Fast Approximate Subgraph Counting and Enumeration

George M. Slota Kamesh Madduri
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA, USA
Email: {gms5016, madduri}@cse.psu.edu

***Abstract*—We present a new shared-memory parallel algorithm and implementation called FASCIA for the problems of approximate subgraph counting and subgraph enumeration. The problem of subgraph counting refers to determining the frequency of occurrence of a given subgraph (or *template*) within a large network. This is a key graph analytic with applications in various domains. In bioinformatics, subgraph counting is used to detect and characterize local structure (*motifs*) in protein interaction networks. Exhaustive enumeration and exact counting is extremely compute-intensive, with running time growing exponentially with the number of vertices in the template. In this work, we apply the *color coding* technique to determine approximate counts of *non-induced occurrences* of the subgraph in the original network. Color coding gives a fixed-parameter algorithm for this problem, using a dynamic programming-based counting approach. Our new contributions are a multilevel shared-memory parallelization of the counting scheme and several optimizations to reduce the memory footprint. We show that approximate counts can be obtained for templates with up to 12 vertices, on networks with up to millions of vertices and edges. Prior work on this problem has only considered out-of-core parallelization on distributed platforms. With our new counting scheme, data layout optimizations, and multicore parallelism, we demonstrate a significant speedup over the current state-of-the-art for subgraph counting.**

***Index Terms*—subgraph counting; motifs; color coding**

I. INTRODUCTION

Subgraph isomorphism and its variants (tree isomorphism, subgraph enumeration, subgraph counting, motif finding, frequent subgraph identification) are fundamental graph analysis methods used to identify latent structure in complex data sets. They have far-reaching uses, including several applications in bioinformatics [1], [2], chemoinformatics [3], online social network analysis [4], network traffic analysis, and many other areas. The problem of counting and listing the occurrences of a certain subgraph (or a *template*) within a large graph, commonly termed subgraph enumeration, is widely used in many applications.

Subgraph isomorphism is an NP-complete decision problem. Subgraph counting and enumeration are also computationally very expensive. An algorithm based on exhaustive enumeration and exploring the neighborhood of every vertex in the graph requires $O(n^k)$ time, where n is the number of vertices in the graph, and k is the number of vertices in the template. The current best-known algorithm for obtaining exact counts and enumerating an arbitrary template within a graph is $O(n^{\frac{\omega k}{3}})$, where ω is the exponent of fast matrix multiplication [5]. The *color coding* technique by Alon, Yuster, and Zwick [6] reduces the running time of subgraph counting for bounded treewidth templates with constant treewidth, using a probabilistic algorithm, to $O(m \cdot 2^k \cdot e^k \cdot \frac{\log 1/\delta}{\epsilon^2})$, where m is the number of edges in the graph, and δ and ϵ are confidence and error parameters, respectively (i.e., the estimated count is within $c(1 \pm \epsilon)$ with probability $(1 - 2\delta)$, where c is the exact count). The color coding technique forms the basis for our work.

A. Our Contributions

We present a new parallel implementation FASCIA of the color coding based subgraph counting algorithm. The key computational phase in color coding is a dynamic programming approach to determine the counts of the template in the original graph. The dynamic programming phase is dependent on the subgraph structure. Our new implementation abstracts the main data structure used for maintaining counts with subproblems (subtemplates in this case, for the bottom-up dynamic programming scheme). This lets us experiment with various representations for the data structure, including several hash-based and array-based schemes. We use a combinatorial number-based indexing scheme to simplify the original subgraph counting method. We also investigate subgraph partitioning heuristics to reduce memory utilization and/or improve computation times. We currently fully support arbitrary undirected tree templates, but can also handle tree-like graphs templates with triangles. We quantify error incurred through comparisons with an exhaustive search approach. We also apply this counting strategy to motif finding and graphlet degree distribution estimation with several large-scale networks. Our implementation currently exploits multicore parallelism through OpenMP threading, and we achieve a 12× parallel speedup for the largest experiment considered (12-vertex template, graph with 31.2 million edges).

II. BACKGROUND

Subgraph isomorphism is formally defined as the decision problem of whether or not an input graph G_T can be embedded onto a graph G, such that there is a one-to-one mapping of every vertex and edge within G_T to a vertex and edge within G. If there exists such a mapping, it is said that

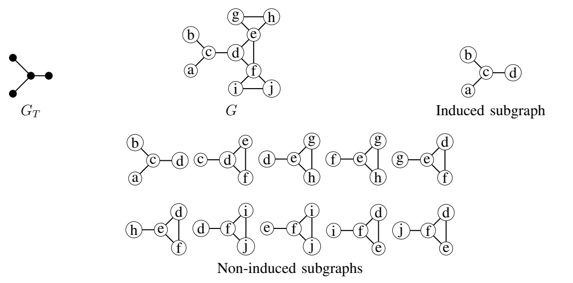


Fig. 1. Consider the query subgraph (or *template*) G_T and the graph G shown above. There is just one induced subgraph in G that is isomorphic to G_T, but there are 10 non-induced subgraphs isomorphic to the template.

G_T is *isomorphic* to G. Subgraph counting, which is the focus of this paper, is to count the total number of such possible embeddings. If we obtain counts for different simple subgraphs for multiple networks, we can then use these counts to compare and characterize the local and global structures of these networks.

The difference between *induced* and *non-induced* subgraph mappings is given in Figure 1. For an *induced* mapping, there exists no edge between any of the vertices in G to which G_T is mapped, except for where that edge correspondingly exists within G_T. A *non-induced* mapping can include such extra edges. The color coding technique (discussed below) counts non-induced occurrences of the subgraph. Counting non-induced subgraphs is challenging because template-specific graph pruning heuristics cannot be used. Enumerating non-induced subgraphs is also desirable in practice because real-world networks may include spurious edges or may be missing some edges. The counts obtained for non-induced occurrences can be much larger than the induced subgraph occurrence counts.

A. Motif Finding

The term *motif* is used to describe a subgraph with a higher than expected occurrence in a network. Motif counting within protein-protein interaction (PPI) and other biomolecular networks, by enumerating all possible subgraphs up to a given size, is of important consideration within the field of bioinformatics. Direct comparison of biological networks, such as through the minimum number of vertex/edge deletions to make the networks isomorphic, is a computationally challenging (NP-hard) problem. Additionally, although networks may share similar global topology, local structure may differ considerably. By comparing the frequency of appearance of possible subgraphs, it is argued that insight into network structure can be obtained. Motif frequency distribution analysis is also commonly utilized for similar reasons within social network analysis.

B. Graphlet Degree Distributions

Graphlet degree distribution [7] is a method for comparison of biological networks. A graphlet degree, analogous to vertex degree, can be thought of as the number of graphlet embeddings in the graph that contain a given vertex, where graphlet is another term for a subgraph. A graphlet frequency distribution is the count of vertices within a network that have a certain graphlet degree over the entire graph (i.e., 10 vertices have a graphlet degree of 7, 12 vertices have a graphlet degree of 5, etc.).

C. Color Coding

The color coding technique can be used to get an approximate count of the number of embeddings of a certain input template (template here being synonymous with the subgraph pattern) in a large graph. The algorithm is based on random coloring of a graph. The number of *colorful embeddings* (embeddings where each vertex in the template has a different color) in the graph are then counted, and that count is then scaled by the probability that the template is colorful.

What makes color coding so powerful is that it allows for a dynamic programming-based algorithm for counting when the template has a bounded treewidth. When the template is partitioned through single edge cuts, the count of the full template at a given vertex, for a given color set, is simply the sum of the counts of the sub-templates produced by the cut with all possible color mappings. It is possible to fully partition a tree template down to single vertices, which allows for a very fast bottom-up counting approach. While the algorithm is also applicable to any template that can be partitioned through a single cut, there is a tradeoff between allowing more complex input templates and a less efficient dynamic programming phase. Additionally, although the algorithm theoretically allows for directed templates and networks, we currently only analyze undirected templates and networks for simplicity. A more detailed overview of the algorithm is given in the next section.

D. Related Work

The color coding scheme forms the basis for several recent serial and parallel implementations. Alon et al. [2] developed a parallel implementation for motif counting in biomolecular networks. They calculate subgraph counts for all templates up to 10 vertices on networks of about 2,400 vertices on an 8 CPU server in 12 hours. Zhao et al. designed a parallel color coding-based subgraph enumeration method targeting distributed-memory platforms called PARSE [8]. They present efficiency and scalability results on a cluster with 400 CPUs, using templates of up to 10 vertices, and graphs with up to 20 million vertices. Zhao et al. further expand the algorithm for labeled templates using their Hadoop-based SAHAD scheme [9]. End-to-end processing times are under an hour for vertex-labeled templates up to 12 vertices, on networks of 9 million vertices.

Several other subgraph enumeration and counting algorithms focus on non-induced occurrences, although the MODA algorithm for biological motif detection by Omidi et al. is able to count both induced and non-induced occurrences simultaneously [10]. Other popular algorithms and software for biological motif detection through searching induced subgraph occurrences includes mfinder by Kashtan et al. [11],

Algorithm 1 Subgraph counting using color coding.

```
1: Partition input template $T$ ($k$ vertices) into subtemplates using *single edge cuts*.
2: Determine $Niter \approx \frac{e^k \log 1/\delta}{\epsilon^2}$, the number of iterations to execute. $\delta$ and $\epsilon$ are input parameters that control approximation quality.
3: for $i = 1$ to $Niter$ do
4:     Randomly assign to each vertex $v$ in graph $G$ a color between $0$ and $k-1$.
5:     Use a dynamic programming scheme to count *colorful* non-induced occurrences of $T$.
6: end for
7: Take average of all $Niter$ counts to be final count.
```

which uses random sampling to search for motifs up to 6 vertices in size, FANMOD by Wernicke [12], which uses an improved sampling method to decrease sampling bias and enhance scaling, and more recently, G-Tries by Ribeiro and Silva [13], which implements the g-trie data structure to represent collections of subgraphs.

III. FASCIA

We now introduce FASCIA (Fast Approximate Subgraph Counting and Enumeration), which is our implementation of the color coding-based subgraph enumeration algorithm, currently optimized for shared-memory workstations. In this section, we will discuss our new improvements on Alon et al.'s technique, including the subtemplate coloring abstraction, our memory-saving dynamic table layout strategy and fast hashing methods, as well as partitioning heuristics and parallelization.

A. Color Coding Technique Description

Algorithm 1 outlines the color coding technique. There are three main steps: template partitioning, random coloring, and the dynamic programming count phase. The FASCIA dynamic programming phase is described in Algorithm 2. The coloring and dyanmic programming steps are repeated for as many iterations as is initially specified. In order to guarantee an error bound of ϵ with confidence δ, we would need to run $Niter$ iterations as is defined in Algorithm 1. However, the number of iterations necessary in practice is far lower, as we will demonstrate in a later section.

In the input template partitioning phase, a single vertex is first specified to be the root of the template. A single edge adjacent to the root is cut, creating two children subtemplates. The child subtemplate containing the original root vertex is termed as the *active child*, with its root specified again as the original root vertex. The other child will be termed as the *passive child*, with its root as the vertex that was connected to the original root vertex through the edge that was cut. We now have two rooted subtemplates. We recursively continue to cut these subtemplates down to single vertices, keeping track of the *partitioning tree*, where each subtemplate greater than one vertex in size has both an active and passive child subtemplate. All subtemplates will have a single parent. This tree can be traced from the bottom up to the original template, which is how we will perform the dynamic programming step of the color coding algorithm. We also sort them in the order in which the subtemplates are accessed, in order to minimize memory usage.

The graph G is next randomly colored. For every vertex $v \in G$, we assign a color between 0 and $k-1$, where k is the maximum number of colors. k needs to be greater than or equal to the number of vertices in T. We will consider k equal to the size of T now for simplicity. Consider first a naïve table structure. We need to be able to store any arbitrary count for any arbitrary vertex for any arbitrary *color set*. For a given subtemplate S of size h, a color set can be considered the mapping of h unique color values to each vertex in S. We create a three dimensional tabular structure and initialize all values to zero. We can then proceed to the inner loops, which contain the dynamic step of the algorithm.

Algorithm 2 details the inner nested loops that we have for the algorithm. The outermost loop will perform, in order, the bottom-up count for each subtemplate, tracing along the partition tree that we created previously. For every single subtemplate, we will then consider every single vertex $v \in G$. If our subtemplate is a single vertex, we know that its count at v is 0 for all possible k color sets of a single vertex, except for the color set that consists of the color equal to the color randomly assigned to v, where it is 1.

If our subtemplate S of size h is larger than a single vertex, we know that it must have an active (a) and passive (p) child. We then look at all possible color sets C of size h with unique values. The count for this color set at v, which we will later store in our table at table[S][v][C], is initialized to zero. Next, we will consider for every neighbor, u, of v, the counts of a rooted at v and p rooted at u. We will then split C into C_a and C_p, which are the mappings onto the active and passive child of the colors in C. The count for S rooted at v with color set C is then the sum over all u and over all possible C_a and C_p of table[a][v][C_a]$\cdot$table[p][u][C_p].

Once we have run through as many iterations as initially specified, we can then take the average over all counts to be our estimate for the total number of embeddings in the graph. We return this value and the algorithm is complete.

B. Combinatorial Indexing System

We now discuss some of the improvements in our implementation over the baseline algorithm. We represent color sets as a single integer. This representation considerably simplifies table accesses and stores for any arbitrary color set of arbitrary size. It also avoids having to explicitly define, manipulate, and pass arrays or lists of color set values. In order to ensure that each combination of colors is represented by a unique index value, these values are calculated based on a combinatorial indexing system. For a subtemplate S of size h with k possible colors, the color set C would be composed of colors $c_1, c_2, \ldots, c_h$, each of possible (unique and increasing)

Algorithm 2 The dynamic programming routine in FASCIA.

```
1:  for all sub-templates $S$ created from partitioning $T$, in
    reverse order they were created during partitioning do
2:    for all vertices $v \in G$ do
3:      if $S$ consists of a single vertex then
4:        Set table[$S$][$v$][$color\ of\ v$] := 1
5:      else
6:        $S$ consists of active child $a$ and passive child $p$
7:        for all colorsets $C$ of unique values mapped to $S$
          do
8:          Set $count$ := 0
9:          for all $u \in N(v)$, where $N(v)$ is the neighbor-
            hood of $v$ do
10:           for all possible combinations $C_a$ and $C_p$
              created by splitting $C$ and mapping onto $a$
              and $p$ do
11:             $count$ +=
12:             table[$a$][$v$][$C_a$]·table[$p$][$u$][$C_p$]
13:           end for
14:         end for
15:         Set table[$S$][$v$][$C$] := $count$
16:       end for
17:     end if
18:   end for
19: end for
20: $templateCount$ = $\sum_v \sum_C$ table[$T$][$v$][$C$]
21: $P$ = probability that the template is colorful
22: $\alpha$ = number of automorphisms of $T$
23: $finalCount = \frac{1}{P \cdot \alpha} \cdot templateCount$
```

values $0, 1, \ldots, k-1$, the corresponding index I would be $I = \binom{c_1}{1} + \binom{c_2}{2} + \cdots + \binom{c_h}{h}$.

In the innermost loops of the algorithm, we also look at all color sets C_a and C_b created by uniquely distributing the colors of C to the two children subtemplates of the partitioned S. By precomputing all possible index values for any given color set C of size h and any given sub-color set of size $1, \ldots, h-1$, we are able to replace explicit computation of these indexes with memory lookups. This reduces the necessary number of operations on these innermost loops by a considerable factor. It also allows these loops to exist as simple *for* loops incrementing through the index values, rather than the slower and more complex loops required with the handling of an explicit color set. The total storage requirements of the complete set of indexes is proportional to 2^k, and the representation only takes a few megabytes even for templates of size 12 vertices.

C. Memory Utilization Optimizations

A major consideration in the color coding algorithm is the memory requirement for the tabular storage of counts. This table grows proportional to $n\binom{k}{\frac{k}{2}}$ (n is the number of vertices in the graph, and k is the number of vertices in the template). For $k = 12$ and $n = 2,000,000$, this could mean up to 32 GB of memory necessary to determine a subgraph count using this algorithm. To this effect, we have implemented a number of different techniques for saving memory.

We initialize our table as a three-dimensional array. The first dimension is for each subtemplate generated through our initial template partitioning. We organize the order of the partitioning tree so that at any instance, the tables and counts for at most four subtemplates need to be active at once. Using the bottom-up dynamic programming approach for counting means that once the counts for a parent subtemplate are completed, the stored counts for the active and passive children can be deleted. We can also exploit symmetry in the template by analyzing possible rooted automorphisms that exist in the partitioned subtemplates. An obvious example can be seen in template U7-2 shown in Figure 2. We reorganize the parent/child relationships in the partitioning tree so that only one of the automorphic subtemplates needs to be analyzed, as the counts will be equivalent for both.

The second dimension in the table is for every vertex in the full graph. For our dynamic table, we only initialize storage for a given vertex v if that vertex has a value stored in it for any color set. This also allows a boolean check to be done when calculating new subtemplate counts for a given vertex. Since the counts for that vertex are based on the active child's count at v and the passive child's counts at $u \in N(v)$, we can avoid considerable computation and additional memory accesses if we see that v is uninitialized for the active child and/or u is uninitialized for the passive child. As we will discuss later, partitioning the graph in a certain way allows considerable algorithmic speedup by exploiting this further.

The third and innermost dimension of our table is for the counts for each coloring value. As discussed, these values are set and read based on the combinatorial number system index for the specific coloring. By organizing the table in this way, accesses can be quickly done as table[$subtemplate$][$vertex$][$color\ index$]. This storage follows the loops of the algorithm, which can help reduce cache misses on the innermost loops.

We have also developed a hashing scheme that can be used in the place of a three-dimensional array for high-selectivity templates. The key values used are calculated for vertex v and color set C as follows, where v_{id} is the numeric identifier of the vertex, I is the color set's combinatorial index, and N_c is the total number of color set combinations for the current subtemplate: $key = v_{id} \cdot N_c + I$. Calculating the key in this way ensures unique values for all combinations of vertices and color sets. Additionally, if we initialize and resize the hash table to simply be a factor of $n \cdot N_c$, where n is the number of vertices in G, we can utilize a very simple hash function of $(key \bmod n)$. This gives a relatively uniform distribution across all possible hash indexes based on the initial random coloring of G. This hashing scheme will generally save memory over the previous method when a template occurs with high regularity rooted at certain vertices within G, but with low regularity relative to the number of possible color sets.

D. Template Partitioning

We have also explored various template partitioning strategies for FASCIA. When possible, we utilize a one-at-a-time approach to partitioning. There are two primary reasons why this is done. The running time of the two innermost loops of the algorithm are dependent on $\binom{k}{S_n} \cdot \binom{S_n}{a_n}$, where k is the number of colors, S_n is the number of vertices in the subtemplate we are getting the count for, and a_n is the number of vertices in the active child of that subtemplate (note that $\binom{S_n}{a_n} = \binom{S_n}{p_n}$, where p_n is the number of vertices in the passive child). The running time of the algorithm grows equivalently to the sum over all $\binom{k}{S_n} \cdot \binom{S_n}{a_n}$ for every different S_n and a_n at each step in the partitioning tree. A one-at-a-time approach can minimize this sum for larger templates (except when exploiting rooted symmetry), as the larger multiplicative factors tend to dominate with a more *even* partitioning.

However, we still observe faster performance from our algorithm when doing a one-at-a-time partitioning approach as opposed to exploiting template symmetry. This is due to the fact that by using this approach and setting the active child as the single partitioned vertex at each step when possible, we can reduce the total number of color sets at each vertex v in the graph G we even have to look at by a factor of $\frac{k-1}{k}$. The count at each v is dependent on the count for the active child with a given color set, and only one color set for a single vertex subtemplate exists that has a non-zero count: the coloring of v.

E. Parallelization

We support two modes of multithreaded parallelism, and the choice is dependent on graph size. For large graphs, we parallelize the loop that calculates counts for all vertices $v \in G$. However, for small graphs and small templates, the multithreading overhead begins to dominate the execution time for each iteration. Therefore, in this instance, we perform multiple outer loop iterations concurrently. Each thread has its own dynamic table and does a full count. The counts are then collected and averaged after the specified number of iterations is completed. Due to the fact that each thread initializes its own table, the memory requirements increase linearly as a function of the number of threads. However, for smaller graphs where this *outer loop* parallelization works better, the vertex counts are small enough that this is unlikely an issue, even while running on a system with limited memory.

IV. EXPERIMENTAL SETUP

We analyze FASCIA across a wide range of different networks with sizes ranging from 252 vertices and 399 edges to about 1.6 million vertices and 31 million edges. Since execution time is expected to scale as 2^k, we use a variety of templates, with k ranging from 3 to 12.

We run most of our experiments on a dedicated shared-memory compute node running CentOS Linux. This node has two 8-core 2.6 GHz Intel EM64T Xeon E5 (Sandy Bridge microarchitecture) processors and 64 GB of DDR3-1333 memory. We use the Intel C compiler (version 12.1.0) to build our code, with the `-O3` optimization and OpenMP library support.

For select tests and comparisons with serial software that required Microsoft Visual Studio, we used a workstation with an Intel Core i5-2400 3.1 GHz quad-core processor, 8 GB memory, and running Windows 7 Professional.

A. Networks

We used three large social networks, an Erdős-Rényi $G(n,p)$ random graph, a network of roads in Pennsylvania, a small electrical circuit, and four different biological PPI networks in our analysis. The biological networks were obtained from the Database of Interacting Proteins, and include networks for three unicellular organisms (*E. coli*, *S. cerevisiae* (yeast), *H. pylori*), and the more complex *C. elegans* (roundworm) [14]. The electrical circuit network is the s420 circuit from the ISCAS89 Benchmark Suite [15]. The Pennsylvania road network was obtained from the Stanford Network Analysis Project (SNAP) large network database [16], [17]. The three social networks used were an email network created from data from the Enron Corporation [16], [18], a February 2009 snapshot of the Slashdot community [16], both also retrieved from the (SNAP) [17] database, and a large synthetic social contact network modeled after the city of Portland, from the Virginia Tech Network Dynamics and Simulations Science Laboratory (NDSSL) [19]. We also considered vertex labels on the Portland network, specifying two genders and four different age groupings for eight total different labels, based on demographic information supplied along with the contact network. The $G(n,p)$ random graph was modeled after the size and average degree of the Enron network. We considered all networks as undirected and only analyzed the largest connected component. Additional details about the networks are given in Table I.

Network	n	m	d_{avg}	d_{max}	Source
Portland	1,588,212	31,204,286	39.3	275	[19]
Enron	33,696	180,811	10.7	1383	[16]–[18]
$G(n,p)$	33,696	181,044	10.7	27	
Slashdot	82,168	438,643	10.7	2510	[16], [17]
PA Road Net	1,090,917	1,541,898	2.8	9	[17]
Elec. Circuit	252	399	3.1	14	[15]
E. coli	2,546	11,520	9.0	178	[14]
S. cerevisiae	5,021	22,119	8.8	289	[14]
H. pylori	687	1,352	3.9	54	[14]
C. elegans	2,391	3,831	3.2	187	[14]

TABLE I
NETWORK SIZES AND AVERAGE/MAXIMUM DEGREES FOR ALL NETWORKS USED IN OUR ANALYSIS.

B. Templates

While analyzing execution times and scaling on the larger networks, we considered two different templates with 3, 5, 7, 10, and 12 vertices. For each size, one template is a simple path and the other one is a more complex structure. The path-based templates are labeled as U3-1, U5-1, U7-1, U10-1, and U12-1. The other templates and their labels are shown in Figure 2.

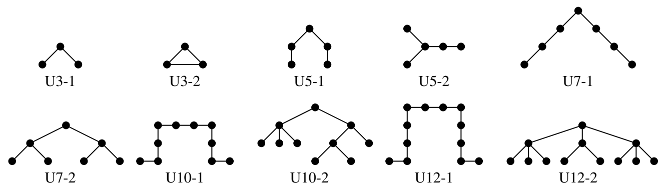


Fig. 2. Some of the templates used in our analysis.

For motif finding, we looked at *all possible tree templates* of size 7, 10, and 12. $k = 7$, 10, and 12 would imply 11, 106, and 551 possible tree topologies, respectively.

V. Results

In this section, we highlight performance results achieved running Fascia. We analyze Fascia in terms of execution time and memory usage for subgraph counting with different templates and networks. Wherever possible, we picked networks and templates so that we can compare our results with prior work. Since color coding gives an approximation to the count, we also looked at approximation error for different template and graph sizes. We also demonstrate how subgraph counting can be used as a graph analytic, by applying it to the problems of motif finding and graphlet degree distribution applications.

A. Execution Time and Memory Usage

We report parallel execution times with several unlabeled templates on the largest network in our collection, the Portland network, in Figure 3. The reported times are for a single iteration of the dynamic programming-based counting scheme. As expected, the time is strongly correlated with the number of vertices in the template. By monitoring performance counter data, we identified that more than 90% of time is spent in step 12 of Algorithm 2, which are the dynamic programming table read accesses.

The single iteration time for smaller templates is extremely low, making it feasible to obtain realtime count estimates. Even for the large templates of size 12, the time was still around 2-4 minutes. We estimate that it would require several days for the brute-force exhaustive count approach on the large templates. The U12-2 template took the longest time as expected. This template was explicitly designed to stress subtemplate partitioning and is therefore gives a practical upper bound for our running times. Another observation was that the running time was fairly independent of the template structure, particularly for the smaller templates. Even for the larger 12-vertex templates, there is just a $2\times$ variation in running time.

The results for *labeled templates* are shown in Figure 4. Labels here refer to integer vertex attributes that are applied to the template as well as the graph. Counting labeled templates in labeled graphs is significantly faster than the unlabeled case because labels help prune the search space by decreasing the overall number of possible embeddings. Our dynamic programming scheme and graph representation exploit labels, and

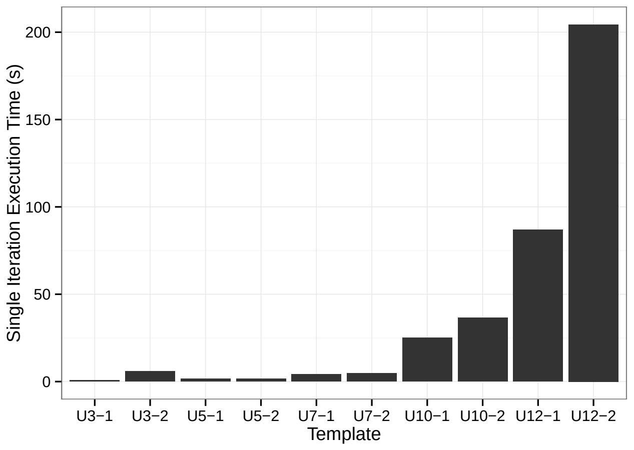


Fig. 3. Execution times for 10 templates on the Portland network.

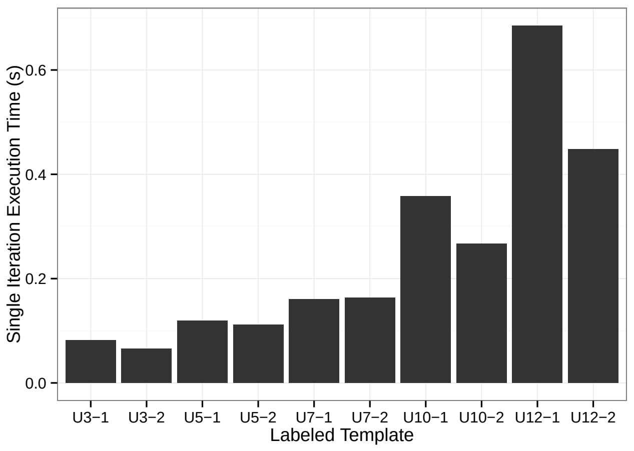


Fig. 4. Execution times for all 10 templates with vertex labels on the Portland network.

we see a commensurate speedup. We do not do an exhaustive analysis with labeled templates, since the cardinality of the label set and the distribution of labels to vertices strongly influence running time. We assume randomly-assigned labels. It is easier to analyze performance with unlabeled templates. Unlabeled templates are also more directly applicable to the problems of motif finding and graphlet degree distribution analysis.

We next report execution times for Fascia for motif finding on four protein interaction networks. For processing all tree templates of 7 vertices (11 of them), execution times per iteration on every network were well under a second. For all templates of 10 vertices (106 different templates), the running times were in the order of seconds, and for 12 vertices (551 templates), the times increased to a few minutes at most. The complete results are given in Figure 5.

A primary consideration with methods based on the color coding technique, or dynamic programming in general, is memory utilization. Figure 6 gives the peak memory require-

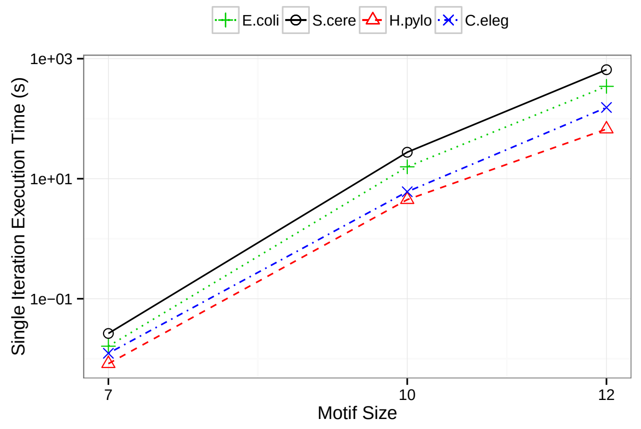


Fig. 5. Execution times per iteration for various templates sizes on all PPI networks.

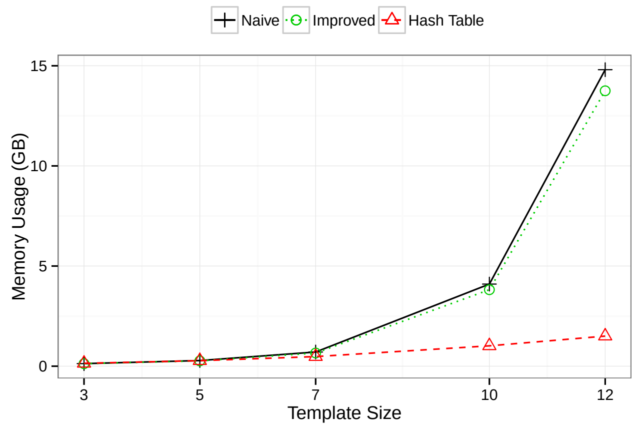


Fig. 7. Peak memory usage with the U3-1, U5-1, U7-1, U10-1, and U12-1 templates on the PA road network with the hash table as well as naïve and improved memory handling.

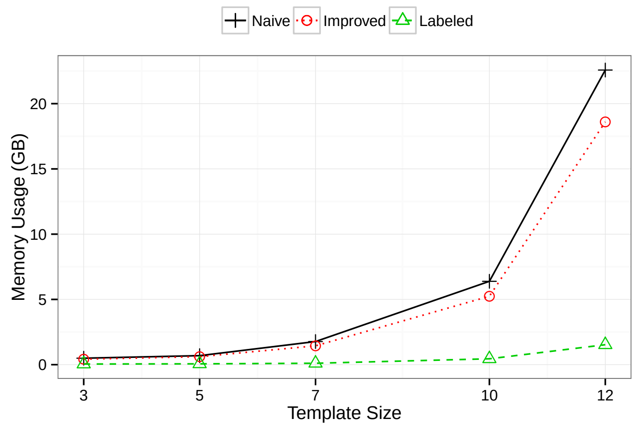


Fig. 6. Peak memory usage with the U3-2, U5-2, U7-2, U10-2, and U12-2 templates on the Portland network with naive and improved memory handling.

ment of the dynamic table on the Portland network for the naïve method (initializing all storage regardless of need), our improved version (initializing vertices only when necessary), and our improved version run with a labeled template. We note about a 20% decrease in peak memory usage using our method with unlabeled templates. With a labeled graph and template, the memory requirements decrease by over 90%. Note that non-induce subgraph counts can be very high (low selectivity) in general. Thus, a 20% saving in the unlabeled case is also considerable. A large reduction can be seen for labeled templates as well as for all highly-selective templates. Note that these results were obtained using our one-at-a-time partitioning approach. Exploiting the inherent symmetry in U7-2 or U12-2 when partitioning would decrease memory usage even further, but with a tradeoff in execution time.

Further reductions in memory requirements are possible using our fast hashing scheme. As we have abstracted the dynamic programming table, we can experiment with various representations. We expect the hash table to perform favorably over the array implementation when a template shows very low vertex selectivity, and this may be seen on networks with a low average vertex degree. Figure 7 gives the peak memory usage while counting the U3-1, U5-1, U7-1, U10-1, and U12-1 templates on the PA road network using our hash table, our array structure, and the naïve implementation. Our improved array method shows about a 2-7% reduction in memory requirements compared to the naïve method, while our hashing scheme shows up to a 90% memory usage reduction with the U12-1 template. For the smaller templates of 3 to 5 vertices, there is minimal to no improvement.

B. *Parallelization*

We next report parallel scaling for a large template. Figure 8 is a representative instance. It gives the execution times for the U12-2 template on the Portland network. The parallelization here is through partitioning the vertices such that each thread computes the counts independently for a subtemplate (loop at line 3 of Algorithm 1). We achieve good parallel scaling up to 16 cores, and this is due to the efficient table accesses (lines 4, 12, 15 of Algorithm 2), and careful NUMA-aware initialization of the table. Overall, there is about a 12× improvement over the serial code, and this is considering aggregate time for processing all subtemplates for this large template.

It is beneficial to parallelize across the entire count (line 3 of Algorithm 1) for smaller graphs, as memory requirements are not an issue and parallelization over a small number of vertices incurs relatively high overhead. The speedup with this approach is illustrated in Figure 9, which shows a comparison between the iteration execution times on the Enron network with inner loop parallelization as well as outer loop parallelization across the entire count (the number of cores given is the total number of independent counts being executed concurrently). Both the total execution and the per-iteration times are given for outer loop parallelization. In this instance,

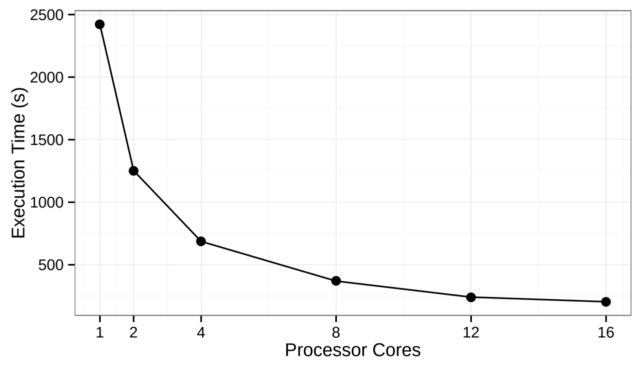


Fig. 8. Execution times versus inner loop parallelization with the U12-2 template on the Portland network.

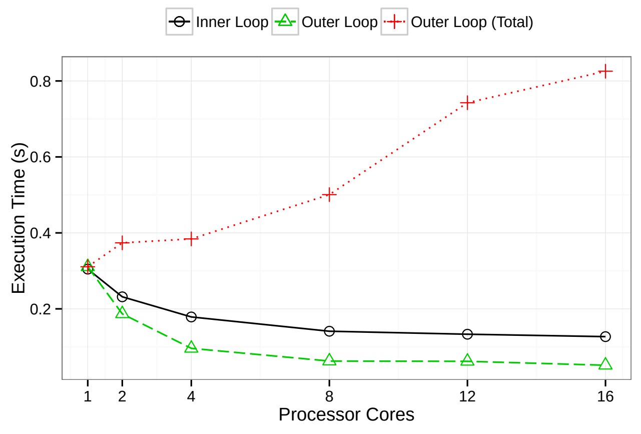


Fig. 9. Execution times with inner and outer loop parallelization for the U7-2 template on the Enron network.

we observe about a 6× parallel speedup versus the serial code with outer loop parallelization, while the parallel speedup is only about 2.5× using inner loop parallelization.

C. Comparison to other schemes

To our knowledge, this is the first work to analyze unlabeled templates of size 12 on *large* (vertex count in millions) networks. Our memory-efficient dynamic table representations enables this on a modest workstation. With the MapReduce-based SAHAD, Zhao et al. [9] report single iteration execution time for an unlabeled 10-vertex template on a 2.1 million vertex network, using a 16 node, 512-core cluster, to be approximately 25 minutes. FASCIA has been able to do counts for 10-vertex templates in about half a minute and 12-vertex templates in under 3.5 minutes on the slightly smaller Portland network. The iteration time with SAHAD with labeled 12-vertex templates is reported to be about 15 minutes. FASCIA's execution time for a 12-vertex labeled template on a single node with the Portland network, using a similar labeling methodology, is under one second.

We further compare FASCIA to MODA [10], which is a recent open-source network analysis tool that reports counts for non-induced subgraph occurrences. We compile and execute MODA on the Windows 7 system previously described, and we run FASCIA and our naïve exact count implementation on the same system. We run all of our code serially for this instance to get a direct comparison. We use the electrical circuit network and set the number of iterations to 1,000 for FASCIA, which gives us a final average error of about 1%. Retrieved total execution for all possible 7 vertex templates is about 147 seconds for the naïve algorithm, 32 seconds for MODA, and 22 seconds for FASCIA. Both MODA and FASCIA show considerable improvement over the naïve algorithm in terms of execution time, while FASCIA shows about a 33% improvement over MODA. However, MODA is unable to scale to much larger networks, and we were unable to determine counts for any of the other networks we analyzed.

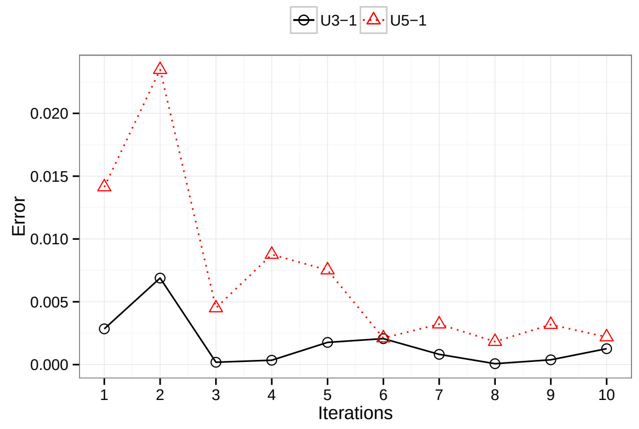


Fig. 10. Approximation error on Enron network ($n = 33,696$, $m = 180,811$) with the U3-1 and U5-1 templates.

D. Error Analysis

We next analyze the accuracy of the color coding approach with the number of iterations run. Accuracy was assessed on the Enron graph for up to 10 iterations with the U3-1 and U5-1 templates. These results are summarized in Figure 10. The observed error falls under 1% for both networks after only three iterations. This represents less than a second of processing for both templates. In comparison, the exact counts required to calculate the errors needed over 5 hours of processing time to complete. We notice that error decreases with increasing network size and average degree, but increases with an increasing template size. In general, the greater the number of template embeddings that exist within the graph, the lower the error. A low error on large graphs after only a few iterations was also previously demonstrated with SAHAD and PARSE [8], [9].

The accuracy of the algorithm for motif finding was also assessed on the *H. pylori* network for various iteration counts across all 11 possible 7-vertex templates. Figure 11 gives average errors across all templates resulting from processing

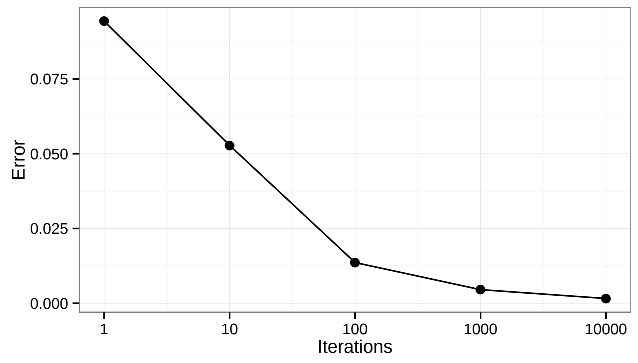


Fig. 11. Average error per iteration for motif finding on the $H.pylori$ network ($n = 687$, $m = 1,352$).

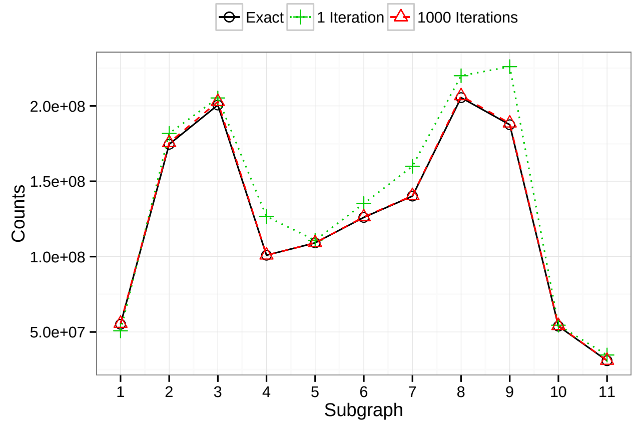


Fig. 12. Motif finding counts after various iterations on the *H. pylori* network.

up to 10,000 iterations. The errors given here are larger than what was observed on the Enron graph. The reason for this is likely due to the *H. pylori* graph being considerably smaller than the Enron graph, so that the random coloring and subsequent count scaling has a much larger effect on the final estimate. However, the average error still falls well below 1% after only 1,000 iterations.

E. Comparative analysis using motif counts

With motif finding, the goal is to obtain relative magnitudes of counts and compare them to what is expected on a random graph. From Figure 12, which shows the exact counts and estimated counts after 1 and 1,000 iterations, it is observed that even after a single iteration the relative magnitudes of counts for all templates are within reasonable bounds. The important difference to note is that the exact counts required hours of processing while the counts for 1,000 iterations were computed in seconds.

Figure 13 shows an overlay of approximate counts after 1,000 iterations for all 7-vertex tree templates, on all 4 PPI networks. Due to the varying sizes of the networks, the counts were all scaled by each of the networks' averages. From

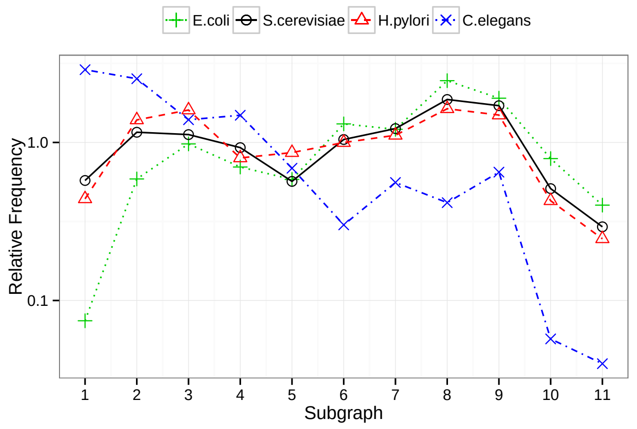


Fig. 13. Motif finding for size 7 templates on all PPI networks.

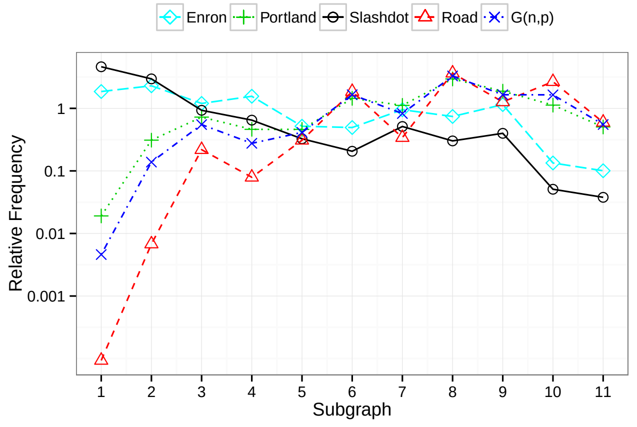


Fig. 14. Motif finding for size 7 templates on the Portland, Slashdot, Enron, PA road, and random networks.

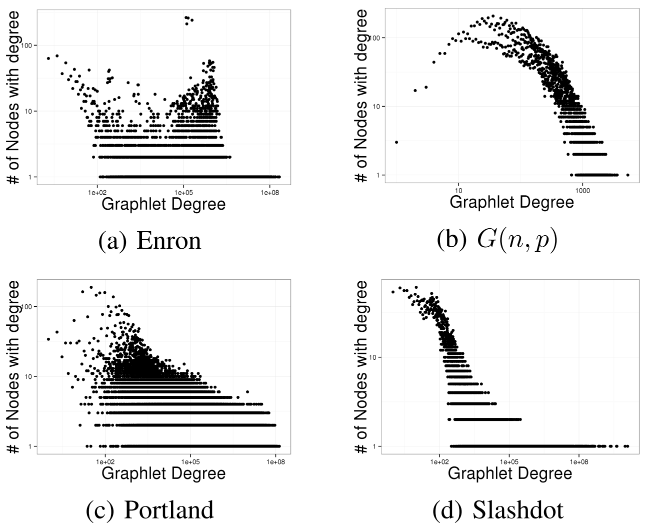
this, we are able to observe the relative frequencies of each subgraph. Our observation is consistent with the one made by Alon et al. [2], that the motif distribution of the three unicellular organisms is similar, and the more complex *C. elegans* motif profile stands out.

We also perform a similar analysis for the social, road, and random networks. These results are depicted in Figure 14, with templates 1 and 2 being very discriminative.

F. Graphlet Degree Distributions

FASCIA can also be applied to determine graphlet degree distributions. Figure 15 shows the varying graphlet degree distributions with the central orbit of the U5-2 template (vertex with degree of 3) on the Enron, Portland, slashdot, and random networks, respectively. Total processing time for all these networks was under 30 seconds.

To explore the efficacy of using the color coding algorithm for determining graphlet degree distributions, a graphlet degree distribution agreement value was calculated for the *E. coli*



(a) Enron (b) $G(n, p)$ (c) Portland (d) Slashdot

Fig. 15. Graphlet degree distribution for template U5-2 on several networks.

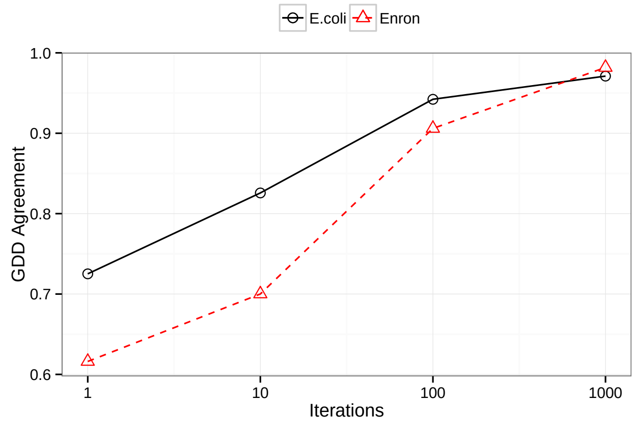


Fig. 16. Agreement of *E. coli* and Enron between their exact and estimated graphlet degree distributions after various iterations.

and Enron networks between the exact distribution and that obtained from the color coding algorithm. The distributions were determined for the orbit specified previously on the U5-2 template. Using a methodology presented by N. Pržulj [7], we determine an agreement value after various numbers of iterations. Figure 16 shows the result of this experiment. An agreement value of 1.0 would be calculated with an exact algorithm. However, we achieve reasonable results after about 1,000 iterations on both networks.

VI. CONCLUSIONS

This paper introduces FASCIA, a shared-memory implementation of the color coding technique for approximate subgraph counting. We demonstrate significant speedup over prior implementations, while decreasing memory consumption, simplifying data layout, and reducing parallel overhead. We also analyze the use of our algorithm to the applications of motif finding and graphlet frequency distribution analysis. In future work, we intend to combine the two OpenMP parallelization strategies, and consider partitioning the dynamic programming table for execution on a distributed-memory platform.

ACKNOWLEDGMENT

This work is supported by NSF grant ACI-1253881 and used instrumentation funded by the NSF grant OCI-0821527. This work also used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by NSF grant OCI-1053575.

REFERENCES

[1] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: Simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[2] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp, "Biomolecular network motif counting and discovery by color coding," *Bioinformatics*, vol. 24, no. 13, pp. i241–i249, 2008.

[3] J. Huan, W. Wang, and J. Prins, "Efficient mining of frequent subgraphs in the presence of isomorphism," in *Proc. IEEE Int'l. Conf. on Data Mining (ICDM)*, 2003, p. 549.

[4] M. Kuramochi and G. Karypis, "Frequent subgraph discovery," in *In Proc. IEEE Int'l. Conf. on Data Mining (ICDM)*, 2001, pp. 313–320.

[5] F. Eisenbrand and F. Grandoni, "On the complexity of fixed parameter clique and dominating set," *Theoretical Computer Science*, vol. 326, no. 1–3, pp. 57–67, 2004.

[6] N. Alon, R. Yuster, and U. Zwick, "Color-coding," *J. ACM*, vol. 42, no. 4, pp. 844–856, 1995.

[7] N. Pržulj, "Biological network comparison using graphlet degree distribution," *Bioinformatics*, vol. 23, no. 2, pp. e177–83, 2007.

[8] Z. Zhao, M. Khan, V. S. A. Kumar, and M. V. Marathe, "Subgraph enumeration in large social contact networks using parallel color coding and streaming," in *Proc. Int'l. Conf. on Parallel Processing (ICPP)*, 2010, pp. 594–603.

[9] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. S. A. Kumar, and M. V. Marathe, "SAHAD: Subgraph analysis in massive networks using Hadoop," in *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, 2012, pp. 390–401.

[10] S. Omidi, F. Schreiber, and A. Masoudi-Nejad, "MODA: an efficient algorithm for network motif discovery in biological networks," *Genes Genet. Syst.*, vol. 84, no. 5, pp. 385–395, 2009.

[11] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon, "Efficient sampling algorithm for esitmating subgraph concentrations and detecting network motifs," *Bioinformatics*, vol. 20, no. 11, pp. 1746–1758, 2004.

[12] S. Wernicke, "Efficient detection of network motifs," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 3, no. 4, pp. 347–359, 2004.

[13] P. Ribeiro and F. Silva, "G-Tries: and efficient data structure for discovering network motifs," in *Proc. ACM Symp. on Applied Computing*. ACM, 2010.

[14] I. Xenarios, L. Salwinski, X. J. Duan, P. Higney, S. M. Kim, and D. Eisenberg, "DIP, the database of interacting proteins: a research tool for studying cellular networks of protein interactions," *Nucleic Acids Research*, vol. 30, no. 1, pp. 303–305, 2002.

[15] Collaborative Benchmarking and Experimental Algorithmics Laboratory: NC State University, "ISCAS89 benchmark suite," http://www.cbl.ncsu.edu/CBL_Docs/iscas89.html, last accessed Feb 8, 2013.

[16] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[17] J. Leskovec, "SNAP: Stanford network analysis project," http://snap.stanford.edu, last accessed July 12, 2013.

[18] B. Klimmt and Y. Yang, "Introducing the Enron corpus," in *Proc. Conf. on Email and Anti-Spam (CEAS)*, 2004.

[19] Network Dynamics and Simulation and Science Laboratory, "Synthetic data products for societal infrastructures and proto-populations: Data set 1.0," 2006, Technical report NDSSL-TR-06-006, Virginia Polytechnic Institute and State University, 1880 Pratt Dr, Building XV, Blacksburg, VA, 24061.