Accurate and Fast Approximate Graph Pattern Mining at Scale

Anna Arpaci-Dusseau
Massachusetts Institute of Technology
Cambridge, MA, USA
annaad@mit.edu

Zixiang Zhou
Massachusetts Institute of Technology
Cambridge, MA, USA
zpeter@mit.edu

Xuhao Chen
Massachusetts Institute of Technology
Cambridge, MA, USA
cxh@mit.edu

ABSTRACT

Approximate graph pattern mining (A-GPM) is an important data analysis tool for numerous graph-based applications. There exist sampling-based A-GPM systems to provide automation and generalization over a wide variety of use cases. Despite improved usability, there are two major obstacles that prevent existing A-GPM systems being adopted in practice. First, the termination mechanism that decides when to terminate sampling lacks theoretical backup on confidence, and performs significantly unstable and thus slow in practice. Second, they particularly suffer poor performance when dealing with the "needle-in-the-hay" cases, because a huge number of samples are required to converge, given the extremely low hit rate of their lazy-pruning strategy and fixed sampling schemes.

We build ScaleGPM, an accurate and fast A-GPM system that removes the two obstacles. First, we propose a novel on-the-fly convergence detection mechanism to achieve stable termination and provide theoretical guarantee on the confidence, with negligible online overhead. Second, we propose two techniques to deal with the "needle-in-the-hay" problem, *eager-verify* and *hybrid sampling*. Our eager-verify method drastically improves sampling hit rate by pruning unpromising candidates as early as possible. Hybrid sampling further improves performance by automatically choosing the better scheme between fine-grained and coarse-grained sampling schemes. Experiments show that our online convergence detection mechanism can precisely detect convergence, and results in stable and rapid termination with theoretically guaranteed confidence. We also show the effectiveness of eager-verify in improving the hit rate, and the scheme-selection mechanism in correctly choosing the better scheme for various cases. Overall, ScaleGPM achieves an `geomean` average of 565× (up to 610169×) speedup over the state-of-the-art A-GPM system, Arya. In particular, ScaleGPM handles billion-scale graphs in seconds, where existing systems either run out of memory or fail to complete in hours.

1 INTRODUCTION

Graph Pattern Mining (GPM) [26, 29, 36, 45, 60, 66, 70, 71], which searches for instances of small patterns (e.g., triangles) in a data graph, is a key building block in graph-based data mining. Despite its wide adoption in real-world applications, GPM is hard to scale to large problem size as it is computationally quite expensive [70]. Fortunately, many real-world use cases do not require exact GPM solutions. For instance, to characterize network structures in biochemistry [54], we just estimate the pattern frequency distribution and use it as a signature for the graph. Therefore we only need Approximate GPM (A-GPM), which trades accuracy for speed, as long as some error bound is met in a given application context.

There have been A-GPM systems, e.g., ASAP [34] and Arya [78], that are built to simplify programming and improve usability. The system responsibility is two fold. First, they provide *generalized* sampling techniques for arbitrary patterns and programming APIs to support a wide variety of use cases. Second, they provide automated mechanisms to determine key sampling parameters, e.g., how many samples required to draw for a given error bound.

However, existing A-GPM systems have two major limitations that prevent them from being adopted in practice. First, the automated termination mechanism to determine when it is confident enough to terminate sampling, does not have strong theoretical backup on confidence, and thus is significantly unstable and slow in practice (see §3.1 for detail). In existing systems, sampling is terminated when the predicted number of samples N_s have been drawn. They predict N_s based on an offline error-latency profiling (ELP) procedure before launching the sampling. However, the ELP prediction of N_s is dependent on the true count, which is supposed to be estimated on the N_s samples, creating a circular dependency. Therefore, ELP cannot theoretically guarantee confidence in bounding the error. Also, ELP returns *unstable* predictions, i.e., huge variances or even failures in predicting N_s, which leads to poor performance.

The second limitation is the poor performance when dealing with the extremely sparse, a.k.a, *needle-in-the-hay* cases, where only a small number of matches of the pattern exist in a big graph (see Fig. 2). In this case, neighbor sampling (`NS`), the sampling scheme used in ASAP and Arya, fails to hit a match in most of the samples, leading to a very low sampling hit rate, e.g., 0.0000017%. This results in slow convergence as a huge number of samples are required to meet the error bound. In some extreme cases, we observe that Arya can be even slower than exact GPM solutions.

To address the unstable termination problem, we propose a novel on-the-fly convergence detection method for `NS`. Instead of offline predicting the termination condition before execution, our *online* method dynamically collects statistics and predicts errors during the sampling execution, until convergence. The convergence is detected when the predicted error is below the user specified error bound. We prove formally in Theorem 1 that the probability of

the true error being smaller than our predicted error is $1 - \delta$, for a given confidence $1 - \delta$. This provides us theoretical guarantee in confidence, which prior systems lack. Meanwhile, our detected termination points are stable across different runs while the ELP method in prior systems is highly unstable. Therefore, our online method can significantly accelerate the execution, because it needs much fewer samples than prior systems, while causing negligible online overhead as statistics can be trivially collected.

As for the low hit rate problem in NS, we find that the root cause is the delayed check on pattern closure in prior systems, which we call lazy-verify. We then introduce *eager-verify* that applying pruning at its earliest possible point to avoid unpromising candidates and thus improve hit rates. We prove theoretically that eager-verify is unbiased, and show that it drastically improves the sampling hit rate, which in turn significantly accelerate convergence. Furthermore, for extremely sparse cases where the fine-grained NS sampling scheme can not handle, we propose a *hybrid sampling* method by adaptively selecting the better sampling scheme, between NS and a coarse-grained scheme, graph sparsification (GS), for various graphs and patterns. For scheme selection, we build a cost model for each sampling scheme to estimate the execution time. With our models, hybrid sampling manages to always select the cheaper one from NS and GS for a wide variety of test cases.

We then build SCALEGPM, an accurate and fast A-GPM system that incorporates our proposed mechanisms: online convergence detection, eager-verify and hybrid sampling. SCALEGPM efficiently leverages parallel hardware and provides flexible modes to meet various accuracy requirements with confidence. Our experiments on a multicore CPU show that, with orders-of-magnitude reduction in required samples and improvement in hit rates, SCALEGPM achieves an average of 565× (up to 610169×) speedup over the state-of-the-art A-GPM system, Arya. The hybrid method further improves performance by 61× on extremely hard (i.e. sparse) cases. Particularly, SCALEGPM handles billion-scale graphs in seconds where previous frameworks either run out of memory or fail to complete in hours. This paper makes the following contributions:

- We conduct analysis and empirical study on sampling schemes, and identify the two major limitations and their root causes in existing A-GPM systems.
- We propose a novel on-the-fly convergence detection method for the NS sampling scheme, which is the first to provide theoretical guarantee on confidence.
- We introduce the eager-verify mechanism to improve hit rate of the NS scheme and thus achieve faster convergence speed.
- We propose hybrid sampling to further improve performance, by adaptively selecting a better scheme based on cost models.
- We build SCALEGPM that incorporates the above novel mechanisms, and evaluation shows that it significantly outperforms the state-of-the-art system, and efficiently handles huge graphs.

2 BACKGROUND

2.1 Approximate Graph Pattern Mining

Graph Pattern Mining (GPM) finds subgraphs that match given pattern(s) $\mathcal{P}$ in a given data graph $\mathcal{G}$. There exist *explicit* GPM tasks like subgraph counting (SC) and motif counting (MC), and *implicit* tasks like frequent subgraph mining (FSM) [27]. GPM has

---

Algorithm 1 Neighbor Sampling

1: **for each** sampler $i \in [1, N_s]$ **do**
2: $\quad C_i \leftarrow 0, \alpha \leftarrow m$
3: $\quad$ Sample an edge e_1 from $\mathcal{E}$, let $E_1 \leftarrow \{e_1\}$
4: $\quad$ **for** j in $[2, k - 1]$ **do**
5: $\quad\quad$ Sample a neighboring edge e_j from E_{j-1}'s neighborhood
6: $\quad\quad E_j \leftarrow E_{j-1} \bigcup \{e_j\}, \alpha \leftarrow \alpha \times c_j$ $\quad \triangleright c_j$ is the neighborhood size
7: $\quad$ **if** closing edges for $(E_{k-1}, \mathcal{P})$ exist in $\mathcal{G}$ **then** $C_i \leftarrow \alpha$ $\quad \triangleright$ closure check
8: $C' = \sum C_i / N_s$ is the estimated count

---

numerous applications in AI and big-data [70], including bioinformatics, chemical engineering, fraud detection, social network analysis, recommender systems.

Exact GPM is solved by enumerating subgraphs in the data graph and searching for matches. The search space can be defined as a *subgraph tree*: each node in the tree is a subgraph of the data graph $\mathcal{G}$. A subgraph H in level l of the tree have l vertices. The root (level 0) is an empty subgraph. A parent node H at level i can be *extended* to a child node H', by adding a vertex/edge in H's neighborhood in $\mathcal{G}$, i.e., $H' = H + \{v\}$, $v \in \mathcal{N}(H)$. Each leaf of the tree is a candidate of match, which is then compared with the pattern $\mathcal{P}$, to test if it is a match. Pruning schemes, e.g., matching order [36, 45, 70], symmetry breaking [26, 27, 44] and decomposition [24, 52], are applied to reduce the search space (i.e., prune the subgraph tree). Nevertheless, this search is extremely expensive, as the computational complexity increases exponentially in the size of the pattern.

Many real-world use cases do not require exact GPM solutions. For example, when we use motif (a.k.a graphlet) distribution as a "signature" (e.g., graph similarity) for network analysis or fraud detection, it is quite sufficient to just provide approximate counts of the motifs. Also in FSM the users only want to find those *frequent* patterns whose occurrences are above some user specified threshold, or simply top-K most frequent patterns, where estimated counts would be sufficient to give high quality solutions. Therefore, for all these use cases, we can perform Approximate GPM (A-GPM) instead to substantially reduce the total amount of computation.

In this paper we focus on sampling based A-GPM approaches. Generally, such an approach first samples a portion of the graph, searches for match in the sample, and makes the estimation by scaling the sampled result. This process can be repeated multiple times to improve the confidence of estimation. Formally, given $\mathcal{G}$ and $\mathcal{P}$, an A-GPM solver aims to use a randomized (ϵ, δ)-approximation scheme, which estimates the number of (non-induced or induced) occurrences of $\mathcal{P}$ in $\mathcal{G}$ within a factor of $(1 \pm \epsilon)$, with probability at least $1 - \delta$, where ϵ and δ are user defined parameters.

There are a large volume of A-GPM applications [31], such as triangle counting [2, 12, 21, 25, 49, 51, 67–69], clique/cycle counting [25, 59, 75], motif counting [5, 14, 18–20, 30, 37, 50, 61, 72, 76], butterfly counting [57], frequent subgraph mining [1, 13, 40, 53, 55]. They all use sampling to reduce computation, though their sampling schemes are *customized* for the specific problems. These custom implementations do not offer system support, like automated termination, generic APIs, or choices in speed and error trade-off.

2.2 Sampling Schemes for GPM Problems

2.2.1 Neighbor Sampling (NS). Algorithm 1 shows how neighbor sampling works. For each sample, it starts with sampling one edge from $\mathcal{G}$ uniformly at random (Line 3), and then repetitively samples

Algorithm 2 Edge Sparsification

1: Randomly select a subset E' of $p \times m$ edges from E, $m = |E|$, $0 < p \le 1$
2: Generate an induced subgraph $G' = (V, E')$
3: $C' \mathrel{+}= \textsc{ExactCounting}(G', \mathcal{P})$ ▷ Exact counting on G'
4: $C = C' \times p^{-l}$ is the estimated count in G ▷ l is # of edges in $\mathcal{P}$

---

one more edge from the neighborhood of the currently sampled edges (Line 5), until the size (number of sampled vertices) is the same as the pattern. It then does *closure check* (Line 7), i.e., check in G the existence of the *closing edges*, which form a match of the pattern together with existing edges. We can draw multiple samples (Line 1) in parallel, and average them for improved accuracy (Line 8).

For a given match $\hat{M}$ where $\hat{e}_1$ is the first edge, $Pr[\hat{M}] = \frac{1}{m \cdot \prod_2^{k-1} c_j}$, where $c_j = |\mathcal{N}(H_{j-1})|$, H_{j-1} is the E_{j-1} induced subgraph, and $\mathcal{N}(H)$ is the neighbor set of H. So the count is scaled by $\alpha = m \cdot \prod_2^{k-1} c_j$ (Line 6). Apparently α varies in different samples, meaning matches are not equally likely to be sampled. Thus α is maintained for each sample and used for normalization.

NS has been widely used in A-GPM applications [17, 50, 51, 69] with customized optimizations. In NS, each time an arbitrary neighbor is sampled from H_i's neighborhood, which may lead to a high failure rate in the closure check. Therefore, restrictions are added to make the sampled subgraph more likely be a match. For example, if $\mathcal{P}$ is a cycle, we can sample a path [37], i.e., restrict the next neighbor to be from the two endpoints' neighborhood. For an arbitrary pattern $\mathcal{P}$, we can do neighbor sampling following $\mathcal{P}$'s spanning tree [19]. Furthermore, automorphism check can be added to avoid redundant subgraphs [38]. More generally, we can sample multiple neighbors each time, instead of a single neighbor [38].

2.2.2 Subgraph Sampling. The idea is to sample a subset of vertices or edges from G with some probability p, to form a subgraph G', and then do exact search in G', and finally scale the count based on p. One popular way to sample a subgraph is to use the *graph sparsification* (GS) technique [10, 32, 63–65], which sparsifies G by randomly removing some edges. Bernoulli Edge Sparsification (BES) [68] is such an example, as shown in Algorithm 2. For each edge in G, include it with probability p (Line 1) to get graph G' (Line 2). For each match M of $\mathcal{P}$ in G, the probability that M exists in G' is p^l, hence the expected count in G' is $C' = p^l \times C$. BES is simple and easy to implement, and can be trivially parallelized.

Color Sparsification [49] (CS), another GS approach, sparsifies G by first randomly assigning a color from $\{1, 2, \cdots, c\}$ to each vertex and then only preserving edges whose two endpoints are in the same color. The probability of choosing an edge is $p = 1/c$, and the probability of choosing a match is p^{l-1}. Hence the expected count in G' is $C' = p^{l-1} \times C$. Apparently, the probability of preserving a match of $\mathcal{P}$ is higher than that in BES. In other words, it could meet the same error bound with more edges removed, thus less computation. The downside is that different edges are not independent of each other any more, which means potentially higher variance. Usually a single sampler is used for sparsification, though more samplers can be used to improve confidence.

Another way to sample a subgraph is Egonet Sampling [56, 57], in which an element (vertex or edge) of G is sampled and the *egonet*, i.e., the local neighborhood, of this element is extracted as a subgraph. It often requires many samplers.
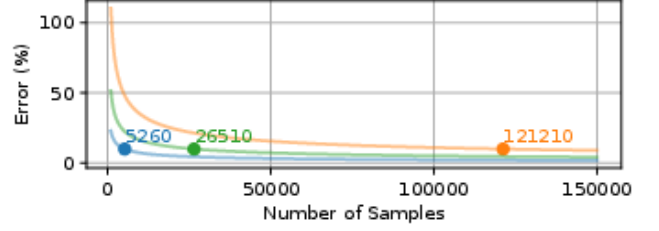

Figure 1: Three different runs (three curves) of Arya's ELP prediction given the `LiveJ` graph and `triangle` pattern. With an error bound of 10%, the curves give dramatically different prediction on the number of samples N_s: 5,260, 26,510 and 121,210. This leads to a 25× performance difference in the sampling execution phase.

In subgraph sampling we can apply a state-of-the-art exact counting algorithm on the sampled subgraph. But the cost is extra time and space to extract the subgraph(s), and the time complexity is still exponential in the pattern size. Subgraph sampling has only been used in A-GPM applications for specific patterns [49, 56–59, 67, 68].

2.3 Approximate GPM Systems

A-GPM systems, such as ASAP [34] and Arya [78], have been proposed to simplify A-GPM programming. These systems provide APIs to the users for them to easily compose various A-GPM applications. As opposed to those case-by-case customized implementations, an A-GPM system provides a generalized, sampling-based approximation method, for arbitrary patterns. Moreover, unlike hand-implemented applications which require hand tuning the key sampling parameters, these systems automatically pick the sampling parameters to meet the user specified error bound. For instance, given ϵ and δ, ASAP and Arya use the Error-Latency Profile (ELP) method to determine N_s, the number of samples needed. N_s is derived from the Chernoff bound in ASAP [34] or Chebyshev's inequality in Arya [78]. For example, the lower bound in Arya is

$$N_s \ge \frac{K \cdot m^\rho}{C \cdot \epsilon^2 \delta} \tag{1}$$

where K is a constant, $m = |E|$, ρ is a pattern specific constant, and C is the true count. ELP then estimates K and C on a sparsified subgraph G' of G. This estimation is repeated for several rounds until convergence (i.e., estimated counts occur within a small range). With estimated K and C, ELP returns a predicted N_s. Note that ELP has no error-confidence bound on the estimate of K and C.

Nevertheless, all the prior systems use fixed sampling schemes to make approximation. For example, ASAP uses the NS scheme, in the *edge streaming* fashion [4, 8, 9, 23, 39] where edges are streamed in, instead of loaded all at once, to save memory space. Arya is also based on NS, but incorporates *pattern decomposition* to reduce the amount of work in each sample for large, easy-to-decompose patterns. Since both ASAP and Arya are based on NS, they both suffer the shortcomings of NS, which are discussed in detail next.

3 UNDERSTANDING SAMPLING TRADEOFFS

3.1 Termination Condition and Confidence

A major responsibility of an A-GPM system is to decide when the sampling can be terminated with confidence to meet the error bound. Existing systems use error-latency profile (ELP), before the
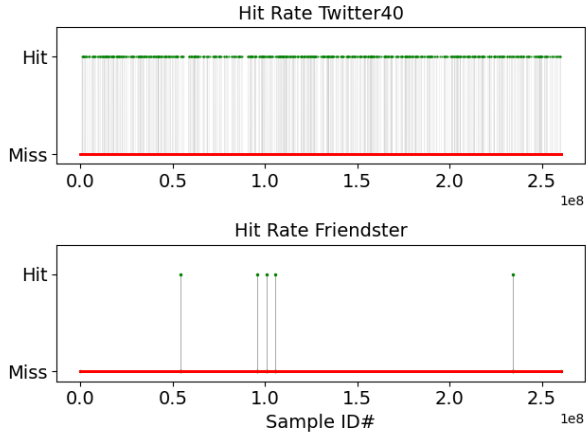
Figure 2: Sample hits and misses in Arya, on `Twitter40` (top) and `Friendster` (bottom) graphs. The pattern is `4-clique` for both. In total 10^8 samples are drawn in both cases. Each green point is a hit sample, while each red point is a miss sample. For `Twitter40`, there are 7,033 hits with a hit rate of 7×10^{-5}. For `Friendster`, there are only 5 hits with a 5×10^{-8} hit rate (i.e. needle in the hay).

execution of the sampling procedure, to pre-determine the number of samples N_s required. The sampling is then terminated when N_s samples have been drawn. However, we observe that the value of N_s that ELP predicts vary dramatically across different runs of ELP. Fig. 1 shows the results of three runs of ELP. Each curve represents the prediction of one run. We get predictions of N_s as 5,260, 26,510 and 121,210, respectively. This huge prediction difference leads to a 25× performance difference in the sampling execution phase.

More importantly, the ELP method for termination condition adds an indirection between the error estimation and the confidence. The problem is, the lower bound in Eq. (1) contains the true count C, which is the output that is supposed to be estimated. Therefore, what ELP does is to essentially first estimate C without theoretical error-confidence (ϵ-δ) guarantees, and then feed it into the lower bound to get N_s. N_s is then used to do sampling and estimate a more accurate C. Although Eq. (1) contains δ, estimating C by ELP *has no involvement of δ*. It means the estimation of N_s loses connection to the confidence. The root cause is the *circular dependency* between C and N_s, i.e., C is used to estimate N_s, and N_s is used to estimate C, which is fundamentally unavoidable in the ELP based approach.

Due to the *circular dependency*, ELP provides only a heuristic rather than a strong theoretical bound. Another limitation of ELP is its own convergence speed. We observe that in some cases, ELP fails to converge within 10 hours (see details in Table 2).

3.2 Characterizing Neighbor Sampling

For the sampling based approximation approach, the estimation difficulty depends on the density and distribution of the matches of $\mathcal{P}$ in the graph $\mathcal{G}$. When there are plenty of matches, defined as the *dense case*, it is easier to make estimation than the *sparse case*, where there exist only a few matches. This is because it is more likely to draw a *successful* sample (i.e., find a match) if there are more matches, and the confidence to meet an error bound depends on seeing enough number of successful samples. Note that the extremely sparse case is known as finding the *needle in the hay*.
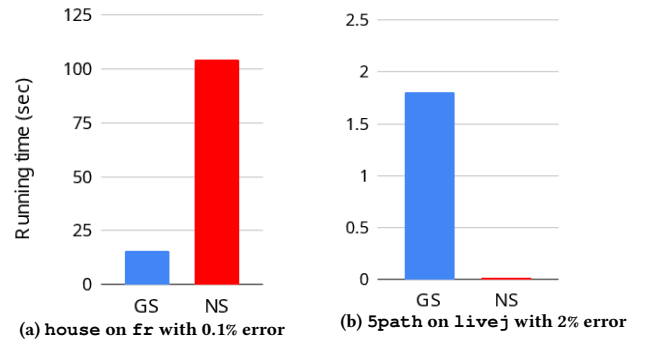

(a) `house` on `fr` with 0.1% error   (b) `5path` on `livej` with 2% error

Figure 3: Execution time variance of Neighbor Sampling (`NS`) and Graph Sparsification (`GS`), under the same error bounds.

We call a sample that finds a match a *hit*, otherwise a *miss*. Fig. 2 shows how the hits and misses are distributed in 1×10^8 samples drawn by Arya [78], for the `4-clique` pattern on graphs `Twitter` (top) and `Friendster` (bottom). The two graphs have similar sizes but quite different degree distribution (see maximum degree in Table 1), and thus different hit rates. There are 7,033 hits in `Twitter` (7×10^{-5} hit rate), while only 5 hits appeared in `Friendster` (5×10^{-8} hit rate) which is a typical needle-in-the-hay case. Since the execution time is roughly linear in N_s, this difference in hit rates would result in a ~700× execution time difference in practice, assuming the same average time per sample.

It is known that `NS` works poorly in the sparse case, for example, when $\mathcal{P}$ is dense and $\mathcal{G}$ is sparse, and particularly in the case of needle in the hay [78]. One can expect that in this case, the closure check fails frequently, which means a large number of samples N_s is required to meet a certain error bound. Moreover, dense patterns are particularly problematic for Arya [78] which decomposes the pattern into sub-patterns, as a dense pattern would be split with many edgecuts, and checking closure is quite expensive. Arya thus suffers significant slowdown in those cases, and in some extreme cases it is even slower than the exact solution.

Note that finding "needle-in-the-hay" patterns is very important. First, A low hit-rate does not mean a small overall count. For instance, even though `Friendster-4clique` has only a 5×10^{-8} hit rate using Arya, there are 8.96 billion instances in total. Second, GPM tasks like clique and motif counting, where "needle-in-the-hay" counting is required, are the most important bases in network analysis applications, such as community detection and dense subgraph discovery [11, 35, 47, 55, 75]. Exact solutions can still take hours to count in these cases (see Table 2 and Table 4a). Lastly, for applications like anomaly detection, a fast and accurate measure of rare patterns is especially useful [3, 48].

3.3 Coarse-grain vs. Fine-grain Sampling

A key difference between `NS` and `GS` is the granularity of each sample. Since each sample contains at most one match, we classify `NS` to be a *fine-grain* sampling scheme, as opposite to *coarse-grain* sampling schemes, each of whose samples can contain multiple matches. For `NS`, each sample is in the size of the pattern size k, and each sample task is lightweight. But we need a lot of samples, i.e., $N_s \gg 1$, to get a meaningful estimation, since each sample contains at most one match. In contrast, subgraph sampling schemes,

including GS, are coarse grained. In GS, a sample is a sparsified graph, which could potentially contain many matches in it. As the sample granularity in GS is much larger, it performs better than NS when given a sparse case, as it is more likely to hit matches in a big region of the graph. However, this advantage comes at the cost of several drawbacks. First, each sample in GS is a much larger computational task, and the complexity is exponential in the pattern size. Second, as multiple matches appear in the same sample, GS may yield worse variance than NS in the worst case.

To summarize, given the distinct characteristics in the input data (graph and pattern), none of these sampling schemes can always be the best solution. Fig. 3 compares the running time of NS with GS, when given the same error bound. On the left we mine the `house` pattern on the `Friendster` graph with an error bound of 0.1%, where GS is 6.7× **faster** than NS. On the right we mine the `5-path` pattern in the `Livej` graph with an error bound of 2%, where GS is 10× **slower** than NS.

4 PROPOSED MECHANISMS

To achieve stable termination with confidence, we propose a novel on-the-fly convergence detection mechanism in §4.1 that is fundamentally different from the existing ELP approach. To improve hit rate in NS, we introduce eager-verify and prove it unbiased in §4.2. To further improve performance in handling needle-in-the-hay cases, we propose a hybrid method that adaptively selects the best-performing sampling scheme. For scheme selection, we establish cost models (a.k.a. performance models) for the NS (§4.3) and GS (§4.4) scheme to estimate their execution time, and select the faster one. We only focus on the two schemes, but this hybrid method can be extended to include other schemes in the future.

4.1 Online Convergence Detection

Due to circular dependency (§3.1), the ELP method breaks the theoretical guarantee on confidence. Also, it is fundamentally hard for ELP to establish confidence because it is done before execution, and there is little information we can leverage. Therefore, we propose an on-the-fly approach to establish confidence. This is based on our observation in NS sampling procedures, that the estimation errors tend to converge over time (see Fig. 10). Instead of predetermining the required number of samples offline (i.e. before execution), we detect online if the estimates have converged, and then terminate the execution subsequently. However, convergence detection is not trivial. A straightforward method is to check if the difference of the error curve is small enough, i.e., within a fixed threshold. But this does not work because the termination point depends on the user defined confidence and error bound. The key challenge is then how to decide termination to meet error bound with confidence.

To address it, we propose to predict the error online periodically with confidence, and terminate when the predicted error is below the error bound. To make predictions with confidence, we collect online statistics during execution, which allows us to formally derive predicted errors based on the probability theory. Our key insight is that confidence can be established by the normal distribution of sampled means (formally proved in Theorem 1), as shown in Fig. 4. Specifically, we keep track of the *mean* μ of the estimated counts and the *standard deviation* σ of the means, and then use μ, σ and δ
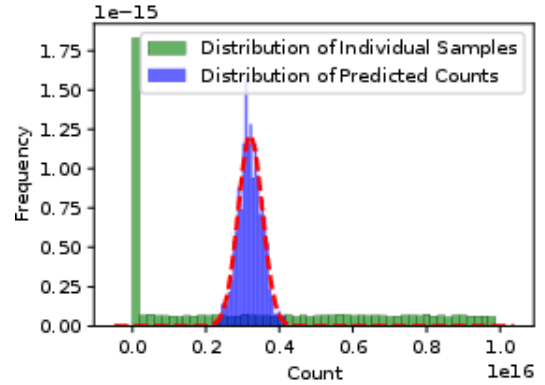

Figure 4: The normal distribution of the means of sampled counts (i.e. our predicted counts) using neighbor sampling (NS). We ran NS to collect 10^6 samples on LiveJ, 4-clique. We obtained a predicted count by taking the mean of a random subset of 100 of these underlying samples. We simulated 1000 of these predicted counts. Although the underlying distribution of the sampled counts (green bars) is not a normal distribution, their means (purple bars), which are our predicted counts, do follow a normal distribution (dashed red line).

to compute a relative error $\hat{\epsilon}$ using Eq. (2), where Φ^{-1} is inverse of the cumulative distribution function of the standard normal. Note that μ is also used as the estimate of the true count C.

$$\hat{\epsilon} = \frac{\Phi^{-1}(1 - \frac{\delta}{2})\sigma}{\mu} \tag{2}$$

When we detect that $\hat{\epsilon}$ is below the user specified error bound ϵ, according to Theorem 1, we can safely terminate the sampling, and conclude that μ is an estimate of C, under an error bound ϵ with a confidence of $1 - \delta$. We refer our approach as NS-online.

THEOREM 1. *Given δ, n samples $X_1, \ldots, X_n$ drawn by using the NS sampling scheme, and the mean of sampled counts $\mu = \frac{1}{n}\sum_{i=1}^{n} X_i$, let C be the true count and $\hat{\epsilon}$ be the estimated error computed by Eq. (2). As $n \to \infty$, the probability of the true relative error being smaller than the estimated relative error is $\mathbb{P}\left(\frac{|\mu - C|}{C} < \hat{\epsilon}\right) = 1 - \delta$.*

PROOF. Let D be the distribution $X_1, \ldots, X_n$ are sampled from. Since the NS estimator is unbiased, the true count C is the mean of the distribution D. Our estimator of C is $\mu := \frac{1}{n}\sum_{i=1}^{n} X_i$. This estimator satisfies $\mathbb{E}[\mu] = \mathbb{E}[\frac{1}{n}\sum_{i=1}^{n} X_i] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[X_i] = \frac{1}{n}\sum_{i=1}^{n} C = C$ and $\mathbb{V}ar[\mu] = \mathbb{V}ar[\frac{1}{n}\sum_{i=1}^{n} X_i] = \frac{1}{n^2}\sum_{i=1}^{n}\mathbb{V}ar[X_i] = \frac{1}{n}\mathbb{V}ar[X_1]$. Because μ is the average of n samples from a distribution D (which clearly has finite mean and variance), the central limit theorem (CLT) applies, so μ follows the normal distribution. Formally, $\frac{\mu - \mathbb{E}[\mu]}{\sqrt{\mathbb{V}ar[\mu]}}$ converges in distribution to a standard normal as $n \to \infty$.

By the law of large numbers, as $n \to \infty$, the sample variance $\frac{1}{n}\sum_{i=1}^{n} X_i^2 - \mu^2$ converges in probability to $\mathbb{V}ar[X_1]$. Letting $\sigma^2 := \frac{1}{n}\left(\frac{1}{n}\sum_{i=1}^{n} X_i^2 - \mu^2\right)$, this means that $\frac{\mathbb{V}ar[\mu]}{\sigma^2} = \frac{\frac{1}{n}\mathbb{V}ar[X_1]}{\frac{1}{n}(\frac{1}{n}\sum_{i=1}^{n} X_i^2 - \mu^2)} = \frac{\mathbb{V}ar[X_1]}{(\frac{1}{n}\sum_{i=1}^{n} X_i^2 - \mu^2)}$ converges in probability to 1. Additionally, μ converges in probability to C, so $\frac{\mu}{C}$ converges in probability to 1. Therefore, by Slutsky's Theorem [73], $\frac{\mu - \mathbb{E}[\mu]}{\sqrt{\mathbb{V}ar[\mu]}} \cdot \sqrt{\frac{\mathbb{V}ar[\mu]}{\sigma^2}} \cdot \frac{\mu}{C} = \frac{\mu - C}{C} \cdot \frac{\mu}{\sigma}$ converges in distribution to a standard normal. This means that for

Algorithm 3 NS-online convergence detection

```
1: sum ← 0, squaredSum ← 0, n = 0, W ← N_min
2: while ê > ε do
3:     for each sampler i ∈ [1, W] do          ▷ W is the window size
4:         X_i ← DrawAsample()               ▷ X_i is the i-th sampled count
5:         sum ← sum +X_i                     ▷ ∑_{i=1}^n X_i
6:         squaredSum ← squaredSum +X_i * X_i  ▷ ∑_{i=1}^n X_i^2
7:         n ← n + 1
8:     μ ← sum / n                            ▷ mean of sampled counts
9:     var ← squaredSum / n − μ * μ           ▷ variance of sampled counts
10:    σ ← sqrt(var/n)                        ▷ standard deviation
11:    ê ← Φ^{-1}(1 − δ/2) * σ/μ              ▷ predicted error
```

any fixed x, we have

$$\mathbb{P}\left(\frac{\mu - C}{C} \cdot \frac{\mu}{\sigma} < x\right) \to \Phi(x) \text{ as } n \to \infty.$$

This implies that

$$\mathbb{P}\left(\left|\frac{\mu - C}{C} \cdot \frac{\mu}{\sigma}\right| < x\right) \to 2\Phi(x) - 1 \text{ as } n \to \infty.$$

Plugging in $x = \Phi^{-1}(1 - \frac{\delta}{2})$, we get

$$\mathbb{P}\left(\left|\frac{\mu - C}{C} \cdot \frac{\mu}{\sigma}\right| < \Phi^{-1}(1 - \frac{\delta}{2})\right) \to 2\Phi\left(\Phi^{-1}(1 - \frac{\delta}{2})\right) - 1 \text{ as } n \to \infty.$$

Rearranging terms and simplifying yields

$$\mathbb{P}\left(\left|\frac{\mu - C}{C}\right| < \frac{\Phi^{-1}(1 - \frac{\delta}{2})\sigma}{\mu}\right) \to 2\left(1 - \frac{\delta}{2}\right) - 1 \text{ as } n \to \infty$$

$$\implies \mathbb{P}\left(\left|\frac{\mu - C}{C}\right| < \hat{\epsilon}\right) \to 1 - \delta \text{ as } n \to \infty.$$

□

Algorithm 3 shows the NS-online algorithm. Each time we draw a sample (Line 4), the only information we need to keep track of is the accumulated sum $\sum_{i=1}^n X_i$ (Line 5) and the accumulated squared sum $\sum_{i=1}^n X_i^2$ (Line 6). At the end of each interval (i.e. W samples in Line 3), we compute the standard deviation σ (Line 10) and predict the error $\hat{\epsilon}$ (Line 11). If $\hat{\epsilon}$ is below the user's error bound (Line 2), sampling is then terminated and it reports the estimated count μ.

When Algorithm 3 terminates, we have $\hat{\epsilon} \leq \epsilon$, so $\mathbb{P}\left(\frac{|\mu - C|}{C} < \epsilon\right) \geq \mathbb{P}\left(\frac{|\mu - C|}{C} < \hat{\epsilon}\right) = \hat{\mathbb{P}}$. Theorem 1 rigorously guarantees that $\hat{\mathbb{P}}$ approaches $1 - \delta$ as n approaches ∞. For a finite n, $\hat{\mathbb{P}}$ is close to $1 - \delta$ if the distribution of μ is close to a normal distribution. A common approximation in statistics is to assume the average of a large, finite number of samples is normally distributed, as this is backed by the Central Limit Theorem. In practice, we observed that the number of samples taken before sampling is terminated is much larger than the number of samples required for a normal distribution shape (like Fig. 4) to emerge. Therefore when Algorithm 3 terminates, $\mathbb{P}\left(\frac{|\mu - C|}{C} < \hat{\epsilon}\right) \approx 1 - \delta$, so $\mathbb{P}\left(\frac{|\mu - C|}{C} < \epsilon\right)$ is close to $1 - \delta$.

4.2 Eager Verify for Neighbor Sampling

A major drawback of NS in prior systems is the low hit rate in dealing with sparse cases. By looking at individual samples, we find that most of these samples fail to pass the closure check (Line 7 in Algorithm 1). Therefore we looked deeper into the failures (i.e. missed samples). Our key observation is that many of the failures have been unpromising candidates even at the early stage of the sampling. For example, if we search for 6-cliques, and the first

Algorithm 4 NS-prune for 4-cycle

```
1: for each sampler i ∈ [1, N_s] do
2:     e(v_0, v_1) ← sample(E), α ← 0          ▷ sample an edge (v_0, v_1)
3:     A ← N(v_1) - {v_0, v_1}, bound by v_0    ▷ set difference
4:     if |A| = 0 then break
5:     v_2 ← sample(A)                          ▷ sample node v_2 from set A
6:     B ← N(v_0) & N(v_2), bound by v_1        ▷ set intersection
7:     if |B| = 0 then break
8:     v_3 ← sample(B)                          ▷ sample node v_3 from set B
9:     X_i ← m * |A| * |B|                      ▷ sampled count
```

four vertices in the sample does not form a 4-clique, this sample is impossible to form a 6-clique. Therefore, in general, the low hit rate in ASAP and Arya is because the closure check is delayed to the very last step, which we refer to as *lazy-verify*. Based on this understanding, we propose a *eager-verify* approach to improve NS performance. The key idea is to sample from promising candidates by verifying the pattern's connectivity constraints as early as possible. This strategy has two advantages. First, since unpromising candidates are pruned at early stage, and we only sample from promising candidates, each sample is more likely to succeed, i.e., hit a match. Second, if a sample starts from an edge whose neighborhood contains no or very few matches (unlikely to hit), eager-verify can minimize the work as this sample would fail at its early stage, while lazy-verify will have to proceed until the end and fail.

The challenge in implementing eager-verify is how to avoid unpromising candidates but still retain unbiasedness in sampling. Based on the subgraph tree abstraction (§2.1), our key finding is that each leaf in the tree corresponds to a unique path. As long as the pruning does not change this one-to-one mapping, we can prove that the sampler is unbiased. We find that two typical pruning techniques, symmetry breaking [44] and matching order [70], meet this requirement. There exist other pruning schemes in the literature [24, 46], which could be applied as well, but we leave this as a future work. We refer NS used in ASAP as *NS-base*, and NS with the two pruning techniques as *NS-prune*. We first give an example to show how pruning avoids unpromising candidates, and then prove NS-prune an unbiased estimator in Lemma 2. Algorithm 4 shows the pseudo-code for finding 4-cycle using NS-prune. In Line 6 we compute a set intersection $N(v_0)$ & $N(v_2)$ which is the candidate set of the third vertex v_3, because v_3 is a common neighbor of v_0 and v_2 in the 4-cycle pattern. In contrast, in NS-base, because both v_0's and v_2's neighbors are possible candidates, v_3 is sampled from set union $N(v_0) \cup N(v_2)$, and in the final step it checks closure between v_3 and v_2 if v_3 is from v_0's neighborhood, otherwise it checks closure between v_3 with v_0. If the closure check fails, it is a miss. However, in NS-prune the closure check is unnecessary because v_3 is guaranteed to be connected to v_0 and v_2, and thus much more likely to hit a match.

LEMMA 2. *NS-prune is an unbiased estimator.*

PROOF. Let $(v_0, \ldots, v_{k-1})$ be the vertices of an occurrence, i.e., a *match*, of the pattern $\mathcal{P}$ in the matching order (e.g. for 4-cycle, $(v_0, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_0) \in E$). Each match corresponds to a unique k-tuple $(v_0, \ldots, v_{k-1})$, e.g., for 4-cycle, we enforce $v_0 = \max(v_0, v_1, v_2, v_3)$ and $v_3 < v_1$. During the execution of NS-prune, the probability that v_0 and v_1 are chosen is $1/m$ (see Line 2 in Algorithm 4), as all edges are equally likely. The probability that v_2 is chosen (see Line 5) is $1/|A|$ as $v_2 \in A$. In general, the probability
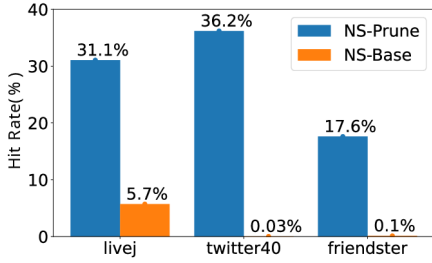
Figure 5: Comparing the hit rate of **NS-prune** with **NS-base**, with the **4-clique** pattern on various graphs.

Figure 6: The execution time for **6clique-Friendster** using Color Sparsification under different numbers of colors. Each point is one run. Red region is the stabilization window.

that v_i is chosen is $1/|S_i|$, where S_i is the candidate set that v_i is drawn from. Therefore the probability that this particular match is sampled by NS-prune is the product of these probabilities (e.g. $1/(m \cdot |A| \cdot |B|)$ for 4-cycle). The scaling factor α is the inverse of this probability, so the expected contribution of this match to the estimated count for one sampler is $\frac{1}{\alpha} \cdot \alpha = 1$. Let C be the true count of $\mathcal{P}$ in $\mathcal{G}$, and let $X_{ij} = \alpha = (m \cdot |S_2| \cdots |S_{k-1}|)$ if the ith sample hits the jth match of $\mathcal{P}$, otherwise $X_{ij} = 0$, where $1 \leq i \leq N_s, 1 \leq j \leq C$. The estimated count is $\mathbb{E}\left(\frac{1}{N_s} \sum_{i,j} X_{ij}\right) = \frac{1}{N_s}\left(\sum_{i,j} \mathbb{E}(X_{ij})\right) = \frac{1}{N_s} \sum_{i,j} 1 = C$. Therefore NS-prune is an unbiased estimator. □

Note that the scaling factor in NS-prune tends to be much smaller than that in NS-base, because it involves the sizes of intersection instead of union sets. This results in lower variances and more stable and faster convergence, as we will show in §6.

4.3 Cost Model for Neighbor Sampling

In NS, the total work is $\sum_{i=1}^{N_s} W_i$, where W_i is the work of the i-th sample s_i. Given that $N_s \gg 1$ in the NS scheme, it is reasonable to assume that the NS execution time is linear in the number of samples N_s. So we can predict the execution time as $t_{tot} = t_{avg} * N_s = c * W_{avg} * N_s$, where W_{avg} is the average work per sample, and c is a hardware specific constant factor to translate work to time. We estimate N_s by profiling §5.3. As this is only used for performance prediction, it does not affect error and confidence. To estimate c, a simple profiler can be run on each hardware machine to determine this constant scale factor. This profiling overhead is negligible, as it can be determined by running only once per machine or once per graph on only a small number of points.

The challenge, however, is to estimate W_{avg} for the given $\mathcal{G}$ and $\mathcal{P}$, as W_i varies for different samples. In fact W_i depends on the local neighborhood structure of s_i. More specifically, the task of each sample is a sequence of set operations and sample operations, for example, see Algorithm 4. Given a specific pattern $\mathcal{P}$, this sequence of operations is fixed, but each set operation in the sequence may take different (worse-case) time depending on the cardinality of the input sets, which overall depends on the structure of $\mathcal{G}$. So it is difficult to get an accurate estimation of W_{avg}, i.e., the slope the linear relationship, without really running NS.

At each break point (e.g., Line 4 in Algorithm 4), we must consider the possibility of an early-exit from the sampling procedure. Each break point is triggered based on the probability of being empty based on the candidate set size. However, it can be hard to predict this probability in practice, as certain graphs
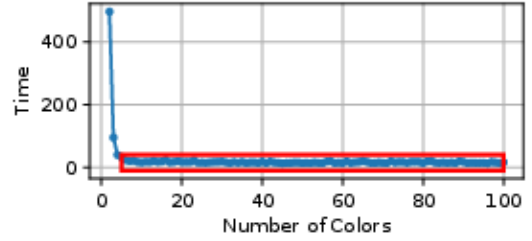
have many dense clustered areas, when sampled in few breakpoints occur, whereas others frequently terminate early. To address this, we model the performance as a *performance cone* (see example in Fig. 13) consisting of two slopes instead of one, where the performance is upper bounded by none of the breakpoints triggering (completing the full work of the sampling procedure) and lower bounded by the first break point.

4.4 Cost Model for Graph Sparsification

GS running time contains two parts, one is the **preprocessing** time to generate the sparsified graph G', the other is the time spent on **exact search** in the sparsified graph G'.

Preprocessing involves looping over every edge in order to remove edges. Therefore, the total work is $|E| \cdot (w_1 + p \times w_2)$, where p is the probability that an edge is kept, w_1 is the work on every edge, w_2 is the work on kept edges. In our implementation w_1 is one read operation, w_2 is one write operation.

For exact search in G', our estimation is again based on set operations. But instead of a sequence of operations in NS, the work in GS consists of nested loops, each of which corresponds to one vertex in the pattern and iterates over the candidates of that vertex. The candidate vertex set is computed by set operations. If we describe a GPM algorithm as a sequence of nested for loops $M = (X_1, \ldots, X_n)$, which refine the candidate set to a possible match. Each nested loop can be described as $X_j = (o_j, i_j)$. Where o_i is the number of operations performed in the inner body of that loop to refine the candidate set. i_j is the number of iterations for the jth loop. $i_j = |Z_i|$, the size of the candidate set at the jth level. o_i is the work of the operations to generate the candidate set $|Z_{i+1}|$. Then the total work can be described as $\prod_j^{i_j} o_j = \prod_j^{|Z_j|} o_j$. To estimate i_j and o_j, we need to estimate the cardinality of each candidate set Z.

If $Z = A - B$, i.e., set difference, $|Z|$ is bounded by $|A|$, which can be estimated as the average degree, $|V|/|E|$. If $Z = A \cap B$, i.e., set intersection, $|Z|$ is bounded by $\min(|A|, |B|)$. To get a better bound, we can use the method in [60] and estimate $|Z|$ as $|V| \cdot p_1 \cdot p_2^{n-1}$, where $p_1 = \frac{2 \cdot |E|}{|V|^2}$, $p_2 = \frac{T \cdot |V|}{2 \cdot |E|^2}$ and T is the triangle count in $\mathcal{G}$. Intuitively, p_1 is the probability of any pair of vertices being neighbors. p_2 is the probability of any pair of vertices in a neighborhood being directly connected to each other. To use this estimation for GS, we make the following adjustments. As discussed in §2.2.2, in a sparsified graph G', $|V'| = |V|$ and $|E'| = |E| \times p$, and $T' = T \times p^2$. Then if $Z = A - B$, $|Z|$ is estimated as $\frac{|V|}{|E| \cdot p}$. If $Z = A \cap B$, $|Z|$ is estimated as $|V| \cdot p_1 \cdot p_2^{n-1}$, where $p_1 = \frac{2 \cdot |E| \cdot p}{|V|^2}$, $p_2 = \frac{T \cdot |V|}{2 \cdot |E|^2 \cdot p^4}$.
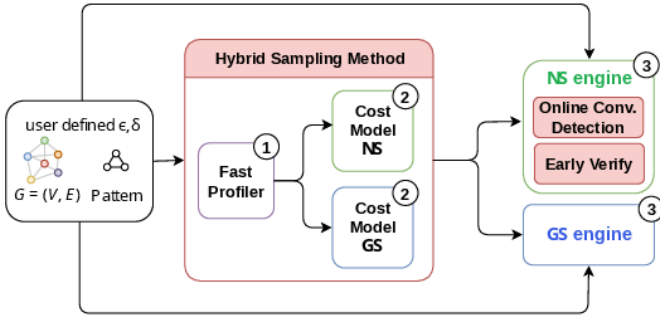
Figure 7: SCALEGPM system overview. The three red boxes are our proposed novel techniques: online convergence detection, early pruning and hybrid sampling. NS: neighbor sampling. GS: graph sparsification. The system execution flow is ① fast profiler estimates input parameters (e.g., #colors, #samples), ② cost models predict performance and select from NS and GS sampling schemes, and ③ the selected (NS or GS) engine is invoked to conduct sampling.

5 SYSTEM DESIGN AND IMPLEMENTATION

We give an overview of our system in §5.1, describe details of the GS engine in §5.2 and our proposed profiling mechanism in §5.3.

5.1 System Overview and Interface

Fig. 7 illustrates the major components in our system. SCALEGPM is composed of a fast profiler, two cost models and two execution engines for NS and and GS respectively. The cost models have been described in §4.3 and §4.4 respectively. The NS engine is enhanced with our novel convergence detection mechanism (§4.1) and is significantly improved by our proven unbiased optimizations (§4.2). Our GS engine is the first generalized color sparsification for arbitrary patterns, as all prior GS-based work are customized for specific patterns, e.g., triangle. To generalize the GS approach for an arbitrary pattern $\mathcal{P}$, we need to (1) generate a pattern specific exact search program, (2) determine the scaling factor for $\mathcal{P}$ and (3) determine the values of its key parameter, i.e., the sparsify probability p or the number of colors c ($c = \frac{1}{p}$). For (1) we can leverage the state-of-art compiler based approach [44]. For (2) we explain in §5.2. For (3) our fast profiler in §5.3 determines it.

To meet various accuracy requirements in applications, SCALEGPM provides two modes for the user to choose, *strict mode* (default) and *loose mode*. The strict mode uses only the NS engine with online convergence detection to guarantee high confidence. In this mode the fast profiling and cost models are bypassed. The loose mode, however, employs our proposed hybrid approach. It uses the fast profiler and cost models to determine if the GS or NS engine is used. When comparing the predicted performance of NS and GS, SCALEGPM uses a thresholding mechanism to check if the predicted performance of GS overlaps with the performance cone of NS, i.e., it is either entirely above or entirely below the cone. If not, SCALEGPM chooses the faster one and activates the corresponding engine. Otherwise, the two schemes should perform similarly well, hence SCALEGPM chooses NS to guarantee high confidence.

5.2 Tradeoff in the GS Engine

For simplicity, we discuss edge sparsification, while color sparsification is similar. The estimated count is $\hat{C} = Y \cdot p^{-l}$, where the random

Table 1: Data graphs (symmetric, no loops or duplicate edges). Maximum degrees are smaller when orientation is applied for cliques.

Graph	Source	|V|	|E|	Avg deg.	Max deg.	Cluster Coeff.
lj	liveJournal [42]	4.8M	43M	17.7	20,333	.18
tw	twitter40 [41]	42M	2.4B	57.7	2,997,487	.0002
fr	friendster [74]	66M	3.6B	55.1	5,214	.005
uk	uk2007 [15]	106M	6.6B	62.4	975,419	.011
gsh	gsh-2015 [16]	988M	51B	52.0	58,860,305	.022
cw	clueweb12 [16]	978M	75B	76.4	75,611,696	.00002

variable Y denotes the number of matches in the sampled graph G', which is sparsified from $\mathcal{G}$ with probability p. For each match M in $\mathcal{G}$, the probability that M exists in G' is p^l, hence the expected number of matches in G' is $\mathbb{E}[Y] = p^l C$, i.e., $\mathbb{E}[Y \cdot p^{-l}] = C$, so we have $\mathbb{E}[\hat{C}] = C$. Although the edges are sampled independently, the matches are not. Consider an edge e shared between two matches M_i and M_j of $\mathcal{P}$. When e gets removed during sparsification, necessarily both M_i and M_j will not be counted. So the Chernoff bounds do not apply directly for GS. We can use Chebyshev's bounds instead. $\mathbb{V}ar[\hat{C}] = C \cdot (p^{-l} - 1) + \mathbb{C}ov$, where $\mathbb{C}ov = \sum_{z=2}^{k-1} t_z \cdot (p^{1-z} - 1)$ and t_z is the number of pairs of matches that share z vertices. The variance depends on both the number of matches C in $\mathcal{G}$ and the number of pairs of matches that share one or more edges.

Apparently, the key knob to tune accuracy is p. As we increase p, the variance decreases and accuracy increases. Since GS speed is insensitive to the number of colors $c = \frac{1}{p}$ within a wide stabilization window (Fig. 6), we can pick a large p to achieve better accuracy. Note that the variance increases exponentially in the pattern size k. This means larger patterns are more difficult to estimate accurately, and so we may need a larger p, to guarantee the same error bound.

5.3 Fast Profiling for Cost Models

To predict performance in our cost models, we need the number of samples N_s for NS and the number of colors c for GS. Our fast profiler is used to quickly determine the values of these parameters. Note that N_s here is only used in the NS cost model, not used for NS sampling (as our NS-online does not need to predict N_s). The profiler first generates a sparsified graph G', e.g., 10% of the original graph. Then, it runs our NS-online engine with an internal error bound (50%) and confidence (99%). By detecting convergence, we can determine $N_s = \frac{N_o \cdot \hat{\epsilon} \cdot \mu \cdot \rho(P,G)}{\bar{S} \cdot \epsilon^2 \cdot \rho(P,G')}$, where N_o is the number of samples NS-online converged with, μ is the count returned by NS-online, $\bar{S}$ is the scaled count from G' to G (see 2.2.2), $\hat{\epsilon}$ is the final predicted error by NS-online, $\rho(P, G)$ is the probability of sampling pattern P in G, which is used in prior systems [34, 78], and determined by properties of G, e.g., Δ and $|E|$.

6 EVALUATION

We implement SCALEGPM in C++ and OpenMP for parallelization. Our GS engine is parallelized among subgraph partitions, as each partition can be searched independently. Within each partition, we further parallelize it over each vertex in the subgraph, which provides enough parallelism. In NS, as each sample is an independent task, it can be embarrassingly parallelized across samples.

We use SCALEGPM-NS, SCALEGPM-GS, SCALEGPM-HY to represent the NS, GS, and hybrid mode of our system respectively.

In this evaluation we focus on comparing with prior A-GPM systems that provide both generalization and automation (see §7 for discussion on non-systematic solutions). We compare ScaleGPM with the state-of-the-art A-GPM system, Arya [78], and exact GPM system, GraphZero [44]. All methods are parallelized across the 48 available cores. We do not include ASAP since Arya always outperforms ASAP. We test on a 3.0 GHz, 48-core (2-sockets, 24 cores per socket) Intel CPU without hyperthreading, with up to 1TB of memory. Table 1 shows the graphs used in our experiments, which are representative real-world graphs with varying sizes and topology characteristics. Note that `li`, `tw`, and `fr` are tested in Arya [78] or ASAP [34]. We add larger graphs `uk`, `gsh` and `cw`, which are evaluated separately in Table 4b, as for most of the cases Arya runs overtime or out of memory on these datasets. In ScaleGPM, graphs are represented in the Compressed Sparse Row (CSR) format. We evaluate two types of GPM tasks: subgraph counting (edge-induced) and motif counting (vertex-induced). For subgraph counting, we test on patterns including k-`cliques`, 5-`path`, `house`, and `dumbbell`. We do not include even larger non-clique patterns, because we can not verify the errors as their exact counts are unknown for most of the graphs. In all the experiments, we time out at 10 hours. We then conservatively use 10 hours for the timed-out cases when calculating speedups over them.

We first compare the overall performance of ScaleGPM with state-of-the-art systems in §6.1. We show how our convergence detection performs in §6.2, and the accuracy the `NS` and `GS` cost models in §6.3. We discuss system efficiency in §6.4.

6.1 Sampling Performance vs. State-of-the-Art

We compare performance with Arya and GraphZero. For the hybrid mode we show its profiling time in §6.4. For ScaleGPM-`GS`, we include the preprocessing time on sparsifying the data graph. We use an error bound of 10% and confidence of 99%, which is a common practice in ASAP and Arya. A lower error bound (e.g., 5%) leads to (10-hour) timeout on many of the cases for Arya, thus we do a separate study by varying the error bounds in Table 5b.

Table 2 compares k-`clique` (k=3,4,6) running time of ScaleGPM with Arya and GraphZero. Overall cliques, ScaleGPM achieves a `geomean` average speedup of 2747× (up to 610169×) against Arya, and 4045× over (up to 65525×) GraphZero respectively. In general, the speedup of ScaleGPM-`NS` over Arya comes from two parts. First, our online convergence detection gives stable and precise termination condition (demonstrated later in Fig. 8 and Fig. 9), while ELP in Arya could suggest very conservative termination conditions that do way more samples than necessary (see Fig. 11). Second, our `NS`-prune approach dramatically improves the hit rate over `NS`-base and thus the total number of samples is reduced. This is evidenced in Fig. 5 and further confirmed in Fig. 10. The significant speedups over Arya are expected because (1) Arya is based on pattern decomposition and cliques are hard to decompose, (2) cliques are more likely to fall in the needle-in-the-hay cases which Arya handles poorly. The speedups are further evidenced by the hit rate and number of samples N_s used. In particular, for 4-`clique`, Arya requires 3 to 5 orders of magnitude more samples, while its hit rates are extremely low, e.g., 1.7×10^{-6}% for `Fr`. Notably, Arya is 28× slower than GraphZero for the sparse case 4-`clique` on

`Fr`. Moreover, Arya's ELP can not converge within 10 hours for the even more sparse case, 6-`clique` on `Fr`, emphasizing the limitation of ELP. For `triangle` on `Fr`, ScaleGPM-`NS` is slightly slower than Arya due to synchronization overhead.

Table 3 compares performance on large, non-clique patterns, including 5-`path`, 5-`house` and 6-`dumbbell`. Note that it is well known that for exact GPM solvers (e.g. GraphZero) it is much more expensive to search for these patterns than cliques, as they are sparser and their sizes are beyond 4. We observe that GraphZero is timed out for most of the cases, which shows that it is critical to use approximation for large (sparse) patterns. Arya enjoys fairly good speedups over GraphZero. This is because, compared to cliques, these patterns are easier to decompose, which is the case that favors Arya. However, for `Fr`, Arya still suffers extremely high N_s and low hit rate, and hence runs more than 10 hours for 5-`house` and 6-`dumbbell`. In all these cases, ScaleGPM significantly improves hit rates and reduces N_s, which lead to fast convergence speed. Across these patterns, ScaleGPM achieves an `geomean` average of 599× and 27641× speedup against Arya and GraphZero respectively. Note that for `Tw` on `house`, as shown in Table 5a, ScaleGPM-`HY` selects the `GS` engine in this case, since `GS` is predicted to be faster, based on our prediction on the number of samples and time per sample.

Table 4a reports the motif counting performance. Note that in motif counting, we look for vertex-induced subgraphs, unlike cases in Table 3 which search for edge-induced subgraphs. The key difference in computation is that we need both set intersection and set difference to find vertex-induced subgraphs, but only set intersection is needed for edge-induced subgraphs. Despite more computation needed, ScaleGPM is still significantly faster than Arya. Across motifs, ScaleGPM achieves a `geomean` speedup of 125× speedup against Arya, and 10563× speedup over GraphZero.

Table 4b compares performance on huge graphs, i.e., uk2007 (`uk`) gsh-2015 (`gsh`) and clueweb12 (`cw`). Note that Arya mostly runs out of memory for `cw` because (1) it has to maintain the set union results (for each of the parallel threads) in memory, and (2) its internal representation of the graph stream is implemented in COO-like format, which is less compact than CSR. ScaleGPM achieves a `geomean` average 130× compared to Arya and a 7245× speedup compared to GraphZero.

Table 5a shows the cases where ScaleGPM-`HY` selects the `GS` mode. The time spent in the pre-processing color sparsification step of `GS` is shown as a percentage of the total time taken. Notably, for `Fr` and $k = 9$ (sparse graph and big dense pattern) which is a needle-in-the-hay case, ScaleGPM successfully chose to use `GS` instead of `NS`. This switch from `NS` to `GS` brings us a 61× speedup. Over all the cases where the `GS` engine is selected, the `GS` engine leads to a `geomean` average 13× speedup over the `NS` engine.

Table 5b shows Arya and ScaleGPM running time with varied error bounds. We observe that the performance gap between Arya and ScaleGPM remains huge (from 31 to 49 thousand times) as we decrease the error bound. Overall, we achieve 565× speedup over Arya, and four orders of magnitude speedup over GraphZero.

6.2 Effectiveness of Convergence Detection

Fig. 8 compares the error predicted by ScaleGPM-`NS` with the actual error throughout the sampling procedure, to show the effectiveness of ScaleGPM-`NS`'s online convergence detection. We

Table 2: k-clique performance (10% error). TO: timed out. Arya uses ELP. ×: ELP does not converge. Fastest time in bold.

Pattern		3-clique (triangle)			4-clique			6-clique		
Graph		Lj	Tw	Fr	Lj	Tw	Fr	Lj	Tw	Fr
SCALEGPM-NS	time (sec)	**0.001**	**0.046**	0.026	**0.003**	**0.068**	**0.090**	**0.059**	**0.707**	**1.132**
	error (observed)	3.04%	3.32%	0.23%	4.86%	0.39%	0.04%	3.82%	–	2.23%
	hit rate	18%	55%	53%	7.3%	46%	31%	6.4%	46%	16%
	# samples	1e4	2e4	1e4	8e4	2e5	5e4	2.1e6	4.1e6	7.7e7
Arya	time (sec)	0.014	1.193	**0.017**	94.2	9656.9	2057.7	TO	TO	TO
	error (observed)	0.76%	4.92%	1.12%	0.03%	0.89 %	7.37%	–	–	–
	hit rate	10.6%	3.9%	2.9%	3.2e-3%	2.7e-3%	1.7e-6%	–	–	–
	# samples	4.3e4	6.3e4	4.6e4	3.3e8	2.6e8	5.1e9	×	×	×
GraphZero	time (sec)	0.434	58.0	83.0	2.1	4004.5	73.1	2502.5	TO	160.4

Table 3: Non-clique pattern performance (10% error). TO: timed out. ×: ELP does not converge. * GS is selected by our hybrid method.

Pattern		5-path			5-house			6-dumbbell		
Graph		Lj	Tw	Fr	Lj	Tw	Fr	Lj	Tw	Fr
SCALEGPM-NS	time (sec)	**0.004**	**0.91**	**0.10**	**0.008**	944.6*	**0.159**	**0.017**	**159.3**	**0.331**
	error (observed)	5.13%	–	8.68 %	2.88%	–	2.28%	2.53%	–	–
	hit rate	29%	99%	90%	43%	43%	15%	16%	43%	32%
	# samples	3e4	1e4	2e4	5e4	1.9e7	3.3e5	1.5e5	1.6e7	6.9e5
Arya	time (sec)	4.422	13.78	77.93	395.6	1297.4	TO	346.2	4644.4	TO
	error (observed)	6.90%	–	7.43%	1.76%	–	–	6.85%	–	–
	hit rate	0.04%	5.8e-03%	1.7%	2.6e-3%	0.02%	–	0.04%	2.2e-3%	–
	# samples	1.1e7	1.4e7	3.4e5	1.5e9	6.8e7	×	2.1e8	1.1e9	×
GraphZero	time (sec)	262.1	TO	TO	TO	TO	TO	TO	TO	TO

Table 4: Running time (sec) (10% error). TO: timed out. ×: ELP does not converge. * the true count is unknown, so accuracy is not verified.

(a) Performance on Motif Counting.

Pattern	3-motif			4-motif		
Graph	Lj	Tw	Fr	Lj	Tw	Fr
SCALEGPM-NS	**0.004**	**0.5**	0.14	**0.06**	**82.7**	**0.2**
Arya	0.020	1.9	**0.08**	231.63	13180.2	6157.9
GraphZero	1.283	16316.2	242.62	1927.27	TO	TO

(b) Performance on huge graphs.

Pattern	triangle			4-clique			5-path*		
Graph	uk	gsh	cw	uk	gsh	cw	uk	gsh	cw
SCALEGPM-NS	**0.09**	**0.28**	**0.19**	**0.11**	**0.88**	**1.09**	**0.1**	**11.2**	**156.0**
Arya	0.2	0.7	OoM	175.5	×	OoM	4.2	×	OoM
GraphZero	73.3	153.6	198.2	TO	TO	TO	TO	TO	TO

Table 5: (a) Comparing NS and GS performance when SCALEGPM-HY selects GS; (b) performance changes when varying error bounds.

(a) running time (sec) of NS and GS, with sparsification time percent

Pattern	8-clique		9-clique		5-house
Graph	lj	fr	lj	fr	tw
SCALEGPM- NS	3.0	1663.6	17.4	8358.0	944.6
SCALEGPM- GS	**0.7 (19%)**	**43.4 (9%)**	**2.1 (6%)**	**134.9 (2%)**	**200.1 (1%)**

(b) running time (sec) under error bounds from 10% to 1%

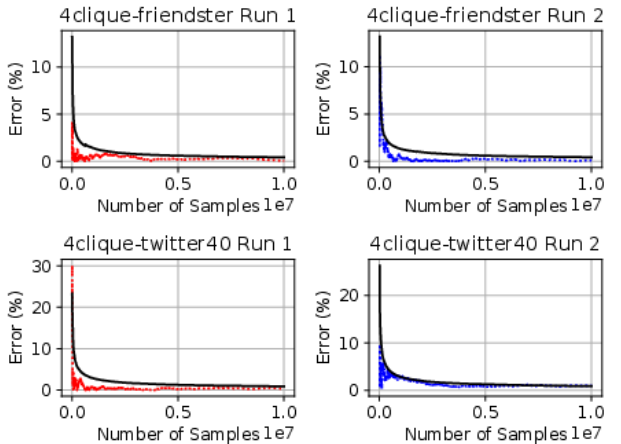| Pattern-Graph	house – Lj				4-clique – Tw			
Error bound	10%	5%	2%	1%	10%	5%	2%	1%
SCALEGPM-NS	**0.008**	**0.053**	**0.3**	**1.1**	**0.07**	**0.13**	**0.69**	**2.68**
Arya	395.6	1639.6	9873.8	TO	9656.8	TO	TO	TO

Figure 8: Comparing SCALEGPM-NS's estimated errors (black curves) with actual errors (red and blue curves). We do two runs of 4-clique counting on two graphs **Fr** and **Tw**.

illustrate two cases here, but we have verified the same trend for all our test cases in the evaluation. In Fig. 8 we observe that for both cases, the predicted error curves strictly bound the actual error, which verifies the high confidence that our method achieves. In addition to our theoretical proof in §4.1, this empirical study further demonstrates that our method provides strong guarantee on confidence, which is critical in applications where users want to have strict accuracy requirement.

Fig. 9 shows the stability of the error estimation method in SCALEGPM-NS. We do three repeated runs on the 4-clique pattern and Lj graph. In contrast to the unstable predictions (Fig. 1) given by ELP in prior systems, our estimated errors are almost the same across three independent runs. Moreover, our online mechanism never fails, while ELP suffers convergence issue that may lead to endless preprocessing (e.g., for 6-clique in Table 2). We observe the same trend in stability on other graphs and patterns. This experiment further demonstrates that our method is highly reliable and can be adopted in practice.

Fig. 10 compares the convergence rate of NS-prune and NS-base, both under our online detection framework. We observe that our
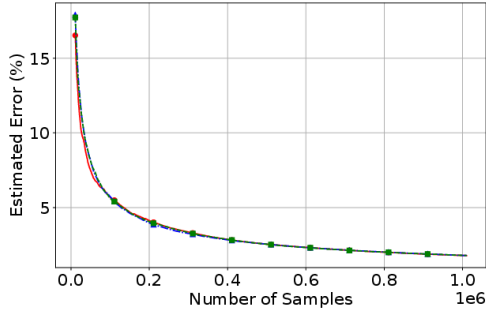
Figure 9: Predicted errors in our convergence detection mechanism across three different runs (`4-clique` on `Lj`) are extremely stable.
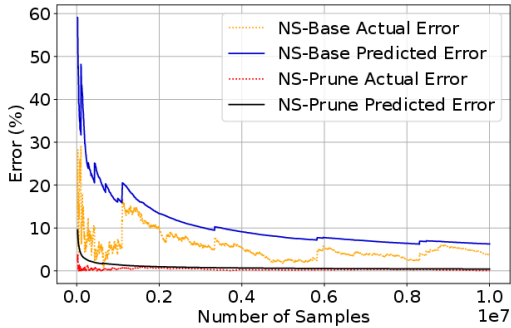

Figure 10: Comparison between the convergence rate of `NS`-prune and `NS`-base. The same formula (with two standard deviations of confidence) was used to generate the error estimate curves.
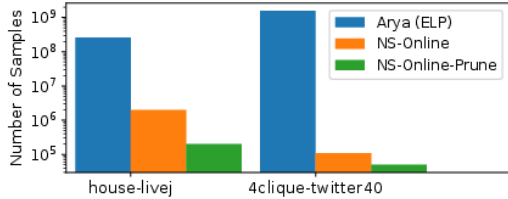

Figure 11: The reduction on the number of samples.

proposed early-pruning mechanism employed in `NS`-prune result in roughly an order of magnitude lower error for the same number of samples. Therefore, with pruning, our system can converge much faster that ASAP and Arya, which is the other major reason why SCALEGPM achieves much better performance.

Fig. 11 shows the reduction on the number of samples N_s, by incrementally applying online convergence detection (orange) and eager-verify (green), against ELP in Arya (blue). Note that in Table 2 and Table 3 we report N_s only for `NS`-online-prune, but here we breakdown the contributions of online detection and pruning. We observe that our online method reduces N_s sharply over Arya. Applying pruning in eager-verify further reduces N_s by a significant amount. Together, we can meet the same error bound with much few samples, and more importantly, with confidence.

6.3 Prediction Accuracy of Cost Models

Fig. 12 shows the effectiveness of our cost model in predicting the running time of the `GS` engine in SCALEGPM. As we increase the

number of colors $c = \frac{1}{p}$ used in `GS`, the total work is exponentially decreased. Thus, the `GS` execution time rapidly becomes dominated by the preprocessing time spent on sparsifying the graph. For `4clique` and `house` on `Lj` and `Fr`, we see that the cost model precisely captures the exponential trend, as well as the stabilization window of the `GS` engine that we discussed in Fig. 6. Note that when c is extremely small, e.g., $c < 5$, it is hard to accurately model the performance due to the steep slope in that curve. However, in this case (which is a needle-in-the-hay case) we would favor the use of `NS` or even exact counting, as `GS` with a small c won't be much faster than exact counting.

Fig. 13 shows the effectiveness of our `NS` cost model. We see that our proposed *performance cone* correctly captures the running time of the `NS` engine at varying numbers of samples. Note that for `Fr`, a relatively sparse graph, the execution time approaches the bottom of the performance cone as expected. We find in practice, that `GS` and `NS` are often predicted to have very distinct performance, so the width of the cone is not an obstacle in making the correct choice. The fluctuation in the sparse case `4clique-Fr` is also expected, as sample hits are less frequent and thus more randomness is involved.

6.4 System Efficiency

Timing Breakdown. Fig. 14 shows the breakdown of the execution time spent on different components of SCALEGPM. We consider the profiling time, the `GS` preprocessing time, and the sampling time (either exact counting in `GS` or drawing samples in `NS`). We see that profiling remains a low percentage of the overall runtime. As for the sampling time, `Lj-8clique` requires less time using the `GS` engine, while `Fr-dumbbell` is processed faster using the `NS` engine, which aligns with our cost model prediction and thus SCALEGPM-HY makes the correct thresholding decision.

Scalability. Fig. 15 shows how the performance of `GS` engine and `NS` engine in SCALEGPM (error bound of 10%) scales in response to the increase of parallel cores (i.e., the number of threads). We evaluate the `NS` engine on `5path-Lj` which is a case that favors the use of the `NS` engine (i.e., `NS` is faster than `GS`). `GS` is evaluated on `8clique-Fr` as `8-clique` is a dense pattern and is rare in `Fr`, preferably executed in `GS`. In both cases, we observe strong scaling that the execution time of both engines increases linearly as we range the number of cores from 1 to 48.

7 RELATED WORK

There exist many sampling methods other than neighbor sampling and graph sparsification. We discuss some of the typical methods in the following. We do not compare with them in §6 because they do not provide either generalization or automated termination (with confidence). However, they can potentially be used to replace the `GS` engine in SCALEGPM. We leave it as a future work.

Color Coding. It first colors each vertex in $\mathcal{G}$ using a color randomly chosen from $\{1,2,\ldots,c\}$ ($c \geq k$), and then counts *colorful* matches, i.e., every vertex in the matched subgraph has a unique color. The requirement of distinct colors allows for heavy pruning: the number of colorful matches Z can be naturally determined by a dynamic programming based counting routine [61]. Color-coding is originally for finding paths or cycles [6, 7], and is then adapted for motif counting [5, 33, 76]. There also exist many parallel and distributed implementations [22, 61, 62, 77]. In addition, the colorful
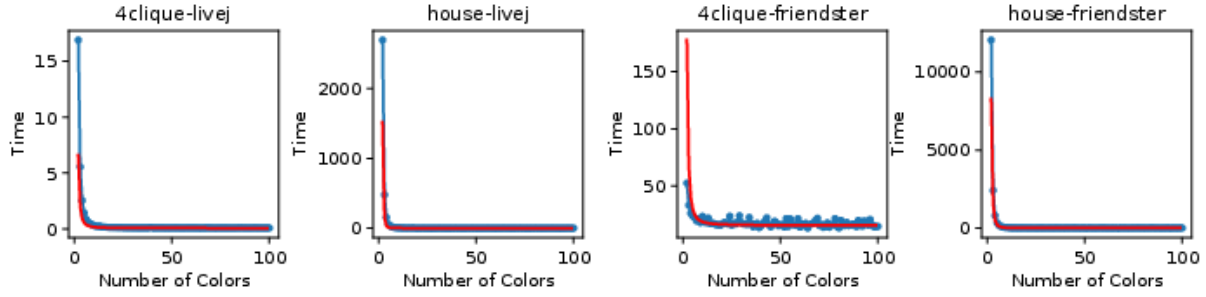
Figure 12: The quality of performance prediction for SCALEGPM's GS Engine. The red lines are our predictions, while blue dots are actual time.
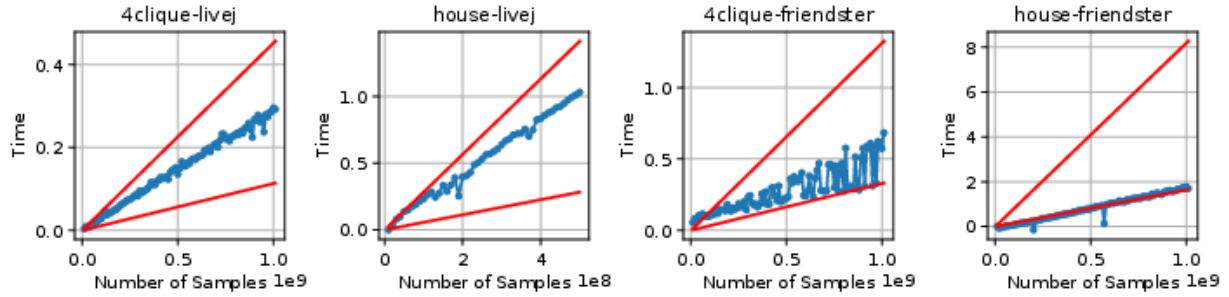

Figure 13: The quality of performance prediction for SCALEGPM's NS Engine. The red lines are our predictions, while blue dots are actual time.
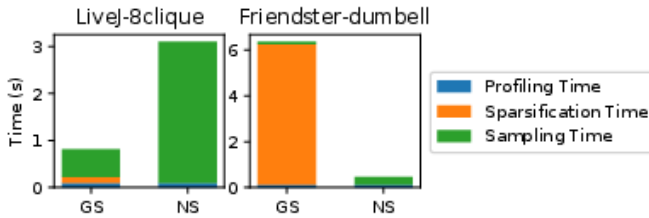

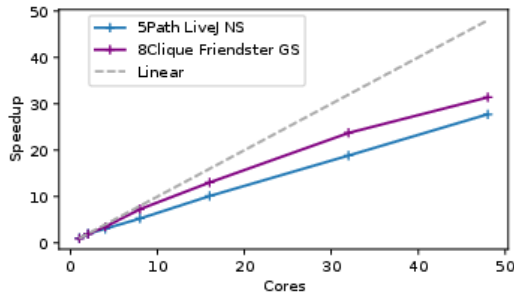**Figure 14: End-to-end time breakdown of SCALEGPM.**

Figure 15: SCALEGPM speedup scaling over single-thread.

matches can be further sampled [18–20], instead of exactly counted, to reduce computation. Like GS, color-coding is also a coarse-grain scheme, which can be included in SCALEGPM.

Loop Perforation. SampleMine [38] proposed to perforate the nested `for` loops in GPM programs, with a certain probability p_i for the i-th loop. The count is then scaled based on the p_i. SampleMine can be thought of as a generalization of the vanilla NS scheme, as it collectively samples multiple candidates, instead of a single one, at a time. Although it has larger sampling granularity than the vanilla NS, its granularity is still limited by the egonet of a vertex or edge,

similar to Egonet Sampling [56]. More importantly, SampleMine does not provide a systematic way for sampling termination, which is instead hand-tuned by executing the sampling procedure multiple times and manually observing convergence (i.e. small variance).

Other Schemes and Approaches. Monte Carlo Markov Chain (MCMC) [12, 14, 28, 72] defines a random walk over the set of subgraphs until it reaches stationarity. MCMC has been used for motif counting. However, it has been shown that MCMC can be extremely inefficient as the random walk may take a huge amount of steps to reach stationarity [18–20]. There also exist GPM algorithms for non-deterministic graphs, whose edges only exist probabilistically [43]. Though also based on probability theory, they are not for sampling, and thus out of scope of this paper.

8 CONCLUSION

Approximate graph pattern mining (A-GPM) systems can be the backbone to support numerous real-world data analytics applications. The key obstacles that prevent A-GPM systems from being adopted in practice is a) the lack of stable, confident termination mechanism and b) poor performance and scalability when dealing with the "hard" cases. We present SCALEGPM, an accurate, high performance and scalable A-GPM systems. SCALEGPM involves two key innovations that remove the obstacles. First, we propose a novel online convergence detection mechanism, which can provide theoretical guarantee on prediction confidence and also yield stable termination condition. Second, we introduce pruning techniques into sampling to improve its success rate, and propose a hybrid method to adaptively select the best-performing sampling scheme from two complementary schemes, based on our proposed cost models. The resulting system, SCALEGPM, achieves an average of 565× speedup over the state-of-the-art A-GPM systems, and manages to rapidly mine billion-scale graphs.

REFERENCES

[1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. Scalemine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 61, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014986

[2] Nesreen K. Ahmed, Nick Duffield, Jennifer Neville, and Ramana Kompella. 2014. Graph Sample and Hold: A Framework for Big-Graph Analytics. 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, 1446–1455. https://doi.org/10.1145/2623330.2623757

[3] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery* 29, 3 (2015), 626–688.

[4] Maryam Aliakbarpour, Amartya Shankha Biswas, Themis Gouleakis, John Peebles, Ronitt Rubinfeld, and Anak Yodpinyanee. 2018. Sublinear-Time Algorithms for Counting Star Subgraphs via Edge Sampling. *Algorithmica* 80, 2 (feb 2018), 668–697. https://doi.org/10.1007/s00453-017-0287-3

[5] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S Cenk Sahinalp. 2008. Biomolecular network motif counting and discovery by color coding. *Bioinformatics* 24, 13 (2008), i241–i249.

[6] Noga Alon, Raphy Yuster, and Uri Zwick. 1994. Color-coding: A New Method for Finding Simple Paths, Cycles and Other Small Subgraphs Within Large Graphs. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing* (Montreal, Quebec, Canada) *(STOC '94)*. ACM, New York, NY, USA, 326–335. https://doi.org/10.1145/195058.195179

[7] Noga Alon, Raphael Yuster, and Uri Zwick. 1995. Color-Coding. *J. ACM* 42, 4 (jul 1995), 844–856. https://doi.org/10.1145/210332.210337

[8] Sepehr Assadi, Mikhail Kapralov, and Sanjeev Khanna. 2018. A Simple Sublinear-Time Algorithm for Counting Arbitrary Subgraphs via Edge Sampling. In *Information Technology Convergence and Services*.

[9] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. 2002. Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, California) *(SODA '02)*. Society for Industrial and Applied Mathematics, USA, 623–632.

[10] András A. Benczúr and David R. Karger. 1996. Approximating S-t Minimum Cuts in Õ(N2) Time. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96)*. Association for Computing Machinery, New York, NY, USA, 47–55. https://doi.org/10.1145/237814.237827

[11] Austin R. Benson, David F. Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (2016), 163–166. https://doi.org/10.1126/science.aad9029 arXiv:https://science.sciencemag.org/content/353/6295/163.full.pdf

[12] Suman K. Bera and C. Seshadhri. 2020. How to Count Triangles, without Seeing the Whole Graph. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 306–316. https://doi.org/10.1145/3394486.3403073

[13] Vandana Bhatia and Rinkle Rani. 2018. Ap-FSM: A parallel algorithm for approximate frequent subgraph mining using Pregel. *Expert Systems with Applications* 106 (2018), 217–232. https://doi.org/10.1016/j.eswa.2018.04.010

[14] Mansurul A. Bhuiyan, Mahmudur Rahman, Mahmuda Rahman, and Mohammad Al Hasan. 2012. GUISE: Uniform Sampling of Graphlets for Large Graph Analysis. In *2012 IEEE 12th International Conference on Data Mining*. 91–100. https://doi.org/10.1109/ICDM.2012.87

[15] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A Large Time-Aware Graph. *SIGIR Forum* 42, 2 (2008), 33–38.

[16] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA) *(WWW '04)*. ACM, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752

[17] Marco Bressan. 2021. Efficient and Near-Optimal Algorithms for Sampling Connected Subgraphs. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing* (Virtual, Italy) *(STOC 2021)*. Association for Computing Machinery, New York, NY, USA, 1132–1143. https://doi.org/10.1145/3406325.3451042

[18] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2017. Counting Graphlets: Space vs Time. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining* (Cambridge, United Kingdom) *(WSDM '17)*. ACM, New York, NY, USA, 557–566. https://doi.org/10.1145/3018661.3018732

[19] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2018. Motif Counting Beyond Five Nodes. *ACM Trans. Knowl. Discov.*

Data 12, 4, Article 48 (April 2018), 25 pages. https://doi.org/10.1145/3186586

[20] Marco Bressan, Stefano Leucci, and Alessandro Panconesi. 2019. Motivo: Fast Motif Counting via Succinct Color Coding and Adaptive Sampling. *Proc. VLDB Endow.* 12, 11 (July 2019), 1651–1663. https://doi.org/10.14778/3342263.3342640

[21] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. 2006. Counting Triangles in Data Streams. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Chicago, IL, USA) *(PODS '06)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/1142351.1142388

[22] V. T. Chakaravarthy, M. Kapralov, P. Murali, F. Petrini, X. Que, Y. Sabharwal, and B. Schieber. 2016. Subgraph Counting: Color Coding Beyond Trees. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 2–11. https://doi.org/10.1109/IPDPS.2016.122

[23] J. Chen, T. Eden, P. Indyk, S. Narayanan, R. Rubinfeld, S. Silwal, D. Woodruff, and M. Zhang. [n. d.]. Triangle and Four Cycle Counting with Predictions in Graph Streams. *Tenth International Conference on Learning Representations (ICLR 2022)* ([n. d.]). https://par.nsf.gov/biblio/10338743

[24] Jingji Chen and Xuehai Qian. 2022. DecoMine: A Compilation-Based Graph Pattern Mining System with Pattern Decomposition. 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, 47–61. https://doi.org/10.1145/3567955.3567956

[25] Justin Y Chen, Talya Eden, Piotr Indyk, Honghao Lin, Shyam Narayanan, Ronitt Rubinfeld, Sandeep Silwal, Tal Wagner, David Woodruff, and Michael Zhang. 2022. Triangle and Four Cycle Counting with Predictions in Graph Streams. In *International Conference on Learning Representations*. https://openreview.net/forum?id=8in_5gN9I0

[26] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining. In *Proceedings of the 35th ACM International Conference on Supercomputing (ICS '21)*. 14 pages.

[27] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proc. VLDB Endow.* 13, 8 (Aug. 2020). https://doi.org/10.14778/3389133.3389137

[28] Xiaowei Chen, Yongkun Li, Pinghui Wang, and John C. S. Lui. 2016. A General Framework for Estimating Graphlet Statistics via Random Walk. *Proc. VLDB Endow.* 10, 3 (nov 2016), 253–264. https://doi.org/10.14778/3021924.3021940

[29] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 1357–1374. https://doi.org/10.1145/3299869.3319875

[30] Ethan R. Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G. Dimakis. 2015. Beyond Triangles: A Distributed Framework for Estimating 3-profiles of Large Graphs. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Sydney, NSW, Australia) *(KDD '15)*. ACM, New York, NY, USA, 229–238. https://doi.org/10.1145/2783258.2783413

[31] Philippe Fournier Viger, Ganghuan He, Chao Cheng, Jiaxuan Li, Min Zhou, Jerry Lin, and Unil Yun. 2020. A survey of pattern mining in dynamic graphs. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 10 (05 2020). https://doi.org/10.1002/widm.1372

[32] Wai Shing Fung, Ramesh Hariharan, Nicholas J.A. Harvey, and Debmalya Panigrahi. 2011. A General Framework for Graph Sparsification. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing* (San Jose, California, USA) *(STOC '11)*. Association for Computing Machinery, New York, NY, USA, 71–80. https://doi.org/10.1145/1993636.1993647

[33] Falk Hüffner, Sebastian Wernicke, and Thomas Zichner. 2008. Algorithm engineering for color-coding with applications to signaling pathway detection. *Algorithmica* 52 (2008), 114–132.

[34] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 745–761. http://dl.acm.org/citation.cfm?id=3291168.3291224

[35] Shweta Jain and C. Seshadhri. 2020. The Power of Pivoting for Exact Clique Counting. In *Proceedings of the 13th International Conference on Web Search and Data Mining* (Houston, TX, USA) *(WSDM '20)*. Association for Computing Machinery, New York, NY, USA, 268–276. https://doi.org/10.1145/3336191.3371839

[36] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: A Pattern-Aware Graph Mining System. In *Proceedings of the Fifteenth EuroSys Conference (EuroSys '20)*.

[37] Madhav Jha, C. Seshadhri, and Ali Pinar. 2015. Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy) *(WWW*

'15). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 495–505. https://doi.org/10.1145/2736277.2741101

[38] Peng Jiang, Yihua Wei, Jiya Su, Rujia Wang, and Bo Wu. 2023. SampleMine: A Framework for Applying Random Sampling to Subgraph Pattern Mining through Loop Perforation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22).* Association for Computing Machinery, New York, NY, USA, 185–197. https://doi.org/10.1145/3559009.3569658

[39] John Kallaugher, Andrew McGregor, Eric Price, and Sofya Vorotnikova. 2019. The Complexity of Counting Cycles in the Adjacency List Streaming Model. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Amsterdam, Netherlands) *(PODS '19).* Association for Computing Machinery, New York, NY, USA, 119–133. https://doi.org/10.1145/3294052.3319706

[40] Michihiro Kuramochi and George Karypis. 2004. GREW-A Scalable Frequent Subgraph Discovery Algorithm. In *Proceedings of the Fourth IEEE International Conference on Data Mining (ICDM '04).* IEEE Computer Society, USA, 439–442.

[41] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10).* ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[42] J. Leskovec. 2013. SNAP: Stanford Network Analysis Platform. http://snap.stanford.edu/data/index.html

[43] Chenhao Ma, Reynold Cheng, Laks V. S. Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. 2019. LINC: a motif counting algorithm for uncertain graphs. *Proc. VLDB Endow.* 13, 2 (oct 2019), 155–168. https://doi.org/10.14778/3364324.3364330

[44] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. GraphZero: A High-Performance Subgraph Matching System. *SIGOPS Oper. Syst. Rev.* 55, 1 (June 2021), 21–37. https://doi.org/10.1145/3469379.3469383

[45] Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing High-level Abstraction and High Performance for Graph Mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19).* ACM, New York, NY, USA, 509–523. https://doi.org/10.1145/3341301.3359633

[46] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (July 2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[47] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. 2002. Network Motifs: Simple Building Blocks of Complex Networks. *Science* 298, 5594 (2002), 824–827. https://doi.org/10.1126/science.298.5594.824 arXiv:https://science.sciencemag.org/content/298/5594/824.full.pdf

[48] Caleb C Noble and Diane J Cook. 2003. Graph-based anomaly detection. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining.* 631–636.

[49] Rasmus Pagh and Charalampos E. Tsourakakis. 2012. Colorful Triangle Counting and a MapReduce Implementation. *Inf. Process. Lett.* 112, 7 (mar 2012), 277–281. https://doi.org/10.1016/j.ipl.2011.12.007

[50] Kirill Paramonov, Dmitry Shemetov, and James Sharpnack. 2019. Estimating Graphlet Statistics via Lifting. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19).* Association for Computing Machinery, New York, NY, USA, 587–595. https://doi.org/10.1145/3292500.3330995

[51] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2013. Counting and Sampling Triangles from a Graph Stream. *Proc. VLDB Endow.* 6, 14 (sep 2013), 1870–1881. https://doi.org/10.14778/2556549.2556569

[52] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. 2017. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17).* International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1431–1440. https://doi.org/10.1145/3038912.3052597

[53] Giulia Preti, Gianmarco De Francisci Morales, and Matteo Riondato. 2021. MaNIACS: Approximate Mining of Frequent Subgraph Patterns through Sampling. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore) *(KDD '21).* Association for Computing Machinery, New York, NY, USA, 1348–1358. https://doi.org/10.1145/3447548.3467344

[54] Natasa Pržulj, Derek G Corneil, and Igor Jurisica. 2004. Modeling interactome: scale-free or geometric? *Bioinformatics* 20, 18 (2004), 3508–3515.

[55] Sumit Purohit, Sutanay Choudhury, and Lawrence B. Holder. 2017. Application-specific graph sampling for frequent subgraph mining and community detection. In *2017 IEEE International Conference on Big Data (Big Data).* 1000–1005. https://doi.org/10.1109/BigData.2017.8258022

[56] Ryan A. Rossi, Rong Zhou, and Nesreen K. Ahmed. 2019. Estimation of Graphlet Counts in Massive Networks. *IEEE Transactions on Neural Networks and Learning Systems* 30, 1 (2019), 44–57. https://doi.org/10.1109/TNNLS.2018.2826529

[57] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. 2018. Butterfly Counting in Bipartite Networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom) *(KDD '18).* Association for Computing Machinery, New York, NY, USA, 2150–2159. https://doi.org/10.1145/3219819.3220097

[58] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2020. Parallel Clique Counting and Peeling Algorithms. In *Conference on Applied and Computational Discrete Algorithms.*

[59] Jessica Shi, Louisa Ruixue Huang, and Julian Shun. 2022. Parallel Five-Cycle Counting Algorithms. *ACM J. Exp. Algorithmics* 27, Article 4.1 (oct 2022), 23 pages. https://doi.org/10.1145/3556541

[60] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: High-Performance Graph Pattern Matching through Effective Redundancy Elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20).* IEEE Press, Article 100, 14 pages.

[61] George M. Slota and Kamesh Madduri. 2013. Fast Approximate Subgraph Counting and Enumeration. In *2013 42nd International Conference on Parallel Processing.* 210–219. https://doi.org/10.1109/ICPP.2013.30

[62] George M. Slota and Kamesh Madduri. 2015. Parallel color-coding. *Parallel Comput.* 47 (2015), 51–69. https://doi.org/10.1016/j.parco.2015.02.004 Graph analysis for scientific discovery.

[63] Daniel A. Spielman and Nikhil Srivastava. 2011. Graph Sparsification by Effective Resistances. *SIAM J. Comput.* 40, 6 (2011), 1913–1926. https://doi.org/10.1137/080734029 arXiv:https://doi.org/10.1137/080734029

[64] Daniel A Spielman and Shang-Hua Teng. 2004. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing.* 81–90.

[65] Daniel A Spielman and Shang-Hua Teng. 2011. Spectral sparsification of graphs. *SIAM J. Comput.* 40, 4 (2011), 981–1025.

[66] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15).* ACM, New York, NY, USA, 425–440. https://doi.org/10.1145/2815400.2815410

[67] Charalampos E Tsourakakis, Petros Drineas, Eirinaios Michelakis, Ioannis Koutis, and Christos Faloutsos. 2011. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining* 1 (2011), 75–81.

[68] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. 2009. DOULION: Counting Triangles in Massive Graphs with a Coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Paris, France) *(KDD '09).* Association for Computing Machinery, New York, NY, USA, 837–846. https://doi.org/10.1145/1557019.1557111

[69] Ata Turk and Duru Turkoglu. 2019. Revisiting Wedge Sampling for Triangle Counting. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19).* Association for Computing Machinery, New York, NY, USA, 1875–1885. https://doi.org/10.1145/3308558.3313534

[70] Xuhao Chen and Arvind. 2022 [pdf]. Efficient and Scalable Graph Pattern Mining on GPUs. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation* **(OSDI).**

[71] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18).* USENIX Association, Berkeley, CA, USA, 763–782. http://dl.acm.org/citation.cfm?id=3291168.3291225

[72] Pinghui Wang, John C. S. Lui, Bruno Ribeiro, Don Towsley, Junzhou Zhao, and Xiaohong Guan. 2014. Efficiently Estimating Motif Statistics of Large Networks. *ACM Trans. Knowl. Discov. Data* 9, 2, Article 8 (sep 2014), 27 pages. https://doi.org/10.1145/2629564

[73] Wikipedia contributors. 2023. Slutsky's theorem — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Slutsky%27s_theorem&oldid=1186968185 [Online; accessed 1-July-2024].

[74] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. *CoRR* abs/1205.6233 (2012). arXiv:1205.6233 http://arxiv.org/abs/1205.6233

[75] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. 2022. Lightning Fast and Space Efficient K-Clique Counting. In *Proceedings of the ACM Web Conference 2022* (Virtual Event, Lyon, France) *(WWW '22).* Association for Computing Machinery, New York, NY, USA, 1191–1202. https://doi.org/10.1145/3485447.3512167

[76] Zhao Zhao, Maleq Khan, V. S. Anil Kumar, and Madhav V. Marathe. 2010. Subgraph Enumeration in Large Social Contact Networks Using Parallel Color Coding and Streaming. In *2010 39th International Conference on Parallel Processing.* 594–603. https://doi.org/10.1109/ICPP.2010.67

[77] Zhao Zhao, Guanying Wang, Ali R. Butt, Maleq Khan, V.S. Anil Kumar, and Madhav V. Marathe. 2012. SAHAD: Subgraph Analysis in Massive Networks Using Hadoop. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium.* 390–401. https://doi.org/10.1109/IPDPS.2012.44

[78] Zeying Zhu, Kan Wu, and Zaoxing Liu. 2023. Arya: Arbitrary Graph Pattern Mining with Decomposition-based Sampling. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*.