ApproxG: Fast Approximate Parallel Graphlet Counting Through Accuracy Control

Daniel Mawhirter, Bo Wu, Dinesh Mehta
Computer Science Department
Colorado School of Mines
dmawhirt@mymail.mines.edu, {bwu,dmehta}@mines.edu

Chao Ai
Huawei Technologies Co. Ltd.
Shanghai, China
aichao@huawei.com

***Abstract*—Graphlet counting is a methodology for detecting local structural properties of large graphs that has been in use for over a decade. Despite tremendous effort in optimizing its performance, even 3- and 4-node graphlet counting routines may run for hours or days on highly optimized systems. In this paper, we describe how a synergistic combination of approximate computing with parallel computing can result in multiplicative performance improvements in graphlet counting runtimes with minimal and controllable loss of accuracy. Specifically, we describe two novel techniques, multi-phased sampling for statistical accuracy guarantees and cost-aware sampling to further improve performance on multi-machine runs, which reduce the query time on large graphs from tens of hours to several minutes or seconds with only <1% relative error.**

I. Introduction

Graphlets are small (ranging from 3-5 vertices) connected non-isomorphic induced subgraphs of a large graph, first described by Pržulj et al. [22] in 2004. Graphlet counting is a methodology for detecting local structural properties to characterize the input large graphs, offering an alternative to using global statistics (such as diameter or degree distribution) to analyze graphs. There are numerous applications of graphlet counting in various domains, such as bioinfomatics [10], [12], [32], social networks [21], webspam detection [3], computer vision [34], and anomaly detection [2]. While graphlet counting has drawn significant attention [12], [12], [20], [32], even polynomial time algorithms to count graphlets may take too long on highly optimized systems. For instance, a state-of-the-art graphlet counting algorithm needs 25.5 hours to process a graph with 447M edges in a 12-core machine [1].

Researchers have proposed many approaches to using parallel hardware to accelerate graphlet counting. Ahmed et al. [1] proposed a novel parallel algorithm to count both 3- and 4-node graphlets, which significantly outperforms several state-of-the-art graphlet counting frameworks, such as FANMOD [32] and Orca [12]. Shun and Tangwongsan [28] carefully optimized locality and parallelism for triangle counting and achieved substantial performance improvements on multi-core CPUs. Distributed graph processing frameworks, such as GraphX and PowerGraph, also support simple graphlet counting algorithms. Despite all the effort, we are not aware of any system that can provide second-level latency for large graphs.

In addition to parallel computing, approximate computing is a strategy which trades off accuracy for performance. Shun and Tangwongsan's work [28] explored approximate triangle counting but does not support 4-node graphlets. GRAFT [23] is a more general framework to perform approximate graphlet counting without parallel computing support. Shang and Yu [27] proposed a compiler-based framework for automatic approximate graph computation, which also demonstrated dramatic performance improvement for triangle counting. However, all these works fail to provide accuracy guarantees, which dampens user's willingness to adopt the proposed approaches. ApproxHadoop [8] and IncApprox [14] are two approximate computing frameworks for distributed systems, which may be used to perform approximate graphlet counting, but they overlook the opportunity to exploit the relaxed accuracy requirement to improve scalability.

Prior work leaves two research questions unaddressed. First, can we modify state-of-the-art graphlet counting algorithms to compute approximate results with an accuracy guarantee? Second, can we leverage the relaxed accuracy requirement to improve the algorithm's scalability in a parallel and distributed system setting?

To address these questions, we present ApproxG, a framework to perform approximate graphlet counting, which can easily integrate existing algorithms. ApproxG applies task sampling, where a task represents the work needed to process a vertex in a vertex-centric framework (e.g., GraphLab [18]), or the work needed to process an edge in an edge-centric framework (e.g., X-stream [25]). Using sampling to reduce execution time is easy; we just sample a small percentage of tasks. However, it is challenging to control the sampling ratio to satisfy a user-defined accuracy requirement. Little is understood about whether it is more beneficial to sample vertex-related tasks or edge-related tasks. Moreover, in a parallel and distributed system where remote communications dominate execution time, it is unclear how to sample tasks to improve scalability.

To overcome these challenges, we propose two novel techniques, which combine parallel computing and approximate computing to produce multiplicative speedups for graphlet counting. The first technique is multi-phased task sampling, which leverages sampling theory, to provide statistical accuracy guarantees. Given an accuracy requirement from the user (e.g., 1% error bounds with 95% confidence interval), ApproxG executes more and more tasks across phases until it

is confident that the requirement is satisfied. We also show that although the idea can be easily applied to perform both edge-centric and vertex-centric task sampling, edge-centric sampling is superior because it needs a smaller number of phases and requires sampling fewer tasks to reach a given accuracy level. The second technique is cost-aware task sampling. The key insight is that in a distributed environment, tasks have non-uniform costs, as they may involve different amounts of remote communications. Hence, we should preferentially sample tasks of smaller costs and meanwhile guarantee that the non-uniform sampling does not introduce bias.

Though the proposed techniques are general enough for k-node graphlet counting, where $K > 4$, we focus on 3-node and 4-node graphlet counting, whose importance has been shown in various existing works [1], [7], [24].ApproxG takes advantage of the proposed techniques and reduces the query time from tens of hours to several minutes for multiple large graphs with marginal accuracy loss (<1% relative error).

We make the following contributions in this paper.

- We propose a simple but practical approach to slightly modify state-of-the-art graphlet counting algorithms to perform approximate computing, whose accuracy is controlled to provide user with statistical guarantees (Section III).
- We parallelize the exact and approximate versions of the algorithm in a distributed system, and propose cost-aware sampling to minimize inter-machine communications for better scalability (Section IV).
- We show through analysis and empirical results that edge-centric sampling typically outperforms vertex-centric sampling, demonstrating another benefit of edge-centric graph processing (Section V and VII).
- We integrate the proposed techniques in the framework, ApproxG, which demonstrates up to 3 orders of magnitude performance improvements on both single-node and distributed platforms with less than 1% average accuracy loss (Section VI and VII).

II. Background and Motivation

In this section, we formally define the graphlet counting problem, followed by a brief description of a state-of-the-art algorithm for exact graphlet counting. We then motivate the work by showing its unsatisfactory performance even if 16 machines are used, and the reasons that cause the inefficiency.

A. Problem Definition and The Exact Algorithm

We consider an undirected connected graph $G = (V, E)$ with vertex set V and edge set E. A subgraph $G' = (V', E')$ is an induced subgraph of G if the subgraph consists of a subset of nodes in G and all the edges that connect them in G (i.e., $V' \subset V$ and $E' = \{u, v \in V', (u, v) \in E\}$). A connected induced subgraph is also called a connected graphlet, which we will refer to as graphlets for short in the remainder of the paper. Figure 1 shows all 3- and 4-node graphlets. There are two 3-node graphlets and six 4-node graphlets. The number of distinct graphlets increases exponentially with the number of vertices in the graphlet. For instance, the number of connected distinct 5-node graphlets is 21, over 3× increase from the 4-node case. Hence, many applications only compute the frequencies for 3- and 4-node graphlets for computation efficiency [1], [20]. We follow those works and only consider 3- and 4-node graphlets.

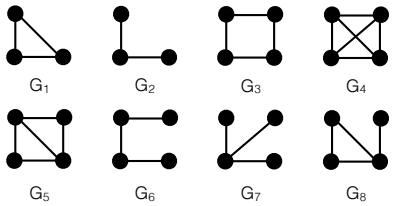


Figure 1: All 3- and 4-node connected graphlets.

Existing studies proposed various graphlet counting algorithms [1], [12], [20], [32]. Conceptually, many of the algorithms follow a structure described in Algorithm 1. The algorithm traverses the edge list, E, and accumulates the number of graphlets incident upon each edge into a result variable N. Before N is returned, it is divided by the number of edges in the graphlet, because it is over-counted by that many times. For instance, consider a graph that is a 4-node clique (G_4 in Figure 1). When an edge is processed, the clique contributes one to the increase in N. Hence, after all edges are processed N's value is 6, which should be divided by 6 (the number of edges in a 4-node clique) to get the correct result (1).

The proposed exact algorithms mainly focus on optimizing the work done by line 4 of Algorithm 1. As far as we know, the algorithm proposed by Ahmed et al. [1] has the lowest complexity and can be easily parallelized, and hence is used as the baseline in this paper. The algorithm directly computes the frequencies of G_1 (triangle), G_2 (wedge), G_3 (rectangle), and G_4 (4-node clique) in Figure 1, and the counting of the other 4-node graphlets depends on those 4 frequencies. Therefore, we only consider those directly counted graphlets.

Algorithm 1: Exact Graphlet Counting

```
   input : G = (V, E), Graphlet pattern: G_k
   output: N : the frequency of G_k.
1  begin
2      N ← 0;
3      for e in E do
4          T ← no. of graphlets incident upon e;
5          N ← N + T;
6      N ← N / (no. edges in G_k);
```

B. Scalability Problem

We implemented the exact algorithm proposed in [1] using OpenMP, which took 2 and 48 hours to count the frequencies of G_3 (rectangle) and G_4 (clique) in graphs LiveJournal and orkut, respectively, on one machine of a Blue Gene/Q cluster (details in Section VII). The result demonstrates that although the algorithm itself is efficient, the execution time can be excessively large due to the large data size.

In light of existing work that uses distributed systems to acclerate graph computation [9], [18], we used MPI to parallelize the algorithm to run on multiple machines, with vertices along with their adjacency lists randomly distributed.

We observe that using 16 machines only brings a 6.23X speedup over the single-machine run (details in Section VII). The source of the scalability issue is poor locality, which is observed in multiple existing studies [9], [15]. Although a more sophisticated graph partitioning approach may improve locality [13], [33], the partitioning may incur unacceptable overhead when the graph inputs are only available during runtime.

We need to alleviate two bottlenecks to substantially improve performance. The first bottleneck is absolute accuracy which may be an overkill in many applications. For example, graphlet counting can be used to compare networks [10], in which a small error (e.g., 1%) may not affect the final result. The second bottleneck is frequent remote accesses in a distributed system environment due to the irregularity of the input graph.

In the next two sections, we propose two approaches to addressing these bottlenecks. We leverage a simple technique from approximate computing to exploit the trade-off between performance and accuracy. We take advantage of the error tolerance and substantially reduce the number of remote communications.

III. Task Sampling with Statistical Accuracy Control

In this section, we apply task sampling to the exact graphlet counting algorithm to perform the approximate computation. We then employ sampling theory to determine how to sample tasks to satisfy user-defined accuracy requirements. We show a multi-phased sampling design, which can be easily implemented to handle real-world workloads.

A. Task Sampling

We view each iteration of the *for* loop in Algorithm 1 as a task, which computes the number of graphlets incident on the processed edge. Similar to a well-known technique called loop perforation [29], we can randomly skip some tasks (i.e., iterations) to reduce the execution time. In this paper, we call this technique *task sampling*. Algorithm 2 shows a simple way to modify Algorithm 1 to perform task sampling. The highlighted statements (lines 4–6 and 10) denote the difference from the exact algorithm. In each iteration, a random number is generated between 0 and 1, which is compared with the supplied sampling ratio P to determine if that task should be processed. Line 10 adjusts the output to estimate the frequency of the given graphlet pattern. The smaller P is, the more tasks are skipped, and hence the more we can reduce execution time.

Theorem 1 below shows that Algorithm 2 results in an unbiased estimate of the count of graphlet G_k in the graph.

Theorem 1. *The expected value of N computed in Algorithm 2 is equal to the frequency of graphlet G_k.*

Proof. Let T_i be the actual count of graphlet G_k associated with edge e_i processed in iteration i. Then the actual count of graphlets is given by $(\sum_i T_i)/|G_k|$, where dividing by $|G_k|$, denoting the number of edges in graphlet G_k, adjusts for overcounting.

Let X_i be the random variable denoting the contribution of edge e_i to N in the algorithm. Then $E[X_i] = (P \times T_i) + (1 - P) \times 0 = PT_i$. The expected number of graphlets counted within the for loop is given by $E[\sum_i X_i] = \sum_i E[X_i] = \sum_i PT_i = P \sum_i T_i$, with the first step following from the linearity of expectations. Dividing by $|G_k|$ and P proves the theorem. □

Algorithm 2: Approximate Graphlet Counting with Task Sampling

```
input : G = (V, E), Graphlet pattern G_k,
        sampling ratio P
output: N : the estimated frequency of G_k.
1  begin
2      N ← 0;
3      for e_i in E do
4          r ← uniform(0, 1);
5          if r > P then
6              continue;
7          T_i ← no. of graphlets incident upon e_i;
8          N ← N + T_i;
9      N ← N / (no. of edges in G_k);
10     N ← N / P;
```

Theorem 1 is interesting because it holds regardless of the degree distribution or the topology of the graph. However, in many real-world applications, an unbiased estimate is not enough, because if the variance is large, the accuracy of one particular run can be too poor to be useful. It is critical to allow the user to provide an accuracy requirement (e.g., 1% error bounds with 95% confidence interval), which should be enforced by the application. Next we show how to leverage sampling theory to reach this goal.

B. Multi-Phased Design for Error Control

Conceptually graphlet counting is performed by summing up a list of L non-negative integers, each of which corresponds to an edge and is equal to the number of graphlets incident upon that edge. Essentially, algorithm 2 samples a sub-list, whose sum is used to estimate the sum of the whole list. We define the estimated sum as S_a and the sum of the original list as S_r. The goal of error control is then to determine a sampling ratio such that $P(|S_a - S_r|/S_a \leq err_{tgt}) = 95\%$, where err_{tgt} is the target error and 95% is the confidence interval.

Given a sampling ratio P and a sampled sub-list with P, we can compute the estimated variance of S_a [31]:

$$var_{est} = \frac{L \times (1 - P) \times s^2}{P} \tag{1}$$

where s is the variance of the sampled sub-list, and the estimated error:

$$err_{est} = \frac{\sqrt{var_{est}} \times z_{\alpha/2}}{S_a} \tag{2}$$

where $z_{\alpha/2}$ represents the z-score at $\alpha/2$. For 95% confidence interval, $\alpha = 0.05$.

If $err_{est} > err_{tgt}$, we can derive a new sampling ratio P' to reduce the error by 1) replacing err_{est} by err_{tgt} in Equation 2, 2) replacing var_{est} according to Equation 1, and 3) solving the equation for P (the solution is P'), which leads to:

$$P' = \frac{z_{\alpha/2}^2 L s^2}{z_{\alpha/2}^2 L s^2 + S_a^2 err_{tgt}^2} \quad (3)$$

It is guaranteed that $P' > P$, because if s, var_{est} and S_a are treated as constants, err_{est} decreases with larger values of P according to Equation 1 and 2.

Given this understanding, we propose multi-phased sampling that works as follows. We set the initial sampling ratio at $P = \frac{1}{10000}$ chosen empirically with the goals of avoiding immediate over-sampling and selecting enough data to get an idea of the variance. We then invoke an iterative process that consists of three steps to determine the final sampling ratio:

Step 1: Take a sub-list sample based on P.

Step 2: Calculate the estimated error err_{est} based on Equation 1 and 2.

Step 3: If $err_{est} \leq err_{tgt}$, use P as the final sampling ratio. Otherwise, use Equation 3 to determine the new value for P, go to step 1.

This iterative approach retains its equivalence to uniform random sampling without replacement by performing the iterated sampling process on the previously unsampled data. For example, let the previous sampling ratio be P_1 and the new sampling ratio be P_2. The unsampled data elements will be selected in the next phase with probability $\frac{P_2 - P_1}{1 - P_1}$. The cumulative result is that any given data element will be selected after phase 1 is P_1 and the probability of any element being selected by the end of the second phase is $P_1 + (1 - P_1) \times \frac{P_2 - P_1}{1 - P_1} = P_2$.

We note that because the sampling ratio is monotone increasing by strides of 0.0001 and a sampling ratio of 1 produces the exact result, ApproxG will eventually reach the accuracy requirement, which needs at most 10,000 phases.

IV. Scaling Approximate Graphlet Counting to Multiple Machines

The last section describes the modified serial algorithm to compute the approximate frequency of graphlets. In this section, we first show how to parallelize the approximate algorithm in a distributed system. We then show the scalability problem of random task sampling. We propose cost-aware task sampling, which does not introduce bias and has the potential to substantially improve scalability.

A. Random Task Sampling and Its Scalability Problem

To parallelize Algorithm 2, we need to partition the graph among multiple machines. Partitioning graphs to minimize communication is an NP-hard problem [5]. Although there exist relatively fast heuristic partitioning algorithms, we are not aware of a fast and effective partitioning algorithm for *very large* graphs. Meanwhile, sophisticated graph partitioning algorithms may introduce too much overhead, outweighing their advantages. In this paper, we use a simple partitioning algorithm, which randomly distributes the vertices along with their adjacency lists to different machines. Each machine runs Algorithm 1 or Algorithm 2 to compute the frequency of graphlets in the partitioned sub-graph, and at the end we aggregate those partial frequencies to produce the final result.

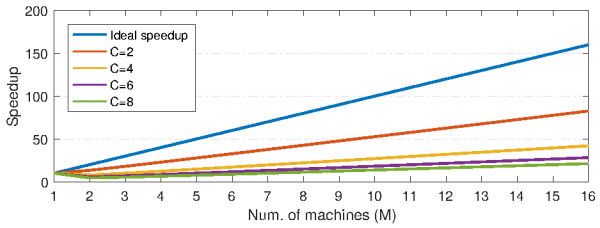


Figure 2: The scalability problem of random task sampling.

To illustrate the scalability problem of naive random sampling, consider approximate triangle counting on M machines with P as the sampling ratio. After distributing the graph data, we have two categories of edges. One category is *inter-machine edges*, whose incident vertices (and their adjacency lists) reside on two different machines; the other category is intra-machine edges, whose incident vertices reside on the same machine. Given an edge (u, v) and a machine A that owns vertex u, the processing of the edge on A needs to retrieve the adjacency list of v if it is an inter-machine edge (i.e., v is on a different machine). Assume that on average retrieving the adjacency list of a remote vertex is C times more expensive relative to retrieving the adjacency list of a local vertex. Since the vertices are randomly distributed, the probability for an edge to be an inter-machine edge is $(M-1)/M$, where M is the number of machines. Hence, the speedup limit of approximate graphlet counting by using M machines over the exact algorithm on one machine can be estimated as:

$$\begin{aligned} Speedup_{random} &= \frac{\text{cost of processing } |E| \text{edges in the single-machine setting}}{\text{cost of processing } \frac{|E| \times P}{M} \text{ edges in the distributed system setting}} \\ &= \frac{|E|}{\frac{|E| \times P}{M} \times (\frac{1}{M} + \frac{C \times (M-1)}{M})} \\ &= \frac{M^2}{P \times (CM - C + 1)} \end{aligned}$$

Figure 2 shows the estimated speedups given different numbers of machines and different values for C. The task sampling ratio is 10%, meaning that the ideal linear speedup that can be obtained from task sampling and parallel execution should be $10 \times M$. We observe that when C increases, the estimated speedup quickly decreases. When $C = 4$, the estimated speedup using 16 machines is about 1/4 of the ideal speedup (i.e., when $C = 1$). The performance even degrades when the number of machines (i.e., parallelism) is too small to compensate the cost of remote communications.

We implement micro-benchmarks to test the cost of remote communications on the Blue Gene/Q cluster (platform details in Section VII). The microbenchmark compares the execution times of reading a 1MB array from local DRAM and a remote machine. We found that the remote communications (memory accesses) are approximately 8X more costly than local

communications (memory accesses). The results, together with Figure 2, show that remote communications may seriously degrade the performance benefits from sampling.

B. Cost-Aware Task Sampling

The pitfall of naive random task sampling is that it does not consider non-uniform costs of tasks. A task that processes an inter-machine edge incurs greater cost compared with one that processes an intra-machine edge. To improve scalability, we propose a methodology for cost-aware task sampling that *crucially* does not impact the uniform random nature of the edge-dropping in Algorithm 2 assumed in Theorem 1.

Algorithm 3: Approximate Graphlet Counting with Cost-Aware Task Sampling

```
   input : G = (V, E), Graphlet pattern G_k, sampling ratio for
           single-machine processing P, no. of machines
           M(M > 1)
   output: N : the estimated frequency of G_k in G.
1  begin
2      N ← 0;
3      P' ← M * P;
4      for e in E do
5          r ← uniform(0, 1);
6          if e is an inter-machine edge then
7              if P' ≤ 1 or r > (M*P−1)/(M−1) then
8                  continue;
9          else
               /* e is an intra-machine edge     */
10             if r > P' then
11                 continue;
12         T ← no. of graphlets incident upon e;
13         N ← N + T;
14     N ← N / (no. of edges in G_k);
15     N ← N / P;
```

Our methodology is described in Algorithm 3, whose main idea is that intra-machine edges are preferentially sampled. We describe how the algorithm works: first it assumes that vertices and their adjacency lists are randomly assigned to the M machines as noted earlier. Any edge (u, v) is then an intra-machine edge with identical probability $1/M$. Hence, when the sampling probability $P \leq 1/M$, the expected number of intra-machine edges is larger than the expected number of sampled edges. In this case, the algorithm skips all tasks that process inter-machine edges (lines 7–8) and samples tasks that process intra-machine edges with probability $P'(P' = M \times P)$ (lines 10–11). When $P > 1/M(i.e., P' > 1)$, the algorithm processes all intra-machine edges (lines 10–11) and samples a portion of inter-machine edges (lines 6–8) determined by the sampling ratio as $\frac{M \times P - 1}{M - 1}$.

Lemma 2. *Algorithm 3 samples any edge with probability P.*

Proof. Recall that any edge (u, v) is intra-machine with probability $\frac{1}{M}$ and inter-machine with probability $\frac{M-1}{M}$ because of the uniform random assignment of vertices to machines.

Case 1: $0 \leq P \leq \frac{1}{M}$. Intra-machine edges are sampled with probability $P' = PM$ and inter-machine edges with probability 0. The probability that edge (u, v) is sampled is given by $(\frac{1}{M} \times PM) + (\frac{M-1}{M} \times 0) = P$.

Case 2: $\frac{1}{M} < P \leq 1$. Intra-machine edges are sampled with probability 1 and inter-machine edges with probability $\frac{PM-1}{M-1}$. The probability that edge (u, v) is sampled is given by $(\frac{1}{M} \times 1) + (\frac{M-1}{M} \times \frac{PM-1}{M-1})$, which is $\frac{1}{M} + \frac{PM-1}{M} = P$. □

Theorem 3. *The expected value of N computed in Algorithm 3 is equal to the frequency of graphlet G_k.*

Proof. The proof follows trivially from Lemma 2 and an argument identical to that used in the proof of Theorem 1. □

Given a specific partitioning of the input graph, the sampling in Algorithm 3 is dependent. For example, assume that all the three vertices of a triangle are placed on the same machine. If two edges of the triangle are sampled, the third one has a higher chance to be sampled. However, we stress that the dependence does not affect the major result of Theorem 3, because the random partitioning itself is part of the algorithm, as shown in Lemma 2.

For triangle and wedge counting, the processing of an edge only needs the adjacency lists of the two incident vertices. If the task sampling ratio (P) is smaller than $1/M$, cost-aware sampling successfully eliminates all remote communications. In the section VII, we will show that for most graphs the needed sampling ratio is indeed smaller than $1/M$ to satisfy the accuracy requirement (1% error bounds with 95% confidence interval). However, for four-node graphlets, to process an edge (u, v) on a machine that owns u, the algorithm has to retrieve all the adjacency lists of u's neighbors. On average Algorithm 3 can only reduce at most 1 remote retrieval, which is only modest improvement. One possible way to further reduce remote accesses is to consider the ratio of remote neighbors for each vertex, which will be explored in the future.

V. Vertex- vs. Edge-Centric Task Sampling

The previous two sections described an edge-centric task sampling approach, which samples tasks that process edges. Some graph processing frameworks take a vertex-centric approach, in which a task processes a vertex. Algorithm 4 shows that task sampling can be easily applied to sample vertex-related tasks. Lines 5–6 control the sampling, and if a task is skipped none of the edges incident on the vertex are processed.

The following theorem shows that, like the previous algorithms, this algorithm produces an unbiased estimate.

Theorem 4. *The expected value of N computed in Algorithm 4 is equal to the frequency of the graphlet G_k.*

We omit the proof, which is similar to that of Theorem 1.

It may seem that vertex-centric sampling should perform as well as edge-centric sampling, because with the same sampling ratio both approaches sample roughly the same percentage of tasks. However, recall that as described in Section III, given an accuracy requirement, the needed sampling ratio is dependent on the variance of the number of graphlets incident on an edge. Similarly, the sampling ratio of vertex-centric sampling

Algorithm 4: Vertex-Centric Approximate Graphlet Counting

input : $G = (V, E)$, Graphlet pattern G_k, sampling ratio P
output: N : the estimated frequency of G_k.

```
1  begin
2      N ← 0;
3      for v in V do
4          r ← uniform(0, 1);
5          if r > P then
6              continue;
7          for e in v.incident_edges() do
8              T ← no. of graphlets incident on e;
9              N ← N + T;
10     N ← N / (no. of edges in G_k);
11     N ← N / P;
```

Table I: Maximum number of graphlets incident on an edge or vertex (D denotes the maximum degree of the input graph)

Graphlet	max. no. of graphlets incident on an edge	max. no. of graphlets incident on a vertex
Wedge	$2 \times (D-1)$	$\binom{D}{2}$
Triangle	$D-1$	$\binom{D}{2}$
Rectangle	$(D-1)^2$	$\binom{D}{2}$
4-node Clique	$\binom{D-1}{2}$	$\binom{D}{3}$

depends on the variance of the number of graphlets incident on a vertex. The larger the variance is, the larger the needed sampling ratio becomes. While the variance depends on the input graph, we analyze the range of the number of graphlets incident on an edge or a vertex. As shown by Hozo et al. [11], a larger range typically implies a larger variance. The minimum number of graphlets incident on an edge or a vertex is 0. Table I shows the maximum possible number of graphlets incident on an edge or a vertex. Observe that the range for vertex-centric sampling is much larger than that for edge-centric sampling except for rectangle. We hence speculate that edge-centric sampling is superior to vertex-centric sampling, which will be empirically evaluated in the next section.

VI. Implementation of the ApproxG Framework

This section describes the major components and the workflow of the ApproxG framework, followed by the presentation of three optimizations to improve its performance.

Figure 3 shows the workflow of the ApproxG framework. The user specifies the graph location, the accuracy requirement, the graphlet pattern, and the initial sampling ratio in the configuration file. After ApproxG partitions the graph, each machine maintains a list of local vertices and a list of remote vertices. Each of the OpenMP threads runs Algorithm 3 to traverse an edge chunk owned by the host machine and maintains the number of incident graphlets to each processed edge and their mean and variance. At the end of the edge processing, the machines reach a global synchronization to exchange the local statistics and use the techniques described in Section III to determine whether the accuracy requirement

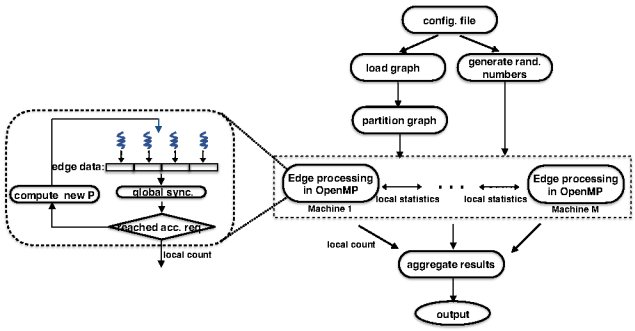


Figure 3: Illustration of the ApproxG framework.

has been satisfied. The local count is sent to the master machine if that is the case. Otherwise, every machine computes a new sampling ratio to traverse all edges again.

Besides sampling, ApproxG implements three optimizations to improve performance.

Reducing the overhead of random number generation: According to Algorithm 3, ApproxG needs to determine whether to process an edge depending on a sampling ratio. Hence, a naive implementation may generate $|E| \times L$ random numbers, where $|E|$ is the number of edges and L is the number of phases. Given the large value for $|E|$ of large graphs, the incurred overhead can be prohibitive. ApproxG reduces the overhead through two approaches. First, ApproxG generates a random number in the range $[0, 1)$ for each edge in parallel with graph loading. Second, in phase $i (i > 0)$, ApproxG processes an edge only if its corresponding random number r satisfies the following condition: $P_{i-1} \leq r < P_i$, where P_i is the sampling ratio in phase i and P_0 is the initial sampling ratio. Therefore, ApproxG only generates $|E|$ random numbers, whose overhead is removed from the critical path.

Reducing space overhead of the maintained statistics: In the distributed setting, ApproxG adopts the hybrid processing model. It uses OpenMP threads to process the edges and MPI to handle inter-machine communication. Each thread processes a large number of edges and needs to compute the variance and mean values for the counts corresponding to the sampled edges. To save space overhead, ApproxG implements an online algorithm [30], which uses only one variable for each statistic.

Reducing remote communication cost: The processing step for any particular edge requires data which may not be available on the local machine. Instead of requiring the remote data immediately to perform the computation, ApproxG instead defers the processing of that edge until all the required remote data is identified. The host of the required data may then send a large buffer containing all the needed data to avoid the otherwise expensive round trip message costs.

VII. Evaluation

In this section, we evaluate ApproxG on real-world graphs. We first explore the trade-off between performance and accuracy. We then evaluate the multi-phased design on a single machine. We use a cluster system to show the improved scalability brought by cost-aware sampling.

Table II: Real-world graphs used in the experiments.

graph	$\|V\|$	$\|E\|$	Size
LiveJournal	4M	35M	550MB
arabic-2005	23M	554M	9.1GB
dewiki-2013	1.5M	33M	474MB
enwiki-2013	4.2M	92M	1.4GB
eu-2015-host	11M	261M	4.1GB
eu-2015-tpd	6.7M	112M	1.7GB
gsh-2015-tpd	31M	490M	8.2GB
hollywood-2009	1.1M	56M	763MB
ljournal-2008	5.4M	50M	789MB
orkut	3.1M	117M	1.7GB
uk-2002	18M	262M	4.3GB

A. Methodology

Programs. We parallelize the exact algorithm proposed in [1] using OpenMP and MPI. OpenMP provides parallelism within one machine and MPI enables inter-machine communications. To minimize the overhead of graph partitioning, ApproxG randomly partitions the vertices along with their adjacency lists to different machines. We parallelize the edge and vertex traversals, respectively, for edge-centric and vertex-centric task sampling. For each exact algorithm, we implement the approaches proposed in previous sections to create the corresponding approximate programs, with cost-aware task sampling enabled by default. For each run of ApproxG that computes the frequencies of multiple graphlets, the accuracy requirement is enforced for all the computed graphlets.

Graph inputs. Table II shows the real-world graphs used in the experiments. LiveJournal and Orkut come from the Stanford's SNAP datasets [17]. The other graphs come from the Laboratory for Web Algorithmics at UNIMI [4]. LiveJournal and ljournal-2008 were extracted from the social network LiveJournal in 2006 and 2008 respectively. Orkut is an online social network. Hollywood-2008 represents actors who co-appear in movies. Wikipedia's German and English pages create the dewiki-2013 and enwiki-2013 graphs respectively. The remaining datasets come from web crawls looking at all pages (no suffix), top private domains (-tpd) or hosts (-host). Since the exact 4-node graphlet counting program requires more than 48 hours to process any of arabic-2005, enwiki-2013, eu-2015-host, eu-2015-tpd, and gsh-2015-tpd on one single BGQ machine, we exclude them for the 4-node graphlet counting experiments. We also point out that the sizes of the graph inputs are larger than those used in most related work.

Parameters and metrics. We set the default accuracy requirement as 1% error with 95% confidence. We use execution time and the error of the estimated frequency to evaluate the approximate programs. The error is defined as $error = \frac{|F_a - F_r|}{F_r}$, where F_a is the approximate frequency and F_r is the real frequency (ground truth). For an approximate program that computes frequencies of multiple graphlet patterns, we report the maximum error over all the graphlets.

Machine environment. The experimental platform is an IBM BlueGene/Q cluster with identical machines connected to the 5-D torus interconnect network. Each machine offers 16 cores and 16 GB main memory. The operating system is Linux (kernel version 2.6.32). The compiler is IBM XLC (version 12.1) used with optimization level O4. The operating system is Ubuntu 14.04.

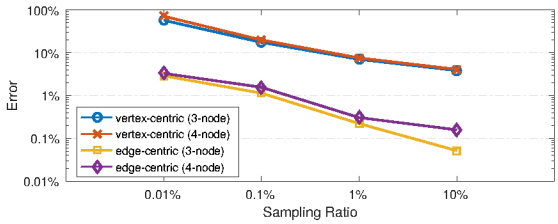
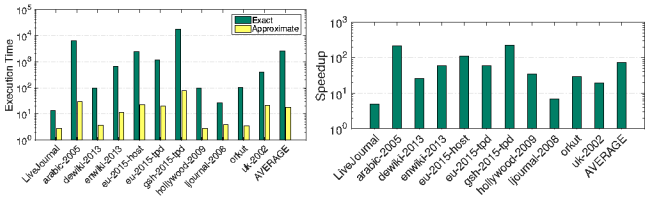


Figure 4: Relation between sampling ratio and accuracy.



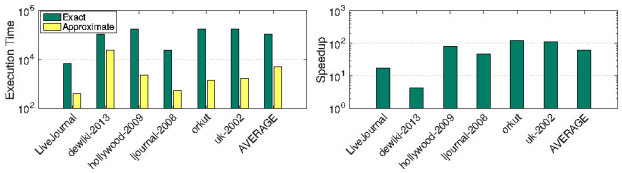
(a) Execution time (b) Speedups

Figure 5: Execution times of the exact and approximate 3-node graphlet counting on a single machine. The speedup of approximate counting over the exact version reaches as much as 224X.

B. Results on a Single Machine

Figure 4 demonstrates the relationship between sampling ratio and accuracy for both edge-centric and vertex-centric task sampling. Each data point represents the geometric mean error for all input graphs. As expected, error decreases as we increase the sampling ratio. Observe that the performance potential for the edge-centric sampling approach is substantial because with 1% sampling, the errors for 3-node and 4-node graphlets are only 0.2% and 0.3%, respectively. Even if only 0.1% of the edges are sampled, the errors are less than 1.6%. However, the vertex-centric approach shows a trade-off which is much more difficult to exploit. When 1% of the vertices are sampled, the errors for 3-node and 4-node graphlet are around 7.1% and 7.6%, respectively, which many applications may not tolerate. If we decrease the sampling ratio to 0.1%, the error for 3-node graphlets jumps to 17.6%, which is 15.4X higher than that of edge-centric sampling. The results echo the speculation in Section V, and we conclude that edge-centric sampling is superior to vertex-centric sampling. Hence, we only evaluate edge-centric sampling in the remaining experiments.

Figure 5 (a) shows the single-machine execution times of the exact and approximate 3-node graphlet counting programs. We observe substantial performance improvement for all 11 graphs. One trend is that larger graphs tend to benefit more from task sampling. For example, the two largest graphs, gsh-2015-tpd and arabic-2005, enjoy the largest speedups, 212X and 224X, respectively as shown in Figure 5 (b). The exact program needs 17,032 seconds to process gsh-2015-tpd, for which the approximate program samples 0.25% of the edges



(a) Execution time (b) Speedups

Figure 6: Execution times of the exact and approximate 4-node graphlet counting on a single machine. The speedup of approximate counting over the exact version reaches as much as 118X.

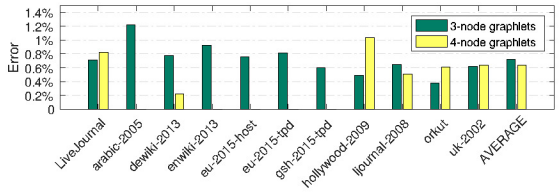


Figure 7: Relative errors produced by the approximate programs.

and takes only 76 seconds. ApproxG reduces the average processing time for a graph from 2591 seconds to 179 seconds. Figure 6 (a) shows the execution times for both the exact and approximate programs for 4-node graphlet counting, which follows a similar trend as in Figure 5 (a). The large graphs benefit more from the approximate program as Figure 6 (b) shows. The approximate program provides an average of 62X speedup and as much as 118X performance improvement for Orkut.

Figure 7 shows the relative errors of the approximate programs with respect to the ground truth. One bar represents the maximum error of the graphlet counts from ApproxG runs on a graph. Recall that we exclude 5 graphs for 4-node graphlet counting, so the figure omits 5 bars. Observe that the multi-phased sampling controls the errors very well, yielding on average 0.72% and 0.64% errors for 3-node and 4-node graphlet counting, respectively. For arabic-2005, the errors can be up to 1.2%, which is larger than the 1% error bound. However, note that the error bound is with the 95% confidence interval, so it is a statistical guarantee. Among the 34 estimated frequencies the error bound is only violated 2 times, and the second largest error is 1.03% for clique counting in hollywood-2009. Therefore, we conclude that the multi-phased sampling is effective in terms of providing substantial performance improvement as well as statistical error control.

Figure 8 shows the average speedup results for all graphs with different error requirements at 95% confidence. Interestingly, we observe that the speedup for 4-node graphlet counting increases much more quickly than that for 3-node graphlet counting. With similar speedups at a 1% error bound, the speedup difference widens to be 4.6X when the error bound is 4%. One plausible reason is that the 4-node graphlet counting is more compute-bounded compared with 3-node graphlet counting, and hence benefits more from running fewer tasks.

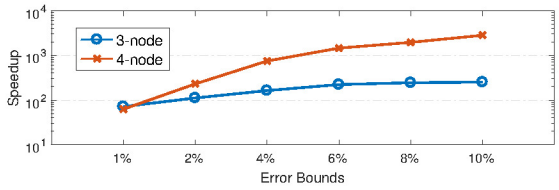


Figure 8: Performance improvement for different accuracy requirements.

Discussion. It may seem surprising that ApproxG achieves $< 1\%$ accuracy loss when sampling less than 1% of the edges. As Freedman et al. point out [6], What fundamentally matters for the variability of a statistic from a random sample is the sample's absolute size rather than its relative size relative to the population. For orkut, ApproxG samples 0.3% of the edges, but because the graph is large, the sample size is around 0.35M. The results demonstrated by ApproxG and the counter-intuitive relation between sampling fraction and variability of statistics indicate a tremendous opportunity for approximate computing for big graph applications.

C. Results on Multiple Machines

Figure 9 shows the performance improvements of both the exact and approximate programs in the distributed system with all graphlets counted in each run. The baseline is the execution time of the exact programs on a single machine. For each setting, we plot the average speedup for all graphs and the error bar, which shows the minimum and maximum speedup. Observe that as the number of machines increases the exact program produces better performance, which aligns well with the research efforts in distributed graph processing. However, due to frequent remote communication, the scalability is far from ideal. When using 16 machines, the exact program runs only 6.23X faster. The approximate program improves the scalability by discarding most of the computation and sampling low-cost tasks. By using 16 machines, the approximate program yields an average of 480X speedup (up to 1061X speedup) over the exact program running on a single machine. Observe that there still exists a gap between the performance produced by the approximate program and the ideal linear speedup when multiple machines are used. The reason is that the sampled tasks for the approximate program still incur a non-trivial amount of remote communications, which bounds the maximum possible performance improvement.

Table III reports the relative errors of the distributed approximate program with respect to the ground truth. Observe that for all six machine configurations, the average error is smaller than 1%, empirically confirming that neither the parallelization nor cost-aware dropping introduces bias. Similar to the error results of the single-machine experiments, the error can be larger than 1%, which happens 2 times for all 144 estimated frequencies.

Figure 10 shows the benefit of cost-aware task sampling over random task sampling for 3-node graphlet counting. Since for most runs the sampling ratio is smaller than $1/M$, where M is the number of machines, cost-aware task sampling eliminates all remote communications in those cases (except for final statistics). Overall, cost-aware task sampling produces

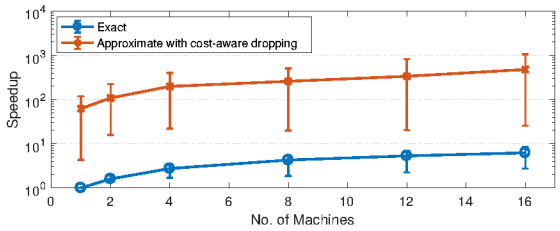


Figure 9: Scalability of exact counting and approximate counting.

Table III: Maximum error values for different machine counts

Machines	1	2	4	8	12	16
min(error)	0.22%	0.27%	0.091%	0.38%	0.14%	0.43%
avg(error)	0.64%	0.65%	0.39%	0.61%	0.41%	0.62%
max(error)	1.03%	1.21%	0.68%	0.77%	0.84%	0.81%

on average 1.37X performance improvement over random task sampling, which means an extra 130× speedup over the exact program. Note that the speedup is far from 8× (i.e., the cost of remote communication), because ApproxG just samples a small portion of the edges, making other parts of the program, such as traversing random numbers, more significant. Theoretically, when more machines are used, the performance of random task sampling should degrade. The reason is that the probability for an edge to be an inter-machine edge increases, and hence the randomly sampled tasks should incur more remote communications. However, we do not observe larger improvement from cost-aware task sampling with more machines. A plausible reason is that several other factors may affect performance, including computation-communication overlapping and load imbalance (due to the random graph partitioning). We point out that the Blue Gene/Q cluster has highly optimized hardware to support remote communications, which is not available in most commodity-hardware-based distributed systems. Therefore, we expect that task-aware sampling would perform significantly better than random sampling in latter systems.

Discussion. Unlike many distributed graph processing systems, ApproxG does not optimize load balance because of two reasons. First, with cost-aware sampling, ApproxG already reduces the processing time to the order of seconds or minutes. However, a partitioning algorithm for load balance may take even longer than the current processing time of ApproxG. For instance, on one Blue Gene/Q node, PowerGraph [9] takes more than 3 minutes to partition Arabic-2005, while ParMETIS [16], a popular multi-threaded graph partitioning library, takes more than 15 minutes. Second, the random partitioning in ApproxG simplifies the enforcement of statistical accuracy guarantees as shown in Algorithm 3.

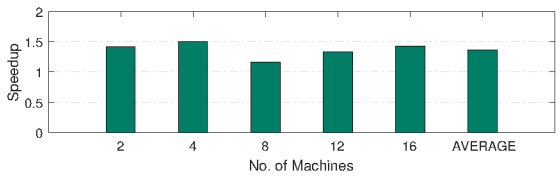


Figure 10: Performance improvement of cost-aware sampling over random sampling.

VIII. Related Work

In this paper, we combine parallel computing and approximate computing to produce multiplicative performance improvement for graphlet counting with statistical accuracy guarantees. Next, we briefly describe some closely related work.

Graphlet counting. Researchers have proposed various approaches to count or estimate the frequencies of graphlets. Ahmed et al. [1] proposed an algorithm, the one this paper uses as the baseline, to count all 3- and 4-node graphlets, which significantly improved the time complexity compared to other franeworks [12], [20], [32]. Orca [12] is a state-of-the-art graphlet counting framework widely used in Bioinformatics, which is however substantially outperformed by Ahmed's algorithm. Shun and Tangwongsan [28] designed a cache-friendly exact triangle counting algorithm, which achieved up to 50X speedup over previous exact triangle counting algorithms on a 40-core machine. They further proposed an approximate version of the algorithm based on random color assignment to vertices. Only edges whose incident vertices have the same color are processed. Although the estimate is unbiased, their approach does not provide any accuracy guarantee. Graft [23] is a framework to estimate both 3- and 4-node graphlets, but similar to Shun and Tangwongsan's work it does not provide statistical accuracy guarantees. In addition, Graft does not support parallel processing. These exact graphlet counting approaches are suitable for applications that can not tolerate errors in the results.

Parallel graph processing. Scalable distributed graph processing is challenging due to the irregularity of the data inputs and the high overhead of remote data accesses. PowerGraph [9], the core technique of GraphLab [18], proposed a novel vertex cut based partitioning algorithm to reduce remote data accesses. GPS [26] is an open-source implementation of Google's Pregel [19], which partitions the adjacency list of high-degree vertices. For each partition, a mirror of the vertex is created to optimize remote data accesses. In this paper, we show that error tolerance provides an extraordinary opportunity to eliminate or substantially reduce remote data accesses.

Approximate computing. Multiple studies in the approximate computing area endeavor to change existing applications to trade off accuracy for great reduction on execution time. Sidiroglou et al. [29] used iteration skipping for application in various domains and demonstrated substantial performance improvement. ApproxHadoop [8] combined task dropping with data sampling and provides statistical error bounds. However, the work mainly considered regular applications and did not optimize remote communications in particular. Shang and Yu [27] proposed a compiler-based approach to enable automatic approximate graph computation. Although the work involved triangle counting, it did not provide accuracy guarantee. Moreover, they did not study the connection between approximation and scalability in a distributed system.

IX. Conclusion

In this paper, we presented a framework named ApproxG to apply an approximate computing technique, called task sampling, to graphlet counting. We proposed simple, yet practical approaches to slightly modify an exact algorithm to perform approximate computation with controllable accuracy. Those approaches can be easily implemented in both vertex-centric and edge-centric processing models, but we demonstrated that edge-centric sampling performs substantially better than vertex-centric sampling. To scale the computation in a distributed system, we proposed cost-aware task sampling, which preferentially samples tasks that involve fewer remote communications. The experiments on 11 real-world graphs validated the effectiveness of the proposed approaches, which showed the great potential of approximate computing for graph computation.

X. Acknowledgement

We thank the anonymous reviewers for their insightful comments and suggestions. The effort of this project is funded by National Science Foundation Grants 1464216 and 1618912.

References

[1] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick G. Duffield. Efficient graphlet counting for large networks. In *2015 IEEE International Conference on Data Mining, ICDM 2015, Atlantic City, NJ, USA, November 14-17, 2015*, pages 1–10, 2015.

[2] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph-based anomaly detection and description: A survey. *CoRR*, abs/1404.4679, 2014.

[3] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 16–24, 2008.

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, pages 587–596. ACM Press, 2011.

[5] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Inf. Process. Lett.*, 42(3):153–159, May 1992.

[6] Roger Purves David Freedman, Robert Pisani. *Statistics (4th edition)*. W.W. Norton & Company, 2007.

[7] Talya Eden, Amit Levi, Dana Ron, and C. Seshadhri. Approximately counting triangles in sublinear time. In *FOCS*, 2015.

[8] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 383–397, 2015.

[9] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, 2012.

[10] Wayne Hayes, Kai Sun, and Natasa Przulj. Graphlet-based measures are suitable for biological network comparison. *Bioinformatics*, 29(4):483–491, 2013.

[11] Stela Pudar Hozo, Benjamin Djulbegovic, and Iztok Hozo. Estimating the mean and variance from the median, range, and the size of a sample. *BMC Medical Research Methodology*, 5(1):13, 2005.

[12] Tomaž Hočevar and Janez Demšar. A combinatorial approach to graphlet counting. *Bioinformatics*, 30(4):559–565, 2014.

[13] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, January 1998.

[14] Dhanya R. Krishnan, Do Le Quoc, Pramod Bhatotia, Christof Fetzer, and Rodrigo Rodrigues. Incapprox: A data analytics system for incremental approximate computing. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 1133–1144, 2016.

[15] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, 2012.

[16] Dominique Lasalle and George Karypis. Multi-threaded graph partitioning. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 225–236, 2013.

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

[18] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[19] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146, 2010.

[20] D. Marcus and Y. Shavitt. Rage - a rapid graphlet enumerator for large networks. *Comput. Netw.*, 56(2):810–819, February 2012.

[21] M. E. J. Newman. The structure and function of complex networks. *SIAM REVIEW*, 45:167–256, 2003.

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

[23] Mahmudur Rahman, Mansurul Bhuiyan, and Mohammad Al Hasan. Graft: An approximate graphlet counting algorithm for large graph analysis. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 1467–1471, 2012.

[24] Ryan A. Rossi and Rong Zhou. Leveraging multiple gpus and cpus for graphlet counting in large networks. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, pages 1783–1792, 2016.

[25] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 472–488, 2013.

[26] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, 2013.

[27] Zechao Shang and Jeffrey Xu Yu. Auto-approximation of graph computing. *Proc. VLDB Endow.*, 7(14), October 2014.

[28] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 149–160, 2015.

[29] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, 2011.

[30] Phil Spector. *Introduction to S and S-Plus*. Wadsworth Publ. Co., Belmont, CA, USA, 1st edition, 1995.

[31] Steven K. Thompson. *Sampling (3rd Edition)*. WILEY, 2012.

[32] Sebastian Wernicke and Florian Rasche. Fanmod: A tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, May 2006.

[33] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[34] Luming Zhang, Mingli Song, Zicheng Liu, Xiao Liu, Jiajun Bu, and Chun Chen. Probabilistic graphlet cut: Exploiting spatial structure cue for weakly supervised image segmentation. In *CVPR*, 2013.