FuseIM: Fusing Probabilistic Traversals for Influence Maximization on Exascale Systems

Reece Neff
rwneff@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Mostafa Eghbali Zarch
meghbal@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Marco Minutoli
marco.minutoli@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

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

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

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

Michela Becchi
mbecchi@ncsu.edu
North Carolina State University
Raleigh, NC, USA

ABSTRACT

Probabilistic breadth-first traversals (BPTs) are used in many network science and graph machine learning applications. In this paper, we are motivated by the application of BPTs in stochastic diffusion-based graph problems such as influence maximization. These applications heavily rely on BPTs to implement a Monte-Carlo sampling step for their approximations. Given the large sampling complexity, stochasticity of the diffusion process, and the inherent irregularity in real-world graph topologies, efficiently parallelizing these BPTs remains significantly challenging. In this paper, we present a new algorithm to fuse a massive number of concurrently executing BPTs with random starts on the input graph. Our algorithm is designed to fuse BPTs by combining separate probabilistic traversals into a unified frontier. To show the general applicability of the fused BPT technique, we have incorporated it into two state-of-the-art influence maximization parallel implementations (gIM and Ripples). Our experiments on up to 4K nodes of the OLCF Frontier supercomputer ($32,768$ GPUs and 196K CPU cores) show strong scaling behavior, and that fused BPTs can improve the performance of these implementations up to 182.13× (avg. 75.15×) and 359.86× (avg. 135.17×) for gIM and Ripples, respectively.

CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**; **Massively parallel and high-performance simulations**; • **Software and its engineering → Massively parallel systems**.

KEYWORDS

Parallel Graph Algorithms, Influence Maximization

ACM Reference Format:
Reece Neff, Mostafa Eghbali Zarch, Marco Minutoli, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, and Michela Becchi. 2024. FuseIM: Fusing Probabilistic Traversals for Influence Maximization on Exascale Systems. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650200.3656621

1 INTRODUCTION

Given a graph $G(V, E)$, a diffusion model M and an integer budget $k > 0$, the objective of influence maximization is to compute a seed set $S \subseteq V$ of k vertices which, when activated, is likely to lead to the maximum number of activations in the graph under the diffusion model M [17]. An important class of approximation algorithms for solving this NP-hard problem are based on sampling [8, 40, 41] (detailed in §2). Here, each "sample" is a result of a single breadth-first traversals (BPT) on G, and the number of samples (θ) determines the approximation guarantee of the solution. In practice, θ as high as 10^6 is necessary to approximate close to the theoretical optimal [8, 28]. Consequently, a dominant fraction of the runtime (up to 90% [29]) is spent performing probabilistic traversals on G.

Graph traversals are a fundamental building block of graph algorithms and graph analytics [2, 39]. In particular, breadth-first searches (BFSs), probabilistic BFSs, and random walks are commonly employed in graph analytics, machine learning and deep learning [5, 34, 35, 47]. For example, variants of BFS are essential to compute matchings [36], network alignment [18], and maximum flow [16], among many other examples. Random walks and probabilistic searches are now fundamental tools in graph representation learning, including the emerging area of graph neural networks [34, 47]. We are motivated by the application of probabilistic breadth-first traversals (BPT) for stochastic diffusion-based graph problems. Such applications arise in graph isomorphism tests [3] and influence maximization [40]. The stochasticity of the diffusion process is usually implemented through a Monte-Carlo sampling step that leads to performing a large number of probabilistic traversals of the graph. For instance, in influence maximization [9, 17], which

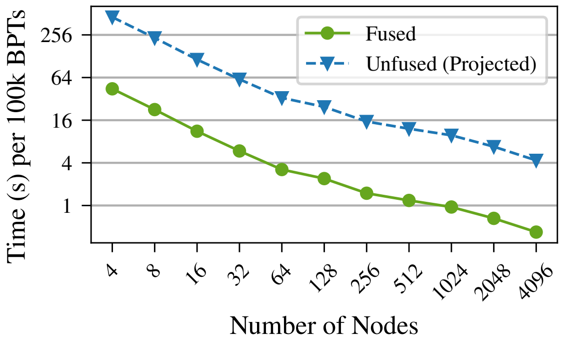

Figure 1: Distributed heterogeneous strong scaling on OLCF Frontier on the soc-LiveJournal1 graph for fused BPTs (this work, 64 batch size) compared to hypothetical performance of an unfused baseline extrapolated from smaller runs. The plot shows: 1) $\sim$ 10× **speedup for fused traversals, and** 2) **strong scaling for up to** 32,768 **GPUs and** 196K **CPU cores.**

has numerous applications in viral marketing and computational epidemiology [9, 17, 24, 29], we are interested in observing a stochastic diffusion process over an input graph in order to identify top influential nodes on the network.

Modern supercomputers, such as Frontier at the Oak Ridge Leadership Computing Facility (OLCF) – currently the #1 system on the Top 500 list and #3 on Graph500 – leverages a large number of GPUs to achieve high parallelism and high computational density. OLCF Frontier consists of more than 8K nodes, where each node has 8 AMD MI250X GPU dies, interconnected with HPE Slingshot network interfaces in a Dragonfly topology (detailed in §6). Two key challenges that limit scaling BPTs on such systems are: (*i*) the large number of BPT traversals; and (*ii*) irregular and skewed access of memory (edges), due to probabilistic traversals and irregular structure of graph topology.

We note that, in practical scenarios, thousands to hundreds of thousands of BPTs are requested. Traditional parallelization techniques process each BPT separately from one another. However, this leads to redundant edge accesses, where the sets of edges traversed by different BPTs overlap. In this work, we leverage this observation to propose a more memory-efficient implementation that reduces edge access redundancy by processing multiple BPTs together.

Contributions: Our work introduces `FuseIM`, which utilizes the technique of *fused breadth-first probabilistic traversals* for heterogeneous distributed systems. `FuseIM` can be applied to any parallel use-case that executes multiple BPTs [5, 34, 35, 40, 47]. We make the following contributions.

- *Algorithms:* We present the *fused BPT* algorithm that fuses many BPTs with the goal of reducing the net number of visits per edge, thereby reducing time-to-solution (§3).
- *Heuristics:* We present several heuristics (vertex reordering, workload balancing) to improve the performance of our parallel implementation (§5).
- *Implementations:* We show the efficacy of fused BPT by incorporating it into two state-of-the-art parallel influence maximization implementations, namely Ripples [28] and gIM [37].
- *Results:* Our experiments were conducted on 4K nodes of the OLCF Frontier supercomputer. Our results on real-world inputs show up to 359.86× and 182.13× speedup over Ripples and gIM respectively, with an average speedup of 135.17× and 75.15×. We demonstrate the effectiveness of fused BPTs to decrease the number of edge accesses (§7).

To the best of our knowledge, this work represents the first use of fused-BPT for influence maximization and implementation on OLCF-Frontier (Fig. 1), the first exascale system. We believe that this work will not only benefit the application and use of influence maximization, but also motivate the use of fused traversals in other scientific applications.

2 BACKGROUND ON INFLUENCE MAXIMIZATION

The Influence Maximization Problem (Inf-Max) is the problem of finding a small cohort of vertices that optimize the outcome of a diffusion process in activating vertices over the input network (or graph). More formally:

Definition 1 (Inf-Max)**.** Let $G = (V, E)$ be a (di)graph where V is the set of vertices and E is the set of edges, M a diffusion process, and k a budget. The *Influence Maximization Problem* is to find a set of vertices $S \subseteq V$, called seeds, such that

$$\arg\max_{S \subseteq V} \sigma(S), \quad \text{s.t. } |S| \leq k \tag{1}$$

where $\sigma(S)$ is the expected influence function over G when the diffusion process M starts from the seed set S.

The problem is known to be NP-hard [17]. However, the expected influence function $\sigma(.)$ is a non-decreasing monotone submodular [17]—i.e., for subsets $A \subseteq B \subseteq V$ and a vertex $x \in V$, $\sigma(A \cup \{x\}) - \sigma(A) \geq \sigma(B \cup \{x\}) - \sigma(B)$. This resulted in a greedy hill climbing algorithm that provides $1 - 1/e$ approximation [10, 17]. An alternative class of approximation algorithms was developed using the notion of Reverse Inverse Sampling (RIS) [8]. The RIS algorithms use the notion of reverse reachability to assess influential vertices. In particular, RIS approaches build a collection of Random Reverse Reachable sets (RRR sets) by simulating the diffusion process M backward. The intuition is that if a vertex u appears in an RRR set that was generated by starting the diffusion process at vertex v, then u also has a chance of activating v during the diffusion process; and the greater number of RRR sets that u appears in, the more influential it can be. Consequently, the problem of selecting the k seeds in S reduces to computing a maximum-k-cover over the collection of RRR sets [8].

The current state-of-the-art algorithm based on RIS is the IMM algorithm of Tang et al. [40]. Tang et al. have proved a lower bound on the sample complexity (the number of RRR sets: θ) that, given the size of the input graph G, the number of seeds k, and a parameter ε, guarantees achieving a $1 - 1/e - \varepsilon$ approximation bound. In practice, θ ranges between 10^5-10^6 [28, 40]. The works by Minutoli et al. [27, 28] and Shahrouz et al. [37] provide efficient parallel implementations of this algorithm, which we use to validate our BPT fusing approach.

The network diffusion literature has generally used two simple but expressive diffusion processes: the Linear Threshold Model (LT) and the Independent Cascade Model (IC). Under LT, the probability of a

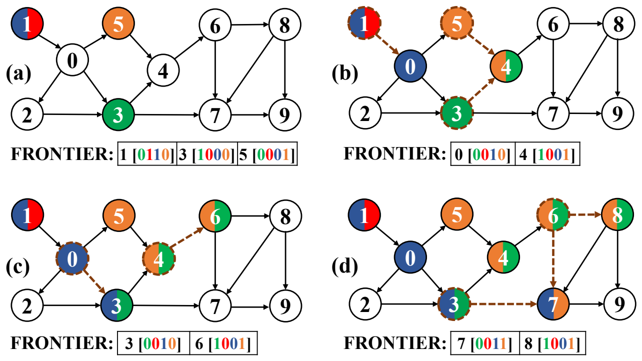
(a)
FRONTIER: 1 [0110] 3 [1000] 5 [0001]
(b)
FRONTIER: 0 [0010] 4 [1001]
(c)
FRONTIER: 3 [0010] 6 [1001]
(d)
FRONTIER: 7 [0011] 8 [1001]

Figure 2: Example of fused BPT. Four BPTs (each represented by a color) originate from vertices 1, 3 and 5. The active vertices and the edges activated in each traversal step are highlighted in brown dashed lines. The illustration shows the frontier *at the end* of each step.

```
1  /* frontier initialization */
2  for (bpt b)
3      frontier[random(0,|V|-1)].b = 1;
4
5  /* Fused random traversals */
6  for (vertex v in frontier){
7      mask fr_v = clear(frontier[v]);
8      visited[v] = visited[v] | fr_v;
9      for (edge e in v.edges){
10         vertex u = mate(e,v);
11         mask fr_u = fr_v & !visited[u];
12         for (bpt b in fr_u)
13             if (random(0,1) > e.prob) clear(fr_u, b);
           frontier[u] = frontier[u] | fr_u;  //fusing
15     }
16 }
17
18 /* RRR sets construction */
19 for (vertex v)
20     for (bpt b)
21         if (visited[v].b) RRRset(b).add(v);
```

Listing 1: Fused BPT algorithm

node activation depends on a group threshold parameter; whereas under IC, the probability of a node v activating its neighbors u is a constant $p(e = (v, u))$ that is independent from the history of the diffusion process. More specifically, at each step t, the newly activated vertices at step $t - 1$ will have a single attempt at activating their 1-hop neighbors, and they will succeed with probability $p(e)$. Moreover, the process assumes *permutation invariant*, in that the final result of activation (or not) is independent from the relative ordering of the attempts made to activate a vertex. Minutoli et al. [27] observed that the IC model is the more computationally challenging, as it could lead to an irregular and often deeper propagation into the graph, among multiple concurrently advancing probabilistic traversal fronts. We now define RRR sets under the IC model.

Definition 2 (RRR set)**.** Let v denote a vertex in G and $\hat{G}$ denote a subgraph obtained by removing each edge e of G with probability $1 - p(e)$. Then, the *Random Reverse Reachable set* for v in $\hat{G}$, denoted by $RR_{\hat{G}}(v)$, is given by:

$$RR_{\hat{G}}(v) = \{u | \text{ a directed path from } u \text{ to } v \text{ in } \hat{G}\} \qquad (2)$$

Definition 2 implies that RRR sets can be computed without explicitly generating the subgraphs $\hat{G}$. In fact, RRR sets can be equivalently computed as the visited array of a Probabilistic Breadth-First Traversal (BPT) that visit edges with probability $p(e)$ as prescribed by the IC diffusion process.

3 FUSEIM

To efficiently address the problem of performing a large number (e.g., θ for Inf-Max) of BPTs concurrently, we present a new algorithm, which we call FuseIM. The BPTs originate at vertices that are selected uniformly at random from V.

Illustrative example: Figure 2 illustrates the operation of the fused BPT algorithm. The example assumes four probabilistic traversals (each represented by a color) starting at vertices 1, 3 and 5. Each *BPT* is associated to a traversal. Note that multiple traversals can originate from the same vertex (vertex 1 in the example). For each traversal step, the figure shows the frontier queue (i.e., the active vertices) at the end of that step. For each active vertex, the mask in the frontier queue shows the BPTs *that need to be propagated* in the next step. Due to their probabilistic nature, traversals will follow only a subset of the edges outgoing from the active vertices. Accordingly, in each traversal step only a subset of the BPTs is propagated. For example, in step (b) only the blue BPT is propagated from vertex 1 to vertex 0, while the red traversal stops at vertex 1. The same vertex can be traversed multiple times (as part of different traversals), leading it to be added to the frontier in different traversal steps. For example, vertex 3 is added to the frontier in steps (a) and (c); first time as part of the green traversal, and second time as part of the blue one.

The key idea of fusing is as follows. When a vertex in the frontier is associated with multiple BPTs (as indicated by its frontier's mask), the corresponding traversals are *fused*, enabling work savings. For example, in step (b) vertex 4 is added to the frontier with two BPTs (orange and green). This causes the orange and green traversals to be fused, leading to a single traversal of vertices 6, 7 and 8. At the end of the traversal process, the vertices with the same BPT are associated to the same RRR set. For example, the RRR set of vertex 5 (where the orange traversal originated) contains vertices $\{4, 5, 6, 7, 8\}$.

Pseudocode: Listing 1 shows the pseudocode of the fused BPT algorithm. The `frontier` array is used to identify the set of active vertices. Each element of that array is associated to a vertex and contains a bitmask that identifies the BPTs that need to be propagated from that vertex. If a vertex `v` is not active, `frontier[v]` does not contain any set (i.e., bit 1) BPTs. The `visited` array indicates, for each vertex `v`, the traversals (i.e., BPTs) passing through `v` (up to the current traversal step). For the example in Fig. 2, the `visited` array encodes the BPTs of the vertices, while the `frontier` array encodes the frontier, which includes only the BPTs to be propagated the next time an active vertex is processed.

During initialization (lines 1-3 of the pseudocode) a random set of vertices is selected as starting points of different traversals; accordingly, each of the selected vertices is associated to a different BPT. The core of the traversal algorithm is encoded in lines 5-16. The traversal continues as long as the frontier is not empty. For each active vertex `v`, the corresponding frontier bitmask is read (in `fr_v`)

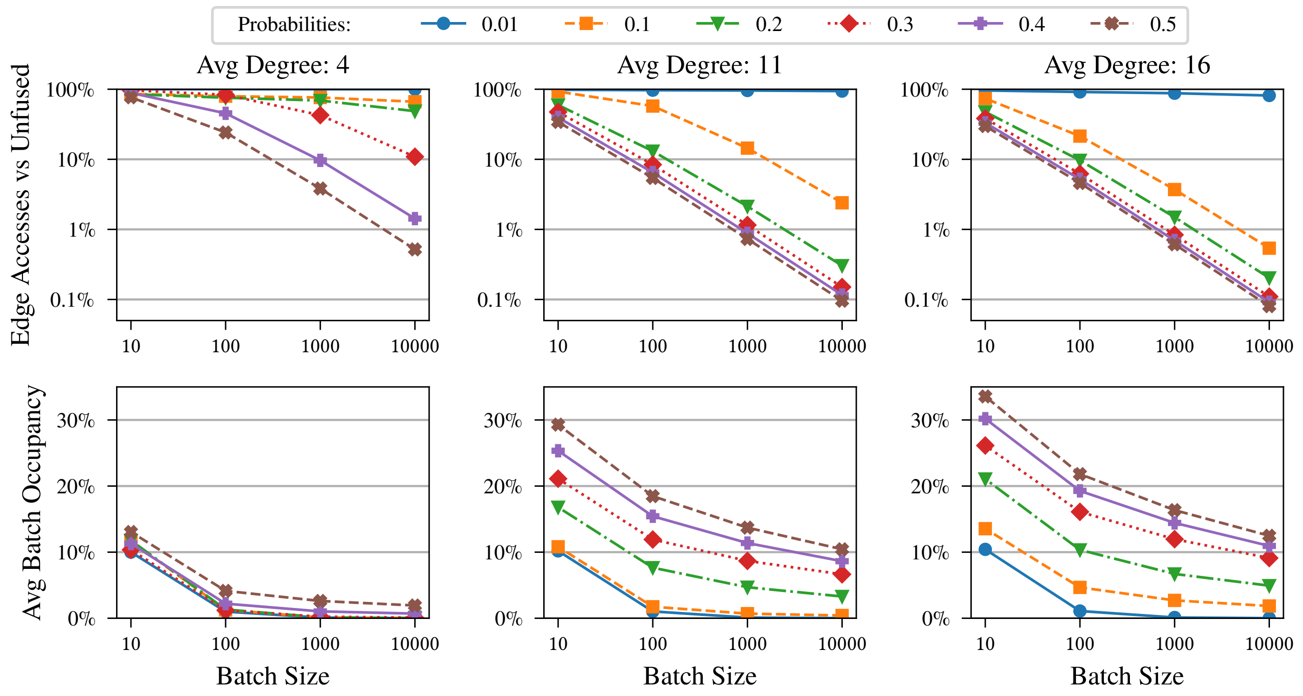

Figure 3: Edge accesses of 10,000 BPTs compared to unfused (top) and average batch occupancy (bottom) for various vertex degrees, batch sizes, and traversal probabilities.

and then cleared (line 7). The BPTs in the frontier of v are then added to the corresponding visited array to update the list of traversals passing through v (line 8). All the edges outgoing from v are then traversed with random probability (lines 9-15). For each edge e from vertex v to vertex u, e.prob indicates its traversal probability, while bitmask fr_u indicates the set of BPTs that will be propagated from v to u. BPTs already visited by u are excluded from the traversal (line 11), and the other BPTs in fr_v are included with probability e.prob (lines 11-13). The BPTs in fr_u are then added to the frontier of u (line 14), effectively fusing the newly added traversals with the ones already part of the frontier of u. Finally, the RRR sets are updated according to the traversal outcome (lines 18-21).

3.1 Analysis on Synthetic Graphs

Since the upper bound of edge accesses is equivalent to performing unfused BPTs, we perform experiments to assess the amount of work saved, in terms of accessed edges, using synthetic graphs. To this end, we generate several graph configurations of the LFR benchmark, leading to graphs with vertex degrees and community sizes that follow a power law distribution [19], mirroring characteristics found in real-world networks. Using NetworkX [15], we generate graphs with 10,000 vertices and degrees 4, 11, and 16. For each configuration, we use three graph generation seeds. We then perform a BPT per vertex using edge probabilities 0.01, 0.1, 0.2, 0.3, 0.4, and 0.5. These traversals are repeated three times, each time using a different starting seed. This results in around 1.6 million BPTs in total, traversed level-synchronously (i.e., active vertices are processed level-by-level).

We perform runs varying the number of BPTs from 10 to 10,000. Accordingly, BPTs are fused in batches, where the batch size is equal to the number of BPTs used in that experiment. We then calculate the work savings (in terms of edge accesses) for each batch compared to the unfused version and average across the three runs. Since we perform a level-synchronous traversal, fusing occurs only if BPTs within the same batch visit a vertex in the same traversal step. Figure 3 shows the resulting plots. The top plot shows that higher activation probabilities and fused batch sizes result in better work savings. This was expected, as higher activation probabilities result in larger activation of the graph, increasing the chances that frontiers will be shared amongst traversals.

The bottom plots show the batch occupancy, defined as the fraction of BPTs (i.e., traversals) that any visited vertex is part of. The average batch occupancy is the average over all the vertices and traversal steps. Intuitively, this term measures how much sharing can be exploited to fuse traversals. An ideal batch occupancy would be as close to 100% as possible, as this maximizes the potential for fusing. As can be seen, although it is not feasible to fuse all BPTs within a batch, the batch occupancy increases with the average vertex degree and edge traversal probability. A higher batch occupancy is indicative of better edge sharing in a fused batch of BPTs.

4 FUSED BPT IMPLEMENTATION

We incorporated fusing into two existing GPU implementations of BPT: gIM [37] and Ripples [27]. Both codes perform many BPTs as the first step of the RIS algorithm for Influence Maximization. However, they differ in their parallelization approach. gIM performs

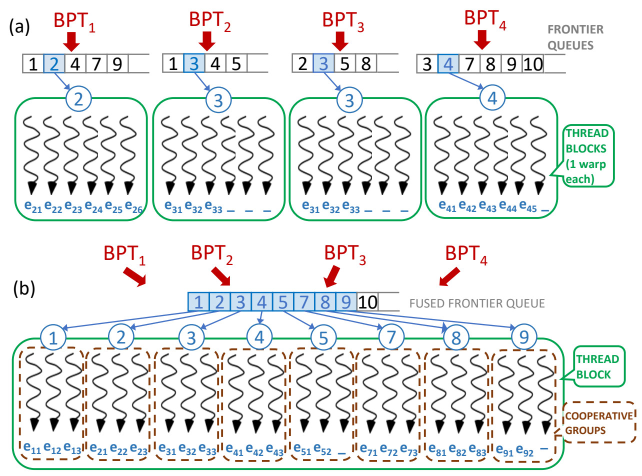

Figure 4: Parallelization scheme for gIM. a) is unfused, while b) is fused. e_{ij} represents the j-th edge of node i. The BATCH_SIZE is 4. For readability, the illustration assumes 6 threads per warp.

multiple level-asynchronous traversals within a single kernel. In contrast, Ripples parallelizes a *single*, *level-synchronous* traversal across the whole GPU and performs multiple kernel calls for each BPT.

4.1 Incorporating Fused BPTs into gIM

The baseline implementation of gIM performs multiple BPTs within a single CUDA kernel. The requested BPTs are distributed across thread blocks and performed in parallel. Figure 4(a) illustrates gIM's parallelization. Each thread block consists of one warp (32 threads) and during a BPT, it processes the vertices in the frontier sequentially. For each active vertex, gIM performs edge-level parallelization and distributes the outgoing edges to the threads within the thread block. The frontier is implemented as a queue of identifiers for the active vertices, while the "visited" mask is an array stored in global memory—one bit per vertex. To avoid shared memory overflow,the frontier is stored split between shared and global memory, and sections are moved between the two memory units as needed. RRR sets are implemented as a linked list of fixed-size buffers stored in global memory.

Bugs and fixes: Before incorporating fusing in gIM, we fixed two existing bugs. The first bug caused gIM to lose part of the frontier queue after offloading it to global memory. The second was a concurrency bug leading some RRR sets to be generated multiple times, thus causing extra BPTs to be executed. In addition, we noticed that the large global memory utilization prevented gIM from scaling to larger graphs or inputs with larger edge traversal probabilities. To address this issue, we changed the implementation to store the RRR sets in CUDA managed memory (UVM), which enables automatic offloading onto host memory, thus limiting global memory pressure. When the RRR sets fit global memory, the use of UVM does not lead to performance degradation because RRR sets are written only once by the GPU kernel.

Modifications to fuse traversals: To support fusing, we expanded the `visited` array to hold one BPT bitmask per vertex. For the frontier, we kept gIM's frontier queue and added an extra global memory array to store the bitmasks associated to the active vertices (similar to the `frontier` array in Listing 1). The parallelization of the fused-gIM implementation is illustrated in Figure 4(b). Fusing allows `BATCH_SIZE` traversals to be performed synchronously, with `BATCH_SIZE` being the number of BPTs fused. Accordingly, in the fused implementation, each thread block performs `BATCH_SIZE` BPTs while processing a single frontier queue. To keep the relative use of shared memory per BPT unmodified, we increased the thread block size from one to `BATCH_SIZE` warps. With this increased block size, however, edge-level parallelization over the whole thread block can lead to warp underutilization, especially for low outdegree vertices. To maintain the same utilization as in the original gIM, we assigned to each warp a different active vertex, effectively parallelizing the frontier's processing. We note that edge-level parallelism can cause warp underutilization for low outdegree vertices even with this vertex-to-warp mapping. To this end, we used CUDA cooperative groups [31] to implement finer-grained parallelization, where one vertex is assigned to only 16 or 8 threads. In our experiments, however, vertex assignment at a sub-warp granularity reported a (slight) performance improvement only on the smallest graph considered.

We note that the use of bitwise operations and integer intrinsic on 32 or 64 bit masks allows for efficient BPT processing. On the other hand, further parallelizing the for-loop at line 12 of Listing 1 by distributing BPTs across threads would cause warp underutilization (when only few BPTs are set) and require extra synchronization on the bitmasks' updates, negatively affecting performance. In the RRR sets construction step (lines 18-21 of Listing 1), each thread processes an element of the *visited* array and updates the corresponding RRR sets atomically, leading to some synchronization cost. Finally, we observed that limiting the register utilization to 32 registers per thread allows increasing the GPU occupancy by doubling the number of resident thread blocks per Streaming Multiprocessor (SM), improving performance despite some added register spilling.

4.2 Incorporating Fused BPTs into Ripples

The baseline implementation of Ripples distinguishes each CPU core as a CPU worker or a GPU worker. A CPU worker is responsible for performing its own BPT, while a GPU worker handles kernel launches and memory movement between the host and GPU. Both kinds of workers perform a single BPT at a time. To handle the irregular workloads common in BPTs, an atomic variable denoting the number of required BPTs is located in the host, with each CPU and GPU worker performing an `atomicAdd` before performing a BPT to determine if there is more work.

Compared to gIM's threadblock granularity, Ripples utilizes the whole GPU to perform a single level-synchronous BPT, returning to the host between kernel calls. There are two main steps in the traversal: (1) *Generation of Frontiers*: The host launches a device-wide kernel to perform an XOR between the visited and frontier arrays to determine which BPTs are in the frontier queue, and reduces these frontier nodes to one of four bins, with each bin corresponding to a different range of degrees. The frontier array is then written to the visited array. (2) *Edge Traversal*: The host then launches four separate streams, each with varying levels of thread block granularity, to scatter the edges and perform randomized traversal. Nodes with smaller degrees

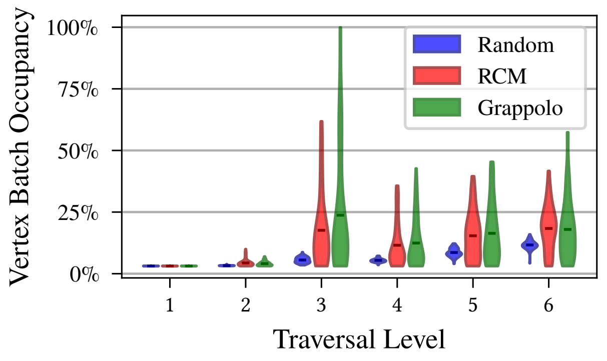

Figure 5: Batch occupancy on a fused web-BerkStan traversal with a batch size of 32 using RCM and Grappolo (§5).

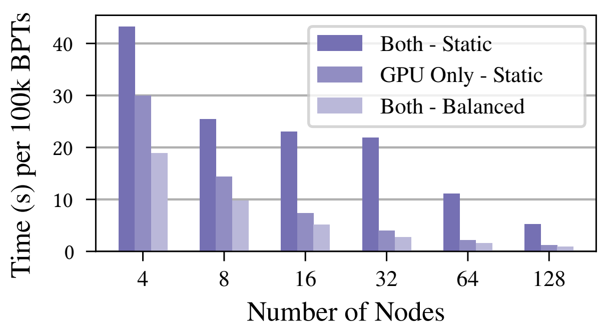

Figure 6: Multi node scaling on soc-pokec-relationships.

have all their edges traversed by a single thread within the warp, and nodes with larger degrees have all their edges traversed by a thread block of size 32, 256, or 1024. This implementation is similar to Merrill et al. [26]. Afterwards, the visited array is moved to the host, and the host CPU traverses the array to push the visited vertices to the appropriate vector.

Modifications to fuse traversals: To perform fusing on the baseline implementation of Ripples, we turned the `visited` and `frontier` arrays (lines 7-8 of Listing 1) into a blocked bitmask, with each block containing N 32-bit values, where N contains a block of 32 BPTs, one bit per BPT. This blocking ensures proper memory alignment between warps. Each bit within the block corresponds to a different BPT, or BPT, within the fused group. Increasing the number of variables, however, resulted in threads handling BPTs across multiple variables, which could lead to memory access inefficiencies. To alleviate this, we assign N threads per blocked bitmask where, as in fused gIM, each thread utilizes integer intrinsics to process their BPT (line 12 of Listing 1). Because the number of BPTs for all edges are the same, there is no workload imbalance in the warp/block-level hierarchical queue. During frontier queue generation, where vertices are filtered to a degree-based workload queue, we perform a localized warp-level reduction to determine the queue offset. Then, the leading thread broadcasts the proper offset to each thread to reduce the number of global atomic operations.

5 PARALLEL HEURISTICS

Vertex Reordering Techniques:

A key factor that influences parallel performance of fused BPTs is the number of common vertices that will visited during concurrent execution of fused BPTs, in other words, the *locality* of fused BPTs. For example, in Figure 2(b), the yellow and green BPTs from vertices 5 and 3 respectively, need to converge on the shared vertex 4 around the same time in order to benefit from fusing. However, if these vertices are stored in a non-local fashion in memory, then the probability of temporal shared visits is reduced. It is important to increase locality for better performance.

A classical technique to improve vertex locality is *vertex reordering*, which aims to obtain a locality-preserving permutation, and consequently maximizes locality for fused BPTs in a given batch, and in the process increases vertex batch occupancy. Reordering algorithms aim to either explicitly minimize the gap (e.g., minimum linear arrangement algorithm) or use heuristics (e.g., reverse Cuthill-McKee (RCM), degree-based sorting and partitioning-based sorting) to accomplish similar goals in an efficient manner [6]. Using average gap profile and average graph bandwidth as a metric, Barik et al. [6] demonstrated that METIS, Grappolo and RabbitOrder were the top three, and Slashburn and Gorder were the worst two, reordering schemes among several methods for reordering. Since Grappolo scales to large graphs, we use it as our reordering scheme. For example, comparison for web-BerkStan at different traversal levels is summarized in Fig. 5. We observe significantly larger batch occupancy for RCM and Grappolo relative to a random vertex reorder as the baseline. All three schemes consider the sorted variant, which pre-generates and sorts the random start vertices. When workers retrieve BPTs to fuse, they pull from this sorted list of source vertices for better locality and, thus, more opportunities for fusing.

CPU-GPU Workload Balancing: Originally, the GPU and CPU workers of Ripples would obtain a batch size—meaning a single node would process up to 3,584 BPTs at a time, if the batch size is 64 (56 batches). When scaling to multiple nodes, the number of BPTs each node needed to generate was scaled down proportionately. However, we encountered an issue where the heterogeneous CPU-GPU setup was lacking in performance compared to a GPU-only setup (Fig. 6). Upon examination, we found that the CPU workers were causing workload starvation, with a comparative test run on web-BerkStan showing the CPU implementation to be up to 16× slower than the GPU implementation.

To alleviate this issue, we designed a lightweight micro benchmarking scheme where, at the beginning of each run, the host times several batches to be run by each CPU and GPU worker. Then, the host calculates the difference between the average CPU and GPU worker times. This difference is split, increasing/decreasing the CPU worker batch size, until the timings between CPU and GPU workers are similar. This method worked for smaller graphs, but in larger graphs, the micro benchmarking was setting the CPU batch size to 0, i.e., the CPU would cause starvation even when retrieving a single BPT. To enable the CPU workers to assist with BPT generation even if a single core would cause starvation, we group CPU workers in the same L3 cache region (6 CPU cores each) to collaborate on one BPT group, resulting in 8 total CPU worker groups executing

BPTs. The results of this workload balancing are shown Fig. 6 (see Section 7.2.4).

6 EXPERIMENTAL SETUP

Table 1: SNAP Graphs

Graph	# Nodes	# Edges	Avg. Degree
web-BerkStan (BS)	685,230	7,600,595	22.18
web-Google (Go)	875,713	5,105,039	11.66
soc-pokec-relationships (PR)	1,632,803	30,622,564	37.51
wiki-topcats (TC)	1,791,489	28,511,807	31.83
com-Orkut (Ok)	3,072,441	117,185,083	76.28
soc-LiveJournal1 (LJ)	4,847,571	68,993,773	28.47

Hardware: We study the performance and scalability of Ripples on Crusher and Frontier Computing Systems hosted at Oak Ridge Leadership Computing Facility. These systems have the same configurations, where each compute node has a 64-core AMD EPYC 7A53 CPU, 512 GB DDR4 memory, and 4 AMD MI250X GPUs. Each MI250X contains two Graphics Compute Dies (GCDs) that, to the host runtime, appear and operate as two separate GPUs. For this reason, we refer to a GPU as a single GCD. We perform scaling experiments up to 4,096 compute nodes, utilizing HPE Cray MPICH 8.1.23 for multi-node setups. At the time of this writing, OLCF Frontier is the #1 system on the Top500 list and #3 on the Graph500 list.

Our experiments are run with "Low-Noise mode" enabled to minimize operating system noise. This setting restricts system processes to run on the first core of each L3 cache region (8 L3 regions in total). However, using the low-noise mode implies that applications are restricted to using only the remaining 56 CPU cores on each node. This leaves 6 cores per L3 cache region as a CPU team with the remaining core (for a total of 7 per L3 cache region) handling GPU operations. The gIM framework supports only a single NVIDIA GPU. For our experiments, we use a Tesla A30 GPU with 24 GB of HBM2 memory.

Software: We have implemented the fused BPT approach in gIM using CUDA 12.2 and GCC 11.4.0. For Ripples, we implemented the fused BPT approach using AMD HIP for GPUs and OpenMP for CPUs. The host code for scheduling work on GPUs and CPUs also uses OpenMP. We compiled Ripples with hipcc (from AMD ROCM 5.1.0).

To facilitate direct comparison with published results ([27, 37]), we use the same or similar input datasets as shown in Table 1. For more detailed experiments, we show a subset of datasets that vary in size.

Unless otherwise specified, we assigned edge weights from a uniform distribution between 0 and 1. We performed the graph generation process once, and consistently reused the same inputs. For multi-node setups, workload balancing as discussed in Section 5 is performed as a pre-processing step.

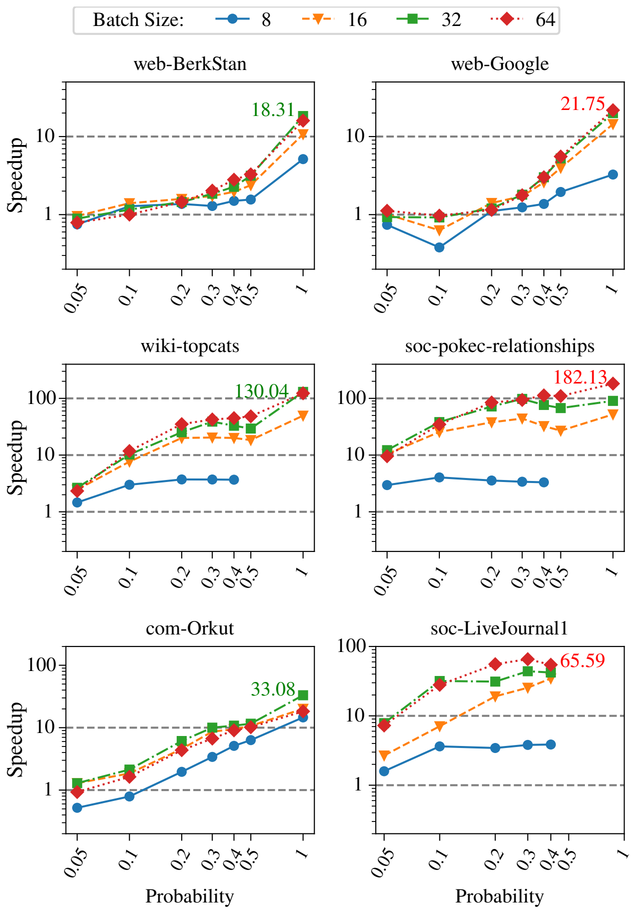

Figure 7: Speedup of fused over unfused gIM with different batch sizes and traversal probabilities.

7 EXPERIMENTAL EVALUATION

7.1 gIM with Fused BPTs

Fig. 7 shows the effect of fusing BPTs on gIM's performance for the graphs in Table 1. Missing data points correspond to experiments where our system ran out of memory to store RRR sets despite allocating the maximum available UVM size. We recall that we used UVM only for RRR sets, while we kept the original gIM implementation for dynamically allocated data structures (such as the frontier queue). In gIM, an increase in peak frontier sizes at higher probabilities leads to extreme memory pressure. This impact is higher in smaller batch sizes as there are fewer fusing opportunities among BPTs. We conducted experiments with various edge traversal probabilities and batch sizes. We make the following observations.

First, incorporating fusing of BPTs in gIM is beneficial in most cases, and yields speedups up to 18×, 21×, and 130× on web-Berkstan, web-Google, and wiki-topcats, respectively. The performance benefits are mainly due to two advantages of the fused gIM implementation. First, fusing decreases the number of vertices and edges traversed, also reducing the number of accesses to global memory. Second, since fused-gIM uses a single frontier queue for all BPTs in the same group, fusing can reduce the probability of overflowing the

Table 2: Impact of fusing BPTs on GPU resource utilization across different probabilities.

Probability	Mem. Utilization (MB)		Achieved Occupancy	
	gIM	Fused gIM	gIM	Fused gIM
0.05	0	0	3.77%	84.66%
0.1	0	0	4.45%	72.98%
0.2	0	10	4.56%	78.11%
0.3	12	38	4.58%	83.39%
0.4	67	71	4.57%	83.02%
0.5	173	104	4.59%	67.12%
1.0	879	88	4.64%	82.58%

shared memory allocated to the frontier queue, thus limiting the data movements between the shared and global memory.

Second, in the absence of enough fusing opportunities, the *overhead of the extra code* added to incorporate fusing (see §4.1) can lead to performance degradation over the baseline gIM. For example, when using only a batch size of 8 and traversal probabilities below 0.2, web-Google incurs up to 62% performance degradation. However, as the fusing opportunities increase (e.g., higher traversal probabilities), fusing BPTs can considerably improve gIM's performance.

Third, increasing the *batch size* from 8 to 32 results in a significant performance improvement. We recall that the batch size determines the size of each BPT group, i.e., each set of BPTs that can be potentially fused. Larger groups allow more BPT sharing opportunities, enabling fusing. For example, on web-Berkstan, fusing 32 BPTs results in a speedup up to 18.3×, while fusing 8 BPTs results in a maximum speedup of 5.1×.

Furthermore, lower *traversal probabilities* reduce the chance of the same edge being traversed by multiple BPTs. This hinders the performance gains from fusing. For example, when using a batch size of 32 on wiki-topcats, increasing the traversal probability from 0.05 to 0.2 causes the speedup over the unfused gIM to increase from 2.6× to 25.2×.

7.1.1 Adaptive Mapping and Utilization Efficiency. Finally, we explored the potential of implementing an adaptive mapping scheme for task scheduling on the gIM implementation with fused BPTs. This approach allows thread blocks that have completed their BPTs to start processing work from other thread blocks. Such a strategy aims to enhance the utilization of the GPUs multiprocessors and limits the growth of the frontier queue in the GPU's global memory. For applications using fused BPTs to gain from adaptive mapping, they must exhibit low GPU warp occupancy. According to the Nvidia Nsight Compute profiler, GPU occupancy is defined as the ratio of active warps per multiprocessor to the maximum number of active warps. This information is crucial for understanding and implementing efficient scheduling strategies [32]. Furthermore, it's important to note that implementing an adaptive mapping scheme necessitates communication between thread blocks through global memory. Therefore, only substantial performance improvements justify the use of this approach.

To assess if an adaptive mapping could benefit the fused gIM, we measured warp occupancy, and the size of dynamically allocated

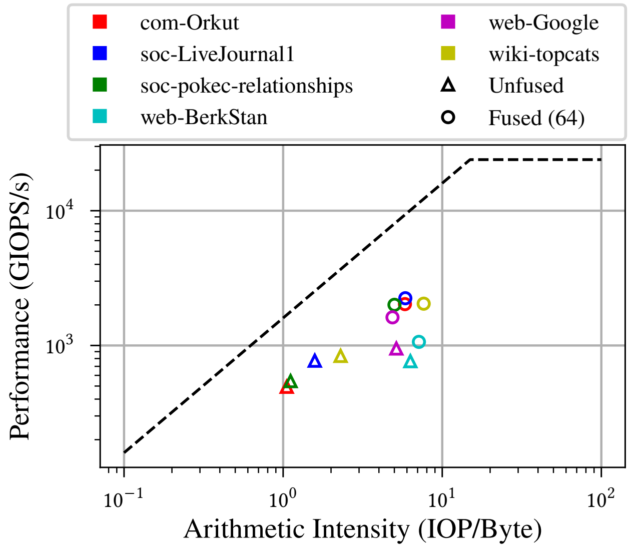

Figure 8: Roofline plot on a single GCD of the MI250x for each input dataset for unfused vs fused (batch size = 64).

memory on GPU (for frontier queues) during traversal. We used the web-BerkStan graph as our test case for this experiment. Table 2 exemplifies the impact of our algorithm on gIM resource utilization. As seen in Table 2, the baseline gIM model exhibits notably low GPU occupancy (averaging ~4.5%), whereas our fused algorithm boosts the occupancy substantially (to an average of ~78.83%). Furthermore, our study illustrates the improvement in dynamic memory allocation on gIM achieved by fusing 32 BPTs in higher probabilities. Our algorithm effectively mitigates memory constraints that previously restricted running gIM on larger graphs with higher traversal probabilities, thus broadening its applicability. For instance, in the case of web-Berkstan input graph, fusing 32 BPTs at a time leads to approximately ten times less dynamic memory allocation when processing a graph with high traversal probabilities on edges. Based on the metrics collected, our conclusion is that the fusing BPTs on gIM already achieves high GPU utilization while maintaining low GPU memory utilization, even at elevated probabilities. Consequently, this scenario indicates that the incorporation of an adaptive mapping scheme would not yield additional benefits in this application.

7.2 Ripples with Fused BPTs

The Ripples framework approaches the problem of BPT generation from a device-wide perspective. We evaluate the impact of fusing on Ripples, including its behavior on single- and multi-node scaling.

7.2.1 GPU Roofline. Figure 8 demonstrates the effectiveness of fusing, where decreasing the edge accesses increases the arithmetic intensity and, therefore, allow for higher performance.

7.2.2 Sensitivity Analysis. Applying the principle of fused BPTs to Ripples provides two key improvements: (1) *BPT Concurrency*: While fusing BPTs in gIM doubles the number of BPTs on the GPU at any given time, applying the approach to Ripples increases the number of BPTs on the GPU from 1 to the batch size. For example,

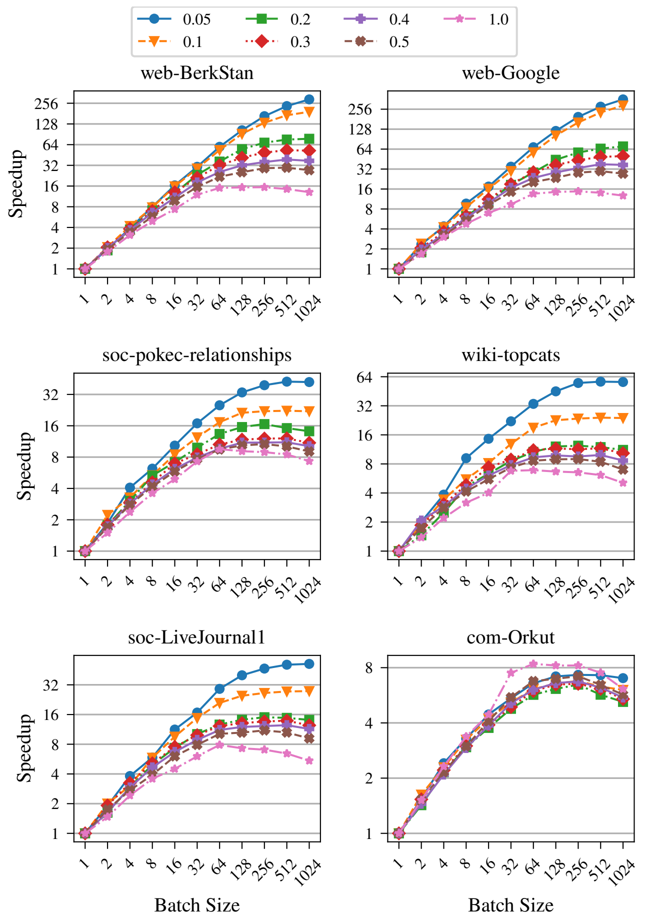

Figure 9: Speedup of fused over unfused Ripples when varying batch size and traversal probabilities. Probability is shown in the top legend.

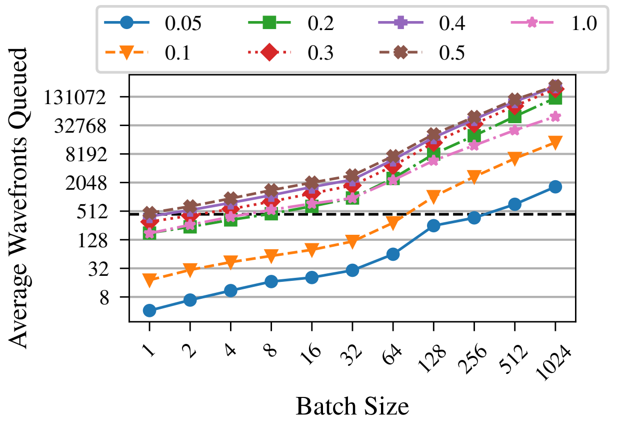

Figure 10: Profiling of frontiers on 10K BPTs on web-BerkStan. The y-axis denotes the number of wavefronts queued per iteration of the BPT. The horizontal black dashed line denotes the number of SIMD units present on one GCD of the MI250x (440), where each SIMD unit processes one wavefront.

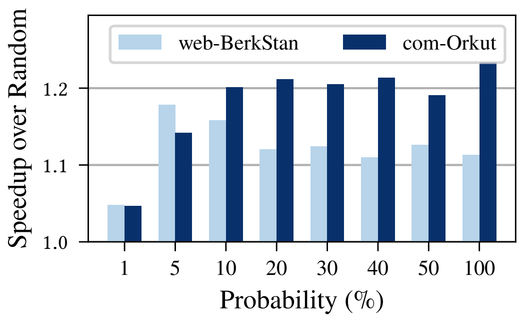

Figure 11: Speedup of Grappolo vertex reordering over random ordering (10k BPTs)

in the case where a batch contains 1024 BPTs, this can lead to a 1024-fold increase in BPT concurrency on the GPU. (2) *Edge Sharing*: Ripples benefits from edge sharing like gIM, as fusing BPTs provides opportunities to reduce the amount of work; however, a large batch size might reduce the GPU utilization per BPT, and increase the overheads for processing empty BPTs (see Section 3.1).

Fig. 9 shows how Ripples scales with varying probabilities and increasing batch size (i.e., BPTs). The speedup significantly increases with batch size over the unfused (batch size of 1) approach. This is due to increased concurrency. We also observe that traversals with lower probabilities benefit more from fusing with respect to traversals with higher probabilities for the majority of input graphs, even when considering probability of 1. Traversals with lower edge-probabilities reduce vertex activations, thus leading to smaller frontiers of the BPTs and to lower GPU utilization. To better understand this behavior, we profiled the sizes of the hierarchical frontier queues. Fig. 10 shows that with low probabilities and low batch size, there are not enough wavefronts to keep an entire GCD fully utilized. A wavefront is the equivalent of a CUDA warp, representing the minimal scheduling unit in terms of parallel threads simultaneously executed by a SIMD (single instruction, multiple threads) unit of the GPU (either a GCU for AMD or a SM for NVIDIA). An AMD wavefront corresponds to 64 threads, a CUDA warp has 32 threads. For lower edge probabilities, increasing the batch size provides a larger improvement in performance. GPU utilization increases from the increase in concurrency of BPTs and edge sharing,

7.2.3 *Vertex Reordering.*

Figure 11 shows the speedup of sampling using Grappolo to perform vertex reordering compared to a random ordering. For Ripples, using Grappolo results in around a 1.1-1.2× speedup. This benefit comes from both increased sharing at higher probabilities and a reduced average max traversal depth, where the max traversal depth per batch is averaged over all batches. BPTs from the same cluster have more similar traversal depths than those from different clusters, especially at lower probabilities.

7.2.4 *Scaling.*

We evaluate the scaling of Ripples with fused BPT on multiple GPUs and multiple nodes.

Single-Node Multi-GPU: First, we evaluate strong scaling on a single node at a batch size of 32 when progressively increasing the number of GPUs from 1 to 8. Fig. 12 shows that the performance of Ripples scales almost linearly when employing fused BPTs. We only report the results of two graphs, but all our benchmarks follow the same trends. Beyond a greater potential for workload imbalances due to increased workload granularity, fusing does not impact the scaling behavior.

Heterogeneous Workload Balancing: We then evaluate the strong scaling on multiple nodes. When increasing the number of nodes, the number of BPTs that each node processes accordingly reduces. This allowed us to identify a workload imbalance between CPU and GPU workers on the same node. CPU workers are slower than GPU ones, thus starving the GPU workers if not enough work units are available. After implementing the load balancing mechanisms described in §5, we achieved a better balance between CPU and GPU workers, allowing CPUs to effectively help in the BPT generation and thus providing a speed up with respect to a GPU-only setup.

From an energy-usage standpoint, a high level analysis suggests adding CPUs for traversal is beneficial. Each MI250x has a TDP of 500W for 2000W per node, and the CPU's TDP is 280W which is 14% of each node's total power. In Figure 6, we see around a 50% performance boost over "GPU only" when adding in the balanced CPUs, which is greater than the 14% power increase.

Multi-Node Scaling: Finally, we evaluate Ripples with the fused BPT approach when increasing the number of nodes from 4 to 4096 nodes. At 4096 nodes, Ripples uses a total of 196,608 CPU cores and 32,768 GPUs for RR set generation. Fig. 13 shows that our approach keeps scaling as we increase the number of nodes, reaching at 4096 nodes a speedup of 128× over the 4 node version.

8 RELATED WORK

Breadth-First Search (BFS) is considered as one of the core primitives of graph algorithms as well as a prototypical irregular kernel used in benchmarking [1], and it has received extensive attention in literature. BFS on GPU architectures has also been explored extensively [38], with notable works such as: Merrill et al. [26] who employed hierarchical queues for fine-grained task management to scale traversals; efficient thread scheduling, degree-based sorting and direction-switching in Enterprise [22]; a data-centric abstraction using bulk synchronous model to enable programming productivity in Gunrock [43]; and a collection of techniques to address the data-specific challenges that are dynamic in nature in XBFS [11].

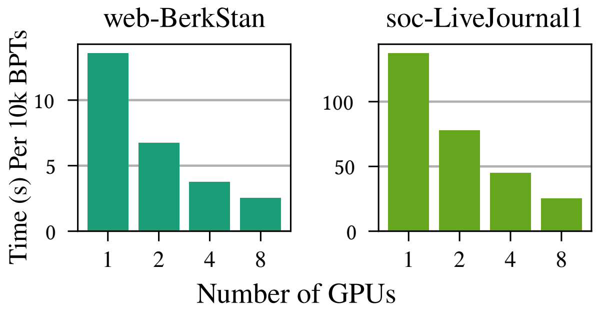

Figure 12: Single node scaling with 32 batch size.

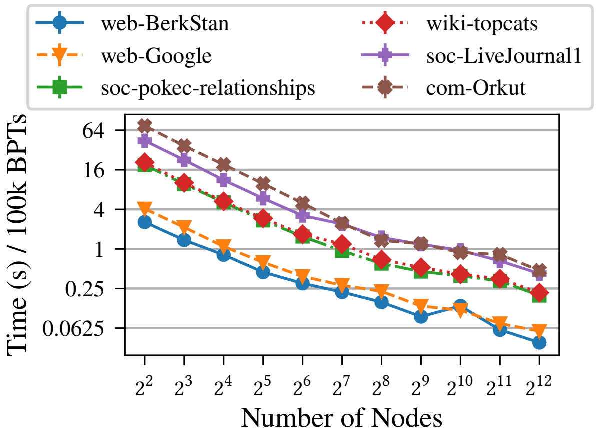

Figure 13: Multi node scaling from 4 to 4096 nodes. All times are averaged over three runs.

Multi-threaded [4, 33], distributed [46], and algebraic approaches for BFS [44] have also been deeply studied.

Several efforts consider performing multiple concurrent deterministic graph traversals, which serve as inspiration for `FuseIM`. MS-BFS, introduced by Then et al. [42], is a single-threaded CPU-only method for running concurrent BFSs. MS-BFS employs a bitmask coloring scheme, and introduces techniques to improve performance such as aggregated neighbor processing, direction-optimized travel, neighbor prefetching, and degree-based vertex reordering. Liu et al. [23] proposed iBFS, an extension of Enterprise [22], to perform parallel concurrent BFSs on GPUs. iBFS also utilizes a bitmask-based data structure and introduces GroupBy rules for improved sharing along with early termination for their bottom-up approach.

While both MultiLyra [25] and Glign [45] pioneered techniques for fusing concurrent graph queries in distributed settings, their focus lies primarily on deterministic, iterative queries traversing significant portions of the graph. Notably, their optimizations, such as query pausing for workload exposition, depend on this deterministic and comprehensive exploration. Our work builds upon these foundational ideas, expanding their applicability to a broader realm of graph algorithms and analytics. Specifically, the challenge of merging randomization and subgraph sampling increases stochasticity and chances of not visiting the entire graph.

Influence Maximization: The seminal work of Kempe et al. [17] has sparked fervent research around the Influence Maximization Problem. At the time of this writing, the algorithms with the best theoretical efficiency are those leveraging the Reverse Influence Sampling (RIS) introduced by Borgs et al. [8]. The current state-of-the-art is the IMM algorithm by Tang et al. [40] and it constitutes the starting point of the parallel algorithms used in our work.

The body of work of Göktürk and Kaya [12, 13, 14] represents the first steps in using the idea of fused simulations of the diffusion process in Influence Maximization algorithms. The algorithms that they propose are variations on the lazy-greedy approach of Leskovec et al. [20] where, at each iteration, the algorithm needs to perform many simulations of the diffusion process from a node in the graph to recompute its score. To speed up the computation, Göktürk and Kaya [12, 13, 14] combine label propagation techniques, memoization, vectorization, and sketches to expose more parallelism and reuse work in their proposed approaches. However, their use of sketches renders their heuristics and approach as unsuitable for Influence Maximization algorithms based on RIS. This is due to the numerous simulations inherent in this class of approaches, each having different (randomly selected) starting points. Our work bridges these gaps and brings the idea of fusing diffusion process simulations to RIS-based methods while preserving their approximation guarantees.

9 CONCLUSION AND FUTURE WORK

In this work, we have proposed FuseIM, a fused BPT algorithm, where we share frontiers between separate BPTs. We implemented our algorithm on two frameworks, gIM and Ripples, that use two different approaches for generating BPTs. Through our experiments, we show the benefits of fusing over their unfused counterparts. We also identify source binning, vertex reordering, and workload balancing as key heuristics for improving the performance of fused BPT on single-accelerator, heterogeneous, and distributed systems. Additionally, we show strong scaling results of both single-node multi-GPU and multi-node heterogeneous systems.

Future research directions include (but are not limited to): a) exploration of adaptive techniques for improved performance in probabilistic traversals—e.g., directional switching [7], hybrid scan vs. queue frontier management methods [21, 30], and dynamic batch sizes; b) potential ways to pause and resume workloads so that finished BPTs can offload their RR sets early, and new start nodes can be injected mid-traversal; and c) exploiting any higher order structural information of a graph into fused BPTs.

ACKNOWLEDGMENTS

This work is supported by the U.S. Department of Energy through the Exascale Computing Project (17-SC-20-SC) (ExaGraph) at the Pacific Northwest National Laboratory; the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphic Intelligence Logical Computing Environment (AGILE) research program, contract number 77740; and by NSF award CNS-1812727 at North Carolina State University.

REFERENCES

[1] 2023. Graph500 Kernel Description. http://www.graph500.org/

[2] Seher Acer, Ariful Azad, Erik G Boman, Aydın Buluç, Karen D. Devine, SM Ferdous, Nitin Gawande, Sayan Ghosh, Mahantesh Halappanavar, Ananth Kalyanaraman, Arif Khan, Marco Minutoli, Alex Pothen, Sivasankaran Rajamanickam, Oguz Selvitopi, Nathan R Tallent, and Antonino Tumeo. 2021. EXAGRAPH: Graph and combinatorial methods for enabling exascale applications. *The International Journal of High Performance Computing Applications* 35, 6 (2021), 553–571. https://doi.org/10.1177/10943420211029299

[3] László Babai. 1979. Monte-Carlo algorithms in graph isomorphism testing. *Université tde Montréal Technical Report, DMS* 79-10 (1979).

[4] David A. Bader and Kamesh Madduri. 2006. Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. In *2006 International Conference on Parallel Processing (ICPP'06)*. 523–530. https://doi.org/10.1109/ICPP.2006.34

[5] Aditya Ballal, Willow B. Kion-Crosby, and Alexandre V. Morozov. 2022. Network community detection and clustering with random walks. *Phys. Rev. Res.* 4 (Nov 2022), 043117. Issue 4. https://doi.org/10.1103/PhysRevResearch.4.043117

[6] Reet Barik, Marco Minutoli, Mahantesh Halappanavar, Nathan R. Tallent, and Ananth Kalyanaraman. 2020. Vertex Reordering for Real-World Graphs and Applications: An Empirical Evaluation. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 240–251. https://doi.org/10.1109/IISWC50251.2020.00031

[7] Scott Beamer, Krste Asanović, and David A. Patterson. 2012. Direction-optimizing Breadth-First Search. *2012 International Conference for High Performance Computing, Networking, Storage and Analysis* (2012), 1–10. https://api.semanticscholar.org/CorpusID:5242266

[8] Christian Borgs, Michael Brautbar, Jennifer T. Chayes, and Brendan Lucier. 2014. Maximizing Social Influence in Nearly Optimal Time. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*. 946–957. https://doi.org/10.1137/1.9781611973402.70

[9] Pedro Domingos and Matt Richardson. 2001. Mining the Network Value of Customers. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California) *(KDD '01)*. Association for Computing Machinery, New York, NY, USA, 57–66. https://doi.org/10.1145/502512.502525

[10] M. L. Fisher, G. L. Nemhauser, and L. A. Wolsey. 1978. An analysis of approximations for maximizing submodular set functions—II. *Polyhedral Combinatorics* (1978), 73–87. https://doi.org/10.1007/bfb0121195

[11] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. 2019. XBFS: eXploring Runtime Optimizations for Breadth-First Search on GPUs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, Phoenix AZ USA, 121–131. https://doi.org/10.1145/3307681.3326606

[12] Gökhan Göktürk and Kamer Kaya. 2021. Boosting Parallel Influence-Maximization Kernels for Undirected Networks With Fusing and Vectorization. *IEEE Trans. Parallel Distributed Syst.* 32, 5 (2021), 1001–1013. https://doi.org/10.1109/TPDS.2020.3038376

[13] Gökhan Göktürk and Kamer Kaya. 2022. Fast and High-Quality Influence Maximization on Multiple GPUs. In *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022*. IEEE, 908–918. https://doi.org/10.1109/IPDPS53621.2022.00093

[14] Gökhan Göktürk and Kamer Kaya. 2024. Fast and error-adaptive influence maximization based on Count-Distinct sketches. *Inf. Sci.* 655 (2024), 119875. https://doi.org/10.1016/J.INS.2023.119875

[15] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11 – 15.

[16] Richard M. Karp, Rajeev Motwani, and Noam Nisan. 1993. Probabilistic Analysis of Network Flow Algorithms. *Mathematics of Operations Research* 18, 1 (1993), 71–97. http://www.jstor.org/stable/3690154

[17] David Kempe, Jon M. Kleinberg, and Éva Tardos. 2003. 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*. 137–146. https://doi.org/10.1145/956750.956769

[18] Arif M. Khan, David F. Gleich, Alex Pothen, and Mahantesh Halappanavar. 2012. A multithreaded algorithm for network alignment via approximate matching. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2012.8

[19] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. 2008. Benchmark graphs for testing community detection algorithms. *Physical Review E* 78, 4 (oct 2008). https://doi.org/10.1103/physreve.78.046110

[20] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne M. VanBriesen, and Natalie S. Glance. 2007. 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*, Pavel Berkhin, Rich Caruana, and Xindong Wu (Eds.). ACM, 420–429. https://doi.org/10.1145/1281192.1281239

[21] Da Li and Michela Becchi. 2013. Deploying Graph Algorithms on GPUs: An Adaptive Solution. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 1013–1024. https://doi.org/10.1109/IPDPS.2013.101

[22] Hang Liu and H. Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807594

[23] Hang Liu, H Howie Huang, and Yang Hu. 2016. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*. 403–416.

[24] Madhav Marathe and Anil Kumar S Vullikanti. 2013. Computational epidemiology. *Commun. ACM* 56, 7 (2013), 88–96.

[25] Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. 2019. MultiLyra: Scalable Distributed Evaluation of Batches of Iterative Graph Queries. In *2019 IEEE International Conference on Big Data (Big Data)*. 349–358. https://doi.org/10.1109/BigData47090.2019.9006359

[26] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *ACM SIGPLAN Notices* 47, 8 (Sep 2012), 117–128. https://doi.org/10.1145/2370036.2145832

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

[28] Marco Minutoli, Mahantesh Halappanavar, Ananth Kalyanaraman, Arun V. Sathanur, Ryan Mcclure, and Jason E. McDermott. 2019. 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*. 1–12. https://doi.org/10.1109/CLUSTER.2019.8890991

[29] Marco Minutoli, Prathyush Sambaturu, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, and Anil Vullikanti. 2020. Preempt: scalable epidemic interventions using submodular optimization on multi-GPU systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 55. https://doi.org/10.1109/SC41405.2020.00059

[30] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 463–474. https://doi.org/10.1109/IPDPS.2013.28

[31] Nvidia. 2017. Cooperative Groups: Flexible CUDA Thread Programming. https://developer.nvidia.com/blog/cooperative-groups/.

[32] Nvidia. 2023. Nsight Compute CLI. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html.

[33] Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2010. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2010.34

[34] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, New York, USA) *(KDD '14)*. Association for Computing Machinery, New York, NY, USA, 701–710. https://doi.org/10.1145/2623330.2623732

[35] Martin Rosvall and Carl T. Bergstrom. 2008. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences* 105, 4 (2008), 1118–1123. https://doi.org/10.1073/pnas.0706851105

[36] Alexander Schrijver et al. 2003. *Combinatorial optimization: polyhedra and efficiency*. Vol. 24. Springer.

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

[38] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph Processing on GPUs: A Survey. *ACM Comput. Surv.* 50, 6, Article 81 (jan 2018), 35 pages. https://doi.org/10.1145/3128571

[39] Steven S. Skiena. 2008. *The Algorithm Design Manual*. Springer, London. https://doi.org/10.1007/978-1-84800-070-4

[40] Youze Tang, Yanchen Shi, and Xiaokui Xiao. 2015. 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*. 1539–1554. https://doi.org/10.1145/2723372.2723734

[41] Youze Tang, Xiaokui Xiao, and Yanchen Shi. 2014. Influence Maximization: Near-Optimal Time Complexity Meets Practical Efficiency. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 75–86. https://doi.org/10.1145/2588555.2593670

[42] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. 2014. The More the Merrier: Efficient Multi-Source Graph Traversal. *Proc. VLDB Endow.* 8, 4 (dec 2014), 449–460. https://doi.org/10.14778/2735496.2735507

[43] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. *ACM SIGPLAN Notices* 51, 8 (Nov 2016), 1–12. https://doi.org/10.1145/3016078.2851145

[44] Carl Yang, Aydın Buluç, and John D. Owens. 2022. GraphBLAST: A High-Performance Linear Algebra-Based Graph Framework on the GPU. *ACM Trans. Math. Softw.* 48, 1, Article 1 (feb 2022), 51 pages. https://doi.org/10.1145/3466795

[45] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2022. Glign: Taming Misaligned Graph Traversals in Concurrent Graph Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 78–92. https://doi.org/10.1145/3567955.3567963

[46] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. 2005. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 25–25. https://doi.org/10.1109/SC.2005.4

[47] Jingya Zhou, Ling Liu, Wenqi Wei, and Jianxi Fan. 2022. Network Representation Learning: From Preprocessing, Feature Extraction to Node Embedding. *ACM Comput. Surv.* 55, 2, Article 38 (jan 2022), 35 pages. https://doi.org/10.1145/3491206