Reproducibility Report of "Asynchronous Distributed-Memory Parallel Algorithms for Influence Maximization"

Ioana Popa
Department of Mathematics
ETH Zürich
iopopa@ethz.ch

Leonard Schmidt
Department of Computer Science
ETH Zürich
sleonar@ethz.ch

Abstract

This report presents a reproducibility study of the work *"Asynchronous Distributed-Memory Parallel Algorithms for Influence Maximization"*. The original paper introduces new distributed-memory algorithms for the Influence Maximization via Martingales (IMM) framework, redesigning its two key kernels, *GenerateRR* and *SelectSeeds*, and proposing a fine-grained asynchronous bulk synchronous parallel (FA-BSP) execution model built on the HClib runtime. To assess these contributions, we reproduce the core performance experiments on our distributed CPU system *Parmi*, focusing on strong scaling behavior, the impact of the parameter k, and the decomposition of the SelectSeeds2D kernel. Our results consistently match the trends reported in the original work: the Actor-based algorithms provide improved memory scalability through full graph distribution, the FA-BSP execution model reduces synchronization overhead compared to BSP-style designs, and the Actor IMM 2D variant achieves the best overall performance due to its efficient seed-selection phase.

Artifact Description (AD)

The influence maximization (IM) problem seeks to identify a small subset of highly influential individuals in a social network whose adoption of an idea, behavior, or product would maximize its eventual spread through the network. Formally, given a graph in which edges represent social connections and weights quantify how strongly one person influences another, the task is to select k seed nodes that maximize the expected number of individuals who become "activated" through a diffusion process.

We model the diffusion process using the **Linear Threshold (LT) model**. Each node v has a threshold θ_v uniformly sampled at random from the interval $[0, 1]$ and is influenced by its neighbors through weighted edges $b_{v,u}$ such that $\sum_u b_{v,u} \leq 1$. Starting from an initial set of active nodes A_0, a node becomes active if the total weight of its active neighbors exceeds its threshold. The process continues until no new activations occur. The goal of the influence maximization problem is to find a set of k seed nodes that maximizes the expected number of activated nodes under this diffusion model [1].

In essence, IM formalizes the intuition behind **viral marketing**: by optimally seeding a few influential individuals, one can exploit the structure of the social network to achieve the greatest spread at minimal cost.

In their paper [3], Singhal et al. present distributed-memory parallel algorithms for the two main kernels of a state-of-the-art implementation of one IM algorithm, influence maximization via martingales (IMM).

This report aims to reproduce the results obtained by Singhal et al.. In the following section, we provide an overview of the experimental setup, including our hardware and software configuration. We then present our experiments, discuss the results, and conclude with a summary of our findings.

1 Overview of Contributions and Artifacts

This section summarizes the paper's main contributions, the corresponding computational artifacts, and how the reproduced figures support and validate these contributions.

1.1 Paper's Main Contributions

- C_1 The paper proposes two distributed-memory parallel algorithms for the IMM influence maximization framework, redesigning both *GenerateRR* and *SelectSeeds* for large-scale HPC systems.
- C_2 It introduces an FA-BSP (fine-grained asynchronous BSP) execution model, implemented on top of the HClib runtime, to reduce synchronization costs and improve scalability compared to traditional BSP approaches.
- C_3 The authors demonstrate through theoretical analysis and large-scale experiments that their Actor IMM 2D variant achieves substantially better performance and scalability than existing state-of-the-art implementations (up to `29.6x` speedup over the MPI-based implementation on synthetic and real-world network graphs) [3].

1.2 Computational Artifacts

Artifact A_1 corresponds to the implementation and evaluation of the two Actor-based IMM algorithms, `Actor IMM 1D` and `Actor IMM 2D`, as introduced in the reference paper. These algorithms directly realize and experimentally validate the main contributions C_1–C_3 of the original work.

A_1 https://doi.org/10.5281/zenodo.13308653

In this report, we reproduce **Figure 6**, **Figure 8**, **Figure 13**, and **Figure 10** from the original paper, as these figures collectively illustrate the three central contributions:

- Figure 6: Strong scaling of the end-to-end IMM workload (C_1, C_3)
- Figure 8: Impact of the parameter k on runtime scalability (C_1, C_3)
- Figure 13: Detailed decomposition of the SelectSeeds2D kernel (C_2)
- Figure 10: Impact of the parameter ϵ on runtime scalability (C_1, C_3)

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3	Figures 6, 8, 13, and 10

2 Artifact Identification

This section documents the computational artifact A_1 used in our reproduction study. We describe the execution environment, required software and datasets, installation steps, and the workflow used to run the Actor-based IMM implementations. This establishes the complete setup needed to reproduce our experimental results.

2.1 Computational Artifact A_1

Relation To Contributions

Artifact A_1 supports all three contributions of the original paper, as detailed below.

1) **Support for C_1:** Artifact A_1 provides the complete implementations of the redesigned *GenerateRR* and *SelectSeeds* kernels for distributed-memory parallel systems. By distributing the input graph across nodes, instead of replicating it, the artifact reduces memory usage and improves scalability.
2) **Support for C_2:** A_1 is built on top of the HClib FA-BSP runtime, enabling fine-grained asynchronous communication and reduced global synchronization. This allows us to examine how FA-BSP execution improves performance relative to classical BSP as claimed in the paper.
3) **Support for C_3:** Through our reproduction of the key performance figures, artifact A_1 allows us to verify the paper's claims regarding the superior scalability of Actor IMM 2D - particularly its performance on large graphs and high core counts.

Expected Results

Based on the design of the Actor IMM 1D and Actor IMM 2D algorithms presented in the original paper, we expect the 2D variant to outperform the 1D implementation. The 2D approach reduces communication during the seed-selection phase by preprocessing a distributed co-occurrence matrix, whereas the 1D version repeatedly performs collective operations over large data structures.

By analyzing time complexity of `Actor IMM 1D`

$$O\Big(\underbrace{kN/P}_{\text{update count array}} + \underbrace{(k\tau + \mu\rho)\log P}_{\text{global argmax \& broadcast}} \Big)$$

and `Actor IMM 2D`

$$O\Big(\underbrace{(\lambda+\mu)\sum_{i=1}^{|R|} |R_i|^2/P}_{\text{build matrix } C} + \underbrace{kN/P}_{\text{update count array}} + \underbrace{k(\tau+\mu)\log P}_{\text{global argmax}} \Big)$$

we can explain the different runtime behaviors observed in our experiments. Here, λ denotes the local memory access cost, τ the latency of remote communication, and μ the bandwidth cost of remote communication. Moreover, P represents the total number of cores, N the number of vertices in the input graph G, and $R = R_i$ the collection of sampled random reverse reachable (RRR) sets. The theoretical analysis predicts that `Actor IMM 2D` should exhibit significantly better scalability than the 1D implementation. The matrix-construction phase amortizes much of the computational cost, while the subsequent seed-selection iterations require only lightweight local updates and a single global `AllReduce` per iteration.

A comparison of the communication terms in the runtime expressions immediately shows why `Actor IMM 2D` is expected to outperform `Actor IMM 1D`. In the 1D variant, the dominant communication cost is $\mu\rho \log P$, since each iteration requires collective operations over data proportional to the number of RRR sets ρ, which is typically very large. In contrast, the 2D algorithm eliminates this dependence on ρ by constructing the co-occurrence matrix C during preprocessing, allowing all subsequent updates to be performed locally. As a result, each iteration requires only a single `AllReduce` on one integer, leading to a communication cost of just $k(\tau + \mu)\log P$. Because in practice $k \ll \rho$, the 2D algorithm incurs orders of magnitude less communication, and is therefore expected to achieve significantly better performance on distributed-memory systems.

Taken together, these observations lead us to expect that our experiments will reproduce the main finding of the original paper: `Actor IMM 2D` consistently outperforms `Actor IMM 1D` on small, medium, and large graphs, especially at higher core counts where communication becomes the dominant bottleneck.

Expected Reproduction Time (in Minutes)

The expected time for the full setup, including dependency installation, artifact download, execution (i.e., reproducing all plots in this paper with five runs per experiment for confidence intervals), and subsequent analysis is summarized in Table 1

Artifact Process	Time
Setup	15 min
Execution	120 min
Analysis	30 min

Table 1: Estimated time required for setup, execution, and analysis.

Artifact Setup (incl. Inputs)

Hardware. All experiments were carried out on the *Parmi* cluster, used by Team RACKlette during the SC25 competition. The system runs on Rocky Linux 9.5 and consists of four nodes: one head/login node (also usable for computation) and three compute nodes. Each node is equipped with a dual-socket AMD EPYC CPU configuration: the head/login node uses two EPYC 9734 processors with 112 cores each (224 cores total), while the three compute nodes each feature two EPYC 9654 processors with 96 cores each (192 cores per node), all equipped with 768 GB of DDR5 memory. The head node provides local NVMe storage ($\approx$ 14 TB total) which is exported to the compute nodes via pNFS over RDMA.

Each node also hosts four NVIDIA H200 NVL (PCIe) GPUs connected via NVLink, although all experiments in this study were

executed in **CPU-only mode**. Inter-node communication is provided by a 400 Gb/s InfiniBand (ConnectX-7) network. The system is enclosed in a 24U rack and designed for high-performance computing under a total power cap of 10 kW.

Software. A_1 is built using the HClib library that requires OpenSHMEM. For *Parmi*, we used *python@3.9*, *gcc@11.4.1*, and *openmpi@4.1.4*. To compile the different implementations and versions, the Spack package manager was used. In order to compile the application and recreate the software stack used in the original paper we used *Spack*. We used *slurm* to allocate our cluster resources and launch the application. In order to generate the data for our analysis, we use a set of *sbatch* scripts that we automated using *bash* scripts [5].

Datasets / Inputs. We use the same SNAP datasets as the original paper, all publicly available on the Stanford Large Network Dataset Collection [2]. Each dataset is first converted to the Linear Threshold (LT) model using the *dump-graph* tool from Ripples, which relabels vertices and normalizes incoming edge weights. The LT-normalized graphs (e.g., *com-DBLP*, *com-Youtube*, *soc-Livejournal1*) directly serve as inputs to our reproduced implementation. This ensures full compatibility with the evaluation setup of the original work.

Installation and Deployment. To build the Actor IMM implementations, we cloned the official repository used by the authors of the reference paper using the sc25-scc branch [4] . For our experiments, we applied a minimal modification to the setup procedure: we disabled the module-loading commands in `./scripts/setup.sh`, as these are environment-specific and not compatible with our cluster configuration. Instead, we manually load the required compiler and MPI installation via *spack* before invoking the setup script. The following setup in Listing 1 was used for all experiments in this work.

```
git clone https://github.com/singhalshubh/imm_hclib.git
cd imm_hclib/
git checkout sc25-scc
spack load gcc@11.4.1 openmpi@4.1.4
ml slurm/24.11
source scripts/setup.sh
cd src/
export HClib_WF=$PWD
cd lt_1D/
make
cd ../lt_2D/
make
```

Listing 1: Setup procedure used to build the Actor IMM implementations.

The script produces two binaries located in `src/lt_1D/production` and `src/lt_2D/production_2D`, corresponding to the Actor IMM 1D and Actor IMM 2D implementations, respectively.

Artifact Execution

Artifact A_1 implements the two main IMM kernels, *GenerateRR* and *SelectSeeds*, using the Actor-based FA-BSP model. After reading the input graph into a distributed adjacency-list format, the artifact executes the sequence

$$T_1 \rightarrow T_2$$

for $\log_2 |V| + 1$ iterations.

GenerateRR (T_1). T_1 generates RRR sets by advancing many random walks in parallel. Each step selects outgoing edges and asynchronously forwards messages to the process owning the next vertex. A barrier marks completion once no unfinished walks remain.

SelectSeeds (T_2). In the 1D variant, T_2 performs an `AllReduce` to identify the vertex with maximum coverage, broadcasts its RRR-set identifiers, and updates local state. The 2D variant additionally builds a distributed co-occurrence matrix, enabling faster iterative updates of coverage counts and improved scalability.

This summarizes the execution flow of the Actor-based IMM implementations provided in artifact A_1.

Artifact Analysis (incl. Outputs)

`Actor IMM 2D` is expected to outperform `Actor IMM 1D` on almost all configurations, as described in 2.1.

Artifact Evaluation (AE)

3.1 Computational Artifact A_1

Artifact Execution

Before every experiment, we ensured consistent and reproducible CPU behavior by explicitly configuring CPU frequency and turbo-boost settings on all nodes. This was achieved through a helper script invoked with `./cpu_freq.sh boost`, which sets the compute nodes to fixed boost frequencies. The essential logic of this script is shown in Listing 2. The script sets different CPU frequencies for the head and compute nodes to match their respective hardware capabilities. The head node uses AMD EPYC 9734 processors with a maximum boost frequency of 3.0 GHz, while the compute nodes use AMD EPYC 9654 processors with a maximum boost frequency of 3.71 GHz.

```
boost() {
  echo "Setting cpu frequency to boost"
  pdsh -w parmi  sudo cpupower frequency-set -f 3.0GHz
  pdsh -w parmi1,parmi2,parmi3 sudo cpupower frequency-set
  ↪ -f 3.71GHz
}
```

Listing 2: Function used to set CPU frequencies to boost mode on the *Parmi* cluster.

To reproduce the three main scaling figures, we implemented fully automated experiment scripts [5] that systematically sweep core counts or parameter values (e.g., k), launch both `Actor IMM 1D` and `Actor IMM 2D` executables, and store the resulting logs.

Each configuration was executed five times, enabling us to compute reliable averages and 95% confidence intervals for all performance measurements.

3.2 Strong Scaling

3.2.1 Experiment Outline. In this section, we present strong-scaling results for three real-world graphs using both the `Actor IMM 1D` and `Actor IMM 2D` implementations. All experiments were conducted on our *Parmi* cluster, using up to 768 CPU cores. Unless otherwise stated, we fix the parameters to $k = 100$ and $\epsilon = 0.13$. Figure 1 shows the average runtime over five runs plotted in function of the number of cores.

3.2.2 Scalability Observations. Across all three datasets (com-DBLP, soc-Epinions1, and com-Youtube), our results consistently show that `Actor IMM 2D` (blue) is substantially faster than `Actor IMM 1D` (orange) at every core count.

In terms of scalability, both algorithms exhibit the expected downward trend as core counts increase, but Actor IMM 2D scales far more efficiently. Their runtime decreases almost proportionally to the number of cores up to 384 cores, demonstrating strong parallel efficiency. However, as seen in Figure 1, the 2D curve begins to rise again beyond 384-576 cores. This growth in runtime is also observed in the original paper and is a direct consequence of the algorithm's structure: once matrix generation becomes extremely fast due to high parallelism, the relative cost of the successive k seed-selection iterations becomes dominant. These iterations involve repeated global `AllReduce` operations, and when the per-process computational load becomes too small, the latency of global collectives outweighs the benefits of additional cores. At this point, additional parallelism provides diminishing returns and eventually causes a slight increase in total runtime.

This behavior illustrates an important distinction: high scalability does not always imply better end-to-end performance. `Actor IMM 1D` sometimes shows "smoother" strong-scaling curves, but its absolute runtime is always significantly worse. Even with perfect scaling, the 1D variant remains bound by much higher communication costs. In contrast, `Actor IMM 2D` may exhibit a performance minimum followed by a slight increase at very high core counts, but it still remains an order of magnitude faster overall. Thus, absolute runtime, not just scalability, determines overall algorithmic performance, and the `2D IMM` implementation is superior in both respects.

Notably, the `Actor IMM 1D` (orange) curve exhibits sublinear scaling in the log-log plot for com-DBLP and com-Youtube. While the algorithm continues to benefit from additional cores, the rate of improvement diminishes as core count increases. This gradual flattening indicates that communication overhead grows with scale, particularly due to the increasing cost of global collectives executed in each iteration.

In contrast, the soc-Epinions1 graph exhibits a U-shaped scaling curve. This behavior is expected given that soc-Epinions1 is substantially smaller than both com-DBLP and com-Youtube. The runtime begins to increase at a much lower core count compared to the larger graphs, reflecting the point at which communication and synchronization overhead outweigh the benefits of parallelization for smaller problem sizes.

3.2.3 System and Hardware Comparison. Our reproduced results are fully consistent with the hardware differences between our *Parmi* cluster and the *Phoenix* system used in the original IMM paper. A dual-socket Intel Xeon Gold 6226 node on *Phoenix* provides roughly 0.52 TFLOPS (FP64) and ∼280 GB/s of DDR4–2933 bandwidth. In contrast, each of our AMD EPYC 9654/9734 nodes sustains more than 2 TFLOPS (FP64) and approximately 460–500 GB/s of DDR5 bandwidth, i.e., about 4× *higher compute throughput* and 1.7× *higher memory bandwidth* per node.

The cache hierarchy further accentuates this gap. EPYC processors offer ∼64 MB of L1+L2 cache per socket (vs. ∼24 MB on Xeon 6226) and up to 768 MB of L3 cache (vs. ∼38 MB on *Phoenix*), yielding over a 20× *increase* in last-level cache capacity. Such large caches strongly benefit the irregular, memory-intensive access patterns of the `GenerateRR` and `SelectSeeds` kernels.

Finally, *Parmi* employs a 400 Gb/s InfiniBand NDR (ConnectX-7) fabric, compared to 100 Gb/s HDR on *Phoenix*, providing substantially lower communication latency. Given that Actor IMM performs fine-grained asynchronous communication, this interconnect advantage directly improves scalability.

Overall, the newer CPU architecture, much larger caches, higher memory bandwidth, and faster interconnect of *Parmi* fully justify the faster runtimes observed in our reproduced results.

3.3 Strong Scaling of `SelectSeeds2D`

3.3.1 Experiment Outline. We present the strong-scaling results of the `SelectSeeds2D` kernel on the com-DBLP and com-Youtube graphs, conducted on the *Parmi* cluster with up to 768 CPU cores. The parameters were set to $k = 100$ and $\epsilon = 0.13$.

The runtime of the `SelectSeeds2D` kernel is split into two phases: *Matrix Gen*, which builds the co-occurrence matrix C, and *k-loops*, which iteratively selects the most influential nodes.

Figure 2 shows the average decomposed runtime over five runs of the `SelectSeeds2D` kernel plotted in function of the number of cores.

3.3.2 Scalability Observations. The performance results are shown in Figure 2. Figure 13 from Singhal et al. [3], together with our reproduced results in Figure 2, shows that the time spent inside the *k-loops* of SelectSeeds2D increases as the number of cores grows, even though the algorithm performs the same number of iterations. This occurs because the *k-loop* phase is dominated by global collectives, specifically a global `argmax`, whose cost is latency-bound and scales as $O((\tau + \mu) \log P)$. As P increases, the local computation per core decreases proportionally to N/P, and thus becomes too small to hide the growing latency of these collectives. At the same time, both the $\log P$ term and network contention increase, making communication the dominant bottleneck at scale. As a result, the runtime of the *k-loop* phase grows with increasing core counts, producing the upward trend visible in the yellow bars at large P.

The *Matrix Gen* phase exhibits the expected near-linear scaling in the log–log plot. In contrast, the *k-loops* phase scales poorly: its runtime increases with higher core counts, consistent with the collective-communication effects discussed above.

3.3.3 Influence of Edge Density on `SelectSeeds2D`. The experimental results reveal a counterintuitive performance characteristic:

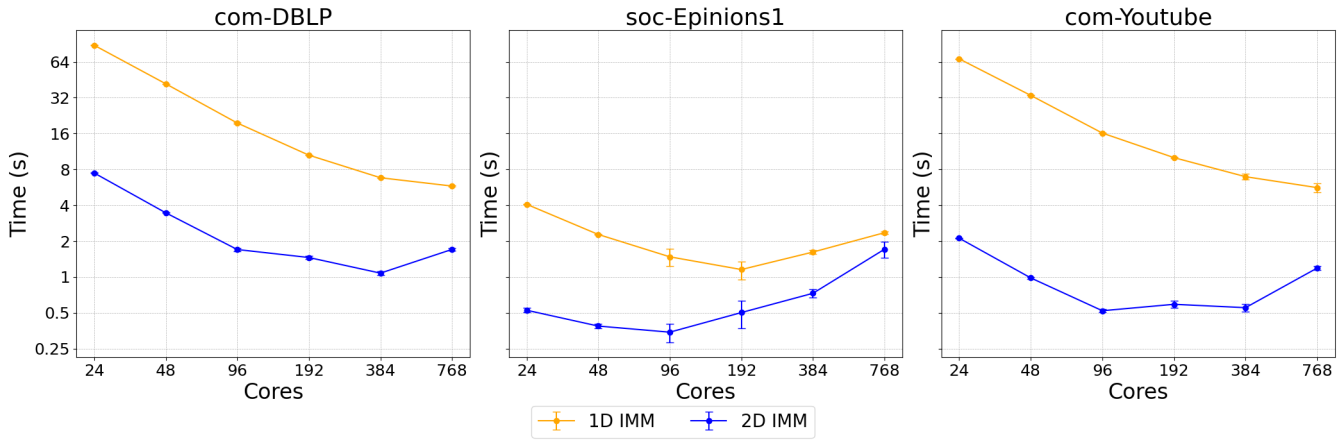

Figure 1: IMM strong scaling on larger real-world graphs [$k = 100$, $\epsilon = 0.13$].

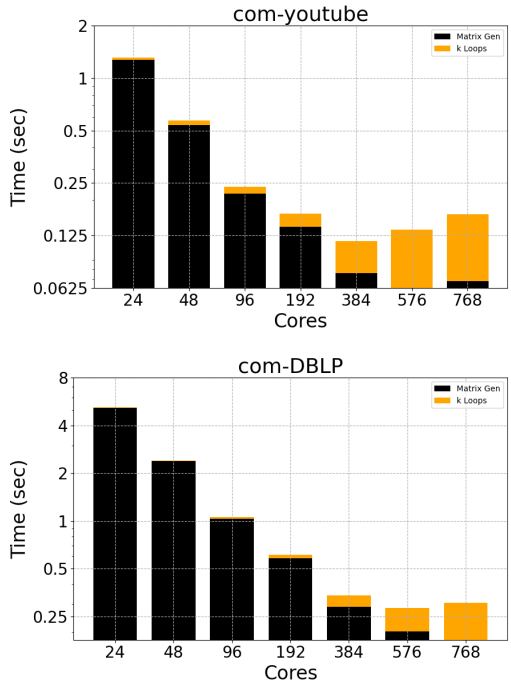

Figure 2: Runtime decomposition and strong scaling of SelectSeeds2D on Parmi. [$k = 100$, $\epsilon = 0.13$].

com-Youtube achieves approximately 4× faster execution than com-DBLP despite being significantly larger (1,134,890 vs. 317,080 nodes and 2,987,624 vs. 1,049,866 edges).

This phenomenon is explained by the structural properties of both graphs, particularly their degree distributions. While both networks exhibit similar average degrees (5 for Youtube, 6 for DBLP), their maximum degrees differ dramatically: com-Youtube has a maximum degree of 28,754 compared to DBLP's 343, an 83× difference. Com-Youtube follows an extreme power-law distribution with a small number of massive hubs, whereas com-DBLP demonstrates a more uniform degree distribution characteristic of collaboration networks. The IMM algorithm generates Random Reverse-reachable (RRR) sets through reverse Breadth-First-Search (BFS), where the computational cost is heavily influenced by node degrees. When reverse exploration encounters a high-degree node, it must process all incoming edges probabilistically. However, com-Youtube's extreme skewness creates a favorable performance characteristic: the majority of nodes have low degrees, resulting in inexpensive RRR set generation for most samples. Although encountering super-hub nodes incurs high computational cost, these expensive cases are sufficiently rare that the overall average cost remains low. In contrast, com-DBLP's more uniform degree distribution results in more consistent computational work per RRR set. While no individual RRR set is as expensive as those encountering Youtube's super-hubs, the lack of trivially cheap cases results in a higher average cost. This is evident in the scaling behavior, where Youtube's *Matrix Gen* component decreases rapidly with increasing core count as the predominantly inexpensive RRR sets parallelize efficiently, while DBLP's *Matrix Gen* component remains significant even at 768 cores due to more uniform work distribution.

This analysis demonstrates that graph size alone is insufficient for predicting algorithm performance, and that extreme heterogeneity in degree distribution can counter-intuitively improve average-case performance when the vast majority of operations occur on low-degree nodes.

3.4 Effects of parameter k

3.4.1 Experiment Outline. We evaluate the scalability of both 1D and 2D actor-based IMM implementations with respect to the number of influencer nodes k to be identified. The experiments compare runtime performance across two real-world datasets: com-Youtube and com-DBLP. We fix the approximation guarantee at $\epsilon = 0.13$ and vary k from 64 to 1024. All experiments are conducted on 4 nodes with a total of 768 cores.

Figure 3 shows the average runtime over five runs plotted in function of k.

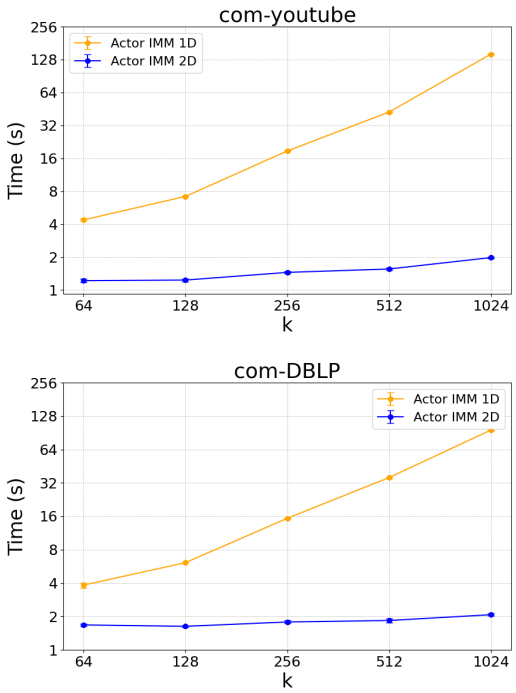

Figure 3: Impact of k on IMM runtime on 768 cores (4 nodes) of Parmi. [$\epsilon = 0.13$]

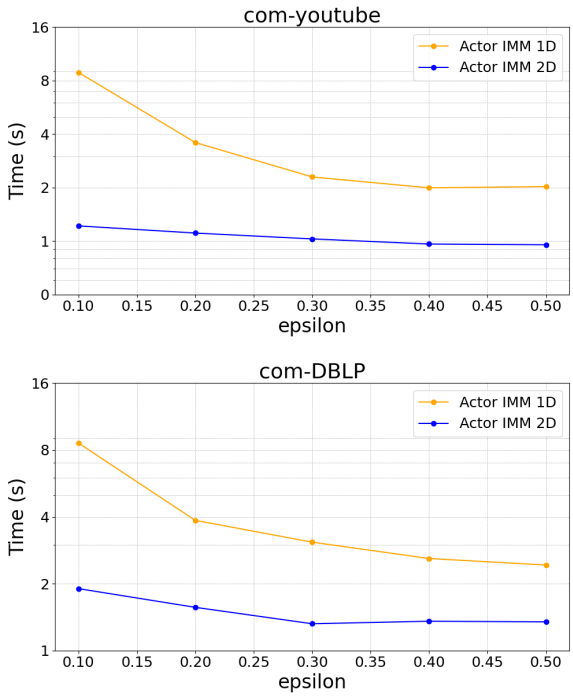

Figure 4: Impact of ϵ on IMM runtime on 768 cores (4 nodes) of Parmi. [$k = 100$]

3.4.2 Scalability Observations. Our results are presented in Figure 3. As discussed in Section 2.1, by analyzing the expected runtimes of `Actor IMM 1D` and `Actor IMM 2D`, we can explain the different runtime behaviors observed in our experiments. For `Actor IMM 1D`, both the computational term $k\frac{N}{P}$ and the communication term $(k\tau + \mu\rho)\log P$ scale linearly with k, since each additional seed selection requires a full scan of the RRR sets and additional collective communication across processes. In contrast, the runtime of `Actor IMM 2D` remains almost constant as k increases. This is because its dominant cost, $\frac{\mu}{P}\sum_{i=1}^{\rho}|R_i|^2$, corresponds to the construction of the co-occurrence matrix, which is performed once and reused throughout the selection phase. Subsequent seed selections only involve local updates and lightweight reductions on this matrix, making their cost nearly independent of k. Hence, the `Actor IMM 2D` implementation exhibits a flat scaling trend with respect to k, while `Actor IMM 1D` shows a steady increase in runtime.

3.5 Effects of parameter ϵ

3.5.1 Experiment Outline. We evaluate both Actor IMM variants (1D and 2D) on 768 cores of the *Parmi* cluster while varying the accuracy parameter ϵ from 0.1 to 0.5. The resulting runtimes are shown in Figure 4.

3.5.2 Observations. For both com-Youtube and com-DBLP, we observe a clear decrease in runtime as the accuracy parameter ϵ increases. This behavior is expected: in the IMM framework, a smaller value of ϵ corresponds to a tighter approximation guarantee and therefore requires generating a larger number of RRR sets ρ, which dominates the total runtime. Conversely, larger ϵ values reduce the required sampling effort and thus lead to faster execution.

Both `Actor IMM 1D` and `Actor IMM 2D` follow this trend, with the 2D variant consistently outperforming the 1D variant across all tested ϵ values. The relative performance gap remains similar on both datasets, indicating that the benefits of the 2D formulation, primarily reduced per-iteration communication during seed selection, are robust across input graph sizes and structures.

4 Conclusion

This reproducibility study successfully validates the core contributions of the work of Singhal et al. on distributed-memory parallel algorithms for influence maximization. Our reproduced results on the *Parmi* cluster consistently confirm the original paper's findings: the Actor-based implementations achieve superior memory scalability through distributed graph storage, the FA-BSP execution model effectively reduces synchronization overhead, and the Actor IMM 2D variant delivers the best overall performance due to its efficient co-occurrence matrix preprocessing. The strong scaling experiments (Figure 1), parameter sensitivity analysis (Figures 3 and 4), and kernel decomposition study (Figure 2) all closely match the trends reported in the original work, demonstrating the robustness of the proposed algorithms across different hardware platforms. Despite using a newer AMD EPYC-based system with significantly higher compute throughput and memory bandwidth than the original Intel-based Phoenix cluster, the fundamental algorithmic behaviors and performance characteristics remain consistent, confirming the generalizability of the authors' contributions.

References

[1] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the Spread of Influence through a Social Network. *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '03)* (2003), 137–146. doi:10.1145/956750.956769

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

[3] Shubhendra Pal Singhal, Souvadra Hati, Jeffrey Young, Vivek Sarkar, Akihiro Hayashi, and Richard Vuduc. 2024. Asynchronous Distributed-Memory Parallel Algorithms for Influence Maximization. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC24)* (2024), 1–14. doi:10.1109/SC41406.2024.00108

[4] Shubhendra Pal Singhal, Souvadra Hati, Jeffrey Young, Vivek Sarkar, Akihiro Hayashi, and Richard Vuduc. 2024. imm_hclib: Actor-based IMM Implementation (SC25 Branch). https://github.com/singhalshubh/imm_hclib/tree/sc25-scc. Accessed: 2025-11-17.

[5] Team RACKlette. 2025. SC25 Reproducibility Challenge Repository. https://github.com/Team-RACKlette/SC25-Reproducibility-Challenge. Accessed: 2025-11-17.