Finding and counting subgraphs using MapReduce

Zhao Zhao, Meng Li, Guanying Wang, Ali Butt, Maleq Khan,
Madhav Marathe, Judy Qiu, Anil Vullikanti

Abstract—Several variants of the subgraph isomorphism problem, e.g., finding, counting and estimating frequencies of subgraphs in networks arise in a number of real world applications, such as genetic network analysis in bioinformatics, web analysis, disease diffusion prediction and social network analysis. These problems are computationally challenging to scale to very large networks with millions of nodes. In this paper, we present SAHAD, a MapReduce based algorithm for detecting and counting trees of bounded size using the elegant color coding technique, developed by N. Alon, R. Yuster and U. Zwick, Journal of the ACM (JACM) 1995. SAHAD is a randomized algorithm, and we show rigorous bounds on the approximation quality and the performance. We implement SAHAD on two different frameworks: the standard Hadoop model and Harp, a more high performance computing environment, and evaluate its performance on a variety of synthetic and real networks. SAHAD scales to very large networks comprising of $10^7 - 10^8$ nodes and $10^8 - 10^9$ edges and tree-like (acyclic) templates with up to 12 nodes. Further, we extend our results by implementing our algorithm using the Harp framework. The new implementation gives an order of magnitude improvement in performance over the standard Hadoop implementation.

Index Terms—subgraph isomorphism, graph partitioning, MapReduce, Hadoop, Harp

1 Introduction

Given two graphs G and H, the subgraph isomorphism problem asks if H is isomorphic to a subgraph of G. The counting problem associated with this seeks to count the number of copies of H in G. These and other variants are fundamental problems in Network Science and have a wide range of applications in areas such as bioinformatics, social networks, semantic web, transportation and public health. Analysts in these areas search for meaningful patterns in networked data; often, these patterns are specific subgraphs such as trees. Three different variants of subgraph analysis problems have been studied extensively. The first version involves counting specific subgraphs, which has applications in bioinformatics [5], [17]. The second involves finding the most frequent subgraphs either in a single network or in a family of networks—this has been used in finding patterns in bioinformatics (e.g., [21]), recommendation networks [22], chemical structure analysis [30], and detecting memory leaks [25]. The third involves finding subgraphs which are either over-represented or under-represented, compared to random networks with similar properties—such subgraphs are referred to as "motifs". Milo et al. [26] identify motifs in many networks, such as protein-protein interaction (PPI) networks, ecosystem food webs and neuronal connectivity networks. Subgraph counts have also been used in characterizing networks [28].

Subgraph Isomorphism problem and its variants are well known to be computationally challenging. In general the decision version of the problem is NP-hard, and the counting problem is #P-hard. Extensive work has been done in theoretical computer science on this problem; we refer the reader to the recent papers by [11], [13], [24] for an extensive discussion on the decision and counting complexity of the problem and tractable results for various parameterized versions of the problem.

The primary focus of this paper is on the three mentioned variants of the subgraph isomorphism problem when k, the number of nodes in H, is fixed. Letting n be the number of nodes in G, one can immediately get simple algorithms with running time $O(n^k)$ to find and count the number of copies of template H in G. When the template is a tree or has a bounded treewidth, Alon *et al*. [5] present an elegant randomized approximation algorithm with running time $O(k|E|2^k e^k \log(1/\delta)\frac{1}{\varepsilon^2})$, where ε and δ are error and confidence parameters, respectively, based on the color coding technique. There result was significantly improved by Koutis and Williams [20] who gave an algorithm with running time of $O(2^k|E|)$.

A lot of practical heuristics have also been developed for various versions of these problems, especially for the frequent subgraph mining problem. An example is the "Apriori" method, which uses a level-wise exploration of the template [19], [21], for generating candidates for subgraphs at each level; these have been made to run faster by better pruning and exploration techniques, e.g., [16],

- *Zhao Zhao, Ali Butt, Madhav Marathe and Anil Vullikanti are with the Network Dynamics and Simulation Science Laboratory, Biocomplexity Institute & Department of Computer Science, Virginia Tech, VA, 24061. E-mail: zhaozhao@vt.edu, butta@cs.vt.edu, mmarathe@vt.edu, vsakumar@vt.edu*
- *Maleq Khan is with the Department of Electrical Engineering and Computer Science, Texas A&M University-Kingsville. E-mail: maleq.khan@tamuk.edu*
- *Meng Li is with the Computer Science Department, Indiana University. Email: li526@umail.iu.edu*
- *Judy Qiu is with the Intelligent Systems Engineering Department, Indiana University. Email: xqiu@indiana.edu*
- *Guanying Wang is working with Google Inc. Email:wang.guanying@gmail.com*

[21], [40]. Other approaches in relational databases and data mining involve queries for specific labeled subgraphs, and have combined relational database techniques with careful depth-first exploration, e.g., [9], [31], [32].

Most of these approaches are sequential, and generally scale to modest size graphs G and templates H. Parallelism is necessary to scale to much larger networks and templates. In general, these approaches are hard to parallelize as it is difficult to decompose the task into independent subtasks. It is not clear if candidate generation approaches [16], [21], [40] can be parallelized and scaled to large graphs and computing clusters. Two recent approaches for parallel algorithms, related to this work, are [9], [41]. The approach of Bröcheler *et al*. [9] requires a complex preprocessing and enumeration process, which has high end-to-end time, while the approach of [41] involves an MPI-based implementation with a very high communication overhead for larger templates. Two other papers [27], [36] develop MapReduce based algorithms for approximately counting the number of triangles with a work complexity bound $O(|E|)$. The development of parallel algorithms for subgraph analysis with rigorous polynomial work complexity, which are implementable on heterogeneous computing resources remains an open problem. Due to the complexity of enumerating subgraphs, people propose to compute some metrics of the subgraph which is anti-monotune to the subgraph size. The algorithm reported in [3] is capable of computing subgraph support on large network with up to 1 Billion edges. However, it requires each machine to have a copy of the graph in memory which limits its scalability to larger graphs. Additionally, computing support requires much less computational effort than counting subgraphs. Another recent work also employs MapReduce to match subgraph [35] which scales to networks with up to 300 million edges.

Other approaches studied in the context of data mining and databases, e.g., [9], [31], [32], are capable of processing large networks, but are usually slow due to limitations of database techniques for processing networks.

Our contributions. In this paper, we present SAHAD, a new algorithm for Subgraph Analysis using Hadoop, with rigorously provable polynomial work complexity for several variants of the subgraph isomorphism problem when H is a tree. SAHAD scales to very large graphs, and because of the Hadoop implementation, runs flexibly on a variety of computing resources, including Amazon EC2 cloud. Our specific contributions are discussed below.

1. SAHAD is the first MapReduce-based algorithm for finding and counting labeled trees in very large networks. The only prior Hadoop based approaches have been on triangles [27], [36], [37] in very large networks, or more general subgraphs on relatively small networks [23]. Our main technical contribution is the development of a Hadoop version of the *color coding* algorithm of Alon *et al*. [5], [6], which is a (sequential) randomized approximation algorithm for subgraph counting. It is a randomized approximation algorithm that for any ε, δ, gives a $(1\pm\varepsilon)$ approximation to the number of embeddings with probability at least $1 - 2\delta$. We prove that the work complexity of SAHAD is $O(k|E_G|2^{2k}e^k \log(1/\delta)\frac{1}{\varepsilon^2})$, which is more than the running time of the sequential algorithm of [5] by just a factor of 2^k.

2. We demonstrate our results on instances generated using the Erdös-Renyi random graph model, the Chung-Lu random graph model and on synthetic social contact graphs for Miami city and Chicago city (with 52.7 and 268.9 million edges, respectively), constructed using the methodology of [8]. We study the performance of counting unlabeled/labeled templates with up to 12 nodes. The total running times for templates with 12 nodes on Miami and Chicago networks are 15 and 35 minutes, respectively; note that these are the *total end-to-end* times, and do not require any additional pre-processing (unlike, e.g. [9]).

3. We discuss how our basic algorithms for counting subgraphs can be extended to compute supervised motifs and graphlet frequency distributions. They can also be extended to count labeled subgraphs.

4. SAHAD runs easily on heterogeneous computing resources, e.g., it scales well when we request up to 16 nodes on a medium size cluster with 32 cores per node. Our Hadoop based implementation is also amenable to running on public clouds, e.g., Amazon EC2 [7]. Except for a 10-node template which produces extremely large amount of data so as to incur the I/O bottle neck on the virtual disk of EC2, the performance of SAHAD on EC2 is almost the same as on the local cluster. This would enable researchers to perform useful queries even if they do not have access to large resources, such as those required to run previously proposed querying infrastructures. We believe this aspect is unique to SAHAD and lowers the barrier-to-entry for scientific researchers to utilize advanced computing resources.

5. We study the performance improvement in extensions of the standard Hadoop framework. The enhanced algorithm is called EN-SAHAD. First, we consider techniques to explicitly control the sorting and inter partition communications in Hadoop. We find that reducing the sorting step by pre-allocating can improve the performance by about 20%, but improved partitioning does not seem to help.

6. Finally, we implement SAHAD within the Harp [29] framework – the new algorithm is called HARP-SAHAD. HARP-SAHAD yields an order of magnitude improvement in performance, as a result of its flexibility in task scheduling, data flow control and in memory cache. We are able to scale to networks with up to hundreds of millions of edges using the HARP-SAHAD with unlabeled templates with 7 nodes.

Organization. Section 3 introduces the background for the subgraph counting problem and MapReduce, the open-sourced implementation Hadoop and the Harp system. Then in Section 4, we give a brief overview of the color coding algorithm proposed by Alon *et. al* in [5]. Then in Section 5 we present our MapReduce implementations. In Section 6 we study the computation cost of our algorithm. Section 7 proposes several variations of the subgraph counting problems that can be computed using our framework. Section 8 discusses experiment results of SAHAD, EN-SAHAD and HARP-SAHAD. Finally, Section 9 concludes the paper.

Extension from conference version. The SAHAD algorithm appeared in [42]. The results on EN-SAHAD and HARP-SAHAD are new additions. Since the publication of [42],

there has been more work on parallelizing the color coding technique, e.g., [33], [34]. However, none of these have been based on MapReduce and its generalizations.

2 Related Work

As mentioned earlier, the subgraph isomorphism problem and its variant has been studied extensively by theoretical computer scientists; see [11], [13], [14], [18], [24], [38] for complexity theoretic results. Marx and Pilipczuk [24] undertake a comprehensive study of the decision problem and provide strong lower bounds including fixed parameter intractability results. They also study the complexity of the problem as a function of structural properties of G and H.

A variety of different algorithms and heuristics have been developed for different domain specific versions of subgraph isomorphism problems. One version involves finding frequent subgraphs, and many approaches for this problem use the Apriori method from frequent item set mining [15], [19], [21]. These approaches involve candidate generation during a breadth first search on the subset lattice and a determination of the support of item sets by subset test. A variety of optimizations have been developed, e.g., using a DFS order to avoid the cost of candidate generation [16], [40] or pruning techniques, e.g., [21]. A related problem is that of computing the "graphlet frequency distribution", which generalizes the degree distribution [28].

Another class of results for frequent subgraph finding is based on the powerful technique of "color coding" (which also forms the basis of our paper), e.g., [5], [17], [41], which has been used for approximating the number of embeddings of templates that are trees or "tree-like".

In [5], Alon *et al*. use color coding to compute the distribution of treelets with sizes 8, 9 and 10, on the protein-protein interaction networks of Yeast. The color coding technique is further explored and improved in [17], in terms of worst case performance and practical considerations. For example, by increasing the number of colors, they speed up the color coding algorithm with up to 2 orders of magnitude. They also reduce the memory usage for minimum weight paths finding, by carefully removing unsatisfied candidates, and reducing the color set storage.

Most of these approaches in bioinformatics applications involve small templates, and have only been scaled to relatively small graphs with at most 10^4 nodes (apart from [41], which shows scaling to much larger graphs by means of a parallel implementation). Other settings in relational databases and data mining have involved queries for specific labeled subgraphs. Some of the approaches for these problems have combined relational database techniques, based on careful indexing and translation of queries, with such depth-first exploration strategy that is distributed over different partitions of the graph e.g., [9], [31], [32], and scale to very large graphs. For instance, Bröcheler *et al*. [9] demonstrate labeled subgraph queries with up to 7-node templates on graphs with over half a billion edges, by carefully partitioning the massive network using minimum edge cuts, and distributing the partitions on 15 computing nodes. A shared-memory parallelization with OpenMP implementation of the color coding approach is given in [33]. This algorithm achieves a speed up of 12 in a graph with 1.5 million nodes and 31 million edges. A more recent work [34] parallelize the dynamic processing of color-coding algorithm to enumerate subgraphs and is able to handle networks as large as 2 billion edges, with template size up to 10.

3 Background

3.1 Preliminaries and problem statement

We consider labeled graphs $G = (V_G, E_G, L, \ell_G)$, where V_G and E_G are the sets of nodes and edges, L is a set of labels and $\ell_G : V \rightarrow L$ is a labeling on the nodes. A graph $H = (V_H, E_H, L, \ell_H)$ is a *non-induced subgraph* of G if we have $V_H \subseteq V_G$ and $E_H \subseteq E_G$. We say that a template graph $T = (V_T, E_T, L, \ell_T)$ is isomorphic to a non-induced subgraph $H = (V_H, E_H, L, \ell_H)$ of G if there exists a bijection $f : V_T \rightarrow V_H$ such that: (i) for each $(u, v) \in E_T$, we have $(f(u), f(v)) \in E_H$, and (ii) for each $v \in V_T$, we have $\ell_T(v) = \ell_H(f(v))$. In this paper, we assume T is a tree. We will consider trees to be rooted, and use $\rho = \rho(T) \in V_T$ to denote the "root" of T, which is arbitrarily chosen. If T is isomorphic to a non-induced subgraph H with the mapping $f(\cdot)$, we also say that H is a non-induced embedding of T with the root $\rho(T)$ mapped to node $f(\rho(T))$. Figure 1 shows an example of a non-induced embedding of template T in a graph G. Let $emb(T, G)$ denote the number of all embeddings of template T in graph G. Here, we focus on approximating $emb(T, G)$.

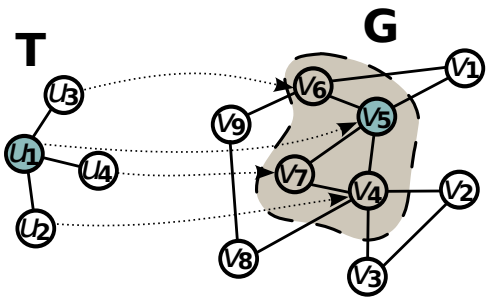


Fig. 1. Here the shaded subgraph is a non-induced embedding of T. The mapping of the template to the subgraph is denoted with the arrow.

An (ε, δ)-approximation to $emb(T, G)$. We say that a randomized algorithm $\mathcal{A}$ produces an *(ε, δ)-approximation* to $emb(T, G)$, if the estimate Z produced by $\mathcal{A}$ satisfies: $\Pr[|Z - emb(T, G)| > \varepsilon \cdot emb(T, G)] \leq 2\delta$; in other words, $\mathcal{A}$ is required to produce an estimate that is close to $emb(T, G)$, with high probability.

Problems studied. We consider the following two problems:

1) *Subgraph counting*: Given a template T and graph G, compute an (ε, δ)-approximation to $emb(T, G)$. When the labels can be disregarded, we refer to this as the *Unlabeled Subgraph Counting* problem. Otherwise, it is referred to as the *Labeled Subgraph Counting* problem.
2) *Graphlet Frequency Distribution (GFD)* [28]: a graphlet is another name for a subgraph. We say a node touches a graphlet T, if it is contained in an embedding of T in the graph G. The graphlet degree of a node v is the number of graphlets it touches. Given a size parameter k, the GFD in a graph G is the frequency distribution

of the graphlet degrees of all nodes with respect to all graphlets of size up to k. The specific problem is to obtain an approximation to the GFD. In this paper, we will focus on "treelets", which only considers all trees of size up to k.

3.2 MapReduce, Hadoop and Harp

MapReduce and its extensions have become a dominant computation model in big data analysis. It involves two stages for data processing: (a) divide the input into distinct *map* tasks and distribute to multiple computing entities, and (b) merging the results of individual computing entities in the *reduce* tasks to produce the final output [12].

The MapReduce model processes data in the form of key-value pairs $\langle k, v\rangle$. An application first takes pairs of the form $\langle k_1, v_1\rangle$ as input to the map function, in which one or more $\langle k_2, v_2\rangle$ pairs are produced for each input pair. Then the MapReduce re-organizes all $\langle k_2, v_2\rangle$ pairs and aggregates all items v_2 that are associated with the same key k_2, which are then processed by a reduce function.

Hadoop [39] is an open-sourced implementation of MapReduce. By defining application specific map and reduce functions, the user can employ Hadoop to manage and allocate appropriate resources to perform the tasks, without knowing the complexity of load balancing, communication and tasks scheduling. Due to the reliability and scalability in handling vast amount of computation in parallel, Hadoop is becoming a *de facto* solution for large parallel computing tasks.

Hadoop falls short in two aspects though: (*i*) the high I/O cost involved within mapper, shuffling and reducer since the data is always read and write from the disk in every stage of a Hadoop job and (*ii*) global synchronization in mapper and reducer, i.e. reducers can start only when all mappers have completed their tasks and vice versa, reduce the efficient usage of the computing resources. To conquer the problems that Hadoop is facing, we further extends our work to use the Harp platform [29].

Harp introduces full collective communication (broadcast, reduce, allgather, allreduce, rotation, regroup or push & pull), adding a separate communication abstraction. The advantage of using in-memory collective communication replacing the shuffling phase is that fine-grained data alignment and data transfer of many synchronization patterns can be optimized.

Harp categorizes four types of computation models (Locking, Rotation, Allreduce, Asynchronous) that are based on the synchronization patterns and the effectiveness of the model parameter update. They provide the basis for a systematic approach to parallelizing iterative algorithms. Since the "Rotation" model partitions the global model parameters among distributed workers, it effectively reduces the memory footprint but requires model synchronization using collective communication. Figure 2 shows the four categories of the computing model.

The Harp framework has been used by 350 students at Indiana University for their course projects. Now it has been released as open source project that is available at public github domain [1]. Harp provides a collection of iterative machine learning and data analysis algorithms (e.g. Kmeans, Multi-class Logistic Regression, Random Forests, Support Vector Machine, Neural Networks, Latent Dirichlet Allocation, Matrix Factorization, Multi-Dimensional Scaling) that have been tested and benchmarked on OpenStack Cloud and HPC platforms including Haswell and Knights Landing architectures. It has also been used for Subgraph mining, Force-Directed Graph Drawing, and Image classification applications.

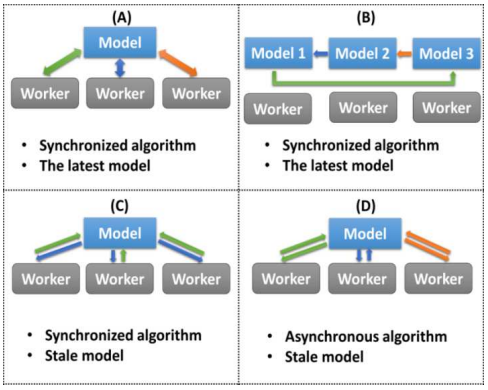


Fig. 2. Harp has 4 computation models: (A) Locking,(B)Rotation, (C) AllReduce, (D) Asynchronous

4 The Sequential Algorithm: Color Coding

TABLE 1
Notations

symbol	description	symbol	description
G	graph	T, T', T''	template and sub-templates
n, m	# nodes, # edges	k	# nodes in T
ρ	root of T	S, s_i	color set, the i^{th} color
$d(v)$	degree of node v	$N(v)$	neighbors of node v

We briefly introduce the color coding algorithm for subgraph counting [6], which gives a randomized approximation scheme for counting trees in a graph. Some of the notation used in the paper is listed in Table 1.

High level description. There are two main ideas underlying the color coding algorithm of [6].

1) **Colorful embeddings**: Color the nodes of the graph with k colors, and only count "colorful" embeddings—an embedding H of the template T is colorful if each node in H has a distinct color. The advantage of this is that the number of colorful embeddings can be counted by a simple and natural dynamic program.
   a) In particular, let $C(v, T(\rho), S)$ be the number of colorful embeddings of T with node $v \in V_G$ mapped to the root ρ, and using the color set S, where $|V_T| = |S|$.
   b) Suppose $(\rho = u_1, u_2)$ is an edge incident on the root node ρ in T. Let tree T be partitioned into trees T_1 and T_2 when the edge (u_1, u_2) is removed, with roots $\rho_1 = u_1$ and $\rho_2 = u_2$ of the trees T_1 and T_2, respectively.
   c) Suppose S_1 and S_2 are disjoint subsets of colors such that $|S_1| = |V_{T_1}|$, $|S_2| = |V_{T_2}|$. Let H_1 and H_2 be two colorful embeddings of T_1 and T_2 using color sets S_1 and S_2, respectively, with ρ_1 and ρ_2 mapped to neighboring nodes $v_1 \in V_G$ and $v_2 \in V_G$, respectively.

Then, H_1 and H_2 must be *non-overlapping*, because they have distinct colors.

d) Therefore,

$$C(v_1, T, S) = \sum_{v_2 \in N(v_1)} \sum_{S = S_1 \cup S_2} C(v_1, T_1(v_1), S_1) \cdot C(v_2, T_2(v_2), S_2),$$

where the first summation is over all neighbors v_2 of v_1 and the second summation is over all partitions $S_1 \cup S_2$ of S.

2) **Random colorings**: If the coloring is done randomly with $k = |V_T|$ colors, there is a reasonable probability that an embedding is colorful—this allows us to get a good approximation to the number of embeddings.

Algorithm 1 The sequential color coding algorithm.

1: **Input:** Graph $G = (V, E)$ and template $T = (V_T, E_T)$
2: **Output:** Approximation to $emb(T, G)$
3:
4: For each $v \in V_G$, pick a color $c(v) \in S = \{1, \ldots, k\}$ uniformly at random, where $k = |V_T|$.
5: Partition the tree T into subtrees recursively to form a set $\mathcal{T}$ using algorithm PARTITION$(T(\rho))$. For each tree $T' \in \mathcal{T}$, we have a root ρ'. Further, if $|V_{T'}| > 1$, T' is partitioned into two trees T'_1, T'_2 with roots $\rho'_1 = \rho'$ and ρ'_2, respectively, which are referred to as the active and passive children of T'.
6: For each $v \in V_G$, $T_i \in \mathcal{T}$ with root ρ_i, and subset $S_i \subseteq S$, with $|S_i| = |T_i|$, we compute $C(v, T_i(\rho_i), S_i)$ using the the recurrence (1) below:

$$c(v, T_i(\rho_i), S_i) = \frac{1}{d} \sum_u \sum c(v, T'_i(\rho_i), S'_i) \cdot c(u, T''_i(\tau_i), S''_i), \quad (1)$$

where d is equal to one plus the number of siblings of τ_i which are roots of subtrees isomorphic to $T''_i(\tau_i)$.
7: For the jth random coloring, let

$$C^{(j)} = \frac{1}{q} \frac{k!}{k^k} \sum_{v \in V_G} c(v, T(\rho), S), \quad (2)$$

where q denotes the number of node $\rho' \in V_T$ such that T is isomorphic to itself when ρ is mapped to ρ'.
8: Repeat the above steps $N = O(\frac{e^k \log(1/\delta)}{\epsilon^2})$ times, and partition N estimates $C^{(1)}, ..., C^{(N)}$ into $t = O(\log(1/\delta))$ sets. Let Z_j be the average of set j. Output the median of $Z_1, ..., Z_t$.

Algorithm 1 describes the sequential color coding algorithm. Figure 3 gives an example of computing Eq. 1.

5 PARALLEL ALGORITHMS

In this section, we present a parallelization of the color coding approach using MapReduce framework, we will first describe SAHAD [42], followed by EN-SAHAD and HARP-SAHAD respectively.

5.1 SAHAD

SAHAD takes a sequence of templates $\mathcal{T} = \{T_0, ..., T\}$ as input. Here $\mathcal{T}$ represents a set of templates generated by partitioning T using Algorithm 2. Then it performs MapReduce variation of Algorithm 1 to compute the number of embeddings of T.

Algorithm 2 *Partition*$(T(\rho))$

1: **if** $T \notin \mathcal{T}$ **then**
2: **if** $|V_T| = 1$ **then**
3: $\mathcal{T} \leftarrow T$
4: **else**
5: Add T to $\mathcal{T}$
6: Pick $\tau \in N(\rho)$, the set of the neighbors of ρ, and partition T into two sub-templates by cutting the edge (ρ, τ)
7: Let T' be the sub-template containing ρ (name as *active child*) and T'' the other (name as *passive child*)
8: Partition$(T'(\rho))$
9: Partition$(T''(\tau))$

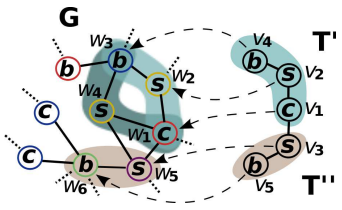


Fig. 3. The example shows one step of the dynamic programming in color coding. T in Figure 1 is split into T' and T''. To count $C(w_1, T(v_1), S)$, or the number of embeddings of $T(v_1)$ rooted at w_1, using color set $S = \{\underline{r}ed, \underline{y}ellow, \underline{b}lue, \underline{p}urple, \underline{g}reen\}$, we first obtain $C(w_1, T'(v_1), \{r, y, b\}) = 2$ and $C(w_5, T''(v_3), \{p, g\}) = 1$. Then, $C(w_1, T(v_1), S) = C(w_1, T'(v_1), \{r, y, b\})C(w_5, T''(v_3), \{p, g\}) = 2$. The embeddings of T are subgraphs with nodes $\{w_3, w_4, w_1, w_5, w_6\}$ and $\{w_3, w_2, w_1, w_5, w_6\}$. Here s, c, b represents the label of the nodes. Details of labeled subgraph counting can be found at [42].

As shown in Equation 1, the counts of all colorful embeddings isomorphic to T rooted from a single node v is computed by aggregating the same measurement of T' and T'', i.e., the two sub-templates, with T' rooted from v and T'' rooted from $\forall u \in N(v)$. We can parallelize color-coding algorithm by distributing the computation among multiple machines, and sending data related with v and $N(v)$ to a computation unit for the aggregation. In our MapReduce algorithm, we manage this by assigning v as the key for both the counts of T' rooted at v and the counts of T'' rooted at v's neighbors, such that all data required for computing counts for T rooted at v has the same key and will be handled by a single reduce function.

Let $X_{T,v}$ be a sequence of color-count pairs $(S_0 = \{s^0_1, s^0_2, ..., s^0_k\}, c_0), (S_1 = \{s^1_1, s^1_2, ..., s^1_k\}, c_1), ...$, where S_i represent a color set containing k colors, and c_i is the counts of the subgraphs isomorphic to T and rooted at v that are colored by S_i. Here $k = |V(T)|$, and each subgraph is a colorful match.

There are 3 types of Hadoop jobs in SAHAD, which are 1) colorer (Algorithm 3) that performs line 4 of Algorithm 1; 2) counter (Algorithm 4, 5) which performs line 6 of Algorithm 1 and 3) finalizer (Algorithm 6, 7) that performs line 7 of Algorithm 1.

The first step is to random color network G with k colors. The map function is described in Algorithm 3:

Here "Collect" is a standard MapReduce operation that

Algorithm 3 *mapper*$(v, N(v))$

1: Pick $s_i \in \{s_1, \ldots, s_k\}$ uniformly at random
2: color v with s_i
3: Let T_0 be the single node template
4: Let $c(v, T_0, \{s_i\}) = 1$ since v is the only colorful matching
5: $X_{T_0,v} \leftarrow \{(\{s_i\}, 1)\}$
6: Collect($\underline{key \leftarrow v}, \underline{value \leftarrow X_{T_0,v}, N(v)}$)

will emit the key-value pairs to global space for further process such as shuffling, sorting or I/O. $N(v)$ represents the neighbors of v. Note that template T_0 is a single node, therefore $X_{T_0,v}$ contains only a single color-count pair $(s_v, 1)$

According to Equation 1, to compute $X_{T_i,v}$, we need $X_{T'_i,v}$ for sub-template T'_i and $X_{T''_i,u}$ for all $u \in N(v)$ for sub-template T''_i. We use a mapper and a reducer function to implement this as shown in Algorithm 4 and 5, respectively.

Algorithm 4 *mapper*$(v, X_{t,v}, N(v))$

1: **if** t is T'_i **then**
2: Collect($\underline{key \leftarrow v}, \underline{value \leftarrow X_{t,v}, flag'}$)
3: **else**
4: **for** $u \in N(v)$ **do**
5: Collect($\underline{key \leftarrow u}, \underline{value \leftarrow X_{t,v}, flag''}$)

Note that in Algorithm 4, the second *Collect* emits $X_{T''_i,v}$ to all its neighbors. Therefore, as shown in Algorithm 5, $X_{T'_i,v}$ and $X_{T''_i,u}$ from all $u \in N(v)$ are handled by the same reducer, which is sufficient for computing Eq. 1. Also note that for a given node v, the number of entries with $flag'$ is 1, and the number of entries with $flag''$ equals $|N(v)|$.

Algorithm 5 *reducer*$(v, (X, flag), (X, flag), ...)$

1: pick X_1 where $flag = flag'$
2: **for** all colorset S'_i from X_1 **do**
3: **for** each X other than X_1 **do**
4: **for** all colorset S''_i from X **do**
5: **if** $S'_i \cap S''_i = \emptyset$ **then**
6: $c(v, T_i, S'_i \cup S''_i) += 1$
7: Collect($\underline{key \leftarrow v}, \underline{value \leftarrow X_{T_i,v}, N(v)}$)

The last step is to compute the total count described in Eq. 2, and is shown in Algorithm 6 and 7.

Algorithm 6 *mapper*$(v, X_{T,v}, N(v))$

1: Collect($\underline{key \leftarrow$ "sum''", $value \leftarrow X_{T,v}$}$)$

Note that in Algorithm 6, $X_{T,v}$ only contains one element, which is the count corresponding to the entire color set. Then in the reducer shown in Algorithm 7, all the counts are added together and properly factorized, to obtain the final count. For a comprehensive description of the MapReduce version of color coding, please refer to [42].

5.2 EN-SAHAD

For general MapReduce problem, the set of keys that is processed in Mapper and Reducer varies among different

Algorithm 7 *reducer*$(\text{“}sum'', X_{T,v_1}, X_{T,v_2}, ...)$

1: $Y = \frac{m^m}{m!} \cdot \frac{1}{q} \sum_{\forall v \in V_G} X$
2: Collect($\underline{key \leftarrow$ "sum''", $value \leftarrow X_{T,v}$}$)$

jobs. Therefore, MapReduce uses external shuffling and sorting in-between Mappers and Reducers to deploy the keys to computing nodes.

In our algorithm, however, the dynamic programming aggregates counts based on the root node of the subtree, and therefore the key is the node index v. In EN-SAHAD, we use this pre-knowledge to predefine a reducer that is corresponding to a set of nodes. We also assign the predefined reducers to computing nodes prior to the beginning of the dynamic programming. Therefore, a data entry with key v will be directly sent to the corresponding computing node and processed by designated Reducer. Using this mechanism, we can reduce the cost of shuffling and sorting in intermediate stage of Hadoop jobs.

5.3 HARP-SAHAD

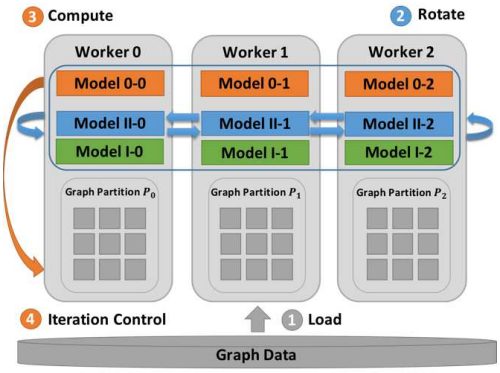


Fig. 4. Here shows one iteration of the Harp implementation.

Figure 4 illustrates HARP-SAHAD for one iteration that counts the embeddings of the template T based on the embeddings of sub-template T' and T''. Graph data is partitioned and stored separately on each worker node in the form of adjacency list. Partitioned graph data is loaded and cached in memory. The model represents a data structure including a color-count pairs for each vertex in graph. Since it follows the dynamic programming scheme, computation runs with static partitioned graph and multiple template models. Model updates are achieved with Model I (embeddings of sub-template T') and model II (embeddings of sub-template T'') generating embeddings of template T on model 0. Note that model I and model II are partitioned and stored separately on the cluster. The model distribution approach with rotation can effectively reduce memory usage per node, thereby addressing the issue when graph template models become too large to fit into memory.

Note that only model II rotates among the worker nodes. As such, it takes three rounds to complete updating model 0 in Figure 4. In the first round, it joins model I-0 and model II-0 to generate new values for model 0-0. Then all partitions of model II rotate so that model II-2 is accessible

on worker 0. In the second round, it joins model I-0 and model II-2 and updates model 0-0. Again, it rotates so that model II-1 is now accessible to worker 0. It then joins model I-0 with model II-1 and updates model 0-0. After this rotation, worker 0 is given access to all partitions of model II and complete updating model 0 in one iteration. Worker 1 and worker 2 run in the same way in parallel. At each iteration, the embeddings of sub-template T' and T'' are used for updating the embeddings of the parent template T. Harp model rotation achieves scaling for fine-grained graph model synchronization with in-memory collective communication.

6 Performance Analysis

In this section, we discuss the performance of SAHAD in terms of the overall work and time complexity. Throughout this section, we denote the number of nodes and edges in the network by n and m respectively. We use k to represent the number of nodes in the template.

Lemma 6.1. For a template T_i, suppose the sizes of the two sub-templates T_i' and T_i'' are k_i' and k_i'', respectively, the sizes of the input, output, and work complexity corresponding to a node v are given below:

- The sizes of the input and output of Algorithm 4 are $O(\binom{k}{k_i'} + \binom{k}{k_i''} + d(v))$ and $O(\binom{k}{k_i''}d(v))$, respectively.
- The size of the input to Algorithm 5 is $O(\binom{k}{k_i''}d(v))$.

Proof For a node v, the input to Algorithm 4 involves the corresponding $X_{T_i',v}$ and $X_{T_i'',v}$ for T_i' and T_i'', as well as $N(v)$, which together have size $O(\binom{k}{k_i'} + \binom{k}{k_i''} + d(v))$. If the input is for T_i'', Algorithm 4 generates multiple key-value pairs for a node v, in which each key-value pair corresponds to some node $u \in N(v)$. Therefore, the output has size $O(\binom{k}{k_i''}d(v))$.

For a given v, the input to Algorithm 5 is the combination of the above, and therefore, has size $O(\binom{k}{k_i''}d(v))$. ■

Lemma 6.2. The total work complexity is $O(k|E_G|2^{2k}e^k \log(1/\delta)\frac{1}{\varepsilon^2})$.

Proof For node v and each neighbor $u \in N(v)$, Algorithm 5 aggregates every pair of the form (S_a, C_a) in $X_{T_i',v}$, and (S_b, C_b) in $X_{T_i'',u}$, which leads to a work complexity of $O(\binom{k}{k_i'}\binom{k}{k_i''}d(v))$. Since $|\mathcal{T}| \leq k$, the total work, over all nodes and templates is at most

$$O(\sum_{v,T_i}\binom{k}{k_i'}\binom{k}{k_i''}d(v)) = O(\sum_v k2^{2k}d(v)) = O(k|E_G|2^{2k}) \tag{3}$$

Since $O(e^k \log(1/\delta)\frac{1}{\varepsilon^2})$ iterations are performed in order to get the (ε, δ)-approximation, the lemma follows. ■

Time Complexity. We use P to denote the number of machines. We assume each machine is configured to run a maximum of M Mappers and R Reducers simultaneously. Finally, we assume a uniform partitioning, so that each machine processes n/P nodes.

Lemma 6.3. The time complexity of Algorithm 3 and 4 is $O(\frac{n}{PM})$ and $O(\frac{m}{PM})$, respectively.

Proof We first consider Algorithm 3, which takes as input an entry of the form $(v, N(v))$ for some node v, and perform a constant work. There are $\frac{n}{P}$ entries processed by each machine. Since M Mappers are run simultaneously, this gives a running time of $O(\frac{n}{PM})$. Next, we consider Algorithm 4. Each Mapper outputs (v, X) for input T_i' and d entries for input T_i'' for each $u \in N(v)$, where d is the degree of v. Therefore, each computing node performs $O(\sum_{i=1}^{n/P} d_i) = O(m/P)$ steps. Here d_i is the degree for v_i. Again, since M Mappers run simultaneously, the total running time is $O(\frac{m}{PM})$.

■

Lemma 6.4. The time complexity of Algorithm 5 is $O(\frac{m \cdot 2^{2k}}{PR})$.

Proof Suppose $|S_i'| = k_i'$ and $|S_i''| = k_i''$. The number of possible color sets S_i' and S_i'' is $\binom{k}{k_i'}$ and $\binom{k}{k_i''}$, respectively. Line 2 of Algorithm 5 involves $O(\binom{k}{k_i'}) = O(2^k)$ steps. Similarly, line 4 also involves $O(2^k)$ steps and Line 3 involves $O(d)$ steps. Therefore the totally running time is $O(d) \cdot 2^{2k}$. Each machine processes $\frac{n}{P}$ entries corresponding to different nodes, leading to a total of $O(\frac{nd \cdot 2^{2k}}{P})$ steps. Since R reducers run in parallel on each machine, this leads to a total time of $O(\frac{m \cdot 2^{2k}}{PR})$. ■

Lemma 6.5. The time complexity of Algorithm 6 and 7 is $O(\frac{n}{PM})$ and $O(n)$, respectively.

Proof Algorithm 6 maps out a single entry for each input. Following the same outline as the proof of 6.3, its running time is $O(\frac{n}{PM})$. Algorithm 7 will take $O(n)$ time since we have only one key "sum", and only one Reducer will be assigned for the summation for all $v \in V(G)$, which takes $O(n)$ time. ■

Lemma 6.6. The overall running time of SAHAD is bounded by

$$O(\frac{k2^{2k}m}{P} \cdot (\frac{1}{M} + \frac{1}{R})e^k \log(1/\delta)\frac{1}{\varepsilon^2}) \tag{4}$$

Proof Algorithm 3 takes $O(\frac{n}{PM})$ time. Algorithm 4 and 5 run for each step of the dynamic programming, i.e., joining two sub-templates into a larger template as shown in Figure 3. Since the number of total sub-template is $O(k)$ when T is a tree, Algorithm 4 and 5 run $O(k)$ times. Therefore the total time is $O(k \cdot (\frac{m}{PM} + \frac{m \cdot 2^{2k}}{PR})) = O(\frac{k2^{2k}m}{P} \cdot (\frac{1}{M} + \frac{1}{R})$. Finally, the entire algorithm as to be repeated $O(e^k \log(1/\delta)\frac{1}{\varepsilon^2})$ times, in order to get the (ε, δ)-approximation, and the lemma follows. ■

6.1 Performance Analysis of Intermediate Stage

With SAHAD, a major bottleneck of a hadoop job in terms of running time is the shuffling and sorting cost in the intermediate stage between Mapper and Reducer, due to the high I/O and synchronization cost as shown by the black bar in Figure 5.

we observe that the external shuffling and sorting stage takes roughly twice the time of the reducing stage in Reducers, which dramatically increase the overall running time. Given that the keys in Mappers and Reducers are always the index of all the node $v \in V(G)$, we can enhance SAHAD by removing the shuffling and sorting in the intermediate

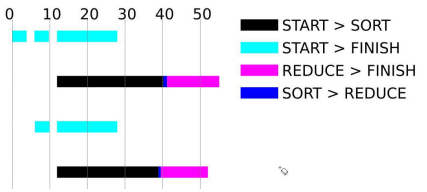


Fig. 5. The figure shows the time spent in each stage of a running hadoop job to produce count for a 5-node template, by aggregating the 2-node and 3-node sub-tree. The black bar is the time for intermediate stage, which is for shuffling and sorting.

stage. Instead, we can designate Reducers and directly send the data to corresponding Reducers.

7 Variations of Subgraph Isomorphism Problems

So far we have discussed the basic framework of the algorithm. We have also discussed how to compute the total number of subgraph embeddings in Algorithm 7. We now discuss a set of problems that are closely related with the subgraph isomorphism problem, including finding supervised motif and computing graphlet frequency distribution, which can be computed using our framework.

Note that our algorithm is specifically suitable for computing on multiple templates if they have common sub-templates, since those common sub-templates only need to be computed once. This is the case in many problems, where common sub-templates such as single node, edge, or simple paths are shared.

7.1 Supervised Motif Finding

Motifs of a real-world network are specific templates whose embeddings occur with much higher frequencies than in random networks and are referred as building blocks for networks. They have been found in many real-world networks [26]. Our algorithm can reduce the computational cost for a group of templates since the common sub-templates are only computed once, therefore is amenable to be applied in supervised motif finding.

7.2 Graphlet Frequency Distribution

Graphlet frequency distribution has been proposed as a way of measuring the similarity of protein-protein networks [28], where common properties such as degree distribution, diameter, etc., may not suffice. Unlike "motifs", graphlet frequency distribution is computed on all selected small subgraphs regardless of whether they appear frequently or not.

Graphlet frequency distribution $D(i, T)$ measures the number of nodes from which i graphlets that are isomorphic to T are touched on. The number of graphlet touched on a single node v can be computed using a number of counts of the same templates T with root placed at different nodes of T.

8 Experimental Analysis of SAHad, En-SAHad & Harp-SAHad

We carry out a detailed experimental analysis of SAHad, En-SAHad and Harp-SAHad. We focus on three aspects:

(*i*) *Quality of the solution*: We measure the empirical approximation error of our algorithms and show that the error is very small so in the following experiments we run the program for a single iteration;

(*ii*) *Scalability of the algorithms as a function of template size, graph size and computing resources:* We carried out experiments using templates with size ranging from 3 nodes to 12 nodes, including both labeled and unlabeled templates. The graphs we use go from several hundreds of thousands of nodes to tens of millions. We also study how our algorithm scales in terms of computing resources including number of threads per node, number of computing nodes, different settings of mappers and reducers, etc.

(*iii*) *Variations of the problem:* Our framework has the ability to extend to a variety of measurements related with the subgraph counting problem. In the experiments, we show the unlabled/labeled subgraph counting and graphlet distribution results.

(*iv*) *Enhancing overall performance by system tuning:* We also investigate different components of the system and their impact to the overall performance. For example, En-SAHad studies the communication and sorting cost in the intermediate stage of the system and gives approaches for improvement. We also propose a degree based graph partitioning scheme that can improve the performance of Harp by imposing better load balancing in terms of computations within each partition. Table 2 listed main results we obtained with various methods.

TABLE 2
Comparison on SAHad, En-SAHad and Harp-SAHad

Method	Networks	Templates	Performance
SAHad	268M edges	$\leq$ 12 nodes	10s of min for 7 node template on Chicago
En-SAHad	12M edges	5 nodes	20% improvement over SAHad
Harp-SAHad	0.5B edges	7 nodes	7 - 8 times faster than SAHad

8.1 Experimental Design

8.1.1 Datasets

For our experiments, we use synthetic social contact networks of the following cities and regions: Miami, Chicago, New River Valley (NRV), and New York City (NYC) (see [8] for detains). We consider demographic labels – {kid, youth, adult, senior} (based on the age) and gender for individuals. We also run experiments on a $G(n, p)$ graph (denoted GNP100) with n nodes, each pair of nodes connected with probability p, and randomly assigned node labels. We also experiment on few other networks: Web-Google [2], RoadNet (rNet) [2] and Chung-Lu random graphs [10]. Table 3 summarizes the characteristics of the networks.

8.1.2 Templates

The templates we use in the experiments are shown in Figure 6. The templates vary in size from 5 to 12 nodes,

TABLE 3
Networks used in the experiments

Network	No. of Nodes(in million)	No. of Edges(in million)
Miami	2.1	52.7
Chicago	9.0	268.9
NYC	18.0	480.0
NRV	0.2	12.4
rNet	2.0	2.8
GNP100	0.1	1.0
Web-Google	0.9	4.3

in which *U5-1,. . .U10-1* are the unlabeled templates and *L7-1 ,L10-1* and *L12-1* are the labeled templates. In the labels, *m*, *f*, *k*, *y*, *a* and *s* stand for *male*, *female*, *kid*, *youth*, *adult* and *senior*, respectively.

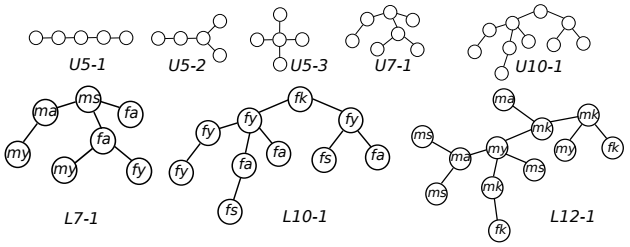


Fig. 6. Templates used in the experiments.

8.1.3 Computing Environment

For experiments with SAHAD, we use a computing cluster **Athena**, with 42 computing nodes and a large RAM memory footprint. Each node has a quad-socket AMD 2.3GHz Magny Cour 8 Core Processor, i.e., 32 cores per node or 1344 cores in total, and 64 GB RAM(12.4 TFLOP peak). The local disk available on each node is $750GB$. Therefore, we can have maximum $31.5TB$ storage for the HDFS. In most of our experiments, we use up to 16 nodes, which give up to $12TB$ capacity for the computation. Although the number of cores and RAM capacity on each node can support a large number of mappers/reducers, the availability of a single disk on each node limits aggregate I/O bandwidth of all parallel processes on each node. To make it worse, aggregate I/O bandwidth of parallel processes doing sequential I/O could result in many extra disk seeks and hurt overall performance. Therefore, disk bandwidth is the bottleneck for more parallelism in each node. This limitation is further discussed in section 8.2.2. We also use the public Amazon Elastic Computing Cloud (**EC2**) for some of our experiments. EC2 enables customers to instantly get cheap yet powerful computing resources, and start computing business with no upfront cost for hardware. We allocated 4 *High-CPU Extra-Large* instances from EC2. Each instance has 8 cores, 7 GB RAM, and two 250 GB virtual disks (Elastic Block Store Volume).

For experiments with HARP-SAHAD, we use Juliet cluster (Intel Haswell architectures) with 1,2,4,8 and 16 nodes. Juliet cluster contains 32 nodes each with two 18-core 36-thread Intel Xeon E5-2699 processors and 96 nodes each with two 12-core 24-thread Intel Xeon E5-2670 processors. All the nodes used in the experiments are with Intel Xeon E5-2670 processors and 128 GB memory. All the experiments are performed on standard cooper 1 $Gbps$ Ethernet.

8.1.4 Performance metrics

We carry out experiments on SAHAD, EN-SAHAD and HARP-SAHAD. For SAHAD, we measure the approximation bounds, the impact of Hadoop configuration including number of Mapper/Reducers and performance on queries related with various templates and graphs. For enhanced SAHAD, we measure the performance improvement gained by eliminating the sorting in the intermediate stage. We also measures the impact with difference partitioning schemes. Then with Harp, similar to SAHAD, we measure the performance impact with various templates and graphs, as well as the system performance regarding number of computing nodes. We also compare HARP-SAHAD and SAHAD to study the improvement Harp brings.

8.2 Performance of SAHAD

In this section, we evaluate various aspects of the performance. Our main conclusions are summarized below. Table 4 summarizes the different experiments we perform, which are discussed in greater details later.

1. Approximation bounds: While the worst case bounds on the algorithm imply $O(e^k \log(1/\delta)\frac{1}{\varepsilon^2})$ rounds to get an (ε, δ)-approximation (see Lemma 6.2), in practice, we find that far fewer iterations are needed.

2. System performance: We run our algorithm on a diverse set of computing resources, including the publicly available Amazon EC2 cloud. We find our algorithm scales well with the number of nodes, and disk I/O is one of the main bottlenecks. *We posit that employing multiple disks per node (a rising trend in Hadoop) or using I/O caching will help mitigate this bottleneck and boost performance even further.*

3. Performance on various queries: We evaluate the performance on templates with sizes ranging from 5 to 12. We find labeled queries are significantly faster than unlabeled ones, and the overall running time is under 35 minutes for these queries on our computing cluster (described below). We also get comparable performance on EC2.

8.2.1 Approximation bounds

As discussed in Section 3, the color coding algorithm averages the estimates over multiple iterations. Figure 7 and 8 show the error for each iteration in counting *U5-1* for Miami and Web-Google, respectively. It is observed that the standard deviation for the error is 2% and 0.4% for Miami and Web-Google, which is very small.

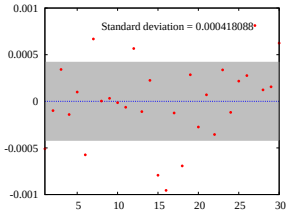


Fig. 7. Error in counting *U5-1* for 30 interations for Miami.

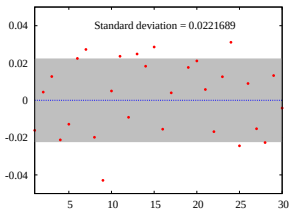


Fig. 8. Error in counting *U5-1* for 30 interations for Web-Google.

In Figure 9, we show that the approximation error is below 0.5% for the template *U7-1* for the GNP100 graph,

TABLE 4
Summary of the experiment results (refer to Section 8.1 for the terminology used in the table)

Experiment	Computing resource	Template & Network	Key Observations
Approximation bounds	Athena	U7-1 & GNP100	error well below 0.5%
Impact of the number of data nodes	Athena	U10-1 & Miami, GNP100	scale from 4 hours to 30 minutes with data nodes from 3 to 13
Impact of the number of concurrent reducers	Athena & EC2	U10-1 & Miami	performance worsen on Athena
Impact of the number of concurrent mappers	Athena & EC2	U10-1 & Miami	no apparent performance change
Unlabeled/labeled templates counting	Athena & EC2	templates from Figure 6 and networks from Table 3	all tasks complete in less than 35 minutes
Graphlet frequency distribution	Athena	U5-1 & Miami,Chicago	complete in less than 35 minutes

even for one iteration. The figure also plots the results based on using more than 7 colors, which can sometimes improve the running time, as discussed in [17]. In the rest of the experiments, we only use the estimation from one iteration, because of the small error shown in this section. The error for i iterations is computed using $\frac{|(\sum_i Z_i)/i - emb(T,G)|}{emb(T,G)}$.

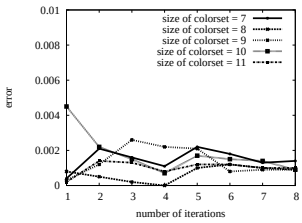


Fig. 9. Approximation error in counting *U7-1* on GNP100.

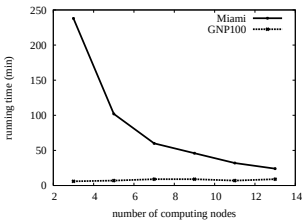


Fig. 10. Running time for counting *U10-1* vs number of computing nodes.

8.2.2 Performance Analysis

We now study how the running time is affected by the number of total computing nodes and number of reducers/mappers per node. We carry out 3 sets of experiments: (i) how the total running time scales with the number of computing nodes; (ii) how the running time is affected by varying assignment of mappers/reducers per node.

1. Varying number of computing nodes Figure 10 shows that the running time for Miami reduces from over 200 minutes to less than 30 minutes when the number of computing nodes increases from 3 to 13. However, the curve for GNP100 does not show good scaling. The reason is that the actual computation for GNP100 only consumes a small portion of the running time, and there are overheads from managing the mappers/reducers. In other words, the curve for GNP100 shows a lower bound on the running time in our algorithm.

2.Varying number of mappers/reducers per node We consider two cases.

2.a. Varying number of reducers per node Figure 11 and 12 show the running time on Athena when we vary the number of reducers per node. Here we fix the number of nodes to be 16 and the number of mappers per node to be 4. We find that running 3 reducers concurrently on each node minimizes the total running time. From Figure 12 we find that though increasing the number of reducers per node can reduce the time for the Reduce stage for a single job, the running time increases sharply in Map and Shuffle stage. As a result, the total running time increases with the number of reducers. It is because of the I/O bottleneck for concurrent accessing on Athena, since Athena has only 1 disk per node. This phenomena is not seen on EC2, as seen from Figure 16, indicating that EC2 is better optimized for concurrent disk accessing for cloud usage.

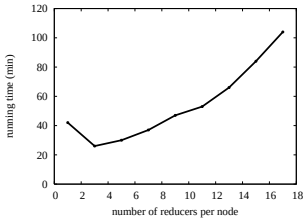


Fig. 11. Total running time versus number of reducers per node.

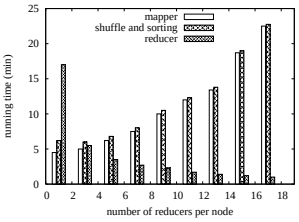


Fig. 12. Running time of different job stages versus the number of reducers per node.

2.b. Varying number of mappers per node Figure 13 and 14 show the running time on Athena when we vary the number of mappers per node while fixing the number of reducers as 7 per node. We find that varying the number of mappers per node does not affect the performance. This is also validated in EC2, as shown in Figure 15.

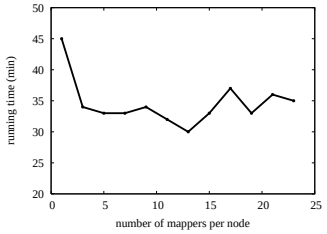


Fig. 13. Total running time versus the number of mappers per node.

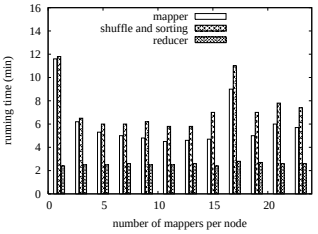


Fig. 14. Running time of different job stages versus the number of mappers per node.

2.c. Reducers' running time distribution Figure 17, 18, 19 and 20 show the distribution of the reducers' running time on Athena. We observe that when we increase the number of reducers per node, the distribution becomes more volatile; for example, when we concurrently run 15 reducers per node, the reducers' completion time vary from 20 minutes to 120 minutes. This also indicates the bad I/O performance on Athena for concurrent accessing.

8.2.3 Illustrative applications

In this section, we illustrate the performance on 3 different kinds of queries. We use Athena and assign 16 nodes as

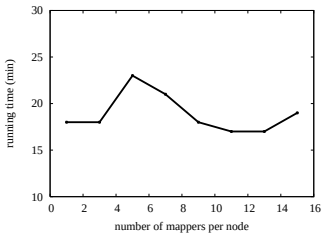


Fig. 15. Total running time versus number of mappers per node on EC2.

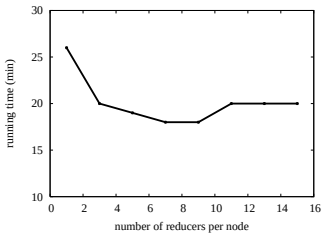


Fig. 16. Total running time versus number of reducers per node on EC2.

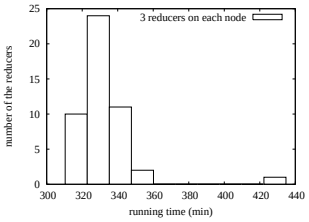


Fig. 17. 3 reducers per computing node.

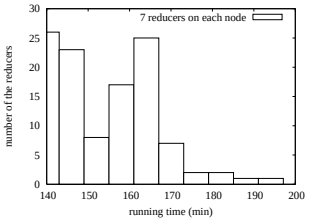


Fig. 18. 7 reducers per computing node.

the data nodes; for each node, we assign a maximum of 4 mappers and 3 reducers per node. Our experiments on EC2 for some of these queries are discussed later in Section 8.2.4.

1. Unlabeled subgraph queries: Here we compute the counts of templates *U5-1*, *U7-1* and *U10-1* on GNP100 and Miami, as shown in Figure 21. We also plot how the running time scales to different templates and networks, as shown in Figure 22 – we observe that for unlabeled template with up to 10 nodes on the Miami graph, the algorithm runs in less than 25 minutes.

2. Labeled subgraph queries: Here we count the total number of embeddings of template *L7-1*, *L10-1* and *L12-1* in Miami and Chicago. Figure 24 shows that the running time for counting templates up to 12 nodes is around 15 minutes on Miami, less than 35 minutes needed for Chicago. The running time is much less for the labeled subgraph queries than that for the unlabeled subgraph queries. It is due to the fact that that labeled templates have much less number of embeddings due to the label constraints.

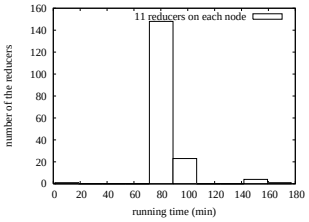


Fig. 19. 11 reducers per computing node.

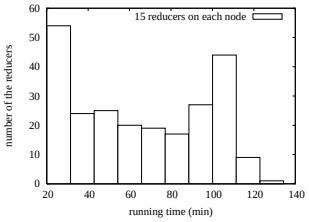


Fig. 20. 15 reducers per computing node.

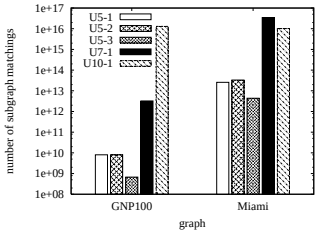


Fig. 21. The counts of unlabeled subgraphs in GNP100 and Miami.

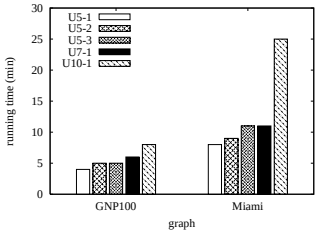


Fig. 22. Running time for counting unlabeled subgraph in GNP100 and Miami.

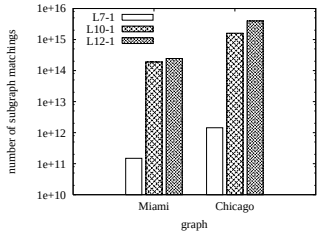


Fig. 23. The counts of labeled templates in Miami and Chicago.

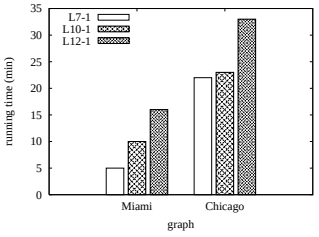


Fig. 24. Running time for counting labeled templates in Miami and Chicago.

3. Computing graphlet frequency distribution: Figure 25 and 26 show the graphlet frequency distribution of the networks of Miami and Chicago, respectively. The template is *U5-1*. It takes 15 minutes and 35 minutes to compute graphlet frequency distribution on Miami and Chicago, respectively.

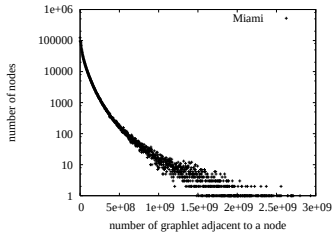


Fig. 25. Graphlet frequency distribution of Miami.

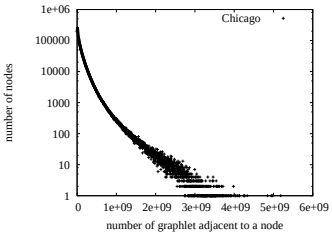


Fig. 26. Graphlet frequency distribution of Chicago.

8.2.4 *Performance Study with Amazon EC2*

On EC2, we run unlabeled and labeled subgraph queries on Miami and GNP100 for templates *U5-1*, *U7-1*, *U10-1*, *L7-1*, *L10-1* and *L12-1*. We use the same 4 EC2 instances as discussed previously, and each node runs up to a maximum of 2 mappers and 8 reducers concurrently. As shown in Figures 27 and 28, the running time on EC2 is comparable to that on Athena, except for *U10-1* on Miami, which takes roughly 2.5 hours to finish on EC2, but only 25 minutes on Athena. This is because for such a large template and graph as large as Miami, input/output data as well as the I/O pressure on disks are tremendous. EC2 uses virtual disks as local storage, which hurt overall performance when dealing with such a large amount of data.

8.3 Performance of EN-SAHAD

In this section we experiment our algorithms on two real-world networks *nrv* and *rNet* and a number of their shuffled versions. We generate shuffled networks with 20, 40, 60, 80 and 100 percent shuffling ratio. , and name them as *nrv20* to *nrv100*, and *rNet20* to *rNet100*.

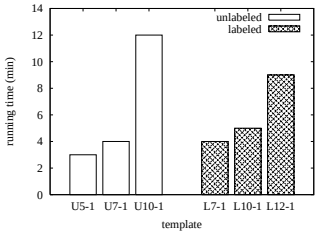


Fig. 27. Running time for various templates on GNP100.

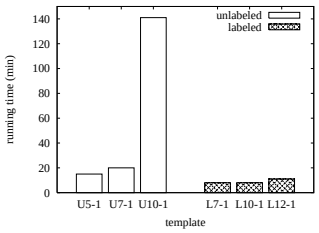


Fig. 28. Running time for various templates on Miami.

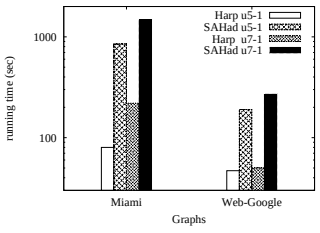


Fig. 31. Comparing the running times of HARP-SAHAD and SAHAD on templates *u5-1* and *u7-1*.

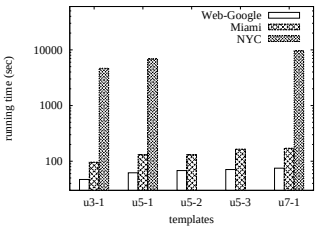


Fig. 32. Running time of HARP-SAHAD with different templates and graphs.

As discussed in Section 5.2, a major factor that impact the overall performance is the heavy shuffling and sorting cost in the intermediate stage of a Hadoop job. We mitigate this factor by designating node index v to Reducers, and pre-allocate Reducers among computing nodes. In such a way, the key-value pairs from Mappers can be directly sent to corresponding Reducers without being shuffled and sorted.

Figures 29 and 30 show the overall running time of our algorithm on *NRV*, *RoadNet* and their variations. Here we generate the variations of the graph by shuffling a proportion of the edges in the graph, e.g., *nrv40* is a *NRV* with 40% of its edges being shuffled. We see that pre-allocating Reducer can bring roughly 20% performance improvement.

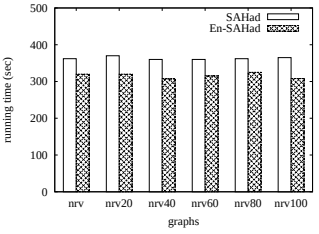


Fig. 29. SAHAD vs EN-SAHAD on NRV and its variations

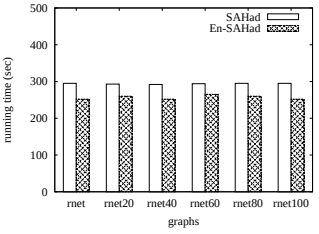


Fig. 30. SAHAD vs EN-SAHAD on RoadNet and its variations

8.4 Performance of HARP-SAHAD

In this section, we discuss the performance of HARP-SAHAD and compare its performance with SAHAD.

8.4.1 HARP-SAHAD *versus* SAHAD

For network Web-Google, HARP-SAHAD runs about 5 times faster than SAHAD; and for network Miami, HARP-SAHAD runs about 9 times faster than Sahad. In addition, Figure 31 shows high efficiency in implementations the total execution time increases 2 times when the graph dataset increases to about 11 times from Web-Google to Miami.

8.4.2 *Scalability of* HARP-SAHAD

In this section, we discuss how Harp performs with different templates and graphs. We start our experiment by running Harp against different templates with $|k|$ from 3 to 7, and on 3 graphs with number of nodes varying from 0.9 millions to 18 millions. Figure 32 shows that the computational cost on Harp is more sensitive to the size of the graph than the size of the templates.

Then we studies the performance by controlling the number of nodes in a graph and increase the number of edges. Here we use Chung-Lu model [4] to generate a series of random graphs given the degree sequence and its variations of Miami and NYC. The average degree of generated random graphs ranging from 50 to 150 for Miami and 10 to 100 for NYC. As we can see from Figure 33 and Figure 34, generally the running time increases according to the number of edges of the graph, which meets the time complexity we propose in Section 6. Specifically, the running time on the Chung-Lu graphs based off NYC shows a quadratic curve with increasing number of edges, which matches Equation 4. The results on Miami does not show quadratic increase due to the relative small computational cost comparing to other overhead of the system.

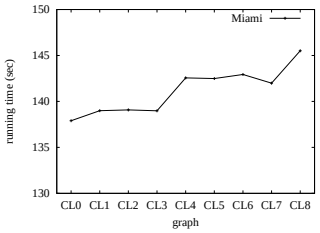


Fig. 33. Running time of HARP-SAHAD given a series of Chung-Lu random graph based off *Miami* with increasing number of edges.

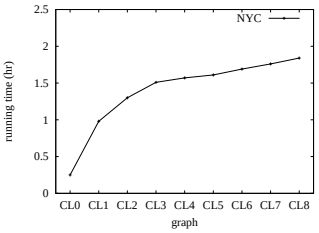


Fig. 34. Running time of HARP-SAHAD given a series of Chung-Lu random graph based off *NYC* with increasing number of edges.

8.4.3 *Varying number of computing nodes*

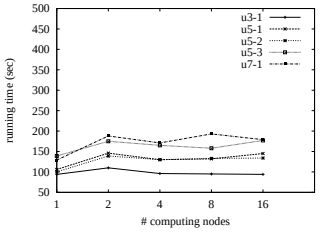


Fig. 35. Running time of HARP-SAHAD given different number of computing nodes and templates on Miami

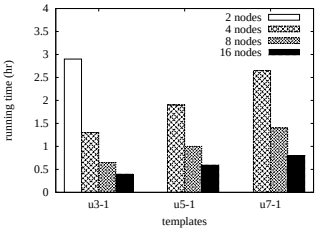


Fig. 36. Running time on NYC scales with number of computing nodes

In this section, we study the performance of HARP-SAHAD as a function of computing resources, i.e., computing nodes and threads per node. Figure 35 shows that for Miami the performance does not improve with increasing

number of the computing nodes. It is due to the low computation cost within each nodes and high communication cost. An increasing number of nodes brings in more communication cost that tradeoffs the gain on computation. To further investigate the performance challenges on big dataset, we run experiments on the NYC dataset, which is about 9 times larger than the Miami dataset. Figure 36 presents the running time on different number of machine nodes (2, 4, 8 and 16 nodes). It clearly shows a good scalability of the Harp implementation.

8.4.4 Degree based partitioning schemes

In the above experiments, we evenly partition the graphs without considering the nature of the problem and the structure of the graphs. With that, each partition has the same number of nodes.

In this section, we experiment a new partition scheme based on a degree-related metric D_p as shown in Equation 5. Given a node with degree d, there are totally $\binom{d}{2}$ different pairs of edges that sub-templates τ' and τ'' can resides at, or $O(d^2)$ ways to join 2 sub-templates. To have roughly equal computational cost within each partition p we partition the graph such that each partition has similar D_p. We expect that the computation in each partition will be roughly the same with this partitioning scheme, therefore there is less overhead in synchronization and unbalanced load.

$$D_p = \sum_{\forall v \in p} {d_v}^2 \tag{5}$$

Here d_v is the degree of node v.

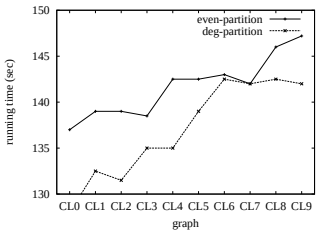


Fig. 37. Running time of Harp on Miami with even and degree based partitioning

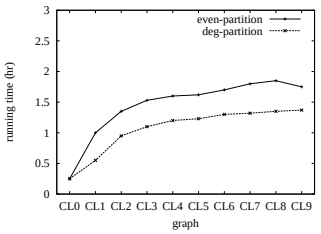


Fig. 38. Running time of Harp on NYC with even and degree based partioning

Figures 37 and 38 show the performance gain obtained by partitioning the graph based on D_p. We noticed that on Miami, the performance improvement is merely 3%, which is largely due to the relatively small size of the graph and computational cost. However, degree based partitioning improve the performance on NYC by an average of 21%, showing that for larger graphs that can incur high computational cost, how the graph is partitioned plays a major role.

9 Conclusion

We described an efficient parallel algorithm to compute the number of isomorphic embeddings of a subgraph in very large networks using MapReduce and the color coding technique. We first developed SAHAD – a Hadoop based implementation. We give performance analysis in terms of work and time complexity. We further explore two approaches to reduce the sorting and communication cost. We also implement our algorithm using the Harp framework which employs collective communication and shared memory to facilitates the computation. Our experiments show that HARP-SAHAD has significantly improved performance when compared to SAHAD — by almost an order of magnitude and simultaneously achieves good scalability. The new algorithm can process networks with 0.5 Billion edges and 7 node templates. As directions for future research, it would be interesting to device new algorithms that scale to larger instances. Additionally, it would be interesting to implement variant of these algorithms for restricted classes of networks.

References

[1] Harp. https://dsc-spidal.github.io/harp/.
[2] Snap – stanford network analysis project. http://snap.stanford.edu/index.html.
[3] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour. 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*, page 61. IEEE Press, 2016.
[4] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 171–180. ACM, 2000.
[5] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):i241, 2008.
[6] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):856, 1995.
[7] Amazon. Elastic computing cloud (ec2). http://aws.amazon.com/ec2.
[8] C. Barrett, R. Beckman, M. Khan, V. Kumar, M. Marathe, P. Stretz, T. Dutta, and B. Lewis. Generation and analysis of large synthetic social contact networks. In *Winter Simulation Conference*, 2009.
[9] M. Bröcheler, A. Pugliese, and V. Subrahmanian. Cosi: Cloud oriented subgraph identification in massive social networks. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 248–255. IEEE, 2010.
[10] F. Chung and L. Lu. Connected components in random graphs with given expected degree sequences. *Annals of combinatorics*, 6(2):125–145, 2002.
[11] R. Curticapean and D. Marx. Complexity of counting subgraphs: Only the boundedness of the vertex-cover number counts. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 130–139. IEEE, 2014.
[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[13] J. Flum and M. Grohe. The parameterized complexity of counting problems. *SIAM Journal on Computing*, 33(4):892–922, 2004.
[14] F. V. Fomin, D. Lokshtanov, V. Raman, S. Saurabh, and B. R. Rao. Faster algorithms for finding and counting subgraphs. *Journal of Computer and System Sciences*, 78(3):698–706, 2012.
[15] L. Getoor and C. Diehl. Link mining: a survey. *ACM SIGKDD Explorations Newsletter*, 7(2):3–12, 2005.
[16] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 581–586. ACM, 2004.
[17] F. Hüffner, S. Wernicke, and T. Zichner. Algorithm engineering for color-coding with applications to signaling pathway detection. *Algorithmica*, 52(2):114–132, 2008.
[18] H. B. Hunt III, M. V. Marathe, V. Radhakrishnan, and R. E. Stearns. The complexity of planar counting problems. *SIAM Journal on Computing*, 27(4):1142–1167, 1998.
[19] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. *Principles of Data Mining and Knowledge Discovery*, pages 13–23, 2000.
[20] I. Koutis and R. Williams. Limits and applications of group algebras for parameterized problems. In *Proc. ICALP*, pages 653–664, 2009.
[21] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data mining and knowledge discovery*, 11(3):243–271, 2005.

[22] J. Leskovec, A. Singh, and J. Kleinberg. Patterns of influence in a recommendation network. *Advances in Knowledge Discovery and Data Mining*, pages 380–389, 2006.

[23] Y. Liu, X. Jiang, H. Chen, J. Ma, and X. Zhang. Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. *Advanced Parallel Processing Technologies*, pages 341–355, 2009.

[24] D. Marx and M. Pilipczuk. Everything you always wanted to know about the parameterized complexity of subgraph isomorphism (but were afraid to ask). In *31st International Symposium on Theoretical Aspects of Computer Science*, page 542, 2014.

[25] E. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 115–124. ACM, 2010.

[26] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824, 2002.

[27] R. Pagh and C. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Arxiv preprint arXiv:1103.6073*, 2011.

[28] N. Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177, 2007.

[29] J. Qiu, S. Jha, A. Luckow, and G. C. Fox. Towards hpc-abds: an initial high-performance big data stack. *Building Robust Big Data Ecosystem ISO/IEC JTC 1 Study Group on Big Data*, pages 18–21, 2014.

[30] J. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of computer-aided molecular design*, 16(7):521–533, 2002.

[31] R. Ronen and O. Shmueli. Evaluating very large datalog queries on social networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 577–587. ACM, 2009.

[32] S. Sakr. Graphrel: A decomposition-based and selectivity-aware relational framework for processing sub-graph queries. In *Database Systems for Advanced Applications*, pages 123–137. Springer, 2009.

[33] G. M. Slota and K. Madduri. Fast approximate subgraph counting and enumeration. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 210–219. IEEE, 2013.

[34] G. M. Slota and K. Madduri. Parallel color-coding. *Parallel Computing*, 47:51–69, 2015.

[35] B. Suo, Z. Li, Q. Chen, and W. Pan. Towards scalable subgraph pattern matching over big graphs on mapreduce. In *Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on*, pages 1118–1126. IEEE, 2016.

[36] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM, 2011.

[37] C. Tsourakakis, U. Kang, G. Miller, and C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 837–846. ACM, 2009.

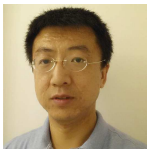
[38] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[39] T. White. *Hadoop: The definitive guide*. Yahoo Press, 2010.

[40] X. Yan, X. Zhou, and J. Han. Mining closed relational graphs with connectivity constraints. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 324–333. ACM, 2005.

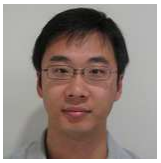
[41] Z. Zhao, M. Khan, V. Kumar, and M. Marathe. Subgraph enumeration in large social contact networks using parallel color coding and streaming. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 594–603, 2010.

[42] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. Kumar, and M. V. Marathe. Sahad: Subgraph analysis in massive networks using hadoop. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 390–401. IEEE, 2012.



Zhao Zhao is pursuing his Ph.D degree in Computer Science at Virginia Tech. He is also a Software Engineer in Verisign Labs, Verisign Inc. His research interests are in Network Science and analytics, especially in the design and analysis of parallel graph algorithms.

Meng Li is a Computer Science Ph.D. student in the School of informatics and Computing at Indiana University. His advisor is Prof. Judy Qiu. His research interest is distributed systems and parallel computing.



Guanying Wang earned his PhD in Computer Science from Virginia Tech in 2012. He is now a software engineer at Google.

Ali Butt received his Ph.D. degree in Electrical & Computer Engineering from Purdue University in 2006. He is a recipient of an NSF CAREER Award, IBM Faculty Awards, a VT College of Engineering (COE) Dean's award for "Outstanding New Assistant Professor", and NetApp Faculty Fellowships. Ali's research interests are in distributed computing systems and I/O systems.

Maleq Khan is an Assistant Professor in the Department of Electrical Engineering and Computer Science at Texas A&M University–Kingsville. He received his Ph.D. in Computer Science from Purdue University. His research interests are in parallel and distributed computing, big data analytics, high performance computing, and data mining.

Madhav Marathe is a professor of Computer Science and the Director of the Network Dynamics and Simulation Science Laboratory, Biocomplexity Institute, Virginia Tech. His research interests include high performance computing, modeling and simulation, theoretical computer science and socio-technical systems. He is a fellow of the IEEE, ACM and AAAS.

Judy Qiu is an associate professor of Intelligent Systems Engineering in the School of Informatics and Computing at Indiana University. Her research interests are parallel and distributed systems, cloud computing, and high-performance computing. Her research has been funded by NSF, NIH, Intel, Microsoft, Google, and Indiana University. Judy Qiu leads the Intel Parallel Computing Center (IPCC) site at IU. She is the recipient of a NSF CAREER Award in 2012.

Anil Vullikanti is an Associate Professor in the Department of Computer Science and the Biocomplexity Institute of Virginia Tech. His interests are in the areas of approximation and randomized algorithms, distributed computing, graph dynamical systems and their applications to epidemiology, social networks and wireless networks. He is a recipient of the NSF and DOE Career awards.