Finding and counting tree-like subgraphs using MapReduce

Zhao Zhao, Langshi Chen, Mihai Avram, 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 web analysis, disease diffusion prediction and social network analysis. These problems are computationally challenging in having to scale to very large networks with millions of vertices. In this paper, we present SAHAD, a MapReduce algorithm for detecting and counting trees of bounded size using the elegant color coding technique developed by N. Alon *et al*. SAHAD is a randomized algorithm, and we show rigorous bounds on the approximation quality and the performance of it. SAHAD scales to very large networks comprising of $10^7 - 10^8$ vertices and $10^8 - 10^9$ edges and tree-like (acyclic) templates with up to 12 vertices. Further, we extend our results by implementing SAHAD in the Harp framework, which is more of a high performance computing environment. The new implementation gives 100x improvement in performance over the standard Hadoop implementation and achieves better performance than state-of-the-art MPI solutions on larger graphs.

Index Terms—subgraph isomorphism, color coding, approximation algorithm, 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 tend to search for meaningful patterns in networked data; and these patterns are often 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 [4], [20]. 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., [24]), recommendation networks [26], chemical structure analysis [34], and detecting memory leaks [29]. 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. [30] 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 [32].

The Subgraph Isomorphism problem and its variants is well known to be as computationally challenging as a NP-complete problem. 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 [13], [16], [28] 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 vertices in the template H, is fixed. Letting n be the number of vertices 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. Note that in this paper we focus on non-induced subgraph matching. When the template is a tree or has a bounded treewidth, Alon *et al*. [4] present an elegant randomized approximation algorithm with running time $O(k|E|2^k e^k \log(1/\ )\frac{1}{2})$, where and are error and confidence parameters, respectively, based on the color coding technique. Their result was significantly improved by Koutis and Williams [23] 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 [22], [24], in generating candidates for subgraphs at each level; these have been made to run faster by better pruning and exploration techniques, e.g., [19], [24], [44]. Other approaches in relational databases and data mining involve queries for specifically labeled subgraphs, and have

- *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*
- *Langshi Chen, Mihai Avram, and Meng Li are with the Computer Science Department, Indiana University. Email: lc37@indiana.edu, mavram@umail.iu.edu, 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*

combined relational database techniques with careful depth-first exploration, e.g., [8], [35], [36].

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. Furthermore, it is not clear if candidate generation approaches [19], [24], [44] can be parallelized and scaled to large graphs and computing clusters. Two recent approaches for parallel algorithms, related to this work, are [8], [46]. The approach of Bröcheler *et al.* [8] requires a complex preprocessing and enumeration process, which has high end-to-end time, while the approach of [46] involves an MPI-based implementation with a very high communication overhead for larger templates. Two other papers [31], [40] develop MapReduce based algorithms for approximately counting the number of triangles with a work complexity bound of $O(|E|)$. 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-monotone to the subgraph size. The algorithm reported in [3] is capable of computing subgraph support on large networks 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 subgraphs [39] which scales to networks with up to 300 million edges.

Other approaches studied in the context of data mining and databases, e.g., [8], [35], [36], 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. In addition, we developed HARPSAHAD+ which is an adaptation of SAHAD in the Harp [33] framework to utilize its advanced MPI-like collective communication. It scales to graphs with up to 1.2 billion edges and achieves two orders of magnitude improvement in performance over SAHAD.

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 [31], [40], [41] on very large networks, or more general subgraphs on relatively small networks [27]. Our main technical contribution is the development of a Hadoop version of the *color coding* algorithm of Alon *et al.* [4], [5], which is a (sequential) randomized approximation algorithm for subgraph counting. It is a randomized approximation algorithm that for any , , gives a $(1\pm\ )$ approximation to the number of embeddings with probability at least $1-2$. We prove that the work complexity of SAHAD is $O(k|E_G|2^{2k}e^k\log(1/\ )\frac{1}{2})$, which is more than the running time of the sequential algorithm of [4] 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 [7]. We study the performance of counting unlabeled/labeled templates with up to 12 vertices. The total running times for templates with 12 vertices 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. [8]).

3. 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 [6]. Except for a 10-vertex template which produces an extremely large amount of data so as to incur the I/O bottleneck on the virtual disk of EC2. It is worth noting here that 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.

4. We study the performance improvement for some 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%.

5. Finally, we implement SAHAD within the Harp [33] framework – the new algorithm is called HARPSAHAD+. HARPSAHAD+ yields two order of magnitude improvement in performance, as a result of its flexibility in task scheduling, data flow control and in memory cache. We are therefore able to scale to networks with up to billions of edges using the HARPSAHAD+ and obtain a comparable performance when compared to a state-of-the-art MPI/C++ implementation.

Organization. Section 3 introduces the background of the subgraph counting problem and MapReduce, as well as 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 [4]. In Section 5 we present our MapReduce implementations and its variations including SAHAD, EN-SAHAD and HARPSAHAD+. In Section 6 we study the computation cost of our algorithm. Section 7 discusses various experiment results and findings. Finally, Section 8 concludes the paper.

Extension from conference version. The SAHAD algorithm appeared in [47]. The results on EN-SAHAD and HARPSAHAD+ are new additions. Since the publication of [47], there has been more work done on parallelizing the color coding

technique, e.g., [37], [38]. 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 [13], [16], [17], [21], [28], [42] for complexity theoretic results. Marx and Pilipczuk [28] 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 [18], [22], [24]. These approaches involve candidate generation during a breadth first search on the subset lattice and a determination of the support of item sets by a subset test. A variety of optimizations have been developed, e.g., using a depth first search order to avoid the cost of candidate generation [19], [44] or pruning techniques, e.g., [24]. A related problem is that of computing the "graphlet frequency distribution", which generalizes the degree distribution [32].

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., [4], [20], [46], which has been used for approximating the number of embeddings of templates that are trees or "tree-like".

In [4], 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 [20], 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. A recent work developed by Venkatesan *et al.* [10] extends color coding to subgraphs with treewidth up to 2, and they scale their algorithm to graphs with up to 2.7 million edges.

Most of these approaches in bioinformatics applications involve small templates, and have only been scaled to relatively small graphs with at most 10^4 vertices (apart from [46], 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., [8], [35], [36], and scale to very large graphs. For instance, Bröcheler *et al*. [8] demonstrate labeled subgraph queries with up to 7-vertex 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 an OpenMP implementation of the color coding approach is given in [37]. This algorithm achieves a speed up of 12 in a graph with 1.5 million vertices and 31 million edges. A more recent work [38] parallelizes the dynamic processing of the color-coding algorithm to enumerate subgraphs and is able to handle networks as large as 2 billion edges, with template size up to 10 vertices.

3 Background

3.1 Preliminaries and problem statement

We consider labeled graphs $G = (V_G, E_G, L, \ _G)$, where V_G and E_G are the sets of vertices and edges, L is a set of labels and $_G : V \quad L$ is a labeling on the vertices. A graph $H = (V_H, E_H, L, \ _H)$ is a *non-induced subgraph* of G if we have $V_H \quad V_G$ and $E_H \quad E_G$. We say that a template graph $T = (V_T, E_T, L, \ _T)$ is isomorphic to a non-induced subgraph $H = (V_H, E_H, L, \ _H)$ of G if there exists a bijection $f : V_T \quad V_H$ such that: (i) for each $(u, v) \quad E_T$, we have $(f(u), f(v)) \quad E_H$, and (ii) for each $v \quad V_T$, we have $_T(v) = \ _H(f(v))$. In this paper, we assume T is a tree. We will consider trees to be rooted, and use $= (T) \quad V_T$ 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 (T) mapped to vertex $f(\ (T))$. 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, taking automorphisms into account. Therefore, an embedding H will be counted only once here even if there exist multiple mappings $f(\cdot)$ that map T to H. Here, we focus on approximating $emb(T, G)$.

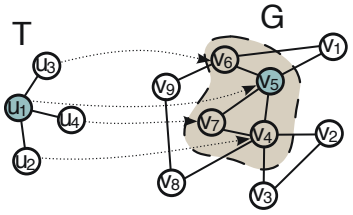


Fig. 1: Here the shaded subgraph is a non-induced embedding of T. The mapping of the template to the subgraph is denoted with the arrow.

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

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) dividing the input into distinct *map* tasks and distributing to multiple computing entities, and (b) merging the results of individual computing entities in the *reduce* tasks to produce the final output [14].

The MapReduce model processes data in the form of key-value pairs k, v . An application first takes pairs of the form k_1, v_1 as input to the map function, in which one or more k_2, v_2 pairs are produced for each input pair. Then the MapReduce re-organizes all k_2, v_2 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 [43] 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 in order to perform the tasks, without knowing the complexity of load balancing, communication and task 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 the mapper, shuffling and the reducer since the data is always read and write from the disk in every stage of a Hadoop job and (*ii*) global synchronization of the mapper and reducer, i.e. reducers can start only when all mappers have completed their tasks and vice versa, thus reducing the efficient usage of the computing resources. To conquer the problems that Hadoop is facing, we further extend our work to use the Harp platform [33].

Harp introduces full collective communication (broadcast, reduce, allgather, allreduce, rotation, regroup or push & pull), adding a separate communication abstraction. The advantage of 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. Figure 2 shows the four categories of the computing model.

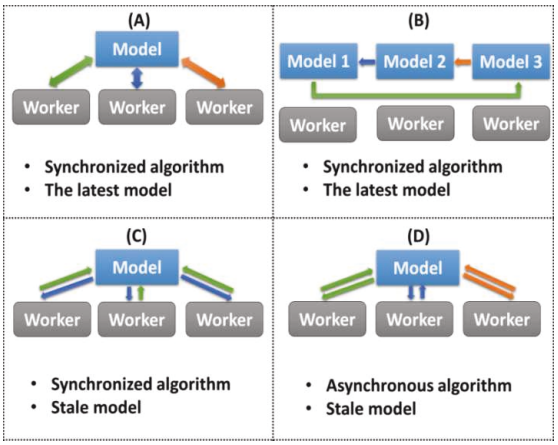


Fig. 2: Harp has 4 computation models: (A) Locking,(B) Rotation, (C) AllReduce, (D) Asynchronous

The Harp framework has been used by 350 students at Indiana University for their course projects. Now it has been released as an open source project that is available at the 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.

4 The Sequential Algorithm: Color Coding

TABLE 1: Notations

symbol	description	symbol	description
G	graph	T, T′, T″	template and sub-templates
n, m	# vertices, # edges	k	# vertices in T
ρ	root of T	S, S_i	color set, the i^{th} color
d(v)	degree of vertex v	N (v)	neighbors of vertex v

We briefly introduce the color coding algorithm for subgraph counting [5], 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 [5].

1) **Colorful embeddings**:
   Color the vertices of the graph with k colors where k $|V_T|$, and only count "colorful" embeddings—an embedding H of the template T is colorful if each vertex 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 $C(v, T(\ ), S)$ be the number of colorful embeddings of T with vertex v V_G mapped to the root , and using the color set S, where $|V_T| = |S|$.
   b) Suppose ($= u_1, u_2$) is an edge incident on the root vertex 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 $_1 = u_1$ and $_2 = u_2$ 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 $_1$ and $_2$ mapped to neighboring vertices v_1 V_G and v_2 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 S_2 of S.

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

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

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 $v \quad V_G$, pick a color $c(v) \quad S = \{1, \ldots, k\}$ uniformly at random, where $k = |V_T|$.
5: Partition the tree T into subtrees recursively to form a set $\mathcal{T}$ using algorithm PARTITION($T(\ )$). For each tree $T' \quad \mathcal{T}$, we have a root $'$. Furthermore, if $|V_{T'}| > 1$, T' is partitioned into two trees T'_1, T'_2 with roots $_1 = '$ and $'_2$, respectively, which are referred to as the active and passive children of T'.
6: For each $v \quad V_G, T_i \quad \mathcal{T}$ with root $_i$, and subset $S_i \quad S$, with $|S_i| = |T_i|$, we compute $C(v, T_i(\ _i), S_i)$ using the the recurrence (1) below:

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

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

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

where q denotes the number of vertices $' \quad V_T$ such that T is isomorphic to itself when $\ $ is mapped to $'$.
8: Repeat the above steps $N = O(\frac{e^k \log(1/\ )}{2})$ times [5], and partition N estimates $C^{(1)}, ..., C^{(N)}$ into $t = O(\log(1/\ ))$ sets. Let Z_j be the average of set j. Output the median of $Z_1, ..., Z_t$.

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

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

5 PARALLEL ALGORITHMS

In this section, we present a parallelization of the color coding approach using MapReduce framework, we will first describe SAHAD [47], followed by EN-SAHAD and HARPSAHAD+ 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 a MapReduce variation of Algorithm 1 to compute the number of embeddings of T.

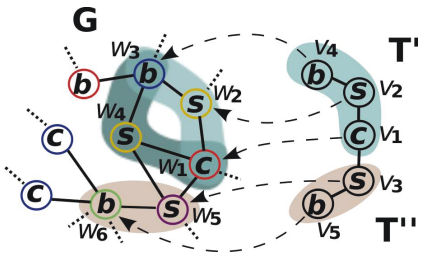


Fig. 3: The example shows one step of the dynamic programming in color coding. T in Figure 1 is split into T' and T''. To count $C(w_1, T(v_1), S)$, or the number of embeddings of $T(v_1)$ rooted at w_1, using color set $S = \{\underline{r}ed, \underline{y}ellow, \underline{b}lue, \underline{p}urple, \underline{g}reen\}$, 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 vertices $\{w_3, w_4, w_1, w_5, w_6\}$ and $\{w_3, w_2, w_1, w_5, w_6\}$. Here s, c, b represents the label of the vertices. Details of labeled subgraph counting can be found at [47].

As shown in Equation 1, the counts of all colorful embeddings isomorphic to T rooted from a single vertex 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 $u \quad N(v)$. 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 represents a color set containing k colors, and c_i represents 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 randomly color network G with k colors. The map function is described in Algorithm 3:

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

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

Here "Collect" is a standard MapReduce operation that 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 vertex, 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*$(v, X_{t,v}, N(v))$

```
if t is T_i' then
    Collect(key ← v, value ← X_{t,v}, flag')
else
    for u ∈ N(v) do
        Collect(key ← u, value ← 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 vertex v, the number of entries with $flag'$ is 1, and the number of entries with $flag''$ equals $|N(v)|$.

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

```
pick X_1 where flag = flag'
for all colorset S_i' from X_1 do
    for each X other than X_1 do
        for all colorset S_i'' from X do
            if S_i' ∪ S_i'' = ∅ then
                c(v, T_i, S_i' ∪ S_i'') += 1
Collect(key ← v, value ← 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*$(v, X_{T,v}, N(v))$

```
Collect(key ← "sum'', value ← X_{T,v})
```

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

```
Y = (m^m / m!) · (1/q) · Σ_{∀v∈V_G} X
Collect(key ← "sum'', value ← 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 [47].

5.2 EN-SAHAD

For general MapReduce problems, the set of keys that are processed in the Mapper and Reducer vary among different 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 program aggregates counts based on the root vertex of the subtree, and therefore the key is the vertex index v. In EN-SAHAD, we use this pre-knowledge to predefine a reducer that corresponds to a set of vertices. We also assign the predefined reducers to computing nodes prior to the beginning of the dynamic program. 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 HARPSAHAD+

HARPSAHAD+ is built upon the Harp framework [45] [11], which adopts a variety of advanced technologies in the research area of high performance Java. HARPSAHAD+ has the following optimizations when compared to the MapReduce Sahad version: 1) It uses a two-level parallel programming model. At the inter-node level, workload is distributed by Harp mappers; At the intra-node level, local workload is divided and assigned to multiple Java threads. 2) For inter-node communication, it utilizes a MPI-AlltoAll like regroup operation owned by Harp. 3) For intra-node computation, it utilizes Habanero Java thread library from Rice University [9] and adopts a Long-Running-Thread programming style [15] to unleash the potential performance of the Java language.

5.3.1 Inter-Node Communication

In SAHAD, the template counts of a vertex v and all of its neighbours $N(v)$ are assigned the same key value v, therefore, they are shuffled into the same reducer to complete the counting process. In HARPSAHAD+ however, we remove the reducer module and replace it by a user-defined mapper function. In this function, the whole set of vertices V is distributed and cached into the memory space of p Harp mappers. Furthermore, each mapper i holds a subset of vertices V_i where $s_i = |V_i|$. In the mapper function, we create a table $LTable$ with s_i entries, and each entry $0 \quad j < s_i$ serves as a “reducer” for vertex v_j. HARPSAHAD+ then uses a regroup operation to “shuffle” the data within the memory in a collective way. Additionally, each mapper function creates another Harp Table object $RTable$, containing multiple partitions, to transfer data. A preprocessing function is fired to record re-usable information required by regroup operations in each iteration. In the preprocessing stage, each mapper holds a copy of all the vertex IDs v and the mapper ID $j, v \quad V_j$ by an allgather communication operation. The mapper then parses the neighbour lists $N(v)$ of all the local vertices V_i and labels each vertex u, where $u \quad N(v)$ but $u \ / \ V_i$, with a mapper ID j that satisfies $u \quad V_j$. Therefore, each mapper i keeps a queue of vertex IDs for each mapper $j = i$ with $v \quad Q_{i,j}, v \quad V_j$. By sending $Q_{i,j}$ to mapper j, finally each mapper j obtains a sending queue $Q_{j,i}$ of vertices.

In each iteration of HARPSAHAD+, the regroup operation fired by mapper i has three steps: 1) For each sending queue $Q_{i,j}$, subtemplate counts of v are loaded for sending queue $Q_{i,j}$ into a partition $Par_{i,j}$ of $RTable$. 2) The sender and receiver mapper identities, i and j, are coded into a

single partition ID for $Par_{i,j}$. During the collective regrouping, a designed Harp partitioner will decode the partition ID and deliver the partition $Par_{i,j}$ to the receiver mapper j. 3) After the regroup operation, the Harp Table $RTable$ of each mapper i now contains counts of vertices $u \quad N(v)$ to update subtemplate counts of local vertices v in the $LTable$.

5.3.2 Intra-Node Computation

HARPSAHAD+ extends the MapReduce framework by taking advantage of the multi-threading programming model in a shared-memory node. We favor the Habanero Java threads instead of the Java.lang.Thread implementation because it allows users to set up thread affinity in multi-core/many-core processors. We also embrace the so-called Long-Running-Thread programming style, where we create the threads at the outermost loop and keep them running until the end of the program. This approach avoids the overhead of frequently creating and destroying threads, and instead uses the java.util.concurrent.CyclicBarrier object to synchronize threads if required.

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 vertices and edges in the network by n and m respectively. Similarly we use k to represent the number of vertices 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. As a result, the sizes of the input, output, and work complexity corresponding to a vertex 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 vertex 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 vertex v, in which each key-value pair corresponds to some vertex $u \quad N(v)$. 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 $O(k|E_G|2^{2k}e^k \log(1/\ )^{\frac{1}{2}})$.

Proof For vertex v and each neighbor $u \quad N(v)$, 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 $|\mathcal{T}| \quad k$, the total work, over all vertices 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 $O(e^k \log(1/\ )^{\frac{1}{2}})$ iterations are performed in order to get the $(\ ,\ )$-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 vertices.

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 vertex 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 $u \quad N(v)$, 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 vertices, 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 $v \quad V(G)$, which takes $O(n)$ time. ∎

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

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

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-templates 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 $O(e^k \log(1/\ )^{\frac{1}{2}})$ times, in order to get the $(\ ,\ )$-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 4.

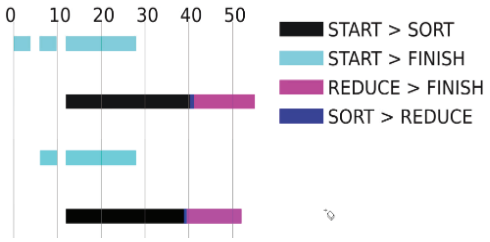


Fig. 4: The figure shows the time spent in each stage of a running Hadoop job to produce a color-count for a 5-vertex template, by aggregating the 2-vertex and 3-vertex sub-tree. The black bar is the time for the intermediate stage, which is for shuffling and sorting.

We observe that the external shuffling and sorting stage takes roughly twice the time of the reducing stage, which dramatically increase the overall running time. Given that the keys in Mappers and Reducers are always the index of all the vertices $v \quad V(G)$, we can enhance SAHAD by removing the shuffling and sorting in the intermediate stages. Instead, we can designate Reducers and directly send the data to corresponding Reducers.

7 Experimental Analysis of SAHad, En-SAHad & HarpSAHad+

We carry out a detailed experimental analysis of SAHAD, EN-SAHAD and HARPSAHAD+, by focusing on three aspects:

(*i*) *Quality of the solution*: We compare the color coding results with exact counts on small graphs in order to measure the empirical approximation error of our algorithms and show that the error is very small (less than 0.5% with one iteration as shown in Figure 7) 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 sizes ranging from 3 to 12 vertices, including both labeled and unlabeled templates. The graphs we use go from several hundreds of thousands of vertices 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, as well as different settings of mappers and reducers, etcetera.

(*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. Table 2 highlights the main results we obtained with various methods.

7.1 Experiment Design

7.1.1 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 [7]

TABLE 2: Comparison on SAHAD, EN-SAHAD and HARP-SAHAD+

Method	Network	Templates	Performance
SAHAD	268M edges	12 vertices	tens of min for 7 vertex template on Chicago
EN-SAHAD	12M edges	5 vertices	20% improvement over SAHAD
HARPSAHAD+	1.2B edges	up to 12 vertices	100-200 times faster than SAHAD

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 vertices, where each pair of vertices are connected with probability p, and are randomly assigned vertex labels. We also experiment on a few other networks: Web-Google [2], RoadNet (rNet) [2], Twitter [25] and Chung-Lu random graphs [12]. Table 3 summarizes the characteristics of the networks.

TABLE 3: Networks used in the experiments

Network	No. of vertices(in million)	No. of Edges(in million)
Twitter	41.7	1202.5
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

7.1.2 Templates

The templates we use in the experiments are shown in Figure 5. The templates vary in size from 5 to 12 vertices, in which *U5-1,. . .U10-1* are the unlabeled templates and *L7-1 ,L10-1* as well as *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.

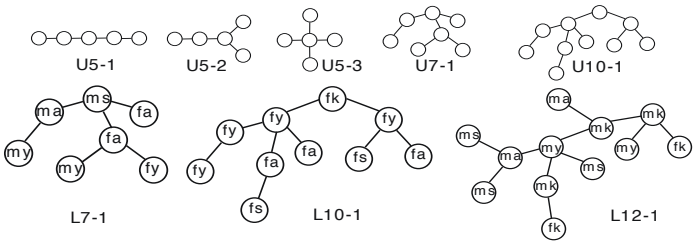


Fig. 5: Templates used in the experiments.

7.1.3 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 7.2.2. 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 the computing process 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 HARPSAHAD+, we use the Juliet cluster (Intel Haswell architecture) with 1, 2, 4, 8 and 16 nodes. The 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 InfiniBand FDR with 10Gbit/s per link.

7.1.4 Performance metrics

We carry out experiments on SAHAD, EN-SAHAD and HARPSAHAD+. 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. Then using 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 HARPSAHAD+ and SAHAD to study the improvement Harp brings.

7.2 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.

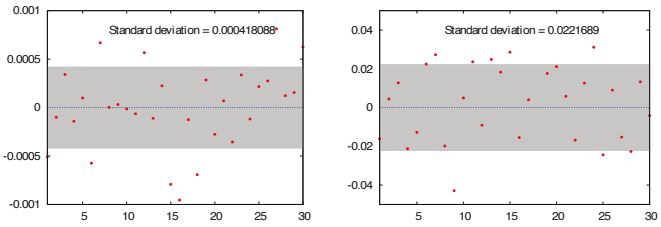
1. Approximation bounds: While the worst case bounds on the algorithm imply $O(e^k \log(1/\ ) \frac{1}{2})$ rounds to get an $(\ ,\ )$-approximation (see Lemma 6.2), in practice, we find that far fewer iterations are needed.

2. System performance: We run our algorithm on a diverse set of computing resources, including the publicly available Amazon EC2 cloud. Here, we find that 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. Here, we find that 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.

7.2.1 Approximation bounds

As discussed in Section 3, the color coding algorithm averages the estimates over multiple iterations. Figure 6 shows 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.



(a) Miami (b) Web-Google

Fig. 6: Error in counting *U5-1* for 30 iteration

In Figure 7, we show that the approximation error is below 0.5% for the template *U7-1* for the GNP100 graph, 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 [20]. 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 $\frac{|(\sum_i Z_i)/i - emb(T,G)|}{emb(T,G)}$.

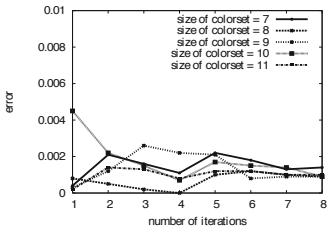


Fig. 7: Approximation error in counting *U7-1* on GNP100.

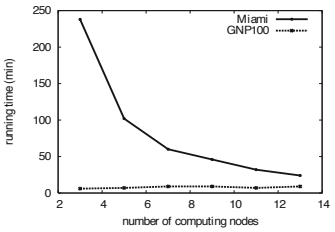


Fig. 8: Running time for counting *U10-1* vs number of computing nodes.

7.2.2 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 8 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

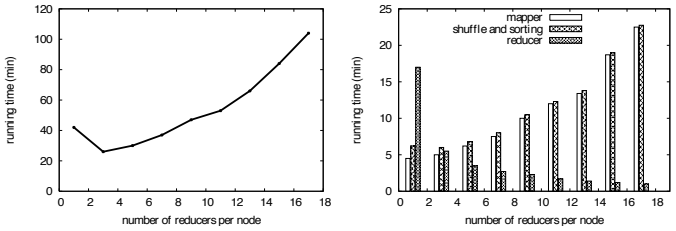
TABLE 4: Summary of the experiment results (refer to Section 7.1 for the terminology used in the table)

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 5 and networks from Table 3	all tasks complete in less than 35 minutes

portion of the running time, and there is overhead 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 Here we consider two cases.

2.a. Varying number of reducers per node. Figure 9 shows 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. In addition we find that although 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. This can be explained by the visible I/O bottleneck for concurrent accessing on Athena, since Athena has only 1 disk per node. This phenomenon is not present on EC2, as seen from Figure 11b, indicating that EC2 is better optimized for concurrent disk accessing for cloud usage.

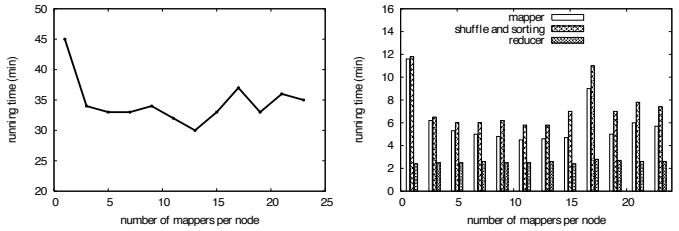


(a) Total running time v.s. number of reducers. (b) Running time of job stages v.s. number of reducers.

Fig. 9: Running time v.s. number of reducers per node

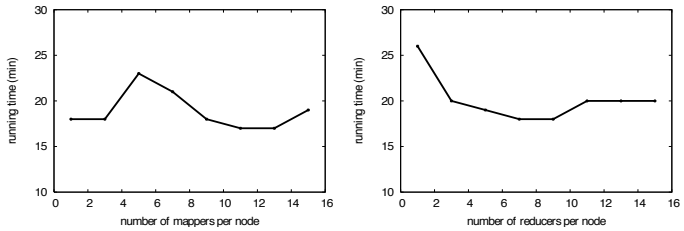
2.b. Varying number of mappers per node. Figure 10 shows the running time on Athena when we vary the number of mappers per node while fixing the number of reducers as 7 per node. We find that varying the number of mappers per node does not affect the performance. This is also validated in EC2, as shown in Figure 11.

2.c. Reducers' running time distribution. Figure 12 shows 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.



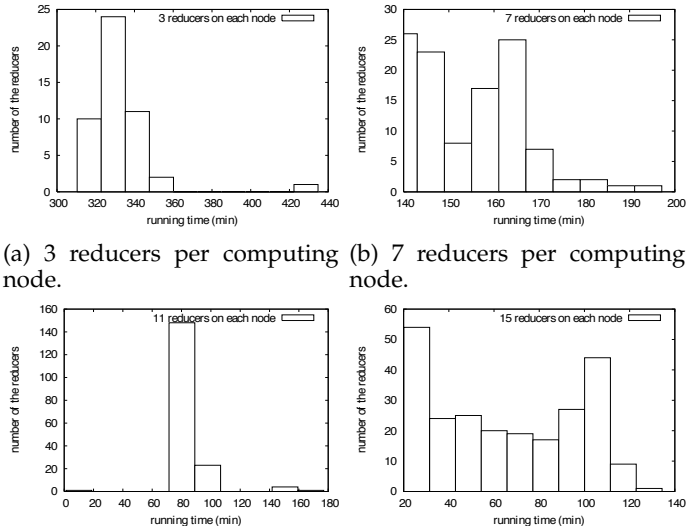
(a) Total running time v.s. number of mappers. (b) Running time of job stages v.s. number of mappers.

Fig. 10: Running time v.s. number of mappers per node



(a) Total running time v.s. number of mappers on EC2. (b) Total running time v.s. number of reducers on EC2.

Fig. 11: Running time w.r.t. number of mappers and reducers on EC2.



(a) 3 reducers per computing node. (b) 7 reducers per computing node.

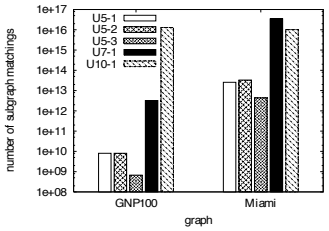
(c) 11 reducers per computing node. (d) 15 reducers per computing node.

Fig. 12: Reducers completion time distribution.

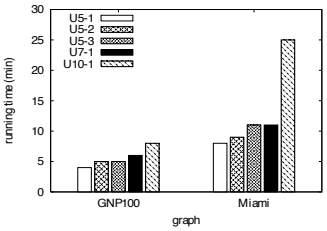
7.2.3 Illustrative applications

In this section, we illustrate the performance on 3 different kinds of queries. We use Athena and assign 16 nodes as 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 7.2.4.

1. Unlabeled subgraph queries: Here we compute the counts of templates *U5-1*, *U7-1* and *U10-1* on GNP100 and Miami, as well as the running time, as shown in Figure 13 – we observe that for unlabeled templates with up to 10 vertices on the Miami graph, the algorithm runs in less than 25 minutes.



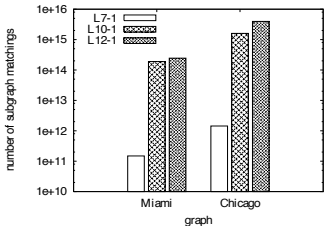
(a) The counts of unlabeled subgraphs.



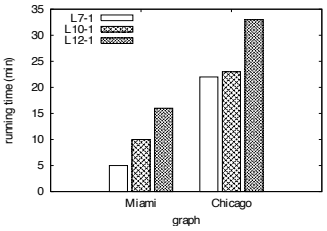
(b) Running time for counting unlabeled subgraphs.

Fig. 13: Querying unlabeled subgraphs on GNP100 and Miami

2. Labeled subgraph queries: Here we count the total number of embeddings of templates *L7-1*, *L10-1* and *L12-1* in Miami and Chicago. Figure 14b shows that the running time for counting templates up to 12 vertices is around 15 minutes on Miami, which is 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. This is due to the fact that labeled templates contain a much fewer number of embeddings due to the label constraints.



(a) The counts of labeled subgraphs.

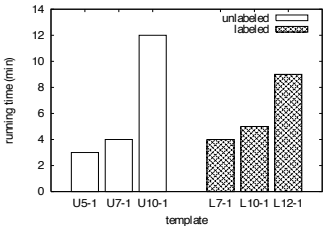


(b) Running time for counting labeled subgraphs.

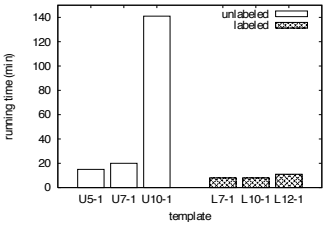
Fig. 14: Querying labeled subgraphs on Miami and Chicago.

7.2.4 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*. Here 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 Figure 15, 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 large templates and graphs as large as Miami, the input/output data as well as the I/O pressure on disks is tremendous. EC2 uses virtual disks as local storage, which hurt overall performance when dealing with such a large amount of data.



(a) GNP100



(b) Miami

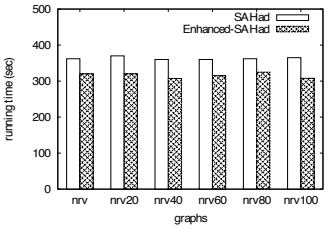
Fig. 15: Running time for various templates on EC2.

7.3 Performance of EN-SAHAD

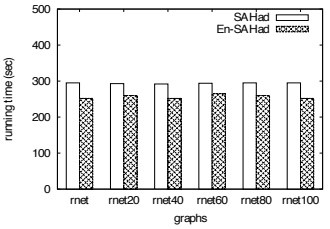
In this section we experiment our algorithms on two real-world networks *NRV* and *RoadNet* and a number of their shuffled versions. We generate shuffled networks with 20, 40, 60, 80 and 100 percent shuffling ratio, and name them as *nrv20* to *nrv100*, and *rNet20* to *rNet100*.

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

Figure 16 shows 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. As a result, we observe that pre-allocating a Reducer can deliver roughly a 20% performance improvement.



(a) NRV and its variations



(b) RoadNet and its variations

Fig. 16: SAHAD v.s. EN-SAHAD on RoadNet and NRV.

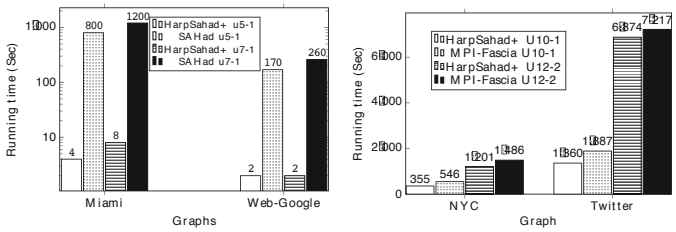
7.4 Performance of HARPSAHAD+

In the following experiments, we evaluate the performance of HARPSAHAD+ by comparing it with a state-of-the-art MPI subgraph counting program called MPI-Fascia. MPI-Fascia is developped by Slota et al. [38], which implements the same color coding algorithm as SAHAD and HARP-SAHAD+. MPI-Fascia uses a MPI+OpenMP programming model. In our tests, it is compiled with g++ 4.4.7 and

compiler option -O3 as well as OpenMPI 1.8.1. Also, we choose InfiniBand instead of Ethernet as the interconnect to test MPI-Fascia and HARPSAHAD+, thus offering more challenges to the Java based communication operation of HARPSAHAD+.

7.4.1 Execution Time

In Figure 17a, we observe that HARPSAHAD+ has a 100x to 200x speedup over SAHAD on a single Haswell node. This tremendous improvement comes from two sides: 1) HARPSAHAD+ has a better utilization of the hardware resources (logical cores) by using Habanero Java threads and affinity binding. 2) Compared to the disk based shuffle process of SAHAD, HARPSAHAD+ caches all of the data in main memory, which significantly reduces the overhead of data access. In Figure 17, we compare HARPSAHAD+



(a) SAHAD vs. HARPSAHAD+ (b) HARPSAHAD+ vs. MPI-Fascia

Fig. 17: (a) Test on 1 Haswell Node and each node running 40 threads; (b) Test on 16 Haswell Nodes and each node running 24 threads

with MPI-Fascia on a Twitter dataset with templates of large size in a distributed environment of 16 Haswell nodes. HARPSAHAD+ achieves comparable or even slightly better performance than MPI-Fascia, which comes from its optimized communication operations. Figure 18 illustrates a breakdown of the execution time into computation and communication on Twitter with template U12-2. Because of the highly intensive computation workload, MPI-Fascia consumes less time in computation thanks to the compiler-level O3 optimization. However, HARPSAHAD+ as a pure Java implementation can still achieve almost the same total counting time with the help of optimized collective communication operations.

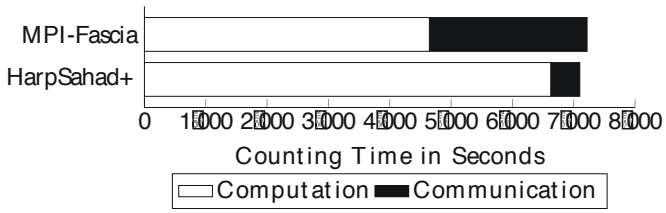
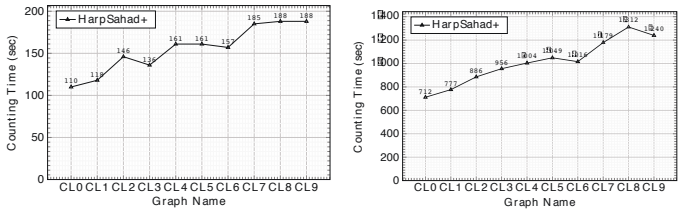


Fig. 18: Breakdown of Time for Twitter-U12-2 on 16 Nodes

7.4.2 Problem Size Scaling

Next we study the performance of HARPSAHAD+ by controlling the number of vertices in a graph while increasing the number of edges. In this experiment, we use the Chung-Lu model [12] to generate a series of random graphs given the degree sequence and its variations of Miami and NYC. The average degree of the generated random graphs range from 50 to 150 for Miami and 10 to 100 for NYC. In Figure 19, the running time generally increases with the number of edges, which meets the time complexity we propose in Section 6. For Miami, when the average degree increases from 50 to 150, the running time only increases by 1.7x. Also, a tenfold (x10) increase in average degree for the NYC graph only accounts for less than 2x of an increase in running time. This indicates that our HARPSAHAD+ implementation maintains good performance in computing the neighbours of vertices in parallel, which is due to the high efficiency of Java threads.



(a) Miami Dataset (b) NYC Dataset

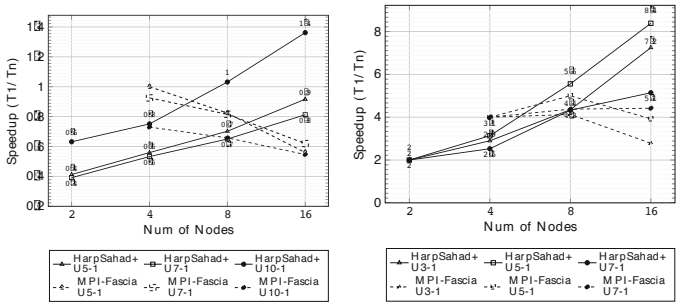
Fig. 19: (a) Test on Miami graph, Template U10-1, 4 Haswell Nodes, and 40 threads/node; (b) Test on NYC graph, Template U10-1, 4 Haswell Nodes and 40 threads/node;

7.4.3 Varying number of computing nodes

In this section, we study the performance of HARPSAHAD+ as a function of computing resources, i.e., computing nodes and threads per node. In Figure 20, we compare the inter-node strong scaling test results between HARPSAHAD+ and MPI-Fascia. For the NYC dataset, we ran strong scaling tests on three templates, and the value of the y-axis represents the speedup on N nodes by dividing the time on a single node by the time on N nodes. Since the NYC dataset is relatively small for HARPSAHAD+ and MPI-Fascia, both of the two implementations are not bounded by the computation overhead, which prevents them from achieving the linear speedup. However, HARPSAHAD+ (solid lines) still obtains a better strong scalability than MPI-Fascia (dashed lines). Furthermore, MPI-Fascia could not run on two nodes due to a memory capacity bottleneck and it shows no scalability after 4 nodes. For the Twitter Dataset, HARPSAHAD+ again outperforms MPI-Fascia after 4 nodes. The speedup is also improved as Twitter gives a much larger workload than NYC and HARPSAHAD+ is more bounded by computation overhead.

8 CONCLUSION

In this paper 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. Hence, we first develop SAHAD – a Hadoop based implementation and also provide performance analysis in terms of work and time complexity. After observing large sorting, communication and I/O costs in SAHAD, we further explore two approaches to remedy



(a) NYC Dataset (b) Twitter Dataset

Fig. 20: (a) Test on NYC graph each node running 24 threads; (b) Test on Twitter graph each node running 24 threads

these problems. The first approach called EN-SAHAD, entails the tight coupling of the number of graph vertices to mappers and reducers, so as to reduce the sorting and shuffling phases of the MapReduce jobs. The second approach is the implementation of the color coding algorithm using the Harp framework, called HARPSAHAD+, which employs collective communication and shared memory to better facilitate computation and communication. Our experiments show that HARPSAHAD+ has significantly improved performance when compared to SAHAD — by almost two orders of magnitude, and simultaneously achieves comparable or even better execution time and scalability than state-of-the-art MPI solutions. HARPSAHAD+ can process networks with 1.2 Billion edges and 12 vertex templates. We also explore the performance of these implementations on different cluster architectures such as EC2 on-demand nodes and Intel Haswell nodes. As directions for future research, it would be interesting to devise new algorithms that scale to larger instances. Additionally, it would be interesting to implement a 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] 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.

[5] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):856, 1995.

[6] Amazon. Elastic computing cloud (ec2). http://aws.amazon.com/ec2.

[7] 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.

[8] 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.

[9] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[10] V. T. Chakaravarthy, M. Kapralov, P. Murali, F. Petrini, X. Que, Y. Sabharwal, and B. Schieber. Subgraph counting: Color coding beyond trees. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 2–11. Ieee, 2016.

[11] L. Chen, B. Peng, B. Zhang, T. Liu, Y. Zou, L. Jiang, R. Henschel, C. Stewart, Z. Zhang, E. Mccallum, T. Zahniser, O. Jon, and J. Qiu. Benchmarking Harp-DAAL: High Performance Hadoop on KNL Clusters. In *IEEE Cloud 2017*, Honolulu, Hawaii, US, June 2017.

[12] F. Chung and L. Lu. Connected components in random graphs with given expected degree sequences. *Annals of combinatorics*, 6(2):125–145, 2002.

[13] 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.

[14] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] S. Ekanayake, S. Kamburugamuve, P. Wickramasinghe, and G. C. Fox. Java thread and process performance for parallel machine learning on multicore HPC clusters. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 347–354, Dec. 2016.

[16] J. Flum and M. Grohe. The parameterized complexity of counting problems. *SIAM Journal on Computing*, 33(4):892–922, 2004.

[17] F. V. Fomin, D. Lokshtanov, V. Raman, S. Saurabh, and B. Rao. Faster algorithms for finding and counting subgraphs. *Journal of Computer and System Sciences*, 78(3):698–706, 2012.

[18] L. Getoor and C. Diehl. Link mining: a survey. *ACM SIGKDD Explorations Newsletter*, 7(2):3–12, 2005.

[19] 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.

[20] F. Hüffner, S. Wernicke, and T. Zichner. Algorithm engineering for color-coding with applications to signaling pathway detection. *Algorithmica*, 52(2):114–132, 2008.

[21] 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.

[22] 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.

[23] I. Koutis and R. Williams. Limits and applications of group algebras for parameterized problems. In *Proc. ICALP*, pages 653–664, 2009.

[24] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data mining and knowledge discovery*, 11(3):243–271, 2005.

[25] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

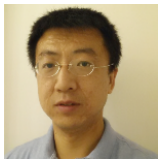
[30] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824, 2002.

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

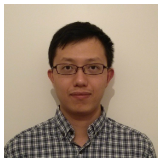
[32] N. Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177, 2007.

[33] 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.

[34] 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.
[35] 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.
[36] 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.
[37] 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.
[38] G. M. Slota and K. Madduri. Parallel color-coding. *Parallel Computing*, 47:51–69, 2015.
[39] 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.
[40] 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.
[41] 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.
[42] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.
[43] T. White. *Hadoop: The definitive guide*. Yahoo Press, 2010.
[44] 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.
[45] B. Zhang, Y. Ruan, and J. Qiu. Harp: Collective communication on Hadoop. *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, pages 228–233, 2015.
[46] 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.
[47] 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.



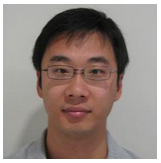
Zhao Zhao is pursuing his Ph.D degree in Computer Science at Virginia Tech. He is also a Software Engineer in Verisign Labs, Verisign Inc. His research interests are in Network Science and analytics, especially in the design and analysis of parallel graph algorithms.



Langshi Chen is a Postdoctoral researcher at the School of informatics and Computing in Indiana University. His research interests include linear solvers for HPC systems, energy efficiency of HPC applications, data intensive machine learning applications on manycore architectures, and so forth.

Mihai Avram is currently a Masters student who is studying Computer Science at Indiana University. His research interests involve applying various CS sub-domains such as Big Data, High Performance Computing, IoT, Machine Learning, HCI, and Data Mining to solve large scale social problems.

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.