cuRipples: Influence Maximization on Multi-GPU Systems

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

Maurizio Drocco
maurizio.drocco@ibm.com
IBM TJ Watson Research Center
Yorktown Heights, NY

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

Antonino Tumeo
antonino.tumeo@pnnl.gov
Pacific Northwest National Lab
Richland, WA

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

ABSTRACT

Influence maximization is an advanced graph-theoretic operation that aims to identify a set of k most influential nodes in a network. The problem is of immense interest in many network applications (e.g., information spread in a social network, or contagion spread in an infectious disease network). The problem is however computationally expensive, needing several hours of compute time on even modest sized networks. There are numerous challenges to parallelizing influence maximization including its mixed workloads of latency- and throughput-bound steps, frequent and irregular access to graph data, large memory footprint, and potential load imbalanced workloads. In this work, we present the design and development of a new hybrid CPU+GPU parallel influence maximization algorithm (`CuRipples`) that is also capable of running on multi-GPU systems. Our approach uses techniques for efficiently sharing and scheduling of work between CPU and GPU, and data access and synchronization schemes to efficiently map the different steps of sampling and seed selection on a heterogeneous system. Our experiments on state-of-the-art multi-GPU systems show that our implementation is able to achieve drastic reductions in the time to solution, from hours to under a minute, while also significantly enhancing the approximation quality.

KEYWORDS

Influence Maximization, Iterative Graph Algorithms, Multi-GPU Systems, Distributed Memory Implementation

ACM Reference Format:
Marco Minutoli, Maurizio Drocco, Mahantesh Halappanavar, Antonino Tumeo, and Ananth Kalyanaraman. 2020. cuRipples: Influence Maximization on Multi-GPU Systems. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3392717.3392750

*Also with Washington State University.

ICS '20, June 29-July 2, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392750

1 INTRODUCTION

Given a graph $G = (V, E, \omega)$, an information diffusion model, and a seed set size k, the goal of influence maximization is to identify a set of k "seed" nodes that maximize the influence in G. The problem is of interest to a number of application domains including in social network analysis, and in the study of infectious disease spread. While the problem is known to be NP-hard [14], methods such as the greedy hill-climbing algorithm of Kempe et al. [14] can compute $(1 - 1/e - \epsilon)$ approximate solutions in polynomial time. However, even with these efficient approximations, the computational cost can be prohibitively high for medium-scale problems—for instance, with current tools, processing real-world networks with millions of nodes could take several hours. While the problem has several traits to benefit from parallelism, it also poses numerous challenges and consequently, there are hardly any efficient parallel implementations available (see §8 for a discussion of related work).

In this work, we target the IMM algorithm of Tang et al. [30] for parallelization. The IMM algorithm is a state-of-the-art approximation algorithm for influence maximization. The two main kernels of this algorithm are: (*i*) *Sampling*, which involves the construction of a set of θ random reverse reachable (RRR) sets; and (*ii*) *Seed selection*, where k most influential nodes are iteratively selected from the RRR sets. Since RRR sets can be constructed independently, the sampling kernel is amenable to parallelization, although the irregularity within the randomized path enumeration procedure can introduce load imbalances. On the other hand, the seed selection phase necessitates synchronized access to data and serialization, making it a challenging step for parallelization. Furthermore, the choice of different information diffusion models can also affect performance.

The emergence of heterogeneous systems, consisting of multiple multicore CPUs and graphics processing units (GPUs), introduces an additional layer of complexity to the set of challenges associated with scaling up influence maximization on modern platforms. In fact, the future exascale architectures are expected to comprise billions of GPU threads and tens of thousands of CPU threads necessitating a fundamental rethinking of graph algorithms and their parallel implementations (§3).

Contributions: This paper presents a new hybrid CPU+GPU parallel influence maximization implementation (`CuRipples`). We target pre-exascale systems comprising of multiple GPU units per node with significant amounts of CPU threads and memory

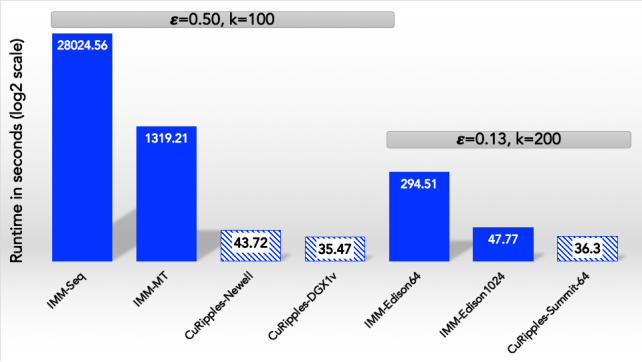


Figure 1: Our new CPU+GPU implementation, `CuRipples`, achieves a speedup of 790× over a state-of-the-art serial implementation (left), while also significantly improving the approximation factor (to $\epsilon = 0.13$) and doubling the number of seeds (right). The input network is com-Orkut. See Table 2 and §7 for more details.

per node. We also target standalone systems featuring substantial amounts of GPUs threads per node that are interconnected by specialized networking devices.

Our approach consists of three key components: (*i*) parallelization of the sampling kernel, where optimization arises from efficiently sharing and scheduling work between the participating heterogeneous (CPU and GPU) processors (§4); (*ii*) parallelization of seed selection, where data access and efficient movement of data, and synchronization between heterogeneous processors become important (§5); and, (*iii*) merging parallelization of sampling and seed selection kernels, where several trade-offs between work distribution and data movement become critical for performance. Using an array of real-world networks, we conduct a thorough empirical evaluation of `CuRipples` on three classes of multi-GPU systems and present the results in §7. The results demonstrate both significant speedups *and* significant improvements in the level of approximation that can be achieved. For instance, as shown in Fig.1, on the com-Orkut social network with 3M nodes and 117M edges, `CuRipples` was able to achieve a speedup of over 790× against the serial implementation of Tang *et al.*, while also significantly improving the approximation factor (from $\epsilon = 0.5$ to $\epsilon = 0.13$) and doubling k (from 100 to 200). The work also represents a significant improvement over the recent (state-of-the-art) MPI-based parallel implementation [25]. Our contribution enables the use of hundreds of GPUs alongside CPUs—exposing a different set of challenges and yielding a fundamentally different implementation in the process.

To the best of our knowledge, this work represents the first effort in designing and building efficient multi-GPU implementations for the execution of influence maximization algorithms at scale. Furthermore, the influence maximization algorithms targeted in this work are prototypical of iterative graph computations that are commonly used to implement various other graph operations (e.g., shortest path, community detection). Consequently, we expect this work to benefit the porting of many other graph algorithms on heterogeneous platforms involving a combination of CPUs and GPUs.

2 INFLUENCE MAXIMIZATION

Definition 2.1 (The Influence Maximization Problem). Given a graph $G = (V, E)$, a diffusion model M, and a positive integer k, the goal of influence maximization is to identify a set $S \subseteq V$ of k nodes such that $\mathbb{E}[I(S)]$ is maximized.

Here, $I(S)$ denotes the influence of the seed set S on G—i.e., the number of nodes in G that are likely to be activated by the nodes in S, as given by the diffusion model M. Two models of diffusion are used in most current implementations.

i) **Independent Cascade (IC) model:** Each edge $e \in E$ is associated with a probability $p(e)$ of activation. The diffusion process proceeds as follows: at every time step t, each node u that was activated at time step $t-1$ gets a single opportunity to trigger the activation of its neighbors with their respective (u, v) edge probabilities, to reveal the next frontier of activated nodes. The number of time steps is bounded by the diameter of G.

ii) **Linear Threshold (LT) model:** Each edge $(u, v) \in E$ is associated with a weight that reflects the degree of influence of node u on v, and each vertex $v \in V$ is associated with a node-specific threshold θ_v[1]. The diffusion process proceeds as follows: at each time step, any vertex $v \in V$ is activated if the sum of the influence from all activated neighbors of v exceeds θ_v. Once activated, the node stays activated for all remaining time steps. The number of time steps is bounded by $|V|$.

Relation to Other Graph Operations: We note here that the iterative computation structure described in the above two diffusion processes of IC and LT, are not necessarily unique to influence maximization, and are in fact commonly observed in other graph codes that are used in the parallel implementations of several other graph operations. More specifically, the wavefront computation pattern described in the IC model is similar to the data-driven algorithmic abstraction [16]—for which examples include topological sort, breadth first search (BFS), the Dijkstra's algorithm for single source shortest path (SSSP), and greedy heuristics for distance-1 coloring [5, 20]. On the other hand, the repeated neighborhood querying computation pattern described in the LT model resembles the topology-driven algorithmic abstraction [16]—for which examples include the Bellman-Ford SSSP algorithm [2, 12, 26], and heuristics for modularity-based community detection [3, 21].

However, what distinguishes influence maximization from these other iterative graph abstractions is the stochasticity that is involved in the underlying diffusion process, which introduces an additional layer of complexity in the way the information spread over the graph is accessed and updated as the algorithm proceeds.

2.1 IMM Algorithm

We use the IMM algorithm by Tang et al. [30] as our serial baseline for parallelization. The IMM algorithm provides $(1 - 1/e - \epsilon)$-approximate solutions. It is an extension of the Reverse Influence Sampling (RIS) algorithm [4], which uses a notion called reverse reachability. While the IMM algorithm and our parallel implementation presented in this paper are defined for both models of diffusion (LT and IC), we use the probability notation of the IC model here simply to ease exposition of the core concepts.

[1]The values for θ_v are an i.i.d. collection of random variables over $[0, 1]$.

Definition 2.2 (Reverse Reachable (RR) Set). Let g denote a graph obtained by removing each edge e in $G(V, E)$ with a probability of $1 - p(e)$, and let v denote an arbitrary node in G. The *reverse reachable set* for v in g is given by:

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

Definition 2.3 (Random Reverse Reachable (RRR) 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 node u appears in R_v, then it implies that u has a chance to influence v, and therefore, nodes appearing in many RRR sets are candidates for highly influential seeds. The IMM algorithm formalizes this intuition into a three step algorithm, as shown in Algorithm 1. The main idea is to generate a certain number (θ) of RRR sets as "samples", and then compute a set of k nodes that provide maximum coverage over the set of samples, as the seed set S. To determine θ, the algorithm runs an estimation procedure that internally calls the sampling and seed selection routines iteratively (up to $\log |V|$ times). For more details on the estimation procedure, refer to Tang et al. [30]. Overall, the dominant contributors to the total runtime are the sampling and seed selection routines (called from either within the estimation procedure or subsequently).

Algorithm 1: IMM Algorithm

```
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 PARALLEL MULTI-GPU SYSTEMS

We focus our work on the following three state-of-the-art multi-GPU systems: (*i*) DGX-1v, (*i*) Newell, and (*i*) Summit. All the three systems feature multiple NVIDIA Tesla V100 boards, based on the NVIDIA Volta architecture (GV100 GPU) and second generation NVLINK interconnect but connected with different topologies.

The GV100 version used in the Tesla V100 implements 80 Streaming Multiprocessors (SMs). Each SM includes: 64 floating point single precision (FP32) cores, 64 integer cores (INT32), 32 floating point double precision cores (FP64), 8 Tensor Cores, and four texture units. Each SM has a combined L1 cache, texture cache, and shared memory of 128 KB. Up to 96 KB of this memory can be sequestered as shared memory (on-chip scratchpad). A key innovation of the Volta SMs is the way warps are executed. While instructions for threads are still issued in warps (group of 32 threads), their execution is now independently controlled, speeding up those cases where they diverge. The design provides a total of 6144 KB of L2 cache and 8 memory controllers at 512-bit width (4096-bit in total) to interface with 4 High Bandwidth Memory 2 (HBM2) stacks. The GPU runs at 1333 MHz but supports (boost) clocks up to 1530 MHz. All the boards in our multi-GPU systems implement NVLINK2, which provides six links with an aggregate 300GB/s bidirectional bandwidth and support for remote atomic memory operations.

DGX-1v. consists of 8 Tesla V100 with 16 GB of HBM2 memory (1750 MHz) each, and two Intel Xeon E5-2698 v4 (Broadwell architecture) at 2.20GHz. The CPUs host 20 HyperThreaded cores (for a total of 80 threads) and 50MB of L3 cache. In the NVIDIA DGX-1 Server design, the GPUs only use point-to-point communication. Each GPU directly communicates with only 4 other GPUs (following the original NVLINK design in Pascal with only 4 links: with NVLINK2 and Volta two connections benefit of an additional link for higher interconnect speed, i.e., 100 GB/s instead of 50 GB/s), while reaching the remaining 3 needs an additional hop. This creates 2 blocks of 4 fully interconnected GPUs, each of which connects to a different CPU socket through PCI-e. Communication across the two CPUs happens through QuickPath Interconnect (QPI). This organization does not allow peer-to-peer memory access and thus remote atomic memory operations for some of the GPUs across the two blocks. The system integrates a total of 512 GB of DDR4 RAM memory at 2133 MHz.

Newell. It is a testbed cluster (5 nodes) hosted at Pacific Northwest National Laboratory mimicking the configuration of the Department of Energy Sierra Leadership-class machine hosted at Lawrence Livermore National Laboratory. Each node is based on the IBM Power AC922 node design and hosts two IBM POWER9 processors and 4 Tesla V100 GPUs with 16 GB of HBM2 memory. The POWER9 processors used in Newell present 16 cores each with 4 threads per core, run at 2.9 GHz and host a 80 MB L3 cache build with eDRAM. A group of 2 GPUs connects each other and with one of the processors using three NVLINK2 channels in peer-to-peer fashion (thus, 150 GB/s to and from CPUs and the other GPU). The two CPUs connects each other through IBM's X-Bus at 64 GB/s, which however allows to route NVLINK packets between sockets. Each node integrates a total of 1 TB of DDR4 memory, and a dual-rail Mellanox Infiniband EDR interconnect (up to 12.5 GB/s for each rail, for a total of 25 GB/s bidirectional).

Summit. It is a Department of Energy Leadership-class machine hosted at the Oak Ridge National Laboratory. The system presents a similar configuration to Livermore's Sierra and the Newell testbed nodes. However, instead of 4, it uses 6 GPUs per node. The two POWER9 Processors are slightly different as well, integrating 22 cores (technically, the processor has 24 cores, but 2 are disabled) with 4 threads each at 3.07 GHz and 110 MB of L3 cache. Each CPU socket thus connects to 3 GPUs. Each block of 3 GPUs and CPUs is fully interconnected with 2 NVLINK2 channels towards each component, reaching 100 GB/s of peer-to-peer bandwidth. Each node has 512 GB of DDR4 memory, and dual rail Infiniband EDR interconnect. Summit has a total of 4,608 nodes, and at the time of this writing is listed as the #1 machine on the TOP500 list[2].

4 PARALLELIZATION OF SAMPLING

The goal of the sampling phase is to generate a predetermined number (say, θ) of RRR sets (for use as samples). As defined in §2, each RRR set (R_v) is defined for some node v in G, and it represents a

[2] https://www.top500.org/resources/top-systems/

collection of other nodes u that have a (directed) path to v, where the path is composed of nodes activated using one of the two diffusion models.

While the generation of RRR paths are independent of one another, there are multiple challenges that underlie parallel design. First, since both diffusion models incorporate stochasticity in their node activation schemes, the computation of RRR sets corresponds to performing θ randomized "visits" of the graph. In the case of IC, this visit corresponds to a probabilistic breadth-first search (BFS), whereas for LT, it corresponds to a random walk. The resulting workloads associated with these visits could vary due to the randomness, and therefore the granularity of tasks to execute between the CPU and GPU resources can impact parallel performance. Second, even within the same visit, the growing and shrinking of frontier nodes can generate non-uniformity. Furthermore, the implementation needs a stable pseudo-random number generator to ensure reproducibility (regardless of the number of CPU and GPU resources used).

To overcome the above challenges, we present a *dynamic work scheduling-based task parallel design*, in which each task performs a subset of the visits. The progress is based on an integer variable, shared by the tasks. Once scheduled, each task increments this variable atomically, until θ is reached. This simple coordination scheme reduces potential load imbalances due to the random nature of visits, and allows tasks to perform arbitrary number of visits.

4.1 Mixed CPU/GPU Workloads

We split tasks into two categories, to match the heterogeneous CPU/GPU hardware configurations: i) *CPU-only* tasks exploit a single CPU to perform the visits; and ii) *CPU+GPU* tasks exploit both a CPU and a GPU. Correspondingly, the code executed by a task is either mapped to a (logical) CPU, or in case of CPU+GPU tasks, is also offloaded to a GPU. We leverage OpenMP parallel regions for multi-threading and thread-to-CPU mapping, and CUDA for GPU processing.

We tune the configuration with respect to specific hardware features (e.g., number of physical CPU cores and GPU boards) by setting the maximum numbers of CPU-only and CPU+GPU tasks that can be in execution at the same time. Allowing more CPU+GPU tasks than physical GPU boards enables the exploitation of parallel offloading to a single GPU (Hyper-Q) —a hardware-accelerated mechanism on recent CUDA boards [9]—to maximize the utilization of GPU devices.

4.2 Parallel Sampling for the IC Model

We design the algorithm for IC visits in CPU+GPU tasks as a minor variant of a classical top-down BFS. More specifically, after selecting a node v from the frontier at level t, we randomly select a subset of its unvisited neighbors—rather than all of them, as in a full BFS—to populate the next frontier at level $t + 1$. On the GPU side, this is implemented as a modification of the optimized BFS included in the CUDA `nvgraph` library[10]. For the generation of pseudo-random number sequences, we perform a parallel generation using the Leapfrog technique[1], which guarantees reproducibility of the sequence over varying number of threads. Moreover, we keep track of how many threads are active within a visit[3], and we *rotate* the sub-sequences among the threads, between consecutive visits, to balance the overall consumption and avoid periodic repetitions.

Each CPU+GPU task performs a single IC visit, since the execution time for visiting a non-trivial graph is sufficient to avoid granularity issues. The visit on GPU produces a binary visited/unvisited mask. Once the mask has been transferred back to the CPU, the task performs a scan to compact it into the list of visited nodes, as requested by the selection phase.

4.3 Parallel Sampling for the LT Model

We designed a similar scheme to implement LT-based sampling using GPU processing. The main difference lies in the random variables associated with nodes (as opposed to edges); however the nature of probabilistic activation is similar to that of IC. The depths of random walks produced in LT are dictated by the probabilities in the diffusion model. However, it is expected that if a simulation with both LT and IC reaches similar depths, the one with the IC model can activate more than one vertex at each level of the spanning tree. For this reason, in practice, we expect LT to yield a non-uniform and sparser coverage of nodes in the graph across the iterations, compared to IC. This was corroborated in our experiments (§7)—while visits for the IC model yielded sets with size uniformly distributed, the LT visits yielded very skewed size distribution with mostly small sets, containing less than 10 nodes in more than 99% of them. Consequently, both the execution time and memory footprint for a single LT visit are extremely limited. To tackle this challenge, we exploit the massive number of GPU cores by performing multiple BFS in parallel, where each search is performed on a given GPU thread. Note that this coarser parallelism is in contrast to the IC model where a single search is performed in parallel by a set of threads.

Each GPU thread performs an LT visit if the visited set does not contain more than 8 nodes. With this constraint, all the working memory, including the output set, fits in the thread's registers on the GPU boards we consider in this work (§3). If the set being visited exceeds 8 nodes, the thread *invalidates* the visit by producing an empty set and marking down the value of the root, so that the CPU will be able to re-compute the visit. Therefore, the post-processing on CPU of the sets produced by the GPU is more complex than in the IC case. Once the results are transferred back to the CPU—8x 64-bit integers for each GPU thread—the CPU scans the sets and re-computes the visits corresponding to the empty sets, starting from the marked roots.

The implementation of the pseudo-random generation step is similar to that of IC, except that we store an additional sub-sequence for each CPU thread executing CPU+GPU tasks (since they could recompute visits exceeding the 8-node limit).

Note that the overall memory requirements of the algorithm is dictated by the space required to store all the RRR sets in the sampling phase. Consequently, the worst-case space complexity is $O(\theta \times |V|)$, which tends to be a loose upper bound in practice.

[3]The BFS implementation from `nvgraph` spawns a fixed number of threads, independently of input size.

5 PARALLELIZATION OF SEED SELECTION

The IMM algorithm selects the set of seeds S of cardinality k by solving a maximum coverage problem over the collection of reverse reachable sets ($\mathbb{R}$) produced as the output of the sampling phase. The method uses a simple greedy iterative approach with k iterations [31]. At each iteration i, the algorithm performs three steps: It counts the number of occurrences of every vertex in $\mathbb{R}$. Next, it adds a/the node with the highest frequency into the seed set S. Next, it updates $\mathbb{R}$ by removing any RRR set that contains this most frequent node. The iterative procedure stops after k iterations, when $|S| = k$.

The greedy strategy to select the next most frequent node at each iteration, makes the algorithm harder to parallelize. To overcome this challenge, we devised a combination of efficient schemes to access and update data (with reuse from previous iterations where possible)—as described below.

5.1 Seed selection on CPUs

We perform counting in parallel, by partitioning the vertex space across all the threads, such that a given CPU thread updates the counts for only those nodes that fall in its interval. This strategy avoids the use of expensive atomic memory operations, but at the expense of scanning the entire $\mathbb{R}$ sequence from each CPU thread. The added advantage, however, is that it makes the memory access more regular across CPU threads, and thus helps improve cache utilization. The computation of the most frequent node is achieved using a simple reduction.

The iterative greedy selection requires that at each iteration i a new seed v_i is to be selected by considering only $\mathbb{R}_+^{(i-1)}$, which is the subset of RRR sets in $\mathbb{R}$ that do *not* contain any node selected so far (i.e., until iteration $i-1$). Obtaining the updated counters for the selection of v_i can be done in two ways: either by building a new histogram from $\mathbb{R}_+^{(i-1)}$; or by starting from the histogram used to select v_{i-1} and decrementing the counters for those nodes contained in $\mathbb{R}_-^{(i-1)}$, which is the subset of paths that contained v_{i-1}. We designed a dynamic adaptive scheme that is aimed at minimizing the number of RRR sets that need to be examined at every iteration; more specifically, by reorganizing $\mathbb{R}$ so that $\mathbb{R}_+^{(i-1)}$ and $\mathbb{R}_-^{(i-1)}$ are contiguous and by only examining the smallest of these two subsets. The corresponding two sequences are built using an in-place parallel partitioning algorithm that uses a divide and conquer approach—where, the divide phase is used to compute partial solutions on each CPU thread, and a parallel merging procedure is used to combine these solutions. At each step of the merge tree, the CPU threads that are not directly involved in the reduction help the other threads to perform data movement as needed.

5.2 Seed Selection on CPU+GPU

For the hybrid CPU+GPU execution, we first note that the computation of histograms is data parallel and therefore well suited for the acceleration on GPUs. However, executing solely on GPUs may become challenging due to large memory constraints. As we will see in the experiments (§7), the IMM algorithm can produce very large $\mathbb{R}$ collections requiring terabytes of memory. To better negotiate this large memory requirement, we implemented an efficient hybrid CPU+GPU parallel approach.

For our hybrid CPUs+GPU implementation, we dedicate one CPU thread per available GPU to perform data movement and the kernel launch on the device. The remaining CPU threads will co-operate as described in §5.1 on their portion.

At the start of seed selection, the $\mathbb{R}$ collection is divided equally among the GPUs. Each of the corresponding manager (CPU) threads will then move as much of the chunk as possible into the device memory. Once data movement is complete, the leftovers are merged into a contiguous sequence using the parallel merge procedure (§5.1). Then, CPUs and GPUs build their local histograms: one shared among all CPU threads, and one per GPU. The local histograms are then reduced into a global histogram by performing a tree reduction on the GPUs. Our implementation uses peer-to-peer communication to reduce the GPU histograms, and uses overlapping movement of the CPU histogram to GPU memory. To cope with the fact that in some of the systems (i.e., DGX-1) not all GPUs are directly connected, we first build a graph of the peer-to-peer connections by asking the CUDA runtime the availability of atomic memory operations between GPUs, and then follow it to perform the tree reduction. On POWER9-based architectures, IBM X-Bus allows GPUs without a direct NVLINK2 link to still perform direct atomic memory operations. Finally, the next seed is selected on the root GPU of the reduction tree.

The portion of $\mathbb{R}$ processed on the GPUs is represented in Coordinate List (COO) format, which allows the alignment of memory accesses from neighboring threads on the GPUs, and helps the memory controller to coalesce memory reads.

6 EXPERIMENTAL SETUP

Implementation. The implementation of `CuRipples`, our hybrid CPU+GPU parallel approach, uses: i) OpenMP + CUDA programming model in the single node implementation; and ii) MPI + OpenMP + CUDA for the distributed implementation. The implementation will be made publicly availableas part of the `Ripples` framework from Minutoli et al. [24] .

Input data. For all our experiments, we used a collection of seven real-world networks, representing a wide range in size, from the SNAP data set [17]. Table 1 shows the basic size and degree statistics of these input networks. In our setup, we randomly assigned the node or edge probabilities (as determined by the model) from the interval [0, 1]. Since the input graph is small compared to the RRR set collection that the algorithm produces, we replicate the graph on each node in the distributed implementation.

Platforms. Experiments were conducted on three machines: DGX-1v, Newell, and Summit (described in §3). `CuRipples` on Newell and Summit were compiled using GCC 8.1 and CUDA 10.1 with optimization enabled using "`-O3`" switch. The parallel implementations on Summit uses the default MPI implementation (spectrum-mpi 10.3). The implementations on DGX-1v use GCC 6 and CUDA 9.0.

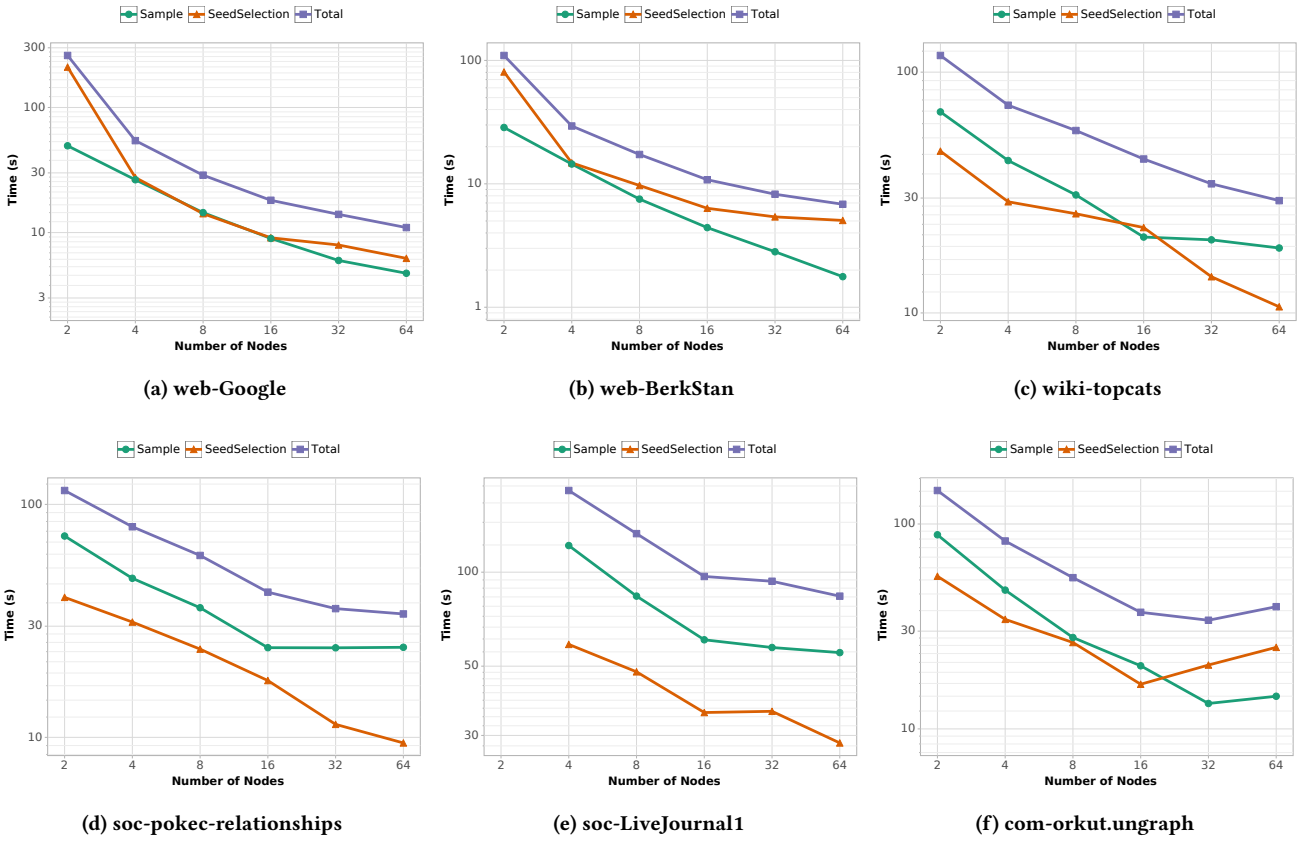


Figure 2: Scaling on Summit with the IC Model. Parameters: $\epsilon = 0.13$, $k = 100$.

Table 1: Input Graphs

Graph	Nodes	Edges	Avg. Degree	Max Degree
cit-HepTh	27,770	352,807	12.70	2,468
web-Google	875,713	5,105,039	11.66	6,353
web-BerkStan	685,230	7,600,595	22.18	84,290
wiki-topcats	1,791,489	28,511,807	31.83	3,907
soc-Pokec	1,632,803	30,622,564	37.51	20,518
soc-LiveJournal1	4,847,571	68,993,773	28.47	22,889
com-Orkut	3,072,441	117,185,083	76.28	33,313

7 EXPERIMENTAL RESULTS

7.1 Comparative Evaluation

We first compare `CuRipples` against state-of-the-art serial and parallel implementations. More specifically, the implementations we use are as follows:

- IMM_{seq}: the original IMM algorithm by Tang et al. [30];
- IMM_{opt}: hand-tuned and optimized CPU implementation [25] of IMM_{seq};
- IMM_{mt}: OpenMP multithreaded implementation of IMM [25]; and
- $\text{IMM}_{\text{edison}}$: distributed implementation using MPI/OpenMP and running on the NERSC Edison supercomputer [25];

We used two configurations of parameters: ($\epsilon = 0.5$, $k = 100$) and ($\epsilon = 0.13$, $k = 200$). Note that the latter configuration was beyond the memory limit of single node implementations (IMM_{seq}, IMM_{opt}, and IMM_{mt}).

Table 2 summarizes the results of our comparative evaluation, for the two largest inputs. Our evaluation shows that `CuRipples` achieves speedups of 790× and 251×, respectively on com-Orkut and soc-LiveJournal1, compared to IMM_{seq}. Against the state-of-the-art single node parallel implementation (IMM_{mt}), `CuRipples` delivers up to 37.19× on com-Orkut and up to 15.72× on soc-LiveJournal1.

As for the distributed implementations, we observe from Table 2 that `CuRipples`, when run on 2.6K CPU cores + 384 GPUs, generates a time-to-solution which is roughly comparable to the time-to-solution achieved by $\text{IMM}_{\text{edison}}$ on 48K CPU cores. In fact, when run on a similar configuration of 64 nodes, `CuRipples` is able to achieve speedups of 8.31× and 1.79× over $\text{IMM}_{\text{edison}}$ (for the two inputs respectively).

Collectively, these comparative results show that our new hybrid CPU+GPU parallel implementation, `CuRipples`, is able to achieve one to two orders of magnitude performance improvement over state-of-the-art single node implementations, while also able to

Table 2: Comparative evaluation of cuRipples relative to previous implementations of IMM—both serial (IMM_{seq}) [30] and parallel ($IMM_{opt/mt/edison}$) [25]. Abbreviations used: No. Cores (C), GPUs (G), Nodes (N).

System	Time (s)	Speedup	Scale
com-Orkut (ϵ=0.5,k=100)			
IMM_{seq}	28024.56	1.00×	1C
IMM_{opt}	9027.50	3.10×	1C
IMM_{mt}	1319.21	21.24×	20C (1N)
$CuRipples_{dgx\text{-}1v}$	35.47	**790.09**×	80C+8G (1N)
$CuRipples_{newell}$	43.72	641.00×	128C+4G (1N)
com-Orkut (ϵ=0.13,k=200)			
IMM_{edison}	294.51	95.16×	3,072C (64N)
IMM_{edison}	47.77	586.61×	49,152C (1024N)
$CuRipples_{summit}$	36.30	**772.03**×	2,688C+384G (64N)
soc-LiveJournal1 (ϵ=0.5,k=100)			
IMM_{seq}	16434.81	1.00×	1C
IMM_{opt}	3954.59	4.16×	1C
IMM_{mt}	1026.21	16.02×	20C
$CuRipples_{dgx\text{-}1v}$	70.23	234.01×	80C+8G (1N)
$CuRipples_{newell}$	65.26	251.84×	128C+4G (1N)
soc-LiveJournal1 (ϵ=0.13,k=200)			
IMM_{edison}	190.94	86.07×	3,072C (64N)
IMM_{edison}	55.12	298.16×	49,152C (1024N)
$CuRipples_{summit}$	106.43	154.42×	2,688C+384G (64N)

demonstrate highly competitive results both in terms of performance and quality (approximation factor, seed set size) on distributed platforms. The results also show that CuRipples is able to process a real-world graph with millions of nodes in just over (or under) a minute.

7.2 IMM versus Greedy Hill Climbing

Since this work is the first to consider performance of IMM on GPUs, we provide a brief comparative assessment of our performance in the context of prior work [19, 27] that is limited to variations of the greedy Hill Climbing algorithm of Kempe et al. [14]. While our work focuses on state-of-the-art multi-GPU systems, prior work is limited to the use of a *single GPU* for accelerating the computation [19, 27]. However, to the best of our efforts, we were unable to find publicly available GPU-based implementations of these prior works to enable a direct comparison of performance. Consequently, we provide a qualitative assessment in this section to demonstrate the superior performance of IMM relative to the greedy Hill Climbing method. Our observations corroborate the observations of Tang et al. [30] in demonstrating the work efficiency of IMM.

We use implementations of greedy Hill Climbing and IMM algorithms in open source tool provided by Minutoli et al. [25] for this comparison. Due to code availability, we only performed CPU-based comparisons using a shared-memory system. The largest input that we were able to process with parallel Hill Climbing algorithm is *cit-HepTh*. We set the edge weights uniformly at random distributed between [0; 1], and the seed set size is 100 seeds. Using 1000 samples, the total time for Hill Climbing algorithm is 6804s, while IMM completed the execution in 140s (a speedup of 48.6×) with the same edge weights and $\epsilon = 0.13$.

Based on a set of smaller inputs (not presented here due to space limitation), our observations corroborate the results obtained in the experimental evaluation of Tang et al. [30]. We further observed that relative to the Hill Climbing algorithm, IMM is not only faster but also provided solutions with better approximation guarantees, for the sample sizes we used due to limitations of compute time. The best known theoretical bounds for the number of samples indicate that over a million samples will be required for *cit-HepTh* to match with the approximation guarantees provided by IMM with $\epsilon = 0.13$ [28]. Consequently, such sampling efforts would render the Hill Climbing method impractical for large scale inputs with any meaningful levels of approximation guarantees.

7.3 Performance Evaluation of the IC Model

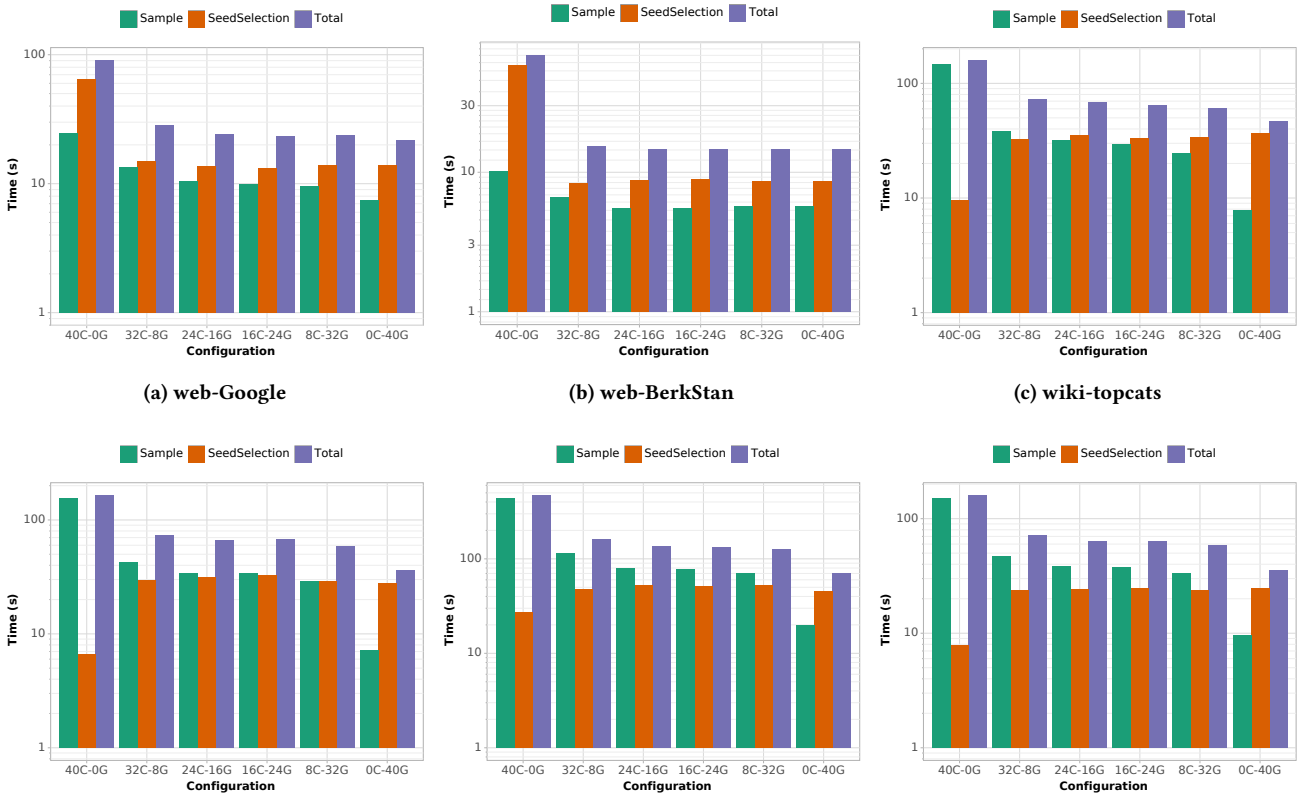
Single Node Evaluation (on DGX-1v). Next, we evaluate the scalability of CuRipples, on DGX-1v. Since the choice of the diffusion model also impacts performance, in this section we present the results of analyzing CuRipples using the IC model.

Figure 3 shows the runtime as a function of the number of CPU cores used. Note that for our hybrid CPU+GPU executions, we used all of the available GPUs[4]. Furthermore, while the sampling procedure was always run in the CPU+GPU hybrid mode, the seed selection kernel has two variants, one using the hybrid mode, and another using the CPU-only mode—as described in §5. Therefore we present both sets of results, with Figure 3a-f showing the results where the seed selection was also run in the hybrid mode, while Figure 4a-f showing the results where the seed selection was run in the CPU-only mode.

The experimental results show that with the exception of the two smallest inputs (web-Google and web-BerkStan), for the IC model, CuRipples generally runs faster when the seed selection kernel is run in the CPU-only mode as opposed to the hybrid mode. This shows the effectiveness of our data access and update mechanisms, while also exposing some of the data movement related challenges inherent in seed selection in the hybrid mode. When the computation is dominated by seed selection, the results show the effectiveness of our hybrid CPU+GPU implementation. In fact for web-Google and web-BerkStan in the IC model (and also all the results for the LT model), a significant boost in performance was observed once the GPU-accelerated seed selection is enabled. The seed selection line then stays flat because the number of GPUs used remains fixed.

Evaluation of Distributed Implementation (on Summit). Next, in Figure 2, we present strong scaling results for running our distributed implementation for CuRipples on Summit using the IC model, with a production-scale setting for ϵ(=0.13). We note that

[4]Note that the zero "CPU threads pushing work to GPUs" means that no work is done on the GPUs and therefore corresponds to a CPU-only execution.



(a) web-Google (b) web-BerkStan (c) wiki-topcats

(d) soc-pokec-relationships (e) soc-LiveJournal1 (f) com-orkut.ungraph

Figure 3: DGX-1v IC Model ($\epsilon = 0.5$, $k = 100$). The configuration reports the number of CPU workers(C) and GPU workers(G)

while running the input soc-LiveJournal1 on 2 nodes, the experiments ran out of memory (over 1TB). As can be observed in Figure 2, the distributed `CuRipples` scales considerably well on up to 64 nodes of Summit, generating speedups between 4× and 23× relative to the performance on two nodes of Summit.

7.4 Performance Evaluation of the LT Model

In this section, we present the results of evaluating `CuRipples` using the LT model.

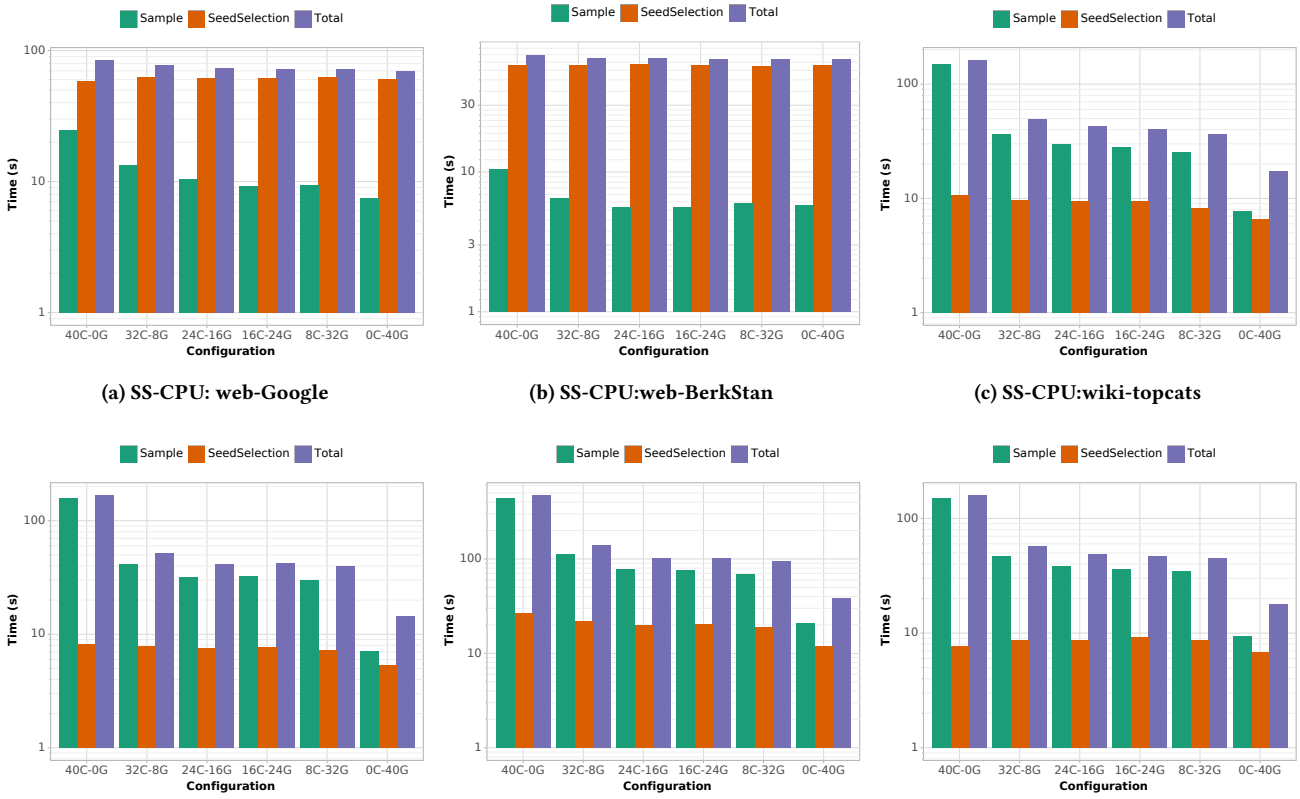
Single Node Evaluation (on DGX-1v). Figure 5a-c show the performance of `CuRipples` on the DGX-1 using the LT model. While we have observed that both seed selection and sampling can be potentially dominating for the IC model, the seed selection algorithm is the clear dominating factor in the case of the LT model. This is because of the sparsity in the set of vertices activated in LT.

We also noticed that by enabling the hybrid CPU+GPU algorithm for seed selection brings a significant (~ 10×) improvement in performance for LT (charts for CPU-only seed selection not shown due to space limitations). Figure 5 also shows that after enabling the use of the hybrid mode (happens after number of CPU cores pushing work to GPU is greater than zero), the performance largely plateaus. In fact, the seed selection portion does not use the same over-subscription mechanism used by the sampling phase because the algorithm tries to maximize the data stored in GPU memory and the steps become data parallel in nature.

Evaluation of Distributed Implementation (on Summit). For the LT model, the runtime results of executing `CuRipples` on up to 64 nodes of Summit are shown in Figure 5d-f. The results show almost linear scaling for the sampling step; however, the overall scaling is affected by the poor performance scaling of the seed selection step. In our experiment, we observed that the LT model tends to produce significantly fewer reverse reachable paths (than IC). Due to the limited size that is peculiar to the LT model, the algorithm performs more iterations of the sampling-seed selection loop. Consequently, the algorithm incurs a much higher communication cost needed for the global reduction to select the next seed, without producing enough work during the sampling phase to amortize the communication cost.

8 RELATED WORK

The seminal work by Kempe et al. [14] formalized the problem of influence maximization. The approach used in their work involves a greedy hill-climbing optimization (submodular maximization)



(a) SS-CPU: web-Google (b) SS-CPU:web-BerkStan (c) SS-CPU:wiki-topcats

(d) SS-CPU:soc-pokec-relationships (e) SS-CPU:soc-LiveJournal1 (f) SS-CPU:com-orkut.ungraph

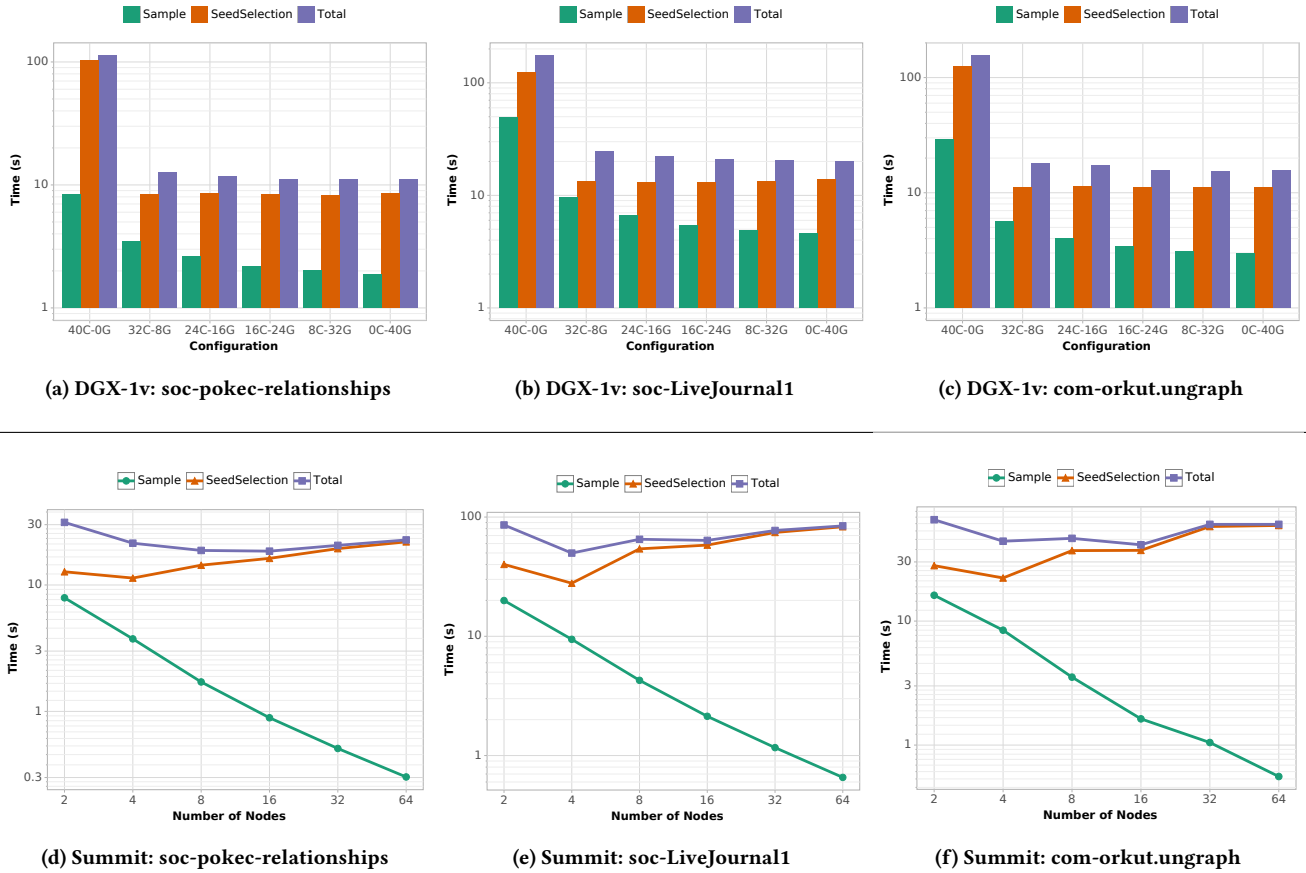
Figure 4: DGX-1v IC Model with Seed Selection on CPU ($\epsilon = 0.5$, $k = 100$). The configuration reports the number of CPU workers(C) and GPU workers(G).

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 ($\sim 10K$) while reachability computations involve BFS kernels. Thus, the combined complexity of the overall method 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 [14] is addressing the scalability aspects of the overall method. Two sub-themes along this line of research focus on accelerating the submodular optimization approach and accelerating the oracle computation.

Leskovec et al. [18] adopt a lazy-greedy submodular maximization approach called CELF (Cost-Effective Lazy Forward), which was later extended into CELF+ by Goyal et al. [13]. By exploiting the submodular property, these works reduce wasteful computations and improve performance [23]. To improve performance and to extend problem size reach to graphs with hundreds of thousands to millions of nodes, heuristic approaches were also introduced (e.g., [6, 7]).

The concept of per-node summary structures called combined reachability sketches are leveraged by Cohen et al. [8] to speedup the influence computations. Borgs et al. [4] introduced 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. [29] where they provide an efficient practical implementation of the algorithm presented in [4] resulting in the ability to analyze billion-edge graphs in hours.

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×–6× on 8 cores. Du et al. [11] report parallel running times of their improved continuous-time influence maximization algorithm but they do not provide any details on the scaling behavior. Lucier et al. [22] 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. [32] propose a parallel algorithm using k-shell decomposition. However, a thorough evaluation of the parallel performance is missing. Liu et al. [19] and Pal et al. [27] have explored the acceleration of Hill Climbing algorithm from Kempe et al. [14] using a *single GPU*.



(a) DGX-1v: soc-pokec-relationships **(b) DGX-1v: soc-LiveJournal1** **(c) DGX-1v: com-orkut.ungraph**

(d) Summit: soc-pokec-relationships **(e) Summit: soc-LiveJournal1** **(f) Summit: com-orkut.ungraph**

Figure 5: *LT Model*: Top row: DGX-1v ($\epsilon = 0.5$, $k = 100$). The configuration for labels in the x-axis reports the number of CPU workers(C) and GPU workers(G). Bottom row: Scaling on Summit ($\epsilon = 0.13$, $k = 100$). Each node has 6 GPUs and 42 CPU cores.

Please refer to §7.2 for a qualitative comparson between the work presented in this paper and the work of Liu et al. [19] and Pal et al. [27]. Recently, Minutoli et al. [25] developed a parallel CPU-only multithreaded and distributed implementations for the IMM algorithm. The work presented in this paper extends this line of work and presents the first generation of parallel implementations for multi-GPU and CPU/GPU heterogeneous platforms.

9 CONCLUSION AND FUTURE WORK

We presented efficient hybrid CPU+GPU parallel implementations for influence maximization for execution on multi-GPU systems. Through a combination of scheduling techniques, our approach balances the workload between CPUs and GPUs, and with improved data access and synchronization techniques it reduces the burden of irregular data movement. The results show that we are able to execute influence maximization at scale on multi-GPU systems on large real-world input graphs—reducing the time to solution from several hours to a few minutes, while significantly improving precision and increased number of seeds. To the best of our knowledge, this is the first hybrid CPU+(multi)GPU implementation for influence maximization, and consequently, will benefit future research in porting of applications to extreme scale architectures.

Future research directions include: i) designing learning-based schemes to better exploit precision-performance tradeoffs exposed by the influence maximization operation, and to improve the efficacy of dynamic schemes for balanced distribution of work between CPUs and GPUs; ii) characterization and comparative evaluation of the two diffusion models; iii) improved implementation for optimizing the seed selection kernel; and, iv) large-scale real-world application study in key scientific domains.

ACKNOWLEDGMENTS

We used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory (Contract No. DE-AC05-00OR22725) and the National Energy Research Scientific Computing Center (Contract DE-AC02-05CH11231). The research is supported by the U.S. DOE ExaGraph project at the Pacific Northwest National Laboratory (PNNL) and by NSF awards CCF 1815467 and OAC 1910213 at Washington State University (WSU). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

REFERENCES

[1] Heiko Bauke. Tina's random number generator library, 2011.

[2] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1): 87–90, 1958.

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[4] Christian Borgs, Michael Brautbar, Jennifer T. Chayes, and Brendan Lucier. Maximizing social influence in nearly optimal time. In Chandra Chekuri, editor, *Proc. Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 946–957. SIAM, 2014.

[5] Ümit V Çatalyürek, John Feo, Assefaw H Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Computing*, 38(10):576–594, 2012.

[6] Wei Chen, Yajun Wang, and Siyu Yang. 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*, pages 199–208. ACM, 2009.

[7] Wei Chen, Chi Wang, and Yajun Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1029–1038. ACM, 2010.

[8] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F. Werneck. Sketch-based Influence Maximization and Computation: Scaling up with Guarantees. In *Proc. ACM International Conference on Conference on Information and Knowledge Management, CIKM*, pages 629–638. ACM, 2014.

[9] Nvidia Corporation. Nvidia Tesla V100 GPU Architecture. Technical report, Nvidia Corporation, 2017.

[10] Nvidia Corporation. nvGRAPH, last accessed 2020. URL https://developer.nvidia.com/nvgraph.

[11] Nan Du, Le Song, Manuel Gomez-Rodriguez, and Hongyuan Zha. Scalable Influence Estimation in Continuous-Time Diffusion Networks. In *Advances in Neural Information Processing Systems 26*, pages 3147–3155, 2013.

[12] Lester Randolph Ford Jr and Delbert Ray Fulkerson. *Flows in networks*. Princeton university press, 2015.

[13] Amit Goyal, Wei Lu, and Laks V. S. Lakshmanan. CELF++: optimizing the greedy algorithm for influence maximization in social networks. In *Proc. International Conference on World Wide Web, WWW*, pages 47–48. ACM, 2011.

[14] David Kempe, Jon M. Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, pages 137–146. ACM, 2003.

[15] Jinha Kim, Seung-Keol Kim, and Hwanjo Yu. Scalable and parallelizable processing of influence maximization for large-scale social networks? In *Proc. IEEE International Conference on Data Engineering, ICDE*, pages 266–277. IEEE Computer Society, 2013.

[16] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel graph analytics. *Communications of the ACM*, 59(5):78–87, 2016.

[17] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[18] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne M. VanBriesen, and Natalie S. Glance. Cost-effective outbreak detection in networks. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 420–429. ACM, 2007.

[19] Xiaodong Liu, Mo Li, Shanshan Li, Shaoliang Peng, Xiangke Liao, and Xiaopei Lu. IMGPU: GPU-Accelerated Influence Maximization in Large-Scale Social Networks. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):136–145, 2014. doi: 10.1109/TPDS.2013.41. URL https://doi.org/10.1109/TPDS.2013.41.

[20] Hao Lu, Mahantesh Halappanavar, Daniel Chavarría-Miranda, Assefaw Gebremedhin, and Ananth Kalyanaraman. Balanced coloring for parallel computing applications. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 7–16. IEEE, 2015.

[21] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.

[22] Brendan Lucier, Joel Oren, and Yaron Singer. Influence at Scale: Distributed Computation of Complex Contagion in Networks. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 735–744. ACM, 2015.

[23] Michel Minoux. Accelerated greedy algorithms for maximizing submodular set functions. *Optimization Techniques*, pages 234–243, 1978.

[24] Marco Minutoli, Mahantesh Halappanavar, and Ananth Kalyanaraman. `Ripples`: Open source software. https://github.com/pnnl/ripples, 2019.

[25] Marco Minutoli, Mahantesh Halappanavar, Ananth Kalyanaraman, Arun V. Sathanur, Ryan Mcclure, and Jason McDermott. Fast and Scalable Implementations of Influence Maximization Algorithms. In *2019 IEEE International Conference on Cluster Computing, CLUSTER 2019, Albuquerque, NM, USA, September 23-26, 2019*, pages 1–12. IEEE, 2019. doi: 10.1109/CLUSTER.2019.8890991. URL https://doi.org/10.1109/CLUSTER.2019.8890991.

[26] Edward F Moore. The shortest path through a maze. In *Proc. Int. Symp. Switching Theory, 1959*, pages 285–292, 1959.

[27] Koushik Pal, Zissis Poulos, Edward Kim, and Andreas G. Veneris. Fast GPU-Based Influence Maximization Within Finite Deadlines via Node-Level Parallelism. In Petra Perner, editor, *Advances in Data Mining. Applications and Theoretical Aspects - 17th Industrial Conference, ICDM 2017, New York, NY, USA, July 12-13, 2017, Proceedings*, volume 10357 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2017. doi: 10.1007/978-3-319-62701-4_12. URL https://doi.org/10.1007/978-3-319-62701-4_12.

[28] Gal Sadeh, Edith Cohen, and Haim Kaplan. Sample Complexity Bounds for Influence Maximization. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 29:1–29:36. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPIcs.ITCS.2020.29. URL https://doi.org/10.4230/LIPIcs.ITCS.2020.29.

[29] Youze Tang, Xiaokui Xiao, and Yanchen Shi. Influence maximization: near-optimal time complexity meets practical efficiency. In *International Conference on Management of Data, SIGMOD*, pages 75–86. ACM, 2014.

[30] Youze Tang, Yanchen Shi, and Xiaokui Xiao. Influence Maximization in Near-Linear Time: A Martingale Approach. In *Proc. 2015 ACM SIGMOD International Conference on Management of Data*, pages 1539–1554. ACM, 2015.

[31] Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001.

[32] Hong Wu, Kun Yue, Xiaodong Fu, Yujie Wang, and Weiyi Liu. 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*, volume 201 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 27–36. Springer, 2016.