FAST AND SCALABLE IMPLEMENTATIONS OF INFLUENCE MAXIMIZATION ALGORITHMS

A PREPRINT

Marco Minutoli*
Pacific Northwest National Lab
Richland, WA
hala@pnnl.gov

Mahantesh Halappanavar
Pacific Northwest National Lab
Richland, WA
hala@pnnl.gov

Ananth Kalyanaraman
Washington State University
Pullman, WA
ananth@wsu.edu

Arun Sathanur
Pacific Northwest National Lab
Richland, WA
arun.sathanur@pnnl.gov

Ryan Mcclure
Pacific Northwest National Lab
Richland, WA
ryan.mcclure@pnnl.gov

Jason McDermott
Pacific Northwest National Lab
Richland, WA
jason.mcdermott@pnnl.gov

August 2, 2019

ABSTRACT

The Influence Maximization problem has been extensively studied in the past decade because of its practical applications in finding the key influencers in social networks. Due to the hardness of the underlying problem, existing algorithms have tried to trade off practical efficiency with approximation guarantees. Approximate solutions take several hours of compute time on modest sized real world inputs and there is a lack of effective parallel and distributed algorithms to solve this problem. In this paper, we present efficient parallel algorithms for multithreaded and distributed systems to solve the influence maximization with approximation guarantee. Our algorithms extend state-of-the-art sequential approach based on computing reverse reachability sets. We present a detailed experimental evaluation, and analyze their performance and their sensitivity to input parameters, using real world inputs. Our experimental results demonstrate significant speedup on parallel architectures. We further show a speedup of up to $586\times$ relative to the state-of-the-art sequential baseline using 1024 nodes of a supercomputer at far greater accuracy and twice the seed set size. To the best of our knowledge, this is the first effort in parallelizing the influence maximization operation at scale.

***K*eywords** Influence Maximization · Parallel Algorithm · Graph Algorithms · Large Scale Data Analytic

1 Introduction

Given a graph, the objective of the *influence maximization* problem is to identify a small set of k *seed* vertices, which when activated, will likely result in the activation of the maximum number of vertices in the input graph, based on a given diffusion model for influence. The problem was first introduced in 2001 by Domingos and Richardson [1], and was formulated as a combinatorial optimization problem by Kempe *et al.* [2], who also proved that the problem is NP-hard. We review relevant algorithmic work in Section 2.

Influence maximization is better equipped at capturing the dynamics of information diffusion on a complex real-world network, than more traditional centrality-based approaches. Consequently, the operation has found many applications in multiple network science domains that critically depend on identifying influential actors in a network (e.g., viral marketing, influential proteins in a molecular network).

Since the seminal work of Kempe *et al.*, significant strides have been made in improving the performance of influence maximization algorithms, either through exploiting special properties of the problem such as its submodular structure

*Also with Washington State University, Pullmann, WA

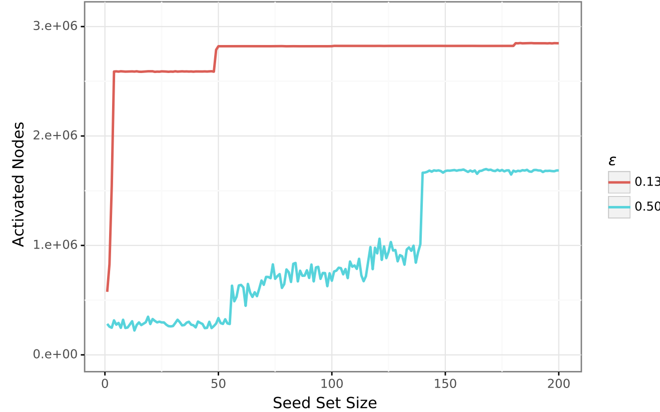


Figure 1: Number of activated nodes (higher is better) on the Orkut graph as a function of the seed set size (k) and approximation quality (ϵ). The blue arc is representative of what the current state-of-the-art can achieve. The red arc is from the parallel implementation presented in this paper. Our parallel implementation enables to improve quality while *also* reducing time to solution (Table 3).

(e.g., [3]) or through devising randomized techniques that provide approximation guarantees on the quality of the solution (e.g.,[4], [5]). A more extended review of these works is presented in Section 2.

Despite these advancements in algorithms, there exists little to no work on efficiently parallelizing the algorithms. State-of-the-art implementations support only a small degree of parallelism, if any, using multithreading at the node level where possible. Parallelism is essential for scaling up influence maximization for a variety of reasons. The size scales of the real-world networks on which the operation needs to be performed, have dramatically increased, with millions to even billions of vertices becoming commonplace. However, it is not just the input size that is a deterrent to performance. Most algorithms for influence maximization rely on approximation; and as the quality of the targeted approximation (ϵ) is made higher, the running time of the algorithms also significantly increase. Furthermore, users typically have to test multiple k values (the seed set size) before identifying an optimal configuration that can maximize their "return on investment" on the seeds. However, the runtime of these algorithms is also sensitive to increases in k. Finally, if the edge probabilities used in the diffusion mechanism are not readily available from the application domain, the burden of experimenting with different edge probability distributions falls on the shoulders of the computational scientist. A fast and scalable parallel implementation that can enable multiple runs of influence maximization at scale is therefore essential for furthering the reach of the algorithms.

In this paper, we present the design and development of efficient parallel implementations for carrying out influence maximization at scale, on both shared and distributed memory platforms. Our parallel implementations are for a state-of-the-art influence maximization algorithm, namely IMM [5]. IMM represents a significant improvement over its predecessors in achieving scale of the network (millions of nodes) and performance. Using IMM as our sequential baseline for comparison, we demonstrate our parallel approach's ability to solve large-scale problems with unprecedented accuracy and performance. Figure 1 illustrates the case in point using an example; it shows the improvement that our parallel implementation (red arc) can achieve compared to the state-of-the-art (blue arc).

In summary, we make the following contributions:

- We introduce new shared and distributed memory algorithms for influence maximization based on the work of Tang *et al.* [5].
- We empirically evaluate the performance and scalability of our implementations on two compute clusters, first with large per-node memory and second, a massively parallel processor.
- We provide a new tool to the research community that enables the study of influence spread on real-world networks, with significantly higher approximation guarantees achieved in orders of magnitude shorter time-to-solution. For instance, on an Orkut network, we demonstrate a speedup of up to $586\times$ with respect to the serial IMM implementation.
- We show a novel application of influence maximization on two case studies from biology: metabolomic and metatranscriptomic data examining the microbial community of a soil ecosystem, and proteomic and transcriptomic data derived from tumor samples from human patients. The findings in this case-study vali-

date the high utility and relevance of our influence maximization implementation compared to other existing approaches.

To the best of our knowledge, this is the first effort in parallelizing the influence maximization operation at scale.

2 Related Work

The seminal work by Kempe *et al.* [2] formalized the problem of influence maximization. The approach used in their work involves a greedy hill-climbing optimization (submodular maximization) procedure operating on an oracle that computes the expected reachability from a set of candidate nodes over a given network. The expectation itself is computed by harnessing a large number of Monte Carlo trials (10000 being a typical number used in literature) while reachability computations involve BFS (Breadth First Search) kernels. Thus, the combined complexity of the overall flow meant that it could be run only on small networks ($<$ 10k nodes). Thus, one of the main themes of the research on the topic of influence maximization following [2] is addressing the scalability aspects of the overall flow. Two sub-themes along this line of research focus on accelerating the submodular optimization approach and accelerating the oracle computation.

Leskovec *et al.* [3] adopt a lazy-greedy submodular maximization approach called CELF (Cost-Effective Lazy Forward). In this approach, by making use of the submodular nature of the reachability objective function, they save on additional computations of the marginal gain [6] that do not contribute to the selection of the top node as the iterations proceed. This method is exact when compared to the greedy algorithm and provides the same approximation guarantee of $(1 - \frac{1}{e})$ (63%). The work by Goyal *et al.* [7] improved on the CELF algorithm, again by exploiting the submodular nature of the objective function and resulted in the CELF++ algorithm. Chen *et al.* [8] combine two complementary approaches, namely an improvement over the CELF algorithm and the introduction of a heuristic called degree discounting. While their heuristics show excellent speedups in run times on relatively large datasets, the introduction of the heuristic also results in the inability to provide approximation guarantees. Chen *et al.* [9] propose a heuristic algorithm based on local arborescence (trees in a directed graph with all edges pointing either towards the root or away from the root) and shortest path algorithms. The authors also demonstrate the scalability of their methods to graphs with millions of edges. The concept of per-node summary structures called combined reachability sketches are leveraged by Cohen *et al.* [10] to speedup the influence computations, resulting in up to two orders of magnitude speedups. Borgs *et al.* [11] introduce the concept of collections of reverse reachable paths; in conjunction with the greedy algorithm operating on the hyper-edges constructed as reverse reachable sets, they provide a near-linear time algorithm to mine for influential nodes. This concept was further developed by Tang *et al.* [4] where they provide an efficient practical implementation of the algorithm presented in [11] resulting in the ability to analyze billion-edge graphs in hours. Note that many of these algorithms work mainly with the Independent Cascade model of local influence.

Another research trend in accelerating the influence maximization flow is to leverage the modular nature or the community structure present in real-world graphs. These approaches mine for influential nodes in the communities which are dense subgraphs. Wang *et al.* [12], at each step of finding the k seeds, first select a community that results in maximum marginal influence gain and then mine the corresponding influential node from that community. In another work, Chen *et al.* [13] first perform community detection as a pre-processing step and then select candidate seeds based on the community features and heuristics followed by a pruning step that finalizes the overall seed set. In a more recent work, Halappanavar *et al.* [14] use community detection, a proportional seed allocation heuristic and OpenMP-based parallelization of the Monte Carlo iterations to achieve scalability. A major shortcoming of these methods is the inability to include the effects of inter-community edges since the subgraphs are disjoint.

One particular area of research along the lines of scalable influence maximization that has not received much interest is the parallelization of the underlying components. Kim *et al.* [15] propose a new algorithm called the Independent Path Algorithm and leveraging OpenMP pragmas, they demonstrate scalability ranging from $3\times$–$6\times$ on 8 cores. Du *et al.* [16] report parallel running times of their improved continuous-time influence maximization algorithm but they do not provide any details on the scaling behavior. Finally, Lucier *et al.* [17] use sampling approaches to provide for scalable implementations of influence maximization that are amenable to distributed processing paradigms such as MapReduce and Wu *et al.* [18] propose a parallel algorithm using k-shell decomposition. However, a thorough evaluation of the parallel performance is missing.

3 Parallel Algorithms

Problem Statement Let $G = (V, E)$ be a directed graph with n vertices in V and m edges in E. A network diffusion model M is a stochastic process that defines how information originating from a small subset of initially "active" vertices (called the *seed set* and denoted by S) spreads through the rest of the network. Each edge $e \in E$ from

Table 1: Notation

	Description
n, m	The number of vertices and edges in G, respectively
p	The number of ranks (i.e., no. threads or no. processes)
θ	The number of Random Reverse Reachable sets to be computed
ϵ	The parameter controlling the approximation factor
S	The seed set of k vertices for initial activation
R_v	A Random Reverse Reachable set obtained from source vertex v
$\mathbb{R}$	A collection of Random Reverse Reachable sets
R_i	The ith Random Reverse Reachable set in $\mathbb{R}$
$\mathbb{R}_i$	A subset (partition) of $\mathbb{R}$ stored at (process or thread) Rank i
LT	Linear Threshold model for diffusion
IC	Independent Cascade model for diffusion

u to v is associated with an activation probability of $p(e)$. Intuitively, the diffusion process specifies how a vertex gets activated based on the state of its neighbors. We consider two models: (i) Linear Threshold (LT) model where a vertex can get activated if a fraction of neighboring vertices that are active is greater than a threshold, and (ii) Independent Cascade (IC) model where there is a one-shot chance for an activated vertex to activate its neighbor. We refer you to Kempe *et al.* [2] for details.

From an implementation perspective, the diffusion process can be described as a probabilistic variant of the Breadth First Search (BFS) from S. We use A_i to denote the set of vertices in G activated *during* time step t_i; initially, $A_0 = S$. To compute A_i, where $i > 0$, the following procedure is used. For each vertex $u \in A_{i-1}$, consider every (outgoing) edge e of the form (u, v) such that v is not active (yet). Then, for each such edge e, v is added to the active set A_i with a probability of $p(e)$. Once activated, a vertex remains active for all remaining time steps. The algorithm proceeds until the time step when no more vertices are activated. Let t_c denote the time step at which this *convergence* criterion is achieved (i.e, $A_c = \emptyset$). Then, the *influence set* of the seed set S in G, denoted by $I(S)$, is given by $\bigcup_{i=0}^{c} A_i$.

Definition 1 (The Influence Maximization Problem)**.** *Given a graph G, a network diffusion model M, and a positive integer k, the influence maximization problem is to identify a seed set S of cardinality k such that $\mathbb{E}[|I(S)|]$ is maximized.*

IMM In this section, we describe the IMM algorithm [5]. We consider the diffusion model M described in Section 3. Table 1 shows the notation we use in this paper.

Definition 2 (Reverse Reachable Set)**.** *Let v denote a vertex in G and g denote a graph obtained by removing each edge e in G with a probability of* $1 - p(e)$*. Then, the* reverse reachable set *for v in g, denoted by $RR_g(v)$, is given by:*

$$RR_g(v) = \{u \mid \exists \text{ a directed path from } u \text{ to } v \text{ in } g\}$$

Definition 3 (Random Reverse Reachable Set)**.** *A* random reverse reachable (RRR) set *for v, denoted by R_v, is an $RR_g(v)$ where g is randomly sampled graph, drawn from a distribution of graphs induced by the randomness of edge removals.*

In other words, if a vertex u appears in R_v, then u has a chance to influence v. Consequently, the larger the number of random reverse reachable sets a given vertex u appears in, the larger its influence is expected to span in the input network. In [11], Borgs *et al.* take advantage of this intuition to construct a random sampling based algorithm, called *Reverse Influence Sampling* or *RIS*. Their algorithm first generates a certain number (denoted by θ) of random reverse reachable sets (as samples[2]), and then computes a set of k seed vertices that cover the maximum number of those samples. Here, the number of samples to generate, θ, depends on a user-defined threshold defined over the number of vertices and edges visited during the sample generation process. We use $\mathbb{R}$ to denote the set of θ samples. The *RIS* algorithm is a randomized algorithm with an approximation guarantee of $(1 - 1/e - \epsilon)$.

The IMM algorithm [5] is an approximation algorithm that extends the RIS algorithm by removing the need for the use of a threshold. Instead, the algorithm provides a way to estimate θ. The estimation process depends on the input graph G, the seed set size k, and the parameter ϵ. Tang *et al.* [5] prove that IMM provides $(1-1/e-\epsilon)$-approximate solution with probability at least $1 - 1/n^l$ (l is set to 1 in [5]). Algorithm 1 shows the major steps of the IMM algorithm. In what follows, we describe our parallel algorithm for IMM.

[2]Henceforth, we use the terms "sample" and "RRR set" interchangeably.

Algorithm 1: InfluenceMaximization

```
Input  : G, k, ε
Output: S
begin
    ⟨ℝ, θ⟩ ← EstimateTheta(G, k, ε)
    ℝ ← Sample(G, θ − |ℝ|, ℝ)
    S ← SelectSeeds(G, k, ℝ)
    return S
```

3.1 Parallel IMM

In order to effectively scale the influence maximization algorithms, we need to address multiple key challenges. These challenges will become clear when describing the individual steps of the algorithm, but at the outset they emerge from: i) the need to explicitly store the RRR sets, thereby involving a significantly larger memory footprint compared to input; ii) the need to synchronize among the process or thread ranks during the greedy iterative selection of k seeds; and iii) the complexities that arise in the context of a distributed memory implementation, including the need to efficiently generate random numbers in parallel and in optimizing communication.

In this section, we present our approach to overcome these challenges toward a scalable implementation of influence maximization. Wherever possible, we express our algorithms as generic parallel pseudocodes without assuming a particular parallel programming model. The details of implementing these algorithms on shared memory should be more evident from these pseudocodes. Section 3.2 expands on the details for our distributed memory implementation.

The key phases of the algorithm are: i) estimation of θ, ii) computation of θ samples ($\mathbb{R}$), and iii) selection of the k most influential seeds (S), as illustrated in Algorithm 1.

Estimation Algorithm 2 shows the algorithm for estimating θ. The algorithm iterates by first computing an initial estimation, $\hat{\theta}$, followed by the computation of $\hat{\theta}$ reverse reachable sets originating from vertices selected at random by invoking Algorithm 3. The algorithm then selects a seed set, S, by invoking Algorithm 4. The analysis in [5] shows that $F_{\mathbb{R}}(S)|V|$ is an unbiased estimator of the unknown optimum (OPT), where $F_{\mathbb{R}}(S)$ is the fraction of reverse reachable sets covered by the seed set S computed by Algorithm 4. The martingale strategy in Algorithm 2 uses the estimator to establish a lower bound (LB) on OPT. The lower bound LB is then used to derive an tighter bound on θ. We refer the reader to Tang *et al.*[5] for further details.

Sampling A significant portion of the execution time is spent in the computation of the reverse reachable sets illustrated in Algorithm 3. While the computation can be easily parallelized by generating each R_i in parallel, the key challenge emerges from the large memory footprint in storing the reverse reachability information ($\mathbb{R}$). As illustrated in Figure 2, θ grows nonlinearly with the number of vertices and target accuracy value (ϵ). Previous implementations [5] store this information in two directions using the notion of a hypergraph, where each RRR set (or sample) is a

Algorithm 2: EstimateTheta

```
Input  : G, k, ε
Output: ⟨ℝ, θ⟩
begin
    for x ∈ 1 → log |V| do
        θ̂ ← f(x, k, ε, |V|)                          % Refer to Tang et al. [5]
        ℝ ← Sample(G, θ̂ − |ℝ|, ℝ)
        S ← SelectSeeds(G, ℝ)
        if F_ℝ(S) > 1/2^x then
            LB ← F_ℝ(S)|V| / √2
            break
    θ ← f′(k, ε, |V|, LB)                            % Refer to Tang et al. [5]
    return ⟨ℝ, θ⟩
```

hyperedge consisting of a subset of vertices in the input graph. Information for each vertex about the samples that it participates in is also maintained. Thus, each association between a sample and a vertex is stored twice. While this information aids in faster selection of seed set later, the memory footprint can become a limitation. Therefore, we only store the information in one direction, where each sample in $\mathbb{R}$ is stored as a list of vertices in the corresponding RRR set—sorted by the vertex ids.

The key operation in Algorithm 3 is the call to function `GenerateRR` for a given source vertex v, that traverses incoming edges from their destination to their source. The insertion policy into the next frontier varies according to the diffusion model (IC vs. LT in our implementation). The BFS is performed probabilistically over random graph samples g generated from G. Our implementation achieves this without explicitly generating each g and instead choosing to probabilistically select each edge as the traversal progresses. The function returns the vertices traversed sorted by ids. Sorting the traversed vertices has two main advantages during the seeds selection. First, it ensures that each R_i set stores the partition $\{v_l, \ldots, v_h\}$, assigned to a rank P_t, in contiguous memory so that the counting steps in Algorithm 4 will proceed in cache order. Second, v_l and v_h can be efficiently found using binary search and, therefore, each rank P_t can avoid to traverse R_i entirely to check for vertices in the partition assigned to it.

Seed set generation The last phase of the IMM algorithm is the computation of the seed set. We first describe our multithreaded implementation in Algorithm 4. The selection algorithm is a two-step procedure. First, we maintain a counter for each vertex to keep track of the number of samples any given vertex is part of. This counter is first computed by scanning through the list of θ samples in $\mathbb{R}$. To avoid synchronization during this step, we evenly partition the set of n vertices across the p threads such that each thread rank takes ownership of updating the counters for a distinct interval of vertices $[v_l, v_h]$. This way we avoid synchronization (e.g., alternative would have necessitated atomic updates). Note that each thread rank still has to visit all samples in R. However, since the vertex lists are kept sorted (by vertex ids), navigating to identify vertices that fall in the interval of a rank can be quickly accomplished using binary search.

The second step in seed selection is a greedy iterative procedure which identifies one seed per iteration (over k iterations), by selecting the next vertex that has the maximum counter. However, when a vertex u is identified as a seed, then all RRR sets (samples) that contain that vertex u should be removed from $\mathbb{R}$ because those RRR sets become redundant—i.e., each such RRR set can no longer contribute to increasing the overall influence set size. Consequently, with the removal of such RRR sets, the counters for all vertices that exist in those RRR sets also need to decremented. The `while` loop in Algorithm 4 shows our parallel approach for this step. Here again, we use the idea of partitioning the workload of vertex-level counter updates, with each thread rank assuming responsibility to update the counters for a distinct interval $[v_l, v_h]$—thereby eliminating a need to synchronize across thread ranks.

3.2 Distributed Implementation

For the distributed implementation, we first observe that the number of samples (θ) can grow significantly larger than the input number of vertices (n), as the estimation procedure depends primarily on the target accuracy ϵ and number of seeds k. Empirical evidence of this nonlinear growth is shown in Figure 2. Consequently, the sampling procedure (Algorithm 3) tends to be a dominant contributor to the overall execution time in practice. Given this observation, we devised a distributed memory approach that targets scaling up the enumeration of samples by evenly partitioning the samples to be generated among the p ranks, such that each rank is responsible to generate a distinct batch of $\frac{\theta}{p}$ samples, $\mathbb{R}_i$. In order to compute local samples, each MPI process stores the entire input graph and performs the probabilistic BFS starting from a source vertex chosen uniformly at random. Therefore, accurate generation of pseudorandom numbers in parallel is critical to guarantee the approximation bounds of the algorithm. We employ the

Algorithm 3: Sample

```
Input  : G, θ, ℝ
Output: ℝ
begin
    for i ∈ 1 → θ do in parallel
        Select v ∈ V uniformly at random
        R_i ← GenerateRR(G, v)
        ℝ ← ℝ ∪ {R_i}
    return ℝ
```

Algorithm 4: SelectSeedsMultiThreaded at all ranks $P_t, t \in \{1, \ldots, p\}$

```
Input  : G, k, ℝ
Output: S
v_l ← (|V| * t)/p
v_h ← (|V| * (t + 1))/p
foreach R_j ∈ ℝ do
    foreach u ∈ R_j : v_l ≤ u < v_h do
        counter[u] ← counter[u] + 1
S ← ∅
while |S| < k do
    v ← argmax_{x∈{1,...,|V|}} counter[x]                    % par. reduce
    if i=0 then
        S ← S ∪ {v}
        counter[v] ← 0
    foreach R_j ∈ ℝ do
        if v ∈ R_j then
            foreach u ∈ R_j : v_l ≤ u < v_h do
                counter[u] ← counter[u] - 1
            if i=0 then
                ℝ ← ℝ \ {R_j}
return S
```

linear congruential generator for this purpose by splitting the sequence between ranks using the Leap Frog method implemented in TRNG [19]. The pseudorandom numbers are generated as needed to compute $\mathbb{R}_i$ in parallel.

For the seed selection phase, the distributed implementation initializes an array of n counters—one for each vertex—on each process. Next, the frequencies of all vertices covered in $\mathbb{R}_i$, are computed. These local counts are subsequently aggregated into global counts across all p processes, using the MPI `All-Reduce` collective operation. Subsequently, each process identifies the k seeds, one vertex per iteration, over k iterations. Identifying the next vertex seed in a given iteration is a strictly local operation as all processes have the global counts for all vertices. The subsequent process of removing all RRR samples that contain the identified seed vertex is also a local operation, as each process only needs to purge the redundant RRR sets from its local partition $\mathbb{R}_i$. However, as this purge is carried out, the counters for the vertices contained in those RRR sets will need to be decremented. This is achieved in two steps: i) decrement the local counters for each vertex, and ii) perform an `All-Reduce` to update the global counts of all n vertices.

Consequently, the dominant communication of the distributed implementation is due to the `All-Reduce` operations that happen at each iteration—yielding a net communication complexity of $\mathcal{O}(kn \lg p)$. The sampling procedure, on the other hand, is memory-bound with a complexity of $\mathcal{O}(\frac{\theta}{p}m)$ (since the cost of generating each sample through a probabilistic BFS is $O(m)$). In our experiments, we found the cost of sampling far exceeds the cost of seed selection (Section 4.2). On each process rank, our implementation employs the division of vertex set as shown in Algorithm 4 using a hybrid MPI and OpenMP programming model.

4 Experimental Evaluation

Experimental Setup For empirical evaluation presented in this work, we used used eight networks from the Stanford Large Network Dataset Collection (SNAP) collection [20]. Table 2 summarizes the key features of this dataset.

All experiments were conducted on two compute clusters. On the first cluster named **Puma**, each node is equipped with two 10-cores Intel Xeon E5-2680 v2 CPUs running at $2.8GHz$ (hyper-threading disabled) and has $768GB$ memory. Nodes on Puma are connected through a single Infiniband FDR $4\times$ switch. The second cluster is **Edison** at the NERSC facility. Each node on Edison consists of two 12-cores Intel "Ivy Bridge" CPUs running at $2.4GHz$ with the option of using hyper-threading and has $64GB$ memory. Nodes on Edison are connected through the Cray Aries high-speed interconnect with Dragonfly topology.

Table 2: Serial execution time and memory usage of IMM vs IMM_{OPT} ($\epsilon = 0.5$, $k = 50$)

Graph	Nodes	Edges	Avg. Degree	Max Degree	IMM (s)	IMM_{OPT} (s)	Speedup	IMM (MB)	IMM_{OPT} (MB)	% savings
cit-HepTh	27,770	352,807	12.70	2,468	8.00	2.84	2.82×	357.23	190.80	46.59%
soc-Epinions1	75,879	508,837	13.41	3,079	41.59	14.62	2.84×	2198.25	1170.05	46.77%
com-Amazon	334,863	925,872	5.53	549	521.04	188.48	2.76×	19222.59	10927.92	43.15%
com-DBLP	317,080	1,049,866	6.62	343	526.82	170.32	3.09×	13260.18	5547.77	**58.15%**
com-YouTube	1,134,890	2,987,624	2.63	28,754	1592.08	511.77	3.11×	49710.07	25785.04	48.13%
soc-Pokec	1,632,803	30,622,564	37.51	20,518	5552.37	2350.27	2.36×	63210.72	51643.09	18.30%
soc-LiveJournal1	4,847,571	68,993,773	28.47	22,889	16434.81	3954.59	**4.16**×	○	64501.89	○
com-Orkut	3,072,441	117,185,083	76.28	33,313	28024.56	9027.50	3.10×	○	○	○

Our C++ implementation for shared memory machines uses the OpenMP programming model, while the implementation for distributed memory machines uses a hybrid MPI plus OpenMP programming model. Both implementations are publicly available as part of the Ripples framework [21]. In our experiments, programs were compiled using GCC 7.3.0 with optimizations enabled using "`-O3`" switch. Parallel implementations are enabled using the default OpenMP (libgomp) and MPI libraries (Puma: OpenMPI 2.1.1, Edison: Cray mpich 7.7.3) provided on the system. Finally, the edge weights for probabilistic BFS are generated uniformly at random in the range $[0; 1]$. For the linear threshold (LT) diffusion model, the weights are readjusted such that the sum of the probabilities of traversing one of the neighboring edges and of not traversing any of them, is one; we implement an equivalent model as described in Kempe *et al.* [2] for this purpose. Alternatively, for the Independent Cascade (IC) model, the probability of diffusion from an active vertex to any of its inactive neighbors is a constant and independent of history, and therefore, probabilities are used without any modifications. Since edges are chosen without any restrictions in the IC model, significantly larger number of edges need to be traversed for the IC model relative to the LT model. We see the impact of runtime presented later in this section.

Sequential Baseline Construction A sequential implementation of IMM algorithm is available from Tang *et al.* [5]. We refer to this implementation as IMM. Our preliminary experiments with this implementation suggested high runtime and memory demands. Consequently, we implemented our own native version of the IMM algorithm to use not only as a sequential baseline but also to construct the parallel implementations. We refer to our serial implementation as IMM_{OPT}. We compared the two sequential implementations and the results are shown in Table 2. We instrumented the code in order to measure the peak memory footprint using Valgrind's Massif tool and, as a consequence of the instrumentation overhead, we were not able to run it for larger instances. We have marked these instances with an $\circ$ symbol in Table 2. We observe that, compared to IMM, our serial implementation IMM_{OPT} is 2 – $4\times$ faster and consumes 18 – 58% less memory. These improvements in both performance and memory footprint can be attributed to the compact representation of the random reverse reachability sets in our implementation, as described in Section 3.1. We validated the correctness of our implementation by comparing the outputs of both the code using similar parameters and observed high rank-biased overlaps of the two outputs. Minor differences arise due to different pseudorandom number generation skemes but we observed similar total activation numbers such as the one presented in Figure 1.

We use the baseline serial implementation for the multithreaded (OpenMP) implementation. Key differences arise in the construction of $\mathbb{R}$ and in the seed selection phase. The multithreaded implementation forms the basis for the distributed hybrid MPI+OpenMP implementation.

4.1 Effect of Parameters

The number of RRR sets to generate (θ) is a function of the number of seeds, k, and accuracy, ϵ. We illustrate this relationship in Figure 2 for input `citHepTh` for a range of ϵ and k values. As can be observed, θ quickly grows as the target precision is increased (i.e., lower ϵ) or if a larger k is used. It is also noteworthy that θ quickly exceeds n (the number of vertices in the input graph).

Consequently, these two parameters, k and ϵ also affects the runtime. In Figure 3, we present the performance results by varying ϵ for a fixed value of k; and in Figure 4 by varying k for a fixed value of ϵ. A larger value of ϵ implies lower accuracy and therefore a smaller runtime. The total runtime in both the figures is shown decomposed into four sections—*Estimation* (Algorithm 2), *Sample* (Algorithm 3), *SelectSeeds* (Algorithm 4) and other—that map to the corresponding phases of Algorithm 1. While studying these figures, note that there are two sets of invocations to the *Sample* function—one from within the *Estimation* function (Algorithm 2), and another directly from the main skeleton of the algorithm (Algorithm 1). The "Sample" bars shown in all the runtime performance figures of this section *only accounts for the latter invocations from the skeleton*; whereas the cost of the calls to *Sample* from within the *Estimation* function are included as part of the "Estimation" bars.

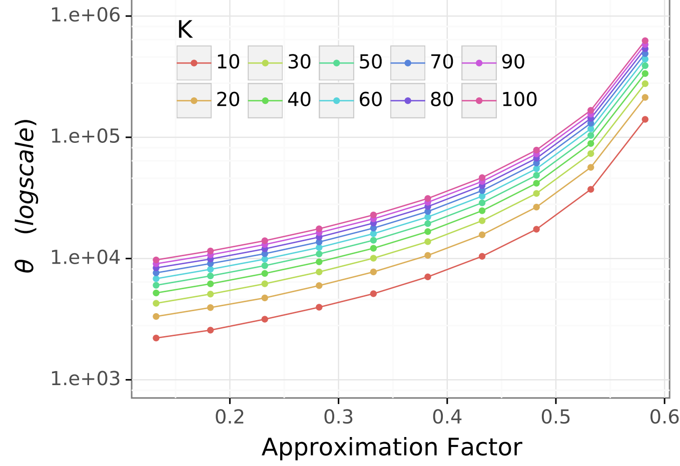
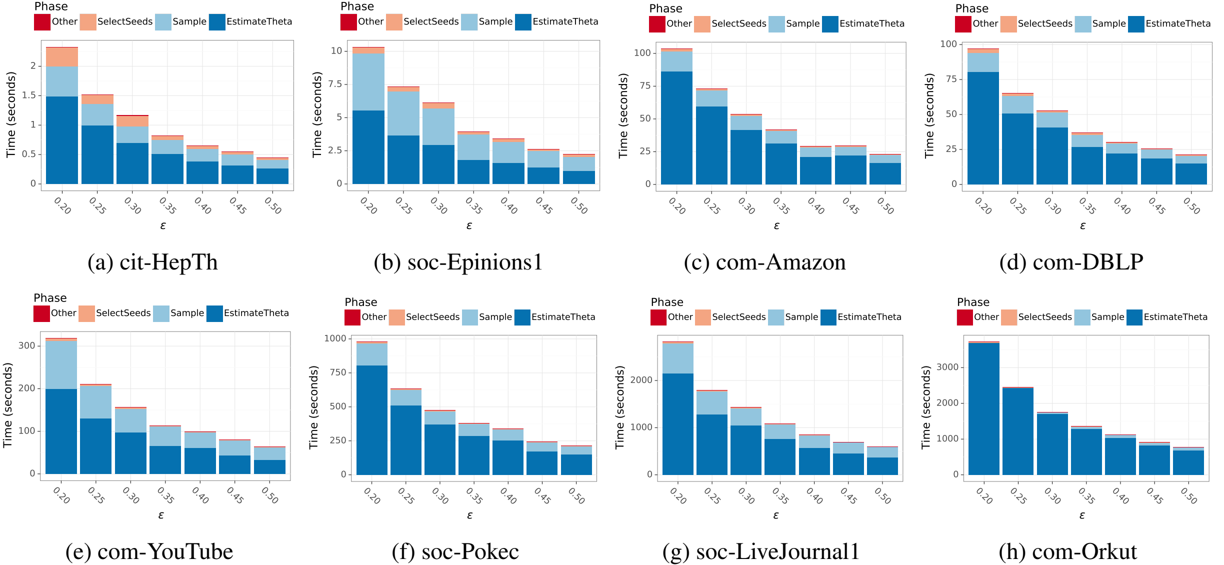


Figure 2: Number of random reverse reachable sets (θ) for cit-HepTh, as function of k and the approximation factor $1 - 1/e - \epsilon$. Note that ϵ decreases from left to right; smaller values of ϵ correspond to a higher precision.



(a) cit-HepTh (b) soc-Epinions1 (c) com-Amazon (d) com-DBLP

(e) com-YouTube (f) soc-Pokec (g) soc-LiveJournal1 (h) com-Orkut

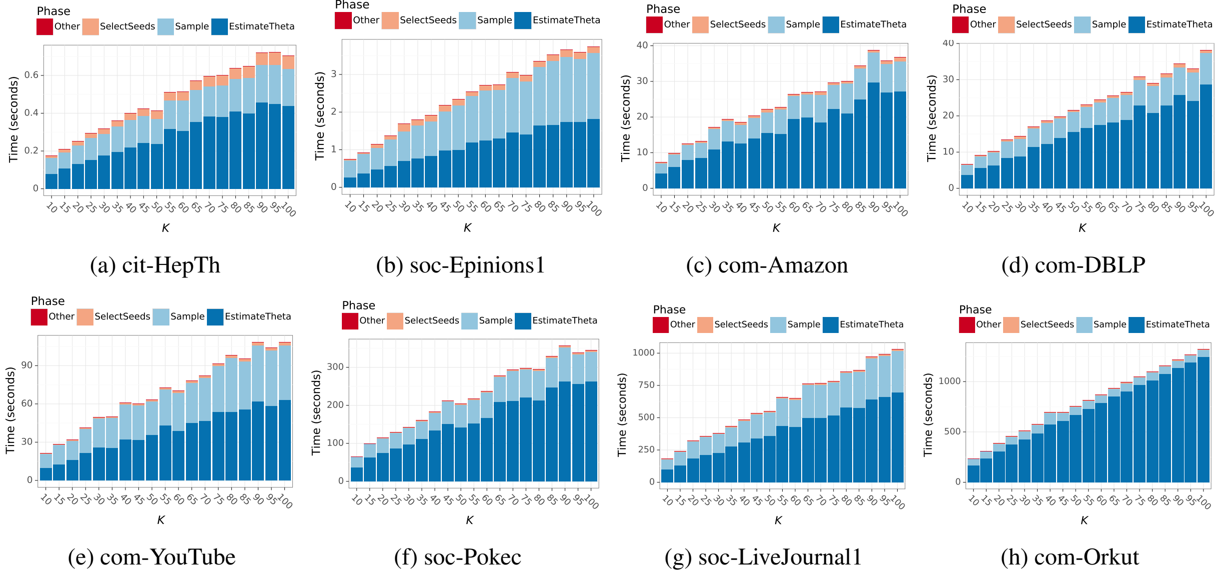
Figure 3: Impact of ϵ on runtime for $k = 50$ using IC model. Runtimes are in seconds using 20 threads on Puma.

Figure 3 shows that the runtime increases when ϵ is decreased (i.e., increase in approximation factor)—which is to be expected. Increasing the size of the seed set k also causes an increase in runtime as shown in Figure 4. These observations match the growth rate of θ shown in Figure 2. While the relationship between ϵ and the seed set size k is different, the phase breakdown in Figures 3 and 4 shows a common reduction of the fraction of the execution time spent in the *Sample* phase by increasing the size of the input.

From these figures, we also observe that the dominant contributors to overall execution time are the Estimation and Sample (in that order). In fact, we observed that most of the time for Estimation is also consumed within its Sample invocations. We also note that the probabilities on the edges have a nonlinear influence on the runtime. We assigned probabilities uniformly at random from $[0, 1]$ while [5] assigned 0.10 to all the edges. Therefore, the runtime reported in the two work are significantly different.

4.2 Strong Scaling Analysis

We now present strong scaling results of our parallel implementations on both shared and distributed memory platforms. We note that previous work [5] limits the experiments to $\epsilon = 0.5$ (approximation factor of 0.13) due to the



(a) cit-HepTh (b) soc-Epinions1 (c) com-Amazon (d) com-DBLP

(e) com-YouTube (f) soc-Pokec (g) soc-LiveJournal1 (h) com-Orkut

Figure 4: Impact of k on runtime for $\epsilon = 0.5$ using IC model. Runtimes are in seconds using 20 threads on Puma.

Table 3: Improvement in runtime relative to IMM [5]

		com-Orkut (s)	**Speedup**
IMM	$(\epsilon = 0.5, k = 100)$	28024.56	1.00×
IMM$_{\mathbf{opt}}$	$(\epsilon = 0.5, k = 100)$	9027.50	3.10×
IMM$_{\mathbf{mt}}$	$(\epsilon = 0.5, k = 100)$	1319.21	21.24×
IMM$_{\mathbf{dist}}$	$(\epsilon = 0.13, k = 200)$	47.77	586.61×
		soc-LiveJournal1 (s)	**Speedup**
IMM	$(\epsilon = 0.5, k = 100)$	16434.81	1.00×
IMM$_{\mathbf{opt}}$	$(\epsilon = 0.5, k = 100)$	3954.59	4.16×
IMM$_{\mathbf{mt}}$	$(\epsilon = 0.5, k = 100)$	1026.21	16.02×
IMM$_{\mathbf{dist}}$	$(\epsilon = 0.13, k = 200)$	55.12	298.16×

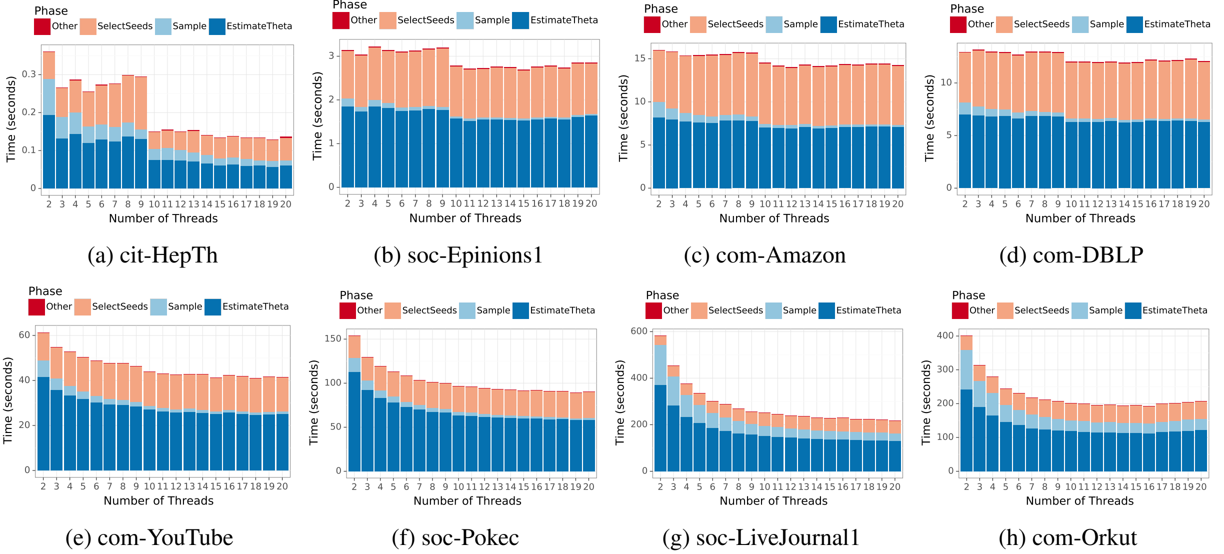
effects that the parameter has on the compute time and on the memory requirements (Figure 2). In contrast, we report results with ϵ up to 0.13 (approximation factor of 0.50) for our parallel and distributed algorithm, which is unprecedented to the best of our knowledge (Table 3, Figures 7 and 8).

Multithreaded Implementation We begin the presentation of scaling results for our multi-threaded implementation on a single node of Puma with up to 20 threads. Results for the LT model are presented in Figure 5 and those for the IC model in Figure 6. We provide execution times from two threads to 20 threads (hyper-threading is disabled) in increments of one.

Figures 5 and 6 show that the speedups improve roughly with the increase in input size, delivering up to a peak speedup of 12.55× relative to the 2-thread execution, for com-Orkut on the IC model. This is to be expected as for the small inputs (such as cit-HepTh) the overheads due to the greedy strategy of seed selection starts to dominate. As the input size increases, the graph traversals from within both the Estimate and Sample functions start to dominate the proceedings. In fact, we start observing near-linear speedups for the larger inputs on the IC model.

The LT model tends to produce very small RRR sets (when compared to the IC model) due to its mechanics and, because of the small workload, it shows limited scalability only for the bigger inputs. In fact, we observed up to 5.67× faster execution time on cit-HepTh and up to 6.37× faster on com-Orkut, compared to the corresponding executions using the IC model.

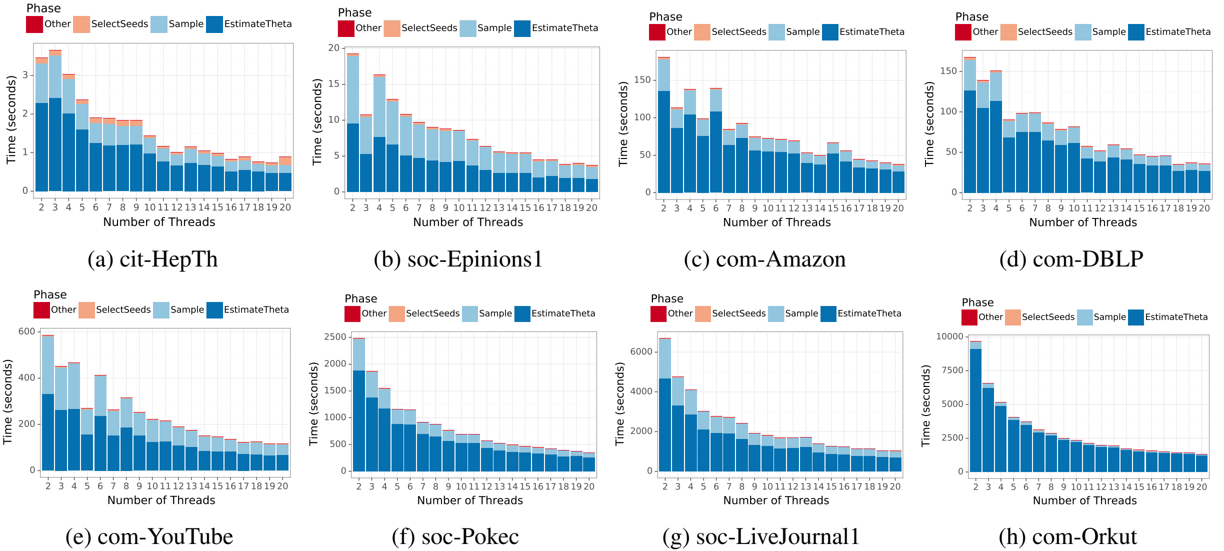
Distributed Implementation We present two sets of distributed scaling results: on up to 16 nodes of Puma, and up to 1024 nodes on Edison. We note that the amount of memory per node on Puma is much larger than the amount



(a) cit-HepTh (b) soc-Epinions1 (c) com-Amazon (d) com-DBLP

(e) com-YouTube (f) soc-Pokec (g) soc-LiveJournal1 (h) com-Orkut

Figure 5: Multithreaded strong scaling using up to 20 threads of Puma. Parameters: $\epsilon = 0.5$, $k = 100$, LT model.



(a) cit-HepTh (b) soc-Epinions1 (c) com-Amazon (d) com-DBLP

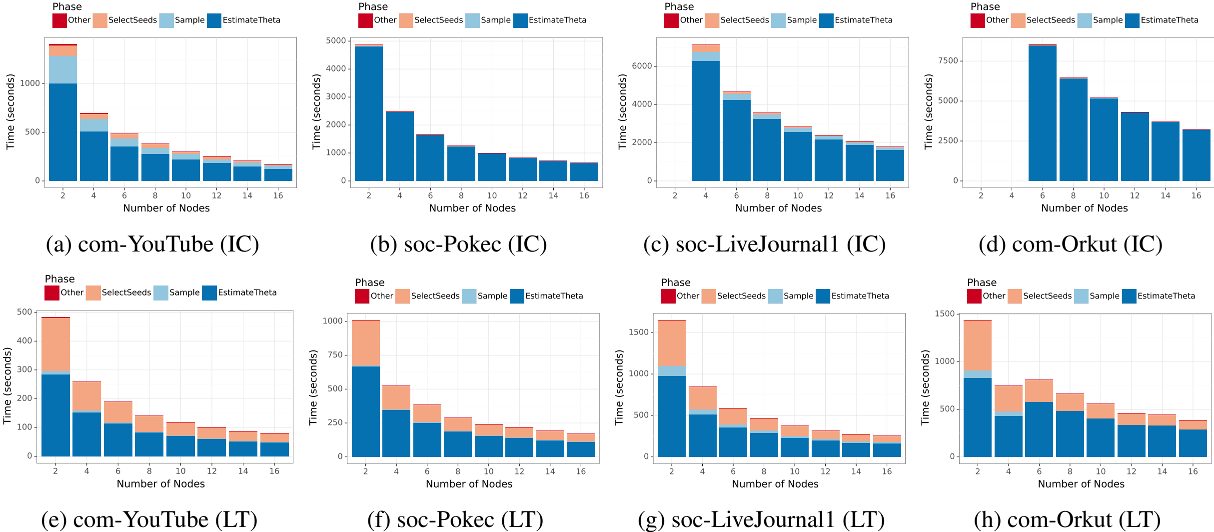
(e) com-YouTube (f) soc-Pokec (g) soc-LiveJournal1 (h) com-Orkut

Figure 6: Multithreaded strong scaling using up to 20 threads of Puma. Parameters: $\epsilon = 0.5$, $k = 100$, IC model.

of memory on Edison and that the processors on Puma are faster. We used hyper-threading on Edison; therefore, the largest run on Edison with $1,024$ nodes amounts to $49,152$ $(1024 \times 24 \times 2)$ threads. On the other hand, the configuration of the system on Puma does *not* allow us to enable hyper-threading, so the number of threads is 320 (16×20).

Figures 7 and 8 shows the scalability study on Puma and Edison clusters for the four biggest graphs considered in our evaluation. (Smaller graphs do not produce sufficient work to justify high processor count).

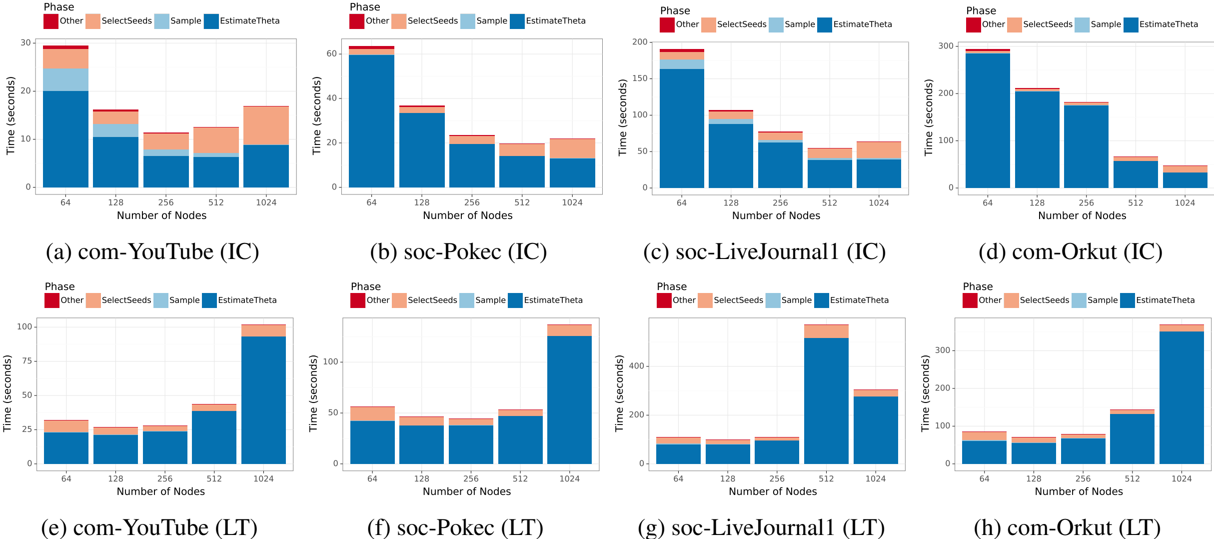
For Puma, Figure 7 shows that our distributed algorithm has scalability on the IC and the LT model generating speedups up to $8\times$. It is important to note that points missing in Figures 7c and 7d are experiments that were killed by the Linux Out of Memory (OMM) killer. These failures show the practical need of algorithms that can scale out to enable to find solution with good approximation guarantees.



(a) com-YouTube (IC) (b) soc-Pokec (IC) (c) soc-LiveJournal1 (IC) (d) com-Orkut (IC)

(e) com-YouTube (LT) (f) soc-Pokec (LT) (g) soc-LiveJournal1 (LT) (h) com-Orkut (LT)

Figure 7: Distributed strong scaling using IC and LT model with up to 16 nodes of Puma. Parameters: $\epsilon = 0.13$, $k = 200$.



(a) com-YouTube (IC) (b) soc-Pokec (IC) (c) soc-LiveJournal1 (IC) (d) com-Orkut (IC)

(e) com-YouTube (LT) (f) soc-Pokec (LT) (g) soc-LiveJournal1 (LT) (h) com-Orkut (LT)

Figure 8: Distributed strong scaling using LT and IC models with up to 1024 nodes of Edison. Parameters: $\epsilon = 0.13$, $k = 200$.

For Edison, Figure 8 shows the scalability results on the IC and LT models. While it can be observed that algorithm is scaling reasonably well for the IC model, we observed that low parallel efficiency under the LT model is caused by the low amount of work with respect to the thread count.

Table 3 summarizes the key contribution and impact with respect to the previous state-of-the-art. We observe scalability in both performance and accuracy of the solution.

5 Case study in biology

As a case of novel scientific application, we present case study of applying influence maximizaiton to two datasets containing multi-omic data from the life sciences. The first is a dataset containing metabolomic and metatranscriptomic data examining the microbial community of a soil ecosystem as it responds to changes in moisture. The second is a dataset containing proteomic and transcriptomic data derived from tumor samples of human patients. We inferred fea-

ture co-expression networks for these datasets using a random forest method implemented in GENIE3 [22]. The edges in these networks link transcripts and proteins, or transcripts and metabolites, based on instances of co-expression across the range of experiments in the dataset under analysis. To determine which features are crucial to the network structure we applied influence maximization to the networks to identify a seed set size of 200. For comparison, we also carried out centrality analysis of each node in the two networks. Several previous works have demonstrated the effectiveness of centrality analysis on biological networks to identify biological important nodes [23]–[26]. As additional measures for analysis, we included vertex degrees and vertex betweenness (a measure of how many shortest paths linking two random nodes pass through the node in question).

To compare the ability of the IMM method with topological measures we ranked nodes in the cancer network based on degree and betweenness centrality and considered the top 200 nodes in statistical enrichment to match the results of the IMM method. We then applied functional enrichment in which Fisher's exact test was applied to pathways (sets of functionally related genes/proteins) from the MSIG database. Results from this comparison revealed that IMM identifies nodes with more significant enrichment (372 pathways enriched with adjusted $p < 0.05$) than betweenness (159 pathways) but fewer than degree (614 pathways). However, examining the most enriched pathways resulting from application of the three methods showed that the top pathways indicated by IMM were cancer related pathways including invasive breast cancer, prostate cancer, and inflammation pathways like complement and coagulation. In contrast, the top pathways indicated by betweenness included focal adhesion and T cell pathways, which are not necessarily specific to cancer. Also, while the top pathways indicated by degree centrality included invasive breast cancer, they also included liver and muscle processes that are not closely related to cancer. Therefore, the IMM method provides complementary, and potentially improved, identification of biological functions important in the network. Extensive further investigation of this intriguing observation is necessary, which we plan to undertake in the near future.

When examining soil networks we also used degree as a measure of centrality and found that several nodes of high centrality were those metabolites that were central to pathways within bacterial species and communities. These include glucose, trehalose, ribose and 3,4-dihydroxybutanoic acid. These metabolites showed the highest number of connections to other transcripts and other metabolites in the network. When looking at the top 30 nodes (transcripts or metabolites) that had the highest degree centrality nine of them ($9/30$, 30%) were also predicted by IMM to have central network positions. These include trehalose and 3,4–dihydroxybutanoic acid as well as transcripts coding for cytochrome C oxidase, a transketolase and maltose transferase. Similar to the analysis of cancer networks, there is complementarity in the standard centrality approaches and IMM as well as possibly improved identification with IMM.

We are currently building large scale networks with tens of millions of nodes, and therefore, the ability to compute high quality seed set will be of significance to biology.

6 Conclusions and Future Work

Influence maximization is rapidly emerging as an effective technique to identify top influential actors in a network in the context of several applications including social networks, advertising, and life sciences. While several algorithms have been proposed in the literature, there is a lack of rigorous implementations in parallel. To the best of our knowledge, the developments reported in this paper represent the first parallel implementations that are designed to execute influence maximization operations at scale. More specifically, our approach parallelizes one of the fastest sequential algorithms, the IMM, on shared and distributed memory systems. Our experiments demonstrate the ability of our parallel implementations to achieve significant improvements in both performance and output precision on large-scale real-world inputs. The fast and scalable performance achieved can enable multiple parameteric experiments and evaluation of complex networks at scale.

Directions for future research, development, and application include (but not limited to): i) extension to settings where the input graph is also partitioned (in addition to $\mathbb{R}$); ii) exploitation of problem properties such as submodularity, and input properties such as communities—toward improving performance and/or precision; iii) extension to other architectures such as GPUs and vector processing units; and iv) application to a wider range of complex real-world networks.

Acknowledgment

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research was in part supported by the U.S. DOE by the U.S. DOE ExaGraph project at Pacific

Northwest National Laboratory (PNNL) and by the U.S. National Science Foundation (NSF) grant 1815467 to Washington State University. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

References

[1] P. M. Domingos and M. Richardson, "Mining the network value of customers," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, San Francisco, CA, USA, August 26-29, 2001*, ACM, 2001, pp. 57–66.

[2] D. Kempe *et al.*, "Maximizing the spread of influence through a social network," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, ACM, 2003, pp. 137–146. DOI: 10.1145/956750.956769.

[3] J. Leskovec *et al.*, "Cost-effective outbreak detection in networks," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15, 2007*, ACM, 2007, pp. 420–429. DOI: 10.1145/1281192.1281239.

[4] Y. Tang *et al.*, "Influence maximization: Near-optimal time complexity meets practical efficiency," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, ACM, 2014, pp. 75–86. DOI: 10.1145/2588555.2593670.

[5] Y. Tang *et al.*, "Influence maximization in near-linear time: A martingale approach," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, ACM, 2015, pp. 1539–1554. DOI: 10.1145/2723372.2723734.

[6] M. Minoux, "Accelerated greedy algorithms for maximizing submodular set functions," *Optimization Techniques*, pp. 234–243, 1978.

[7] A. Goyal *et al.*, "CELF++: optimizing the greedy algorithm for influence maximization in social networks," in *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011 (Companion Volume)*, ACM, 2011, pp. 47–48. DOI: 10.1145/1963192.1963217.

[8] W. Chen *et al.*, "Efficient influence maximization in social networks," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009*, ACM, 2009, pp. 199–208. DOI: 10.1145/1557019.1557047.

[9] W. Chen *et al.*, "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, Washington, DC, USA, July 25-28, 2010*, ACM, 2010, pp. 1029–1038. DOI: 10.1145/1835804.1835934.

[10] E. Cohen *et al.*, "Sketch-based influence maximization and computation: Scaling up with guarantees," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*, ACM, 2014, pp. 629–638. DOI: 10.1145/2661829.2662077.

[11] C. Borgs *et al.*, "Maximizing social influence in nearly optimal time," in *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, C. Chekuri, Ed., SIAM, 2014, pp. 946–957. DOI: 10.1137/1.9781611973402.70.

[12] Y. Wang *et al.*, "Community-based greedy algorithm for mining top-k influential nodes in mobile social networks," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, ACM, 2010, pp. 1039–1048. DOI: 10.1145/1835804.1835935.

[13] Y. Chen *et al.*, "CIM: community-based influence maximization in social networks," *ACM TIST*, vol. 5, no. 2, 25:1–25:31, 2014. DOI: 10.1145/2532549.

[14] M. Halappanavar *et al.*, "Accelerating the mining of influential nodes in complex networks through community detection," in *Proceedings of the ACM International Conference on Computing Frontiers, CF'16, Como, Italy, May 16-19, 2016*, ACM, 2016, pp. 64–71. DOI: 10.1145/2903150.2903181.

[15] J. Kim *et al.*, "Scalable and parallelizable processing of influence maximization for large-scale social networks?" In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, IEEE Computer Society, 2013, pp. 266–277. DOI: 10.1109/ICDE.2013.6544831.

[16] N. Du *et al.*, "Scalable influence estimation in continuous-time diffusion networks," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.*, 2013, pp. 3147–3155.

[17] B. Lucier *et al.*, "Influence at scale: Distributed computation of complex contagion in networks," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, ACM, 2015, pp. 735–744. DOI: 10.1145/2783258.2783334.

[18] H. Wu *et al.*, "Parallel seed selection for influence maximization based on k-shell decomposition," in *Collaborate Computing: Networking, Applications and Worksharing - 12th International Conference, CollaborateCom 2016, Beijing, China, November 10-11, 2016, Proceedings*, S. Wang and A. Zhou, Eds., ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 201, Springer, 2016, pp. 27–36. DOI: 10.1007/978-3-319-59288-6_3. [Online]. Available: https://doi.org/10.1007/978-3-319-59288-6_3.

[19] H. Bauke and S. Mertens, "Random numbers for large scale distributed monte carlo simulations," *CoRR*, vol. abs/cond-mat/0609584, 2006. arXiv: cond-mat/0609584. [Online]. Available: http://arxiv.org/abs/cond-mat/0609584.

[20] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*, http://snap.stanford.edu/data, Jun. 2014.

[21] M. Minutoli *et al.*, *Pnnl/ripples*. [Online]. Available: https://github.com/pnnl/ripples.

[22] A. Irrthum *et al.*, "Inferring regulatory networks from expression data using tree-based methods," *PloS one*, vol. 5, no. 9, e12776, 2010.

[23] R. S. McClure *et al.*, "Species-specific transcriptomic network inference of interspecies interactions," *The ISME Journal*, p. 1, 2018.

[24] J. E. McDermott *et al.*, "The effect of inhibition of PP1 and TNFα signaling on pathogenesis of SARS coronavirus," *BMC systems biology*, vol. 10, no. 1, p. 93, 2016.

[25] J. E. McDermott *et al.*, "Identification and validation of Ifit1 as an important innate immune bottleneck," *PLoS One*, vol. 7, no. 6, e36465, 2012.

[26] J. E. McDermott *et al.*, "Topological analysis of protein co-abundance networks identifies novel host targets important for HCV infection and pathogenesis," *BMC systems biology*, vol. 6, no. 1, p. 28, 2012.