by some other nodes in R. We can prove that the nodes in O will still be in R, by induction. For the first node o_1, it may be probed by a new node other than v. If the probe is successful, o_1 will stay in R. Otherwise, at least o_1 can be traversed by v. Either way, o_1 will still be in R. Similarly, assuming o_i is in R, we can prove that o_{i+1} is still in R. Thus, the nodes in R will not change.

For new roots K' that were not in R, new reverse sampling will be carried out. The states of the edges explored in the sampling are independent of previous sampling that generated R. Thus, the nodes R_Δ that can be traversed by K' in a fresh probing are the same as those traversed by K' given R.

To summarize, the elements in R after adding roots and reverse sampling have the same distribution as an mRR-set generated from scratch.

8.7 Proof of Lemma 3.4

Lemma 3.6 shows that by replacing the activated root with randomly selected ones, the root set is equal to a newly selected root set. Then, in this lemma, we do not consider the case where there are activated roots.

Assume a random mRR-set R with k_{i-1} roots K, which needs $\Delta = k_i - k_{i-1}$ new roots. When adding new roots into R, to mimic the generation of fresh mRR-sets, we only avoid selecting the roots already in R. Then, some non-root nodes in R may be selected as a root. To accommodate this case, we select some new roots from $R \backslash K$ and some from $V_i \backslash R$. According to Alg. 5, to ensure a uniform sampling, the numbers of roots we select from $R \backslash K$ and $V_i \backslash R$ are proportional to their cardinalities. That is, $\Delta \cdot \frac{|R|-k_{i-1}}{n_i-k_{i-1}}$ roots are selected from $R \backslash K$, and $\Delta \cdot \frac{n_i-|R|}{n_i-k_{i-1}}$ from $V_i \backslash K$. Let the roots selected from $R \backslash K$ (resp. $V \backslash K$) be D_1 (resp. D_2), and $D = D_1 \cup D_2$ ($D \cap K = \emptyset$, $D \subseteq V_i$, and $|D| = k_i - k_{i-1} = \Delta$).

We first prove that the k_{i-1} roots from the previous round can be reused. To see it, we consider an intermediate way of root supplementation. That is, we select Δ roots from V_i/K. For any set of k_i nodes $T \subseteq V_i$, the probability that the new root set equals to T is $C_{k_i}^{k_{i-1}} / C_{n_i}^{k_{i-1}} C_{n_i-k_{i-1}}^{k_i-k_{i-1}}$. By similar derivation in Section 8.5, the probability is equal to selecting k_i fresh roots from V_i directly.

Next, we show that the probability of a node being selected as a root by Lemma 3.4 is the same as the way in Lemma 3.6, where the probability is $\Delta/(n_i - k_{i-1})$. As for our way of selecting roots from $R \backslash K$ and $V_i \backslash K$ separately, the probability of a node $u \in R \backslash K$ being selected as a root is $\frac{1}{n_i-|R|} \cdot \Delta \cdot \frac{n_i-|R|}{n_i-k_{i-1}} = \frac{\Delta}{n_i-k_{i-1}}$. For a node $u \in V_i \backslash R$, the probability is $\frac{1}{|R|-k_{i-1}} \cdot \Delta \cdot \frac{|R|-k_{i-1}}{n_i-k_{i-1}} = \frac{\Delta}{n_i-k_{i-1}}$. Thus, the probability of any node in $V_i \backslash K$ being selected as a root is the same as selecting directly from $V_i \backslash K$.

9 PASTI IMPLEMENTATION

We have incorporated cost-awareness into the single-threaded ASTI framework, but it remains time-consuming in our experiments. The dominant computational cost in each round stems from generating substantial mRR-sets as we show in Figures **??-??**. The original ASTI implementation performs mRR-set generation sequentially while maintaining a *hypergraph* structure. In this structure, each mRR-set records its visited nodes, and an inverted index maps each node to the list of mRR-sets containing it. This inverted index allows for efficient coverage updates when evaluating candidate seeds. However, building the inverted index concurrently introduces data races, as multiple threads may simultaneously append their respective mRR-set identifiers to the shared list of the same node. To maximize parallelism in the mRR-set generation phase, we forgo the inverted index entirely, making this phase fully lock-free.

Furthermore, since seed selection proceeds sequentially across rounds and thus limits parallelism to operations within individual rounds, we propose a multi-threaded variant, PASTI, specifically designed to select a fixed batch of seeds per round, as described in Algorithm 6. Given the residual graph G_i, batch size b, and factor ρ_b, PASTI achieves a $\rho_b(1-1/e)(1-\epsilon)$-approximation guarantee seed selection in each round as `MERIT`. In each round, PASTI first computes the maximum number $\theta_{\max}$ mRR-sets required and starts with a small initial number $\theta_i = \theta$. The number of previously existing mRR-sets θ_{i-1}, is initialized to 0. Threads independently generate mRR-sets with dynamic scheduling [6], continuing until a total of $|\theta_i - \theta_{i-1}|$ mRR-sets have been produced (Line 7). After generating the mRR-sets, we proceed to the seed selection phase. We traverse the newly generated mRR-sets and update the coverage counter *cnt* of each node (Lines 8-10). We also maintain the coverage

Algorithm 6: PASTI

Input: Graph G_i, failure probability $\delta = 1/n_i$, estimation error ϵ, batch size b, node costs $c(\cdot)$.

Output: A $\rho_b(1-1/e)(1-\epsilon)$-approximation seed set S_i, where $\rho_b = 1-(1-1/b)^b$.

1 $\hat{\epsilon} \leftarrow (\epsilon-\delta)(1-\delta)$, $cnt[u] \leftarrow 0\ (\forall u \in V_i)$;

2 $\theta_{\max} \leftarrow 2n_i \left(\sqrt{\ln\frac{6}{\delta}} + \sqrt{\left(\ln\binom{n_i}{b} + \ln\frac{6}{\delta}\right)/\rho_b}\right) \Big/ (\hat{\epsilon}^2 b)$;

3 $H \leftarrow \lceil \log_2 (n_i/(\hat{\epsilon}^2 b)) \rceil + 1$;

4 $a_1 \leftarrow \ln(3H/\delta) + \ln\binom{n_i}{b}$, $a_2 \leftarrow \ln(3H/\delta)$;

5 $\mathcal{R} \leftarrow \emptyset, \theta \leftarrow \theta_{\max} \cdot \hat{\epsilon}^2 b/n_i, \theta_i \leftarrow \theta, \theta_{i-1} \leftarrow 0$;

6 **while** $\theta_i < \theta_{\max}$ **do**

7 **Parallel** generate $|\theta_i - \theta_{i-1}|$ mRR-sets $\mathcal{R}_o$ into $\mathcal{R}$;

8 **for** *each* $R_j \in \mathcal{R}_o$ **do**

9 **parallel for** *each* $v \in R_j$ **do**

10 $cnt[v] \leftarrow cnt[v] + 1$;

11 $cnt'[\cdot] \leftarrow cnt[\cdot], S_i \leftarrow \emptyset, \Lambda(S_i) \leftarrow 0$;

12 **while** $|S_i| < b$ **do**

13 $u \leftarrow \text{argmax}_{v \in V_i} \frac{cnt'[v]}{c(v)}$ by **parallel** reduction;

14 $S_i \leftarrow S_i \cup \{u\}, \Lambda(S_i) \leftarrow \Lambda(S_i) + cnt'[u]$;

15 **if** $|S_i| < b-1$ **then**

16 **for** *each* $R_j \in \mathcal{R}$ **do**

17 **if** $u \in R_j$ **then**

18 **parallel for** *each* $v \in R_j$ **do**

19 $cnt'[v] \leftarrow cnt'[v] - 1$;

20 $\Lambda^l(S_i) \leftarrow \left(\sqrt{\Lambda(S_i) + 2a_1/9} - \sqrt{a_1/2}\right)^2 - a_1/18$;

21 $\Lambda^u(S_i^o) \leftarrow \left(\sqrt{\Lambda(S_i)/\rho_b + a_2/2} + \sqrt{a_2/2}\right)^2$;

22 **if** $\Lambda^l(S_i) > \rho_b(1-\hat{\epsilon})\Lambda^u(S_i^o)$ **then**

23 **return** S_i;

24 $\theta_{i-1} = \theta_i, \theta_i \leftarrow 2\theta_i$;

25 **return** S_i;

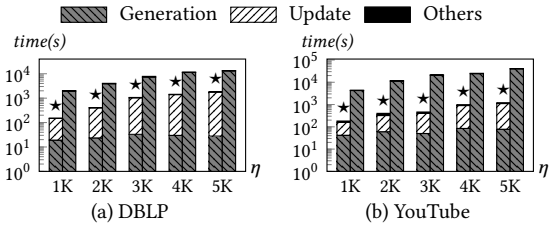
counter for candidate seed selection, and perform seed selection interactively in batches of size b. Within each batch, we perform a parallel reduction over the remaining candidate nodes to append the one with the highest coverage-to-cost ratio to the seed set S_i, and add its coverage to $\Lambda(S_i)$. For each newly selected node, we concurrently iterate over all mRR-sets containing that node and decrement the coverage counts of every node that appears in those sets. This process is repeated until all nodes in the batch are selected. Next, we compute the lower and upper bounds of the $\Lambda(S_i)$ (Lines 20–21). Only when the condition in Line 22 is satisfied do we return the seed set; otherwise, we update θ_{i-1} to current number of mRR-sets θ_i, double θ_i to produce more mRR-sets, and repeat the seed selection process.

Given the optimal policy π^o that spends the same cost in each round as PASTI, the performance guarantee of PASTI is the same as MERIT as presented in Theorem 3.1.

10 ADDITIONAL RESULTS

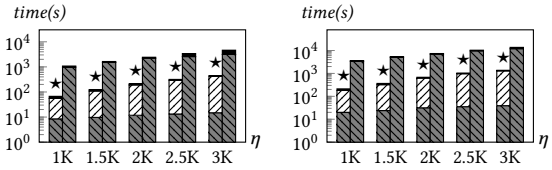
Time breakdown on small datasets. We provide time breakdown results on the DBLP and YouTube datasets under both diffusion models in Figures 14 and 15. As shown, in our method (MERIT) (marked with ⋆), the mRR-set update phase constitutes a large proportion of the total running time, whereas PASTI-16 spends nearly all its time on mRR-set generation on the two datasets, particularly under the IC model and on both datasets. Overall, MERIT consistently achieves superior efficiency compared to PASTI-16, demonstrating the effectiveness of our designed mRR-set generation and update mechanisms.

Varying β under the LT model. Figure 16 illustrates the effect of β under the LT model. The trends observed are generally consistent with those in Figure 9 for the IC model. The running time decreases significantly when β increases from 0 to 1, with speedup achieving up to 1.7× on DBLP, 1.5× on YouTube, 1.3× on Flickr, and 1.2× on LiveJournal. Furthermore, as β increases from 1 to 2, the running time continues to shorten visibly across all datasets, demonstrating our effective load balance approach. When β rises from 2 to 4, the runtime further decreases in YouTube, but the outperformance diminishes on other datasets. At $\beta = 8$, however, we observe an increase in running time for most datasets, primarily due to extra synchronization during the warp-centric probing inside the block and the overhead from frequent kernel launches, which outweighs the work load-balancing benefits. Therefore, we configure $\beta = 2$



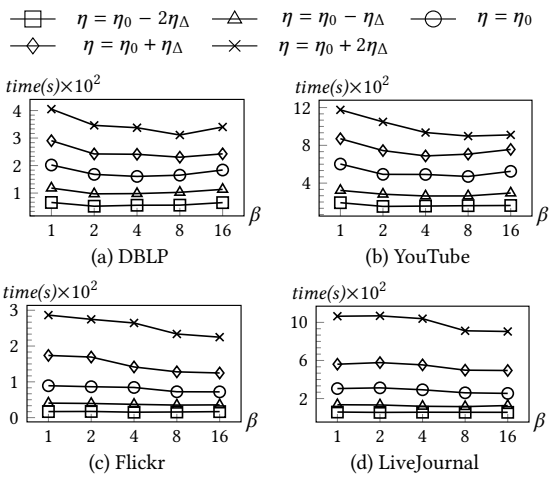
(a) DBLP (b) YouTube

Figure 14: Runtime breakdown under the IC model.



(a) DBLP (b) YouTube

Figure 15: Runtime breakdown under the LT model.



(a) DBLP (b) YouTube

(c) Flickr (d) LiveJournal

Figure 16: Vary β under the LT model (η_0/η_Δ = 2K/0.5K, 4K/1K, 7.5K/2.5K on DBLP+YouTube, Flickr, LiveJournal).

for all datasets under the LT model, as it strikes the best overall balance between load balancing speedup and additional overhead.