HBMax: Optimizing Memory Efficiency for Parallel Influence Maximization on Multicore Architectures

Xinyu Chen
Washington State University
Pullman, WA, USA
xinyu.chen1@wsu.edu

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

Jiannan Tian
Indiana University
Bloomington, IN, USA
jti1@iu.edu

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

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

Dingwen Tao*†
Indiana University
Bloomington, IN, USA
ditao@iu.edu

ABSTRACT

Influence maximization aims to select k most-influential vertices or seeds in a network, where influence is defined by a given diffusion process. Although computing optimal seed set is NP-Hard, efficient approximation algorithms exist. However, even state-of-the-art parallel implementations are limited by a sampling step that incurs large memory footprints. This in turn limits the problem size reach and approximation quality. In this work, we study the memory footprint of the sampling process collecting reverse reachability information in the IMM (Influence Maximization via Martingales) algorithm over large real-world social networks. We present a memory-efficient optimization approach (called `HBMax`) based on Ripples, a state-of-the-art multi-threaded parallel influence maximization solution. Our approach, `HBMax`, uses a portion of the reverse reachable (RR) sets collected by the algorithm to learn the characteristics of the graph. Then, it compresses the intermediate reverse reachability information with Huffman coding or bitmap coding, and queries on the partially decoded data, or directly on the compressed data to preserve the memory savings obtained through compression. Considering a NUMA architecture, we scale up our solution on 64 CPU cores and reduce the memory footprint by up to 82.1% with average 6.3% speedup (encoding overhead is offset by performance gain from memory reduction) without loss of accuracy. For the largest tested graph Twitter7 (with 1.4 billion edges), `HBMax` achieves 5.9× compression ratio and 2.2× speedup.

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms**;
• **Theory of computation** → **Graph algorithms analysis**.

KEYWORDS

Influence maximization, compression, multicore architectures

*Dingwen Tao is the corresponding author.
†Dingwen Tao is also with Washington State University.

PACT '22, October 10–12, 2022, Chicago, IL, USA

ACM ISBN 978-1-4503-9868-8/22/10.
https://doi.org/10.1145/3559009.3569647

ACM Reference Format:
Xinyu Chen, Marco Minutoli, Jiannan Tian, Mahantesh Halappanavar, Ananth Kalyanaraman, and Dingwen Tao. 2022. HBMax: Optimizing Memory Efficiency for Parallel Influence Maximization on Multicore Architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 8–12, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3559009.3569647

1 INTRODUCTION

A graph, $G = (V, E)$, captures complex relationships between a set of entities represented as nodes or vertices (V), through binary relations expressed as edges or links (E). Graph analytics provides a set of algorithms such as centrality measures, community detection, shortest paths, and network flow to enable decision making on data presented as graphs. The ubiquity of massive data from domains such as social networks and life sciences has enabled application of graph analytics on numerous domains with varying degrees of scale. A fundamental limitation to the application of graph analytics is the massive memory requirement of algorithms. Since many graph algorithms have irregular accesses to memory and low (arithmetic) computation, the performance of memory system becomes critical.

Given a directed graph $G = (V, E, \omega)$, where ω represents edge weights corresponding to the influence of node x on node y for an edge (x, y); a diffusion model, and a budget k; the *Influence Maximization* (IM) problem is an optimization problem to identify a set of k vertices which when activated result in a maximal number of expected activation in G, where activation is defined for a given model of diffusion. The IM problem came from viral marketing, where a company tried to create a cascade of product adoption through the word-of-mouth effect by choosing a set of influential individuals and giving them free samples of the product. Now It has numerous applications [14, 16, 23] in domains such as politics, public health, bioinformatics, and sensor networks.

Finding the top k influential vertices (i.e., seeds) in a graph can be formulated as a discrete optimization problem that has been shown to be NP-Hard [14]. Consequently, a few key efficient approximation solutions have been developed. For instance, Kempe *et al.* [14] attempts to find an approximate solution by hill-climbing on a large number of Monte Carlo (MC) diffusion processes (typically around 10^4). Borgs *et al.* [6] greatly improves the efficiency of MC simulations by exploiting the potential overlap in the vertex space among multiple simulated paths of diffusion. Their approach enumerates random reverse reachable (RRR) sets and using them to select influential vertices that occur frequently in them. Tang *et*

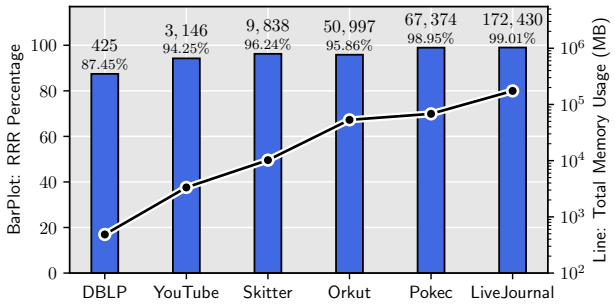

Figure 1: Memory usage breakdown of Ripples [22], where the blue portions represent the computation of random reverse reachable (RRR) paths (87 to 99% of memory usage).

al. [34] further extend this work with a two-phase IMM algorithm (including sample estimation and seed selection) to improve the determination of the sampling effort through a martingale strategy (will be discussed in detail in Section 2). The targeted quality is a $(1 - 1/e - \epsilon)$-approximation. The error factor ϵ plays an important role in the overall performance of the IMM algorithm. Smaller ϵ will produce a more accurate approximation by making the algorithm carry out more MC trials. With the growing size of social networks and the goal of getting high-quality approximation, the IMM algorithm is both computationally challenging and memory demanding. To alleviate this challenge, parallel computing has been introduced to reduce the execution time of the IMM algorithm. Ripples, proposed by Minutoli *et al.* [22] is the state-of-the-art parallel implementation of the IMM algorithm. It can solve the time-cost challenge by parallelizing the computation and scaling up in both the shared-memory and distributed-memory architectures.

However, few studies in the literature address the memory-intensive challenge, which should have a standalone place in the research on the IMM algorithm. More specifically, the reason to address the memory challenge of the IMM algorithm is two-fold: (1) Although the aggregated memory capacity of today's high-performance computing (HPC) systems is ever-increasing, such distributed computing resource is not readily available to most. Thus, reducing the memory usage on shared-memory systems helps to solve larger influence maximization problems with limited computational resources. (2) There is a vast difference between the input graph size and the memory footprint during the computation [12]. This memory inflation (by the algorithm) is particularly pronounced for a stochastic graph application such as IMM. For instance, for the six graphs studied in Figure 1, the ratio of the peak memory usage to the input size varies from 30× to 165×—implying a one to two orders of magnitude increase in memory requirement during the course of computation to store the intermediate results.

To this end, we propose a memory-efficient parallel influence maximization algorithm using Huffman coding and Bitmap coding (called `HBMax`) and implement it based on the state-of-the-art implementation Ripples [21]. Specifically, by profiling the memory footprint of Ripples on large real-world graphs, we identify the most demanding portion of the algorithm and different demands with different types of graphs. With these characteristics, we propose a block-based workflow that leverages the Huffman coding or bitmap coding to save the intermediate MC simulation results in a compressed format. The subsequent analysis can be performed on partially decompressed Huffman encoded data or directly performed on the bitmap encoded data: neither of the two schemes needs to fully decompress the data to the original size, so as to preserve the memory savings. To the best of our knowledge, this work represents the most space-efficient parallel IMM on very large graphs.[1] Our main contributions are summarized as follows.

(1) We conduct a comprehensive characterization and profiling of different real-world graphs to understand the impact of their features on the memory footprint based on the state-of-the-art solution for the influence maximization problem.
(2) We identify the intermediate RRR sets sampled from MC simulations vary from skewed-distributed to flat-head distributed. We propose a scalable "compress-to-compute" based IMM method (called `HBMax`) that leverages the Huffman coding or bitmap coding to reduce the memory footprint of saving intermediate RRR sets based on these characteristics.
(3) We propose two efficient seed selection approaches based on the two encoding methods. Specifically, for Huffman-encoded intermediate RRRs, we exploit data locality to query the compressed RRRs without fully decoding them; for bitmap-encoded RRRs, we directly query the compressed RRRs with bit operations. Moreover, we propose a new parallel max-reduction method for finding the vertex with the maximum frequency to improve the scalability of seed selection.
(4) We evaluate our `HBMax` and compare it with the original Ripples implementation on eight large graphs (Ripples cannot run the largest two graphs on the tested machine with 376 GB RAM). Experiments show that `HBMax` reduces the memory usage by up to 82.1% with 6.3% speedup (the encoding overhead is offset by performance gain from memory reduction). Moreover, `HBMax` can reduce the overall time by up to 79% with the same memory footprint when compared to Ripples.

The rest of this paper is organized as follows. In Section 2, we present background information about the IM problem, state-of-the-art algorithms and implementations, Huffman coding, bitmap coding and related work. In Section 3, we profile the memory footprint of Ripples with different graphs and characterize input graphs into two main categories. In Section 4, we propose our three optimizations for memory footprint reduction and performance/scalability enhancement. In Section 5, we evaluate our optimized IMM solution and compare it with the state-of-the-art method on large graphs. In Section 6, we conclude our work and discuss the future work.

2 BACKGROUND AND RELATED WORK

In this section, we introduce the influence maximization problem, the RIS and IMM algorithms, Ripples software, Huffman/bitmap compression, and compressed-based graph analytics.

2.1 Influence Maximization Problem

Let $G = (V, E)$ be a graph with n vertices and m edges. Given G and a stochastic diffusion process, the influence maximization problem is one of identifying a set S of top k vertices in G (called "seeds") that maximizes the expected influence spread ($\mathbb{E}[I(S)]$) in G, measured by the number of activated vertices. Kempe *et al.* [14]

[1] Our code is available at https://github.com/hipdac-lab/hbmax-pact.

have shown that $\mathbb{E}[I(S)]$ is a submodular function of S for two simple but powerful diffusion models: the independent cascade model (IC) and the linear threshold model (LT). Submodular functions have the property of diminishing marginal gains and leads to efficient approximation algorithms [26]. Leveraging the submodularity framework, they propose a greedy hill-climbing algorithm that offers an approximation guarantee of $1 - 1/e - \epsilon$ [14].

The IC and LT diffusion models are broadly studied in the literature. The IC model comes from the physics of interacting particles. In this model, each newly activated vertex has a single chance to activate its neighbors. Assuming a directed graph $G = (V, E, \omega)$ as an example, each edge $(u, v) \in E$ is assigned with a probability $p(u, v)$ to trigger the activation of v from u. When the vertex u gets activated at time t, then at time $t + 1$, the vertex v gets activated with the probability $p(u, v)$.

In contrast, the LT model captures mass behaviours. Each vertex has a threshold modeling their resistance to adopting the mass behavior and each edge (u, v) has a weight w representing the capacity of u to influence v. At time t, a vertex v gets activated if the sum of the weights on the incoming edges from its already active neighbors exceeds its threshold. It is worth noticing that the edge weights ω are provided by input graphs, but they are not corresponding to the probability of activation. Thus they only affect the LT model instead of the IC model.

The LT diffusion model tends to produce very small RRR sets and more than 99% of them end up with less than 10 vertices [21]. On the contrary, the sizes of RRR sets from the IC model is less skewed and tend to be of larger size. From a utility standpoint, the IC model is more generally applicable to abstract a range of diffusion processes. It can be viewed as a special case of the Susceptible-Infected-Recovered (SIR) models [23] used in epidemics. However, its large memory footprint (as shown in Figure 1) limits its scalability, so we focus on the IC diffusion model in this paper.

2.2 RIS, IMM, and Ripples

Reverse Influence Sampling. Following the seminal work [14], Borgs *et al.* [6] proposed an approximation algorithm based on the idea of Reverse Influence Sampling (RIS). Their scheme attempts to find which are the most likely causes of activation for each vertex in the graph. The algorithm randomly samples vertices v and simulates the diffusion model in reverse collecting sets of vertices which may cause the activation of v. The seed sets S is later decided by solving a maximum coverage problem over the sets collected through RIS. This approach provides the same approximation guarantee as the greedy hill-climbing algorithm [14].

The RIS approach is the fundamental building block of the IMM algorithm from Tang *et al.* [34], and its state-of-the-art parallel implementation Ripples [22] is the starting point for our work. In the following, we give important definitions and a high-level description of the fundamental building blocks of the IMM and Ripples algorithm. We direct the reader to the original work for a more exhaustive presentation.

Definition 2.1 (Reverse Reachable (RR) Set). Let $\hat{G} = (V, \hat{E})$ be the transposed graph obtained from $G = (V, E)$ by inverting the orientation of all the edges in E. The reverse reachable set of a vertex v is the set of vertices $u \in V$ that are reachable from v in $\hat{G}$.

Definition 2.2 (Random Reverse Reachable (RRR) Set). Let $g = (V, E')$ be a subgraph of $G = (V, E)$ obtained by edge removal by retaining only the active edges during a simulation of a diffusion process M on G. A random reverse reachable set $RR_g(v)$ for a vertex v is the set of vertices $u \in V$ that are reachable from v in $\hat{g}$, where $\hat{g}$ is the transposed graph obtained from g.

In the following discussion, we use *RRR sets* to refer to the collection of random reverse reachable sets. We use *rr* or *sample* to refer to one RRR. Since each RRR is a set of vertices, we refer the cardinality of an RRR as the size of RRR.

IMM and Ripples. Although the reverse reachable scheme of Borgs *et al.* greatly improves the efficiency of MC simulations, it could overestimate the required number of MC simulations and waste a lot of computation. The IMM algorithm proposed by Tang *et al.* [34] adopts a two-phase design to algorithmically determine the sampling effort by leveraging a martingale strategy[2]. This strategy greatly improves the performance so that IMM algorithm can analyze large graphs with millions of vertices.

Specifically, the two-phase design works as follows. In the *Sampling* phase, the algorithm will produce θ *RRR*s starting from random vertices in the input graph and simulating the diffusion model (IC or LT) in reverse. In the case of the IC and LT model, the task of generating a RRR set resembles a randomized breadth-first search (BFS) where only a subset of the neighbors of a vertex enter the next frontier. To estimate the required sampling effort (θ) to achieve the approximation guarantee (controlled through the parameter ϵ), the algorithm uses two important results. Borgs *et al.* showed that the fraction of RRR covered during the seed selection process is an unbiased estimator of the influence function while Tang *et al.* were able to prove a lower bound on the sampling effort using an estimation of the influence function. In Equation (1), Tang *et al.* shows the algorithm starts with a guess on θ and doubles the sampling effort at every iteration until the exit condition derived from the lower lower bound is satisfied and the final value of θ is computed. Here n is the number of vertices, k is the number of seeds, ϵ is the error factor, and θ is the required number of sampling. The larger n and k are, the larger θ will be. On the other hand, small error factor ϵ will increase θ non-linearly. This process resembles to the martingale betting strategy.

$$\theta_i = \frac{(2 + \frac{2\sqrt{2}}{3}\epsilon) \cdot (\log \binom{n}{k} \cdot \log(n) + \log\log_2(n)) \cdot 2^i}{2\epsilon^2} \quad (1)$$

The *Seed Selection* algorithm is based on the greedy maximum coverage algorithm [35]. The method iterates k times over the RRR sets to select the vertex v appearing most frequently. At every iteration i, the RRR sets covered by one of the i seeds already selected are ignored. Minutoli *et al.* [21, 22] devise efficient parallel schemes that perform the counting by either updating the state of the previous iteration or recount from scratch when more profitable.

Ripples [29] is a state-of-the-art parallel software framework that provides fast and scalable implementation for the IM problem. According to [21, 22], its CPU version provides a speedup

[2]The martingale strategy is a betting strategy in which a person doubles the bet every time they lose knowing that they must win at some point. It first samples a small number of RRR sets to calculate the achieved influence and then doubles the number of samples until the influence reaches an estimated lower bound.

of 580× over the best sequential baseline using 1024 nodes, and its CPU+GPU version provides a speedup of 760× over the best sequential baseline.

2.3 Huffman Coding and Bitmap Compression

Huffman coding is a classical data compression technique [13]. It assigns variable-length codes to encode target characters based on their relative frequency, which gets better reduction when the data has skewed distribution [30]. The Huffman codes are prefix-free and are typically created as a binary tree with the encoded characters stored at the leaves. There are works that adopt Huffman coding to reduce the memory footprint. For example, Suontausta and Tian [32] applied Huffman coding for efficiently storing decision tree parameters to minimize the memory footprint. Ficara *et al.* [10] adopted Huffman coding to improve counting bloom filters in terms of fast access and limited memory consumption.

Bitmap is a mapping from some domain (e.g., a range of integers) to bits [11]. It not only reduces the data size but also allows efficient direct operations such as binary logic operations [20]. Thus, many efficient bitmap compression schemes have been extensively studied in the database systems, such as BBC, WAH, EWAH, and Roaring [36]. However, none of these work exploited data analytics on the Huffman or bitmap encoded data directly.

2.4 Graph Compression and Data Analytics on Compressed Data

Due to the growing sizes of graphs, researchers have been studying compression techniques for graphs. For example, Randall *et al.* [28] tried to compress the links of web graphs by leveraging the locality of web graphs. Ligra [31] and SlimGraph [2] focused on providing frameworks that can facilitate graph analytics with compression scheme. The Ligra framework mainly utilizes the vertex degrees and graph density to select a scheme to map vertices or edges to integer-arrays or bit-vectors, while SlimGraph focuses more on utilizing statistics of local parts in the input graph to map vertices and edges to higher hierarchies. Thus, its compression kernels can preserve some critical graph properties. However, the use of these frameworks needs to re-program the graph applications with their specific syntax and semantics. This is a non-trivial effort for most of the graph algorithms and applications in the literature.

Moreover, there are a few works that investigate data analytics directly on compressed data without decompression. For example, Zhang *et al.* proposed an approach to perform document analytics (word count, inverted index, and sequence count) directly on compressed textual data on CPUs [42] and GPUs [18, 27, 40]. Moreover, Zhang *et al.* developed a new storage engine, called CompressDB, which can support data processing for databases without decompression [41]. Furthermore, Macko *et al.* [19] proposed LLAMA that performs graph storage and analysis on the compressed sparse row (CSR) representation and achieves performance gain on graph benchmarks (i.e., PageRank, BFS, and triangle counting) due to in-memory execution. In addition, Mofrad *et al.* [25] proposed a compression technique specifically designed for matrix-vector operations based on the compressed sparse column (CSC) representation and leverage this compression to accelerate distributed graph benchmarks (e.g., PageRank, single source shortest path, and BFS). However, no work has been done on using compression to improve memory efficiency and accelerate a real-world diffusion-based graph application such as influence maximization.

3 MEMORY FOOTPRINT PROFILING AND GRAPH CHARACTERIZATION

In this section, we characterize the memory usage of Ripples, the state-of-the-art parallel IMM implementation and discuss the potentials to reduce the memory footprint.

3.1 Memory Usage of RRR Sets

The key component of the Monte Carlo (MC) diffusion process [22, 34] is a probabilistic Breadth First Search. The intermediate collections of vertices that may cause activation of the BFS root are saved in the form of RRR sets. We benchmark six real-world large graphs used in the literature and study the memory usages for the MC diffusion process and generating/saving the intermediate RRR sets.

As shown in Figure 1, the space attributed to storing of the intermediate RRR sets dominate, consuming between 87% to 99% of the memory footprint.

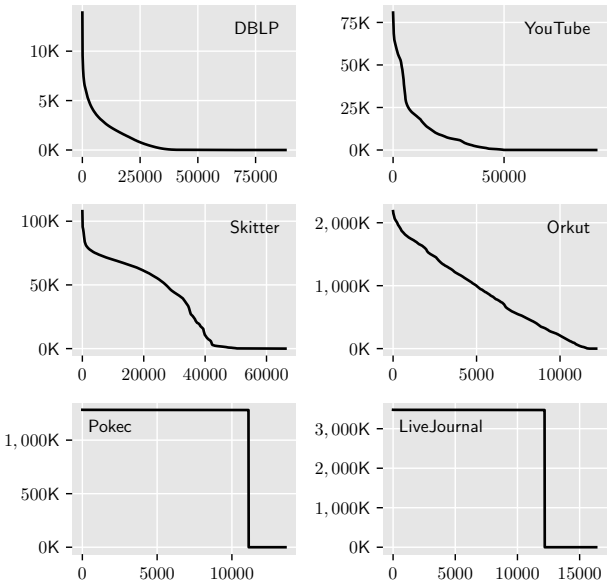

Figure 2: Distributions of RRR set sizes on different graphs (X-axis: the sorted RRR set ID; Y-axis: the number of vertices contained in the RRR set).

To further understand the memory footprint usage patterns in Figure 1, we studied the distribution of the RRR sets by their sizes. Figure 2 shows the sizes of RRR sets, i.e., their vertex counts, for different graphs. The figure illustrates the shapes of RRR sets' distribution based on their sizes: (1) In the top row, the two graphs (i.e., DBLP and YouTube) have long tails, and their shapes correspond to a power-law distribution. (2) In the middle row, the shapes of the two graphs (i.e., Skitters and Orkut) show close to a linear decay without any long tails. (3) In the bottom row, the shapes of the two

Table 1: Skewness $\mathcal{S}$ and density of RRR sets distributions.

Graph	Skewness $\mathcal{S}$	Density $\mathcal{D}$
DBLP	11.46 ± 0.15	$0.261\% \pm 0.001\%$
Youtube	9.01 ± 0.06	$0.630\% \pm 0.001\%$
Skitter	5.38 ± 0.01	$2.030\% \pm 0.001\%$
Orkut	0.75 ± 0.03	$27.73\% \pm 0.03\%$
Pokec	-1.43 ± 0.01	$66.01\% \pm 0.111\%$
LiveJournal	-0.99 ± 0.01	$53.28\% \pm 0.273\%$

graphs (i.e., Pokec and LiveJournal) show a very uniform two-step (flat-head) distribution. The overall memory footprint of the RRR sets corresponds to the area under the curves in the respective plots. This well explains why the last two graphs spends much more memory in saving the RRR sets. Further, these observations raise a critical question to the design of optimizations for memory footprint reduction: What are the characteristics of the various RRR sets of the above three different categories? (see Section 3.2)

3.2 Characterize The RRR Sets

The shapes in Figure 2 roughly categorize the RRR sets' distributions into two types: the first four graphs (i.e., DBLP, YouTube, Skitter and Orkut) have skew-distributed distribution while the last two graphs (i.e., Pokec and LiveJournal) have a uniform two-step distribution. We use two quantities, i.e., skewness and density, to characterize the shapes of RRR sets' distributions.

The skewness score ($\mathcal{S}$) measures how distributions are asymmetric about their mean: When $\mathcal{S} < 0$, the distribution has a longer left tail; When $\mathcal{S} > 0$, the distribution's right tail is longer. Equation (2) shows the calculation of skewness score for the size of RRR sets $X = (X_1, ..., X_\theta)$, where each X_i is a sample of the number of visited vertices starting from a seed vertex in a MC simulation. Thus all X_i's are at least 1 because there will be no empty resultant RRR set (it will at least contain the starting seed vertex); θ is the total number of sampled RRR sets; $\bar{X}$ is the average RRR size, and s is their standard deviation.[3] In Table 1, we show the skewness scores of the first four graphs (i.e., DBLP, YouTube, Skitter and Orkut) are all positive, which help to distinguish them from the last two graphs (i.e., Pokec and LiveJournal) whose skewness scores are negative.

The density ($\mathcal{D}$) measures the proportion of non-zero elements that are required to represent the sampled RRR sets if they are stored in matrix format. Although density is not correspond to the shape of distributions, it can help to select the data structure to store RRR sets. It is sufficient to use 32-bit unsigned integers to represent each vertex in the sampled RRR sets in this study. Thus we can use 3.12% to be the threshold. When the density $\mathcal{D} \leq \frac{1}{32} = 3.12\%$, it is more efficient to use a sparse representation of RRR sets, i.e., to explicitly store each vertex. On the other hand, when the density $\mathcal{D} > 3.12\%$, it is more efficient to use a dense bitmap coding to store RRR sets. Equation (3) shows the calculation of density for the size of RRR sets X (Similarly, θ is the total number of RRR sets; X_i is the number of vertices in each RRR set; n is the number of vertices in graph G). In Table 1, we show the last two graphs (Pokec and LiveJournal) have much higher densities ($\geq 50\%$) than the rest four graphs (DBLP, YouTube, Skitter, Orkut).

$$\mathcal{S} = \frac{\frac{1}{\theta}\sum_{i=1}^{\theta}(X_i - \bar{X})^3}{s^3}, \tag{2}$$

$$\mathcal{D} = \frac{\sum_{i=1}^{\theta} X_i}{\theta \cdot n} \tag{3}$$

3.3 Characterize The Influence of Vertices

The different shapes of RRR sets' distribution also characterize the influence of individual vertices which can help us to find data localities. To verify this intuition, we compare the selected seeds of each graph by varying the random starting vertex for MC samplings. Table 2 shows the Rank-Biased Overlap (RBO) scores [37] (1.0 means highly overlapping rank positions and 0.0 means no overlapping). For the graphs with skew-distributions, the same top-1 influential vertex is consistently selected from different random starting vertices. However, for the graphs with flat-head distributions, many vertices achieve the maximum influence, so the RBO score is zero.

Table 2: Influential seeds with a random start.

Graph	Activated	RBO (Top-1)	RBO (Top-50)
DBLP	0.499	1.0	0.57
YouTube	0.554	0.87	0.24
Skitter	0.794	0.50	0.16
Orkut	0.967	0.46	0.09
Pokec	0.817	0.0	0.0
LiveJournal	0.741	0.0	0.0

From the first four graphs (DBLP, YouTube, Skitter, Orkut), the sampled RRR sets are close to power-law or linear decay distributions. Under this situation, there are only a small number vertices that influence a lot of other vertices; while many vertices only influence one or two other vertices. On the other hand, for the last two graphs (Pokec and LiveJournal), the flat-head shows that many vertices can influence many other vertices. In other words, many vertices are equally influential, which makes the flat-head distributed RRRs lack the data locality in the skew-distributed RRR sets. Thus, they need different optimization strategies to reduce their memory footprints.

In summary, the memory required to store intermediate RRR sets is one to two order of magnitude larger than the memory footprint of the diffusion process. We observe two types of graphs with regards to their different behaviors in diffusion process and hence the data localities. Considering the observed skewed-distributed RRR sets, it is promising to leverage variable-length encoding to reduce the memory footprint of RRR sets for those graphs. On the ther hand, using bitmap coding can reduce the memory footprints for the flat-head distributed RRR sets because of their high density of non-zero elements.

4 OUR PROPOSED OPTIMIZATIONS

In this section, we propose a "compress-to-compute" IMM method to reduce the memory footprint for the Ripples based on our characterization and profiling.

[3]The standard deviation $s \neq 0$ because X_i's are the sizes of RRR sets, which are not all equal. Given the elements in Equation (2) are all non-zero, the skewness score and density will not overflow by dividing zeros for real-world social networks.

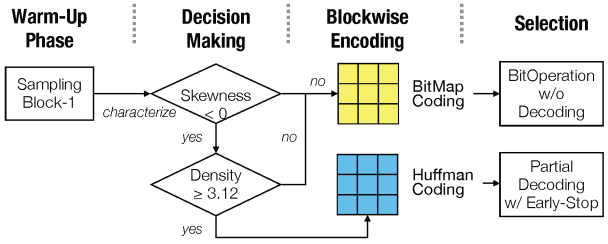

Figure 3: Workflow overview of our proposed solution.

4.1 Overview of Our Proposed Workflow

The overview of the "compress-to-compute" workflow is shown in Figure 3. It contains three main phases: warm-up, sample-and-encode, and decode-and-select. Specifically, we first characterize the skewness score and the density from the distribution of the RRR sets in the warm-up phase. Based on that, we then choose an efficient compression technique to encode the intermediate RRR sets and query on the encoded RRR sets.

More specifically, we introduce a block-based *sampling-and-encoding* approach and an integrated *selection* approach. To achieve a better scalability on multicore architectures, we design two schemes - (1) `Huffmax`: We use Huffman coding to encode skew-distributed RRR sets, and leverage the data locality knowledge acquired from the warm-up phase to accomplish the selection without fully decoding the compressed RRR sets. (2) `Bitmax`: We use bitmap coding to encode flat-head distributed RRR sets. The corresponding selection can be accomplished directly on the encoded data by efficient bit operations. These two schemes are complementary to each other so that the proposed workflow does not rely on the data locality of skewed distributions. It can work with RRR sets that have non-skewed distributed sizes such as normal distributions or uniform distributions (their skewness scores are zero). In Figure 2, we show the flat-headed distributions of Pokec and Livejournal are two special cases that are close to uniform distributions. Furthermore, we use a heuristic approach to optimize the parallel reduction operation to find the vertex with the highest frequency. We use Table 3 to list the notation used in the paper.

4.2 Sampling-and-Encoding

Next, we describe the proposed two encoding methods.

4.2.1 Block-based sampling-and-encoding. We adopt a block-by-block sampling strategy and use the first block as the warm-up phase. This design provides three benefits: First, as aforementioned, the warm-up phase characterizes the graph and enables us to choose between Huffman coding and bitmap coding. Second, the learned characteristics of RRR sets distribution also helps to determine whether the two encoding methods can efficiently compress RRR sets and hence reduce the memory usage of RRR sets. For RRR sets with negative skewness, using the Huffman coding will cause low compression ratio, and the decoding cannot early stop due to the lack of data localities; For RRR sets with density < 3.12%, the bitmap coding will even increase the memory footprints. Third, the block-by-block sampling enables us to interleave the sampling and encoding so that we can release the used memory as soon as possible to maximize the peak memory reduction.

Table 3: Notation used in the paper.

Notation	Description
n	The number of vertices in Graph
RRR	Random Reverse Reachable set
seeds	The selected most influential vertices
k	The number of most influential vertices in seeds
θ	The number of random RRR sets to be sampled
$\mathcal{S}$	The skewness score of RRR sets
$\mathcal{D}$	The proportion of non-zero elements of sampled RRR sets
X_i	The number of vertices in a sampled RRR set
ϵ	The approximation factor
b	The number of blocks that consists the Sampling step
$\mathbb{R}_i$	A block of RRR sets
rr_j	One RR set from the block
$\mathbb{C}_i$	A block of Huffman encoded RRR sets
c_j	The encoded part of one RRR
$\mathbb{CP}_i$	A block of copied buffers
cp_j	The copied part of one RRR
H^*	The Huffman codebook
$\hat{h}$	The shared vertex-frequency table
$\mathbb{B}_i$	A block of bitmap encoded RRR sets
u^*	The current most influential vertex

4.2.2 Encoding of `Huffmax`. We simply check the skewness in the warm-up phase to enable the Huffman coding. Table 1 shows the skewness scores of our benchmark graphs. The skewed RRR sets of the first four graphs have $\mathcal{S} > 0$. They will be encoded with Huffman Codes. Next, we will describe the block-based sampling-and-encoding approach where the Huffman coding method is interleaved with the sampling step. Algorithm 1 presents the block-based sampling-and-encoding method in detail.

Specifically, we split the sampling step into b sub-steps. In each sub-step we get a block of $\frac{\theta}{b}$ RRR sets. Let $\mathbb{R}_1 = \{rr_1, ..., rr_{\theta/b}\}$ be the first block. We build a Huffman codebook H^* based on the block $\mathbb{R}_1$. We use this codebook H^* to encode $\mathbb{R}_1$ into a set of byte-strings $\mathbb{C}_1 = \{c_1, ..., c_{\theta/b}\}$, each of which corresponds to a RRR set. At the same time, we construct a frequency table $\hat{h}$ to store the vertex frequency of the block $\mathbb{R}_1$. In the following sub-steps, we keep updating the frequencies stored in the table $\hat{h}$ and encoding the RRR sets $\mathbb{R}_i$ into $\mathbb{C}_i$, where $i = 2, \cdots, b$. Because of the data localities of the skew-distributed RRR sets, we can keep track of the current most influential vertex u^* and swap it to the beginning position of the encoded byte string whenever it appears in the corresponding RRR set. Note that this swap is beneficial as it enables early-stopping (will be described in Section 4.3). At the end of each sub-step, to reduce the peak memory usage, the memory for $\mathbb{R}_i$ is deallocated once the encoding of the current block completes.

Ideally, the codebook H^* built from the first block $\mathbb{R}_1$ should contain the Huffman codes for all vertices in the entire RRR sets. However, in our block-based sampling-and-encoding approach, some vertices may not be sampled in $\mathbb{R}_1$ but could appear in later blocks due to the skewed distributions. The more skewed distributions the RRR sets have, the more likely some vertices are missing from $\mathbb{R}_1$. For example, 0.2% of DBLP's vertices have no Huffman-codes where the skewness score is 11.46; 0.02% vertices of Skitters have no Huffman-code where the skewness is 5.38; and 0.003% vertices of Orkut's have no Huffman-code where the skewness is 0.75. For those vertices do not have corresponding Huffman codes in the

Algorithm 1 Sampling-and-Encoding(G, θ, b)

$\mathbb{B} = \phi$, $\mathbb{C} = \phi$, $\mathbb{CP} = \phi$, $\hat{h}$ = zeros
for $i \leq b$ **do**
  $\mathbb{R}_i$ = Sampling$\left(G, \frac{\theta}{b}, \text{threshold}\right)$
  if $i \equiv 1$ **then**
    $\mathcal{S}, \mathcal{D}$ = Characterize($\mathbb{R}_i$)
    if $\mathcal{S} < 0$ **then**
      `Bitmax`= *True*
    else
      `Huffmax`= *True*
      H^* = BuildHuffmanCode($\mathbb{R}_i$);
    end if
  end if
  if `Huffmax`$\equiv$ *True* **then**
    $\hat{h}$ = UpdateHistogram$\left(\mathbb{R}_i, \hat{h}\right)$
    $[\mathbb{C}_i, \mathbb{CP}_i]$ = HuffmanEncode$\left(H^*, \hat{h}, \mathbb{R}_i\right)$
    $\mathbb{C} = \mathbb{C} \cup \mathbb{C}_i$; $\mathbb{CP} = \mathbb{CP} \cup \mathbb{CP}_i$
  end if
  if `Bitmax`$\equiv$ *True* **then**
    $\mathbb{B}_i$ = BitmapEncode$\left(\mathbb{R}_i\right)$
    $\mathbb{B} = \mathbb{B} \cup \mathbb{B}_i$
  end if
  Deallocate $\mathbb{R}_i$
end for

codebook H^*, we use an additional array $\mathbb{CP}_i$ to directly save those vertices without encoding. By doing this, we preserve the exact information of RRR sets by encoding and/or copying all the vertices in the original RRR sets. From our empirical studies, the vertices that do not have corresponding Huffman codes only consist of less than 0.2% of the entire vertices. Thus we do not build new Huffman code book in the following steps to reduce the overhead.

4.2.3 Encoding of `Bitmax`. We not only check the skewness (< 0), but also check the density ($> 3.12\%$) to enable the bitmap coding (as mentioned in Section 3.2). For a block of RRR sets $\mathbb{R}_i$, the encoded data is represented as a dense bit matrix $\mathbb{B}_i$, where the shape of matrix $\mathbb{B}_i$ is n rows by $\frac{\theta}{b}$ columns (n is the number of vertices in graph G, $\frac{\theta}{b}$ is the number of RRR sets sampled in this block). If the bit at the r-th row and c-th column in matrix $\mathbb{B}_i$ is set to 1, it represents the r-th vertex appears in the c-th RRR set. In our implementation, we pad $\frac{\theta}{b}$ columns to be the multiple of 32 so that we can save the bit matrix in byte format. Because the padded bits are all zeros, the padded columns do not affect the correctness of the method. Similar to `Huffmax`, we deallocate the $\mathbb{R}_i$ after the bitmap encoding to reduce the peak memory usage.

4.2.4 Parallel implementation. The sampling-and-encoding computation on one sample is independent of other RRR sets. To accelerate the computation, we distribute the sampling-and-encoding workload to multiple threads/cores. For `Huffmax`, the bottleneck is to perform a parallel reduction to build a shared frequency table $\hat{h}$ considering the NUMA effect [15]. Our solution is to follow the first-touch principle: we allocate a local frequency table h^{local} on each thread to store the vertex frequencies and sum them to the global frequency table after the sampling-and-encoding of this block completes (the synchronization happens b times as we split the sampling-and-encoding into b sub-steps). For `Bitmax`, its encoding is embarrassingly parallel on multi-cores for high scalability.

Algorithm 2 `HuffmaxDecodeQuery`$\left(H^*, \hat{h}, \mathbb{C}, \mathbb{CP}\right)$

u^*=argmax($\hat{h}$), seeds=ϕ , deleteRRRflag[θ]=*False*
while |seeds| $\leq k$ **do**
  $\hat{h}$ = zeros; seeds = seeds $\cup\, u^*$
`#pragma omp parallel for`
  for $j = 1 \ldots \theta$ **do**
    if deleteRRRflag$_j \equiv$ *False* **then**
      tmp, findflag = DecodeFind(H^*, c_j, cp_j, u^*)
      if findflag $\equiv$ *True* **then**
        deleteRRRflag$_j$ = *True*
      else
        $\hat{h}$ = UpdateHistogram$\left(tmp, cp_j, \hat{h}\right)$
      end if
    end if
    Deallocate *tmp*
  end for
  u^*=argmax($\hat{h}$)
end while

4.3 Optimized Selection

Lastly, we describe two selection approaches for efficiently querying in `Huffmax` and `Bitmax`, respectively. The selection iterates two core computations for k times to select the most influential seeds. The first computation is to locate and remove the RRR sets which contain the current most frequent vertex u^*. The second computation is to reconstruct the frequency table $\hat{h}$ after every RRR set that contains the previous u^* is removed. After that, a new u^* is selected based on the updated frequency table $\hat{h}$.

4.3.1 Selection of `Huffmax`. For RRR sets encoded by Huffman coding, we leverage the data locality to swap the most frequent vertex u^* to the beginning position during the `Huffmax` encoding steps (described in Section 4.2.2). This enables us to partially decode an encoded RRR set (i.e., c_j where $j = 1, \cdots, \theta$) and stop early whenever it contains u^*. Due to the skew-distributed RRR sets, we only need to decode a small number of encoded RRR sets. As aforementioned in Table 1, graphs of greater skewness scores (DBLP/11.46, YouTube/9.01) also have longer tails than graphs of smaller skewness scores(Skitter/5.38, Orkut/0.75). However, the difference of tails does not affect the selection efficiency: even graph with small positive skewness (Orkut/0.75) still has the data locality that enables `Huffmax` to early-stop in the selection step. Note that when we decode c_j back to a temporary buffer $tmp = \{v_1, \cdots\}$, it implies that the current u^* does not appears in the encoded part of this RRR set; otherwise, we would have early stopped decoding. In this case, we will search the corresponding copied array cp_j.

To reconstruct the new frequency table $\hat{h}$, we only need to count the vertices in the fully decoded buffer *tmp* and the corresponding copied array cp_j if u^* is not found. At this point, the temporary buffer *tmp* is safely deallocated to reduce memory footprints. Algorithm 2 describes this `Huffmax` selection in detail.

Algorithm 3 BitmaxQuery$(\hat{h}, \mathbb{B})$

```
1: u*=argmax(ĥ),  seeds=φ ,  deleteVTXflag[n]=False
2: while |seeds| ≤ k do
3:     ĥ = zeros;  seeds = seeds ∪ u*
4: #pragma omp parallel for
5:     for i = 1...n do
6:         if deleteVTXflag_i ≡ False then
7:             SUBTRACT (v_i, u*)
8:             if POPCOUNT(v_i) ≡ 0 then
9:                 deleteVTXflag_i = True
10:            else
11:                ĥ = UpdateHistogram(POPCOUNT(v_i))
12:            end if
13:        end if
14:    end for
15:    u*=argmax(ĥ)
16: end while
```

4.3.2 Selection of Bitmax. Unlike Huffmax's selection, where we leverage the data localities to partially decode the compressed data, the selection in Bitmax directly operate on the encoded data without decoding. The selection is accomplished by efficient bit operations. Note that the shape of bitmap encoded $\mathbb{B}$ is different from the Huffman encoded $\mathbb{C}$; we will use an example to illustrate the two core selection computations (i.e., locate and remove RRR sets which contains u^*, reconstruct frequency table to find the new u^*).

$$\begin{pmatrix} v_1^*|1 & 1 & 0 & 0 & 1 & 1 \\ v_2|0 & 1 & 1 & 0 & 0 & 1 \\ v_3|1 & 0 & 1 & 0 & 1 & 0 \\ v_4|0 & 1 & 0 & 1 & 0 & 1 \\ v_5|0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} v_1^*|0 & 0 & 0 & 0 & 0 & 0 \\ v_2|0 & 0 & 1 & 0 & 0 & 0 \\ v_3|0 & 0 & 1 & 0 & 0 & 0 \\ v_4|0 & 0 & 0 & 1 & 0 & 0 \\ v_5|0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \quad (4)$$

Equation (4) shows a simple example that has 5 vertices and 6 RRR sets. The frequency table $\hat{h}$ is computed by row-wise POPCOUNT operation (i.e., find the sum of ones in each row). Given their frequencies, v_1 is selected as the current most frequent u^*. We do not need to locate the RRR sets because their locations are explicitly represented by the 1s in v_1's row (for simplicity, we use the notation v_i to represent the v_i's row). To remove these RRR sets, we subtract row v_1 from all the other rows. We use two bit operations to accomplish the SUBTRACT operation on the bitmap-encoded data. Specifically, we use tmp = v_i AND (v_i XOR u^*) to subtract row u^* from row v_i.

After SUBTRACT, we use row-wise POPCOUNT again on the updated bitmap matrix $\mathbb{B}$ to reconstruct the updated frequency table $\hat{h}$. Algorithm-3 describes the selection of Bitmax in detail.

4.3.3 Parallel implementation. The above example in Equation (4) demonstrates the bit operations of POPCOUNT, AND, and XOR on each row of vertices. However, the bit-columns of the encoded bitmap $\mathbb{B}$ are distributed across different threads according to the first touch principle (as described in Section 4.2.4). It is important to consider the NUMA effect to parallelize the selection step of Bitmax along the bit-column direction instead of iteratively applying bit operations on each row. More concretely, each thread keeps track

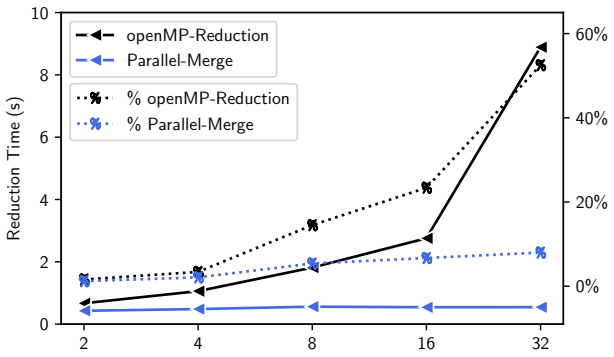

Figure 4: Time comparison of reduction on Skitter's frequency table $\hat{h}$ with different # of threads (solid line: absolute time; dashed line: relative to the time-to-solution).

of a local frequency table h^{local} and these local frequency tables are reduced to the global frequency table $\hat{h}$ after each iteration. The next section describes our solution to address the scalability challenge on getting the global most frequent vertex u^* without reducing the entire global frequency table $\hat{h}$.

4.3.4 Parallel merge. Both selections in Huffmax and Bitmax need to find the new most influential vertex u^* after reconstructing the frequency table. However, we note that to generate the overall reconstructed frequency table $\hat{h}$ from multiple threads, a parallel reduction with k times of synchronizations is needed to sum local frequencies for all vertices. This reduction introduces high time overhead, as shown in Figure 4. The reduction times are measured on the Skitter graph. The dashed line shows the runtime of the OpenMP reduction function with 2~32 threads/cores. It illustrates that the original OpenMP reduction would become the performance bottleneck as the number of threads increases. The reduction time takes upto 52.5% of the entire time-to-solution.

To solve this issue, we propose to use the following approach to greatly decrease the number of reductions so as to make the selection step more efficient and scalable: (1) Select the locally most frequent vertex from each thread (these are local maxima); (2) Perform reductions to get the global frequencies for these locally selected vertices (the local maxima); and (3) Select the vertex with the maximum global frequency from the locally selected vertices (get the global maximum from these local maxima). Note that the total number of reduction operations in (2) is $k \times p$ instead of $k \times n$, where p is the number of threads and n is the number of vertices in graph ($p << n$). Figure 4 illustrates that our proposed parallel-merge-based reduction is highly efficient and scalable with almost constant time cost. In this example, the number of vertices of Skitter is about 1.6 millions. With 32 threads, the OpenMP reduction needs to reduce 650 MB data, whereas our approach only needs to reduce 12.5 KB data[4]. This explains why our approach is more efficient with the constant computation time of about 0.5 seconds.

The heuristic is based on the fact that (1) The global RRR sets is the collection of local RRR sets from each thread. The local RRR

[4]In the skitter example (n=1.6M, k=100), each frequency needs 4 bytes (one float), the baseline reduction needs to sum 1.6M*100*4=650MB data; and with 32 threads, our proposed method sums 32×100×4=12.5 KB data.

sets on each thread is independent because the MC simulations are uniformly sampled. (2) Let $P_v(X)$ be the distribution of vertex v's frequency in the global RRR sets. Then, the distribution of vertex v's frequency on each thread is $P_v(X/p)$. (3) Given two vertices u, v, the corresponding distributions of their frequencies are $P_u(X), P_v(X)$ if they are measured from the global RRR sets. Let $P_u(X) \geq P_v(X)$, without loss of generality, we also have $P_u(X/p) \geq P_v(X/p)$ on each thread. (4) Thus the globally most frequent vertex is among the locally most frequent vertices from each thread.

5 EXPERIMENTAL EVALUATION

In this section, we first present our experimental setup. We then demonstrate the effectiveness of our "compress-to-compute" method. After that, we show the performance of HBMax in memory usage and computation time. Finally, we show that our method has strong scalability on the multicore architectures.

5.1 Experimental Setup

Test datasets. For evaluation purpose, we use the five largest graphs tested in the Ripples software and three large graphs not tested by Ripples[5]. Table 4 shows the main features of these widely studied networks. In detail, (1) DBLP [39] describes a co-authorship network of researchers in computer science; (2) YouTube [24] is a social network of friendship and groups on the video-sharing web site; (3) Skitter [8] is an internet topology graph by depicting the forward paths that actual packets traversed to a destination; (4) Orkut [24] is generated from an on-line social networking site where its primary purpose is to finding and connecting new users; (5) Pokec [33] is an on-line social network in Slovakia and Czech Republic; (6) LiveJournal [1] is a graph from a social networking and blogging site, with explicit user-defined communities; (7) Arabic-2005 [3–5] crawls web pages written in Arabic; and (8) Twitter7 [38] contains the graph of Twitter posts covering a 7-month period. The first six graphs are characterized (as shown in Figure 2) to help us design HBMax, while the last two are not characterized beforehand. We use these two graphs as a test set to verify the effectiveness of HBMax. For the Arabic graph, $\mathcal{S} = -0.25$, $\mathcal{D} = 0.22$, and for the Twitter7 graph, $\mathcal{S} = -3.19$, $\mathcal{D} = 0.62$. Thus, these two graphs use Bitmax. Note that the input data of the two largest graphs uses large amount of storage. Specifically, Arabic has over 22.7 million vertices and 639.9 million edges (requiring more than 11 GB storage overhead), while Twitter7 has 41.6 million vertices and 1.46 billion edges (requiring more than 23 GB storage overhead). These two largest graphs cannot be run by Ripples on our tested platforms due to memory overflow.

Evaluation platforms. We evaluate our proposed method and compare it with the baseline on two platforms. The first platform is a local workstation which has 4 Intel Xeon Gold 6238R CPUs, 376 GB RAM and 1 TB NVMe SSD storage. This local workstation is used to make a fair comparison with Ripples because it has larger memory. The second platform is a Regular Memory (RM) compute node from the Bridges-2 supercomputer [7] at PSC, which has 2 AMD EPYC 7742 CPUs (64 cores per CPU) and 256 GB RAM. The Bridges-2 compute node is used for scalability study because it has more CPU cores. We use GCC-8.3.1 for compilation.

[5]Tested graphs are from the the SNAP collection [17], the UFL sparse matrix collection [9], and Laboratory for Web Algorithms (LAW) [4, 5].

Table 4: Input real-world graphs tested.

Network	#Vertices	#Edges	Avg Deg	Max Deg
DBLP	317,080	1,049,866	3.31	306
YouTube	1,134,890	2,987,624	2.63	28,576
Skitter	1,696,415	11,095,298	6.54	35,387
Orkut	3,072,441	117,185,083	76.28	33,313
Pokec	1,632,803	30,622,564	37.51	20,518
LiveJournal	4,847,571	68,993,773	28.47	22,889
arabic-2005	22,744,080	639,999,458	28.14	575,618
twitter7	41,652,230	1,468,365,182	35.25	770,155

Table 5: Sampling time (in seconds) and reduced page faults (in percentile) of Ripples and HBMax.

Graph	HBMax	Ripples	Reduced Page Faults
DBLP	0.45	0.50	29.5%
YouTube	4.30	5.78	24.3%
Skitter	13.43	17.56	10.5%
Orkut	196.02	245.25	25.2%
Pokec	196.79	257.96	31.0%
LiveJournal	612.60	761.59	42.8%
arabic-2005	1297.60	NA	NA
twitter7	11219.10	NA	NA

Parameter setup. We choose $k = 100$ for all the graphs. We use $\epsilon = 0.2$ for DBLP, YouTube, and Skitter, $\epsilon = 0.5$ for Orkut, Pokec, and LiveJournal, and $\epsilon = 0.7$ for Arabic and Twitter based on the sizes of these graphs. Note that it takes 24+ hours to run Twitter7 using 2 cores on Bridges2.

Our implementation and comparison baseline. We use HBMax to denote our solution; specifically, we use Huffmax to denote our solution using the Huffman encoding with partial decoding for selection and Bitmax to denote our solution using the bitmap encoding with non-decoding selection. We compare HBMax with the original Ripples v.2.1. We implement our optimization in parallel based on this version of Ripples by using OpenMP-4.1.1. Note that Ripples provides a flexibility to configure the number of threads for sampling and selection phases separately. In our evaluation, HBMax uses the same number of threads for both phases, while Ripples can use different number of threads in the selection phase to get the best performance.

5.2 Performance Evaluation

In this section, we evaluate the performance of our proposed HBMax. We first evaluate on the local workstation to compare the memory usage and time-to-solution of HBMax with Ripples because it has larger memory (376 GBs). The experiment results are measured with 16 cores. For a fair comparison, we repeat the experiments on each tested graph for five times and take the averages to compare memory usage and time-to-solution of HBMax with Ripples. We show the averaged experiment results in Table 6, 7 and 8. We then use Bridges-2 to study the scalability of HBMax because its compute

Table 6: Memory footprint (in MB) and reduction ratio (shown in parenthesis).

Graph	DBLP	YouTube	Skitter	Orkut	Pokec	Journal	Arabic	Twitter7
Ripples	424 (1.00)	3,143 (1.00)	9,838 (1.00)	46,506 (1.00)	55,682 (1.00)	163,745 (1.00)	348,606(1.00)	1,193,006(1.00)
Huffmax	316 (1.34)	1,722 (1.83)	5,293 (1.86)	30,130 (1.54)	-	-	-	-
Bitmax	-	-	-	-	10,661 (5.22)	29,329 (5.58)	81,504(4.28)	200,250(5.96)

Table 7: Time-to-solution (in second) and overhead ratio (shown in parenthesis).

Graph	DBLP	YouTube	Skitter	Orkut	Pokec	Journal	Arabic	Twitter7
Ripples	0.95 (1.0)	6.95 (1.0)	20.46 (1.0)	249.35 (1.0)	262.66 (1.0)	775.58 (1.0)	NA	NA
Huffmax	1.10 (1.16)	6.31 (0.91)	17.93 (0.88)	235.14 (0.94)	-	-	-	-
Bitmax	-	-	-	-	222.63 (0.85)	692.70 (0.89)	1,608.48	12,098.30

Table 8: Time-to-solution (in second) and overhead ratio with the same memory footprint.

Graph	DBLP	YouTube	Skitter	Orkut	Pokec	Journal	Arabic	Twitter7
Ripples	1.68 (1.0)	20.69 (1.0)	85.95 (1.0)	669.85 (1.0)	998.85 (1.0)	1930.60 (1.0)	5463.62(1.0)	26825.60(1.0)
HBMax	1.10 (0.66)	6.31 (0.30)	17.93 (0.21)	235.14 (0.35)	222.63 (0.22)	692.70 (0.36)	1608.48 (0.29)	12098.30 (0.45)

node has more CPU cores (i.e., 64 cores per CPU). We show the scalability results in Figure 5 and 6.

5.2.1 Memory Reduction Evaluation. Table 6 shows the memory footprint and memory reduction ratio of our proposed solution compared with Ripples. The reduction ratio is computed as the ratio of our memory usage to Ripples's memory usage. With the proposed "compress-to-compute" techniques, Huffmax achieves an average of 1.6× memory reduction on four skew-distributed RRR sets (i.e., reduces 39.1% memory footprint); Bitmax achieves an average of 5.3× memory reduction (i.e., reduces 81.0% memory footprint)on two flat-head distributed RRR sets. Note that the memory usage of Ripples are colored to grey for the last two graphs (Arabic and Twitter7). The numbers are projected from accumulation. The original Ripples cannot finish due to out-of-memory (OOM) error.

5.2.2 Time-to-Solution Evaluation. Table 5 shows HBMax has better performance than Ripples in the sampling phase. This is because, although the sampling phases of HBMax and Ripples are identical (HBMax only optimizes the storage of intermediate RRR sets in the seed selection phase), reducing memory footprint can decrease the page faults for the sampling phase (the average reduced percentile of page faults is 27.2%[6]) and hence shorten the sampling time and the overall time-to-solution. In other words, the encoding/decoding overheads are offset by the performance gain from the improved sampling phase.

Table 7 shows the time-to-solution of our proposed solution compared with Ripples. We calculate the overhead (in ratio) as $Time_{ours}/Time_{base}$. When our time-to-solution is longer than that of Ripples, the overhead is greater than 1.0; otherwise, it is less than 1.0. The table shows that Huffmax achieves an average overhead of 0.97 for the four skew-distributed RRR sets, while the Bitmax achieves an average overhead of 0.87 for the two flat-head distributed RRR sets. In other words, Huffmax reduces the time-to-solution and the memory footprints at the same time. More concretely, the average reduction on peak memory usage is 52.5% with 6.3% speedup on time-to-solution on all the tested graphs (not including the last two graphs that Ripples gets OOM error). Our solution reduces the peak memory usage by up to 82.1% (on LiveJournal) and the time-to-solution by up to 15.0% (on Pokec).

Moreover, the memory reduction also brings performance benefits to resource-constrained systems. To simulate such an environment, we simply limit the amount of available memory for Ripples to be the same as the memory usage by HBMax. We modify Ripples to let it write the exceeded RRR sets during the sampling step to external SSD storage and read them back for the seed selection step. Table 8 shows the performance gain of HBMax over Ripples with the same limited memory capacity. The in-memory compression technique increases the overall performance on the eight skewed-distributed graphs. The average performance gain is 64.4%, which equals 3.17× speed-up. This is obviously because Ripples increases I/O time due to insufficient memory, while our solution saves RRR sets in a compressed format to avoid data movement.

5.3 Strong Scalability Evaluation

We now present the strong scaling result of HBMax with up to 64 CPU cores, as shown in Figure 6. We break down HBMax into four operations: (1) sampling the RRR sets with MC diffusion process, (2) encoding the RRR sets with Huffmax, (3) encoding the RRR sets with Bitmax method, and (4) selecting the most influential seeds. We analyze their impacts to the overall scalability separately.

First, Figure 5 shows that the parallel sampling step is the dominant part on all evaluated graphs (takes an average of 83.30% of the total time) and has a strong scalability due to the independent nature of each sampling operation; thus, the scalability of HBMax is affected by the encoding step and selection step.

Second, for Huffmax, we note that building the Huffman codebook does not affect the overall scalability. This is because we only build up the codebook once during the warm-up phase, which takes an average of 2.19% of the total time. Also, the Huffman encoding

[6]the reduced page fault percentile = $(\mathrm{PF}_{\texttt{Ripples}} - \mathrm{PF}_{\texttt{HBMax}})/\mathrm{PF}_{\texttt{Ripples}} \times 100\%$

		2	4	8	16	32	64
DBLP	tree	22 (0.60%)	22 (1.06%)	22 (1.81%)	22 (2.57%)	22 (2.73%)	22 (1.96%)
	samp.	2275 (61.22%)	1215 (58.47%)	684 (56.01%)	484 (56.41%)	494 (61.55%)	794 (69.52%)
	enc.	309 (8.30%)	173 (8.35%)	94 (7.76%)	52 (6.11%)	34 (4.19%)	32 (2.84%)
	dec.	1110 (29.12%)	667 (31.08%)	420 (32.28%)	299 (31.90%)	253 (28.67%)	293 (24.14%)
	sum	3717	2078	1222	859	804	1142
YouTube	tree	137 (0.45%)	135 (0.84%)	129 (1.46%)	135.8 (2.42%)	137 (3.39%)	136 (2.88%)
	samp.	26608 (87.03%)	13876 (85.75%)	7268 (81.70%)	4385 (78.35%)	3095 (76.48%)	3396 (71.34%)
	enc.	2848 (9.32%)	1502 (9.28%)	775 (8.72%)	462 (8.27%)	267 (6.60%)	178 (3.74%)
	dec.	981 (3.17%)	668 (4.05%)	722 (7.86%)	613 (10.38%)	547 (12.72%)	1049 (20.89%)
	sum	30575	16183	8896	5598	4047	4760
Skitter	tree	516 (0.46%)	515 (0.90%)	514 (1.68%)	519 (3.04%)	512 (4.69%)	516 (4.79%)
	samp.	100936 (89.90%)	51492 (89.37%)	26818 (87.26%)	14553 (85.13%)	8848 (80.96%)	7428 (69.02%)
	enc.	9743 (8.68%)	4732 (8.21%)	2626 (8.54%)	1304 (7.63%)	738 (6.76%)	490 (4.56%)
	dec.	1079 (0.96%)	878 (1.51%)	776 (2.49%)	717 (4.10%)	829 (7.33%)	2328 (20.88%)
	sum	112276	57619	30736	17094	10930	10764
Orkut	tree	5586 (0.44%)	5503 (0.82%)	5526 (1.59%)	5398 (2.73%)	5584 (4.44%)	5432 (4.90%)
	samp.	1058973 (84.27%)	565117 (84.61%)	292644 (83.94%)	162991 (82.57%)	102617 (81.54%)	92557 (83.47%)
	enc.	191110 (15.21%)	96622 (14.47%)	49723 (14.26%)	28388 (14.38%)	17093 (13.58%)	11846 (10.68%)
	dec.	981 (0.08%)	668 (0.10%)	722 (0.21%)	613 (0.31%)	547 (0.43%)	1049 (0.90%)
	sum	125651	667912	348617	197393	125843	110885
Pokec	samp.	1772380 (96.69%)	895568 (96.19%)	457269 (95.28%)	248158 (93.84%)	136100 (90.43%)	107484 (87.41%)
	enc.	56210 (3.07%)	32725 (3.52%)	20755 (4.32%)	14826 (5.61%)	12543 (8.33%)	12062 (9.81%)
	sel.	4444 (0.24%)	2702 (0.29%)	1910 (0.40%)	1464 (0.55%)	1855 (1.23%)	3425 (2.79%)
	sum	1833035	930996	479935	264450	150499	122972
LiveJournal	samp.	4595970 (96.18%)	2381220 (95.69%)	1370680 (95.23%)	653503 (92.63%)	362924 (86.97%)	276321 (82.59%)
	enc.	167383 (3.50%)	98444 (3.96%)	62422 (4.34%)	46721 (6.62%)	47836 (11.46%)	48252 (14.42%)
	sel.	15230 (0.32%)	8879 (0.36%)	6199 (0.43%)	5286 (0.75%)	6529 (1.56%)	9980 (2.98%)
	sum	4778584	2488544	1439302	705511	417291	334554
Arabic	samp.	4541380 (86.71%)	2318360 (87.01%)	1238270 (86.19%)	670838 (84.19%)	423780 (82.23%)	368972 (81.81%)
	enc.	358265 (6.84%)	205955 (7.73%)	137121 (9.54%)	92893 (11.66%)	61262 (11.89%)	38105 (8.45%)
	sel.	337602 (6.45%)	140188 (5.26%)	61305 (4.27%)	33066 (4.15%)	30298 (5.88%)	43931 (9.74%)
	sum	5237247	2664503	1436697	796798	515341	451009
Twitter7	samp.	48012600 (89.82%)	24223500 (92.78%)	12736200 (93.94%)	6922660 (96.25%)	4138640 (81.66%)	2819580 (80.95%)
	enc.	5349020 (10.01%)	1826720 (7.00%)	778796 (5.74%)	235330 (3.27%)	876097 (17.29%)	631004 (18.12%)
	sel.	89864 (0.17%)	58820 (0.23%)	42471 (0.31%)	34271 (0.48%)	532268 (1.05%)	32347 (0.93%)
	sum	53451485	26109041	13557467	7192261	5068006	3482932

Figure 5: Time breakdown of HBMax with different numbers of threads/cores on our tested graphs. The underscore's length of each operation is proportional to its runtime.

takes an average of 8.77% of the total time as it only includes a small number of synchronizations to build frequency table. Thus, Huffmax's encoding shows a strong scalability up to 64 cores.

Third, for all the large graphs (Pokec, LiveJournal, Arabic, Twitter7), the encoding of Bitmax shows a strong scalability up to 64 cores due to its independent nature. The bitmap encoding takes an average of 8.19% of the total time.

Fourth, the selection operation in both Huffmax and Bitmax does not affect the overall scalability. First, our optimized reduction with parallel merge plays an important role, as shown in Figure 4, to minimize the synchronization between threads. Without this optimization, the selection operation would become the bottleneck due to the original non-scalable parallel reduction. Second, both the selection operations of Huffmax and Bitmax are parallelized considering the NUMA effect so that each thread can maintain its local frequency table by only accessing the locally encoded data. Note that selection of Bitmax scales better than Huffmax because it does not have a decoding step.

Finally, we note that for the two small graphs DBLP and YouTube, Ripples does not scale well. This is because of the highly imbalanced workload between vertices in the seed-selection step, which can be also reflected in its high skewness score. While HBMax achieves better scalability (can scale up to 32 cores) thanks to the proposed

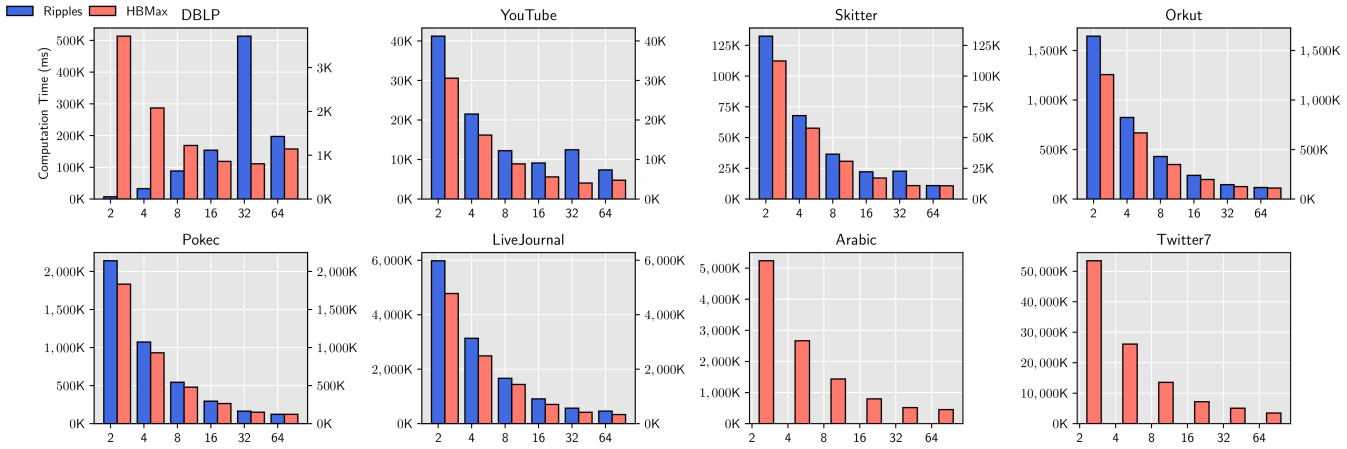

Figure 6: Both HBMax and Ripples shows strong scalability on tested graphs, except Ripples does not scale on the DBLP and YouTube graph. Note that Ripples does not run on the last two graphs - Arabic and Twitter7 because of OOM error.

parallel merge and the consideration of NUMA effect. The performance starts to degrade from 64 cores due to the relatively small workload. For the other six larger graphs, HBMax can scale up to 64 cores and achieve an average speedup of 12.98× on 64 cores.

6 CONCLUSIONS AND FUTURE WORK

Graph analytics has emerged as an important class of data analytics tools. The ubiquity of data from a wide variety of sources has necessitated the development of scalable graph analytics tools. Influence Maximization (IM) is an important graph problem with many applications. Wide use of IM is currently limited due to the limitations from memory and computation requirements. In this paper, we presented a "compress-to-compute" approach to use Huffman or bitmap coding to encode the sampled RRR sets. By exploiting the data locality and leveraging the efficient bit operations, our method is able to reduce the peak memory usage by up to 82.1% and reduce computation by up to 15.0% without noticeable loss of accuracy.

Future research includes: i) establishing analytical bounds on the loss of information so that approximation guarantees of IM algorithms can be established; ii) extension to distributed platforms with GPU accelerators, and iii) extension of compression techniques to other key graph algorithms (with large memory footprints) that address fundamental graph problems such as triangle counting, community detecting, and network alignment, which can potentially benefit from our proposed vertex-encodings.

ACKNOWLEDGMENTS

The research is supported by the U.S. DOE Exascale Computing Project's (ECP) (17-SC-20-SC) ExaGraph codesign center at Pacific Northwest National Laboratory (PNNL) and by the NSF awards OAC-2034169, OAC-2042084, OAC-1910213, SHF-1919122, and CCF-1815467 at Washington State University. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. This work used the Bridges-2 system, which is supported by the NSF award OAC-1928147, at Pittsburgh Supercomputing Center.

REFERENCES

[1] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 44–54.

[2] Maciej Besta, Simon Weber, Lukas Gianinazzi, Robert Gerstenberger, Andrey Ivanov, Yishai Oltchik, and Torsten Hoefler. 2019. Slim graph: Practical lossy graph compression for approximate graph processing, storage, and analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–25.

[3] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience* 34, 8 (2004), 711–726.

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[5] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[6] Christian Borgs, Michael Brautbar, Jennifer 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*. SIAM, 946–957.

[7] Bridge-2 system. 2020. https://www.psc.edu/resources/bridges-2/. (Accessed on 12/22/2021).

[8] CAIDA UCSD. 2008. The CAIDA UCSD Macroscopic Skitter Topology Dataset. https://www.caida.org/catalog/software/skitter [Online; accessed 24-January-2022].

[9] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[10] Domenico Ficara, Andrea Di Pietro, Stefano Giordano, Gregorio Procissi, and Fabio Vitucci. 2010. Enhancing counting Bloom filters through Huffman-coded multilayer structures. *IEEE/ACM Transactions On Networking* 18, 6 (2010), 1977–1987.

[11] James D Foley, Foley Dan Van, Andries Van Dam, Steven K Feiner, John F Hughes, and J Hughes. 1996. *Computer graphics: principles and practice*. Vol. 12110. Addison-Wesley Professional.

[12] Sayan Ghosh, Nathan R Tallent, Marco Minutoli, Mahantesh Halappanavar, Ramesh Peri, and Ananth Kalyanaraman. 2021. Single-node partitioned-memory for huge graph analytics: cost and performance trade-offs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[13] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[14] David Kempe, Jon 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*. 137–146.

[15] Christoph Lameter. 2013. An overview of non-uniform memory access. *Commun. ACM* 56, 9 (2013), 59–54.

[16] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. 2007. The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)* 1, 1 (2007), 5–es.

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

[18] Jiesong Liu, Feng Zhang, Hourun Li, Dalin Wang, Weitao Wan, Xiaokun Fang, Jidong Zhai, and Xiaoyong Du. 2022. Exploring Query Processing on CPU-GPU Integrated Edge Device. *IEEE Transactions on Parallel and Distributed Systems* (2022).

[19] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. 2015. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 363–374.

[20] Norbert Martínez-Bazan, M Ángel Águila-Lorente, Victor Muntés-Mulero, David Dominguez-Sal, Sergio Gómez-Villamor, and Josep-L Larriba-Pey. 2012. Efficient graph management based on bitmap indices. In *Proceedings of the 16th International Database Engineering & Applications Sysmposium*. 110–119.

[21] 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.

[22] Marco Minutoli, Mahantesh Halappanavar, Ananth Kalyanaraman, Arun Sathanur, Ryan Mcclure, and Jason McDermott. 2019. Fast and scalable implementations of influence maximization algorithms. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–12.

[23] Marco Minutoli, Prathyush Sambaturu, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyananaraman, and Anil Vullikanti. 2020. Preempt: scalable epidemic interventions using submodular optimization on multi-GPU systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[24] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. 29–42.

[25] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. 2019. Efficient distributed graph analytics using triply compressed sparse format. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–11.

[26] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. 1978. An analysis of approximations for maximizing submodular set functions—I. *Mathematical programming* 14, 1 (1978), 265–294.

[27] Zaifeng Pan, Feng Zhang, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. Exploring data analytics without decompression on embedded GPU systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 7 (2021), 1553–1568.

[28] Keith H Randall, Raymie Stata, Rajiv G Wickremesinghe, and Janet L Wiener. 2002. The link database: Fast access to graphs of the web. In *Proceedings DCC 2002. Data Compression Conference*. IEEE, 122–131.

[29] Ripples. 2022. https://github.com/pnnl/ripples. (Accessed on 4/20/2022).

[30] Mark A Roth and Scott J Van Horn. 1993. Database compression. *ACM Sigmod Record* 22, 3 (1993), 31–39.

[31] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[32] Janne Suontausta and Jilei Tian. 2003. Low memory decision tree method for text-to-phoneme mapping. In *2003 IEEE Workshop on Automatic Speech Recognition and Understanding (IEEE Cat. No. 03EX721)*. IEEE, 135–140.

[33] Lubos Takac and Michal Zabovsky. 2012. Data analysis in public social networks. In *International scientific conference and international workshop present day trends of innovations*, Vol. 1.

[34] 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*. 1539–1554.

[35] Vijay V Vazirani. 2001. *Approximation algorithms*. Vol. 1. Springer.

[36] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 993–1008.

[37] William Webber, Alistair Moffat, and Justin Zobel. 2010. A similarity measure for indefinite rankings. *ACM Transactions on Information Systems (TOIS)* 28, 4 (2010), 1–38.

[38] Jaewon Yang and Jure Leskovec. 2011. Patterns of temporal variation in online media. In *Proceedings of the fourth ACM international conference on Web search and data mining*. 177–186.

[39] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[40] Feng Zhang, Zaifeng Pan, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. G-TADOC: Enabling efficient GPU-based text analytics without decompression. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1679–1690.

[41] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. 2022. CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases. In *Proceedings of the 2022 International Conference on Management of Data*. 1655–1669.

[42] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text analytics directly on compression. *The VLDB Journal* 30, 2 (2021), 163–188.

A APPENDIX: ARTIFACT DESCRIPTION/EVALUATION

A.1 Experimental Environment

(1) OS: Linux Ubuntu (≥ 18.04)
(2) Compiler: GCC (≥7.4.0)
(3) OpenMP (≥4.5)
  (a) For users in HPC systems (such as Summit) with Slurm, please try "module load gcc/7.4.0" to load the gcc compiler.
  (b) For users in Chameleon Cloud, please request a node in the user dashboard, create an instance using *CC-Ubuntu18.04* image, and launch and login to the instance.

A.2 Step 1: Download Dependencies

A.2.1 Install Python3, CMake, Git, Wget.

```
sudo apt-get update && \
sudo apt-get install -y --no-install-recommends \
     build-essential cmake git wget \
     python3.6 python3-dev \
     python3-pip python3-setuptools gcc
```

A.2.2 Install Conan.

```
pip3 install --no-cache-dir --upgrade pip
pip3 install --no-cache-dir conan==1.51.0
export PATH=$PATH:~/.local/bin
```

A.3 Step 2: Download HBMax and example data

```
HBMAX_ROOT=$(pwd)
git clone https://github.com/hipdac-lab/hbmax-pact
cd $HBMAX_ROOT/hbmax-pact/test-data
wget https://eecs.wsu.edu/~dtao/data/dblp.txt
```

A.4 Step 3: Build HBMax

```
cd $HBMAX_ROOT/hbmax-pact
conan create conan/waf-generator user/stable
conan create conan/trng user/stable
conan install .
./waf configure build_release
```

A.5 Step 4: Test HBMax

A.5.1 Download other dataset (optional).
You can download and extract the example dataset (i.e., pokec) from SNAP with the following commands.

```
cd $HBMAX_ROOT/hbmax-pact/test-data
wget https://snap.stanford.edu/data/\
    soc-pokec-relationships.txt.gz
gunzip soc-pokec-relationships.txt.gz
mv soc-pokec-relationships.txt pokec.txt
```

A.5.2 Set environment variables.

```
export DATADIR=$HBMAX_ROOT/hbmax-pact/test-data
export EXECDIR=$HBMAX_ROOT/hbmax-pact/build/release/tools
```

A.5.3 Run test.
Compare the performance of HBMax and Ripples. Note that you may need to change the number of openMP threads for scalability tests. Other parameters that are changeable are target number of seeds k; approximation error e.

```
cd $HBMAX_ROOT/hbmax-pact/
export task=dblp
export ncpus=8
export OMP_NUM_THREADS=$ncpus
$EXECDIR/imm -i $DATADIR/$task.txt -p \
            -k 100 -d IC -e 0.2 -q 6 \
          >> new_${task}_${ncpus}.txt
$EXECDIR/oimm -i $DATADIR/$task.txt -p \
             -k 100 -d IC -e 0.2 -q 1 \
         >> old_${task}_${ncpus}.txt
echo "======== begin "$task"-hbmax \
     with "$ncpus" threads ============="
  >> /tmp/result
grep 'IMM Parallel :' new_${task}_${ncpus}.txt
    | awk '{print "hbmax   using:" $8}' >> /tmp/result
grep 'IMM Parallel :' old_${task}_${ncpus}.txt
    | awk '{print "ripples using:" $8}' >> /tmp/result
echo "======== finish "$task"-hbmax \
      ================" >> /tmp/result
cat /tmp/result
rm new_${task}_${ncpus}.txt \
   old_${task}_${ncpus}.txt \
   /tmp/result
```

A.5.4 Test Results.
This demo code reads the DBLP graph and uses 8 openMP threads for sampling. It (1) compresses the intermediate RRRs with Huffman Coding (because the skewness is 11.46), (2) selects most influential seeds, (3) runs HBMax/Ripples on the DBLP graph, and (4) show the results with comparison.

You'll expect to see the output like this:
======== begin dblp-hbmax with 8 threads =============
hbmax using: 2093.158376ms
ripples using: 26777.702168ms
======== finish dblp-hbmax =============