Parallel Computing 47 (2015) 51–69

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Parallel color-coding

George M. Slota, Kamesh Madduri *

Department of Computer Science and Engineering, The Pennsylvania State University, United States

ARTICLE INFO

Article history:
Available online 28 February 2015

Keywords:
Color-coding
Parallelization
Protein interaction networks
Motifs
Subgraph counting
Pathways

ABSTRACT

We present new parallelization and memory-reducing strategies for the graph-theoretic color-coding approximation technique, with applications to biological network analysis. Color-coding is a technique that gives fixed parameter tractable algorithms for several well-known NP-hard optimization problems. In this work, by efficiently parallelizing steps in color-coding, we create two new biological protein interaction network analysis tools: FASCIA for subgraph counting and motif finding and FASTPATH for signaling pathway detection. We demonstrate considerable speedup over prior work, and the optimizations introduced in this paper can also be used for other problems where color-coding is applicable.
© 2015 Elsevier B.V. All rights reserved.

1. Introduction

The color-coding method is a simple and elegant graph-theoretic strategy that gives fixed parameter tractable algorithms for several NP-hard optimization problems. Color-coding was first proposed by Alon et al. [1]. In this paper, we present efficient shared- and distributed-memory parallelizations of this strategy, using new data structures and optimizations to reduce peak memory utilization and inter-processor communication. We also create two new software tools, FASCIA and FASTPATH, that use parallel color-coding to solve bioinformatics problems.

The problem of counting the number of occurrences of a *template* or subgraph within a large graph is commonly termed *subgraph counting*. This problem is very similar to the classical subgraph isomorphism problem. Related problems, such as subgraph enumeration, tree isomorphism, motif finding, frequent subgraph identification, etc. are all fundamental graph analysis methods to identify latent structure in complex data sets. They have applications in bioinformatics [2–4], chemoinformatics [5], online social network analysis [6], network traffic analysis, and many other areas.

Subgraph counting and enumeration are compute-intensive problems. A naïve algorithm, which exhaustively enumerates all vertices reachable in k hops from a vertex, runs in $O(n^k)$ time, where n is the number of vertices in the network and k is the number of vertices in the subgraph. For large networks, this running time complexity puts a constraint of the size of the subgraph (value of k). If k is larger than 2 or 3, exact counting becomes prohibitively expensive. Thus, there has been a lot of recent work on approximation algorithms. Approaches are generally based on sampling or on exploiting network topology. Sampling-based methods analyze a subset of the network and extrapolate counts based on the observed occurrences and network properties. Some tools based on sampling are MFINDER [7], FANMOD [8], and GRAFT [9]. The other class of methods impose some constraint on the network or transform the network so that the possible search space is restricted. Examples of tools imposing constraints on the network are NEMO [10] and SAHAD [11]. Tools based on the color-coding method belong to the second category, and this forms the basis of our current work.

* Corresponding author.
E-mail addresses: gms5016@cse.psu.edu (G.M. Slota), madduri@cse.psu.edu (K. Madduri).

http://dx.doi.org/10.1016/j.parco.2015.02.004
0167-8191/© 2015 Elsevier B.V. All rights reserved.

The color-coding method for this problem uses a dynamic programming scheme to generate an approximate count of a given *non-induced tree-structured* subgraph/template (also referred to as a *treelet*) in $O(m \cdot 2^k)$ time, where m is the number of edges in the network. The algorithm can be informally stated as follows: every node in a network is randomly colored with one of at least k possible colors. The number of *colorful* embeddings of a given input template is then counted, where colorful in this context means that each node in the template embedding has a distinct color. The total embedding count is then scaled by the probability that the template is colorful, in order to generate an approximation for the total number of possible embeddings. This colorful embedding counting scheme avoids the prohibitive $O(n^k)$ bound seen in exhaustive search.

Color-coding can also be applied in an entirely different context. Consider the NP-hard optimization problem [12] of finding the *minimum-weight simple path* of path length k in a weighted graph with positive edge weights. This problem is of considerable interest in bioinformatics, specifically in the analysis of paths in protein interaction networks. With an appropriately-defined edge weight scheme, paths with the minimum weight, or in general close to the minimum weight, often have vertices that belong to biologically-significant subgraphs such as signaling networks and metabolic pathways [13,14]. As in the case of subgraph counting, color-coding can only offer an approximate solution to this NP-hard problem. With some confidence and error bounds, it is guaranteed to return simple paths with weight close to the minimum path weight. The low-weight paths returned through color-coding are shown to be good candidates for signaling pathways [12]. We present a shared-memory parallelization of the approximate low-weight path enumeration strategy.

Color-coding can be in general applied to finding any subgraphs with a bounded tree-width in polynomial time, by executing the color-coding algorithm with the tree decomposition of the subgraph [1]. However, in this work, we only consider finding treelets, which are subgraphs with a tree-width of 1. Another application of color-coding that is not included in this work is for finding cycles of length k. We also note that all algorithms using color-coding can be derandomized by using families of perfect hash functions. However, we observe in practice that an optimal, or near-optimal solution, can usually be found much quicker with only the randomized approach.

1.1. Summary of contributions

We present several new optimizations that may be applicable to approximation algorithms that are based on color-coding. We give general methodologies for shared-memory and distributed-memory parallelization. We discuss strategies to reduce overhead in the inner loops of the algorithm, present a combinatorial numbering system to represent unique colorings, and a simple template partitioning method for subgraph counting. We also give a compressed data structure representation to reduce communication and memory costs when analyzing large-scale networks in a distributed environment.

Through these optimizations, we offer orders-of-magnitude speedup relative to prior software tools for subgraph counting and path enumeration [15,11,4]. Additionally, our implementations allow

- Approximate subgraph counts for templates of size up to 7 vertices in 100 million-edge networks, in a few seconds (through shared-memory parallelism and optimizations).
- Approximate subgraph counts for templates of size up to 9 vertices in billion-edge networks, in a few minutes (through additional distributed-memory parallelism and optimizations).
- Low-weight path enumeration for paths of length 9 in protein-interaction networks, in a few seconds.

We provide open-source versions of both our shared-memory approximate subgraph counting (Fascia [16]) and path enumeration (FastPath [17]) tools.

2. Related work

There is considerable prior work on improvements and extensions to color-coding, and applying it to solve various graph-theoretic problems. We focus on the problems of counting tree-structured subgraphs and enumerating low-weight paths in this paper. Color-coding can also be used to count and enumerate cycles, cliques, and bounded treewidth subgraphs [1].

2.1. Subgraph counting

Subgraph counting has recently emerged as a widely-used graph analytic in various domains, especially the biological and social sciences. Pržulj has demonstrated that *graphlets*— all 2–5 vertex induced undirected subgraphs — are a useful analytic for biological network comparisons [3]. Pržulj and Milenkovič et al. have extended this work to several other subgraph-based comparative metrics [18–20]. Bordino et al. used counts of both small undirected and directed subgraphs, similar to graphlets, to cluster networks of various types (e.g. citation networks, road networks, etc.) [21].

Alon et al. implemented color-coding subgraph counting to demonstrate its applicability for finding large tree-structured motifs in biological networks [4]. Zhao et al. implemented distributed color-coding subgraph counting for large graphs via both MPI and MapReduce, with applications in social network analysis [15,11]. We recently designed Fascia, and show that it achieves considerable speedups relative to prior work in both shared and distributed-memory environments. We also use Fascia to demonstrate the applicability of treelets for a number of subgraph counting-based analytics [22,23].

2.2. Minimum-weight paths and related problems

Given a graph with positive edge weights and a path length k, finding the minimum-weighted simple paths (or paths) among all possible paths of length k is a useful graph analytic, particularly in bioinformatics [13,14]. Scott et al. were the first to use the color-coding technique to find low-weight paths, with the use case of detecting signaling pathways in protein interaction networks [12]. Vertices in these networks are proteins, and edge weights are the negative log of the probability that the two proteins interact. Thus, simple paths with low weights correspond to chains of protein that would interact with high confidence. Hüffner et al. expanded on this initial work by offering several optimizations to the baseline algorithm to improve running times, including choosing an appropriate number of colors to decrease iteration counts and implementing a pruning strategy [24] that complements coloring. More recently, Gabr et al. further decreases the number of iterations required for a given confidence bound through per-iteration examination of graph colorings [25]. Color-coding has also been used for querying linear pathways in protein interaction networks by Shlomi et al. [26]. This work was expanded for more complex bounded tree-width queries by Dost et al. [27]. Similar to the aforementioned Gabr et al. work, Gülsoy et al. speed up pathway and small bounded tree-width querying by reducing the number of iterations required for a given confidence bound [28,29].

3. Our color-coding implementations

3.1. Subgraph counting with FASCIA

We first present the algorithmic details of applying the color-coding method for tree-structured subgraph counting [22]. As shown in Algorithm 1, there are three main phases in the algorithm: template partitioning, random coloring, and the dynamic programming count phase. The pseudocode for the dynamic programming phase is described in Algorithm 2. The coloring and dynamic programming steps are repeated for multiple iterations to estimate the subgraph count. Alon et al. [1] prove that to guarantee a count bound of $\mathcal{C}(1 \pm \epsilon)$ with probability $1 - 2\delta$ ($\mathcal{C}$ being the exact count), we would need to run at most *Niter* iterations, as defined in Algorithm 1. Using a topology-aware coloring scheme [29,28,25], prior work has shown that a tighter upper bound can be obtained. We observe that the number of iterations necessary to produce accurate global counts on large networks is far lower in practice [22,11], and we will also demonstrate this in Section 4.

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. δ and ϵ are input parameters that control approximation quality.
3: **for all** $it = 1$ to *Niter* **do** ▷ Outer loop parallelism
4: Randomly assign to each $v \in G$ a color between 0 and $k - 1$.
5: Use a dynamic programming scheme to count *colorful* occurrences of T.
6: Take average count of all *Niter* counts to be final count.

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 called 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. Every subtemplate has a parent. This tree can be traced from the bottom up to the original template, which is how we will perform the dynamic programming phase of the color coding algorithm. We also sort them in the order in which the subtemplates are accessed, in order to reduce memory usage.

The graph G is next randomly colored. For every vertex v, 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. It has been demonstrated that higher values of k can decrease the required iterations for a given error bound [24]. However, note that this considerably increases memory requirements as well.

Consider first a naïve table structure. We need to be able to store non-zero counts for every vertex and for all possible *color sets*. For a given subtemplate S_i of size h, a color set can be considered to be the mapping of h unique color values to each vertex in S_i. 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 programming-based counting 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 previously created. For every subtemplate, we will then consider every vertex $v \in G$. If our subtemplate is of size 1, 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.

Algorithm 2. The dynamic programming step in FASCIA

```
1: for all sub-templates S_i created from partitioning T, in reverse order of their partitioning do
2:   for all vertices v ∈ G do                    ▷ Inner loop parallelism
3:     if S_i consists of a single vertex then
4:       table[S_i][v][color of v] ← 1
5:     else
6:       S_i consists of active child a_i and passive child p_i
7:       for all color sets C of unique values mapped to S_i do
8:         count ← 0
9:         for all u ∈ N(v), where N(v) is the neighborhood of v do
10:          for all C_a and C_p created by uniquely splitting C do
11:            count ← count + table[a_i][v][C_a] · table[p_i][u][C_p]
12:        table[S_i][v][C] ← count
13: templateCount ← Σ_v Σ_C table[T][v][C]
14: P ← probability that the template is colorful
15: α ← number of automorphisms of T
16: finalCount ← 1/(P·α) · templateCount
```

If the size of the subtemplate is greater than 1, we know that it must have an active (a_i) and passive (p_i) 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$[S_i][v][C]$, is initialized to zero. Next, we will consider for every neighbor, u, of v, the counts of a_i rooted at v and p_i 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_i 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$[a_i][v][C_a]$ · table$[p_i][u][C_p]$.

Once we execute 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 execution of the algorithm is complete.

3.2. Color-coding implementation optimizations

We now discuss some improvements to the baseline algorithm presented in the previous subsection. These include the representation of colorings through a combinatorial index system, careful memory management, and partitioning the input template to reduce work performed.

3.2.1. Combinatorial indexing system

We represent a color set as a 32-bit 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 number system [30]. For a subtemplate S_i 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) 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_i. By precomputing all possible index values for any given C, 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 considerably reduces the number of indexing operations on these innermost loops. 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 for the complete set of indexes is proportional to 2^k, and the representation only takes a few megabytes even for templates of size 12.

3.2.2. Memory utilization optimizations

A major consideration in the color-coding algorithm is the memory required for tabular storage of counts. This table grows proportional to $n\binom{k}{Min(\frac{k}{2}, h)}$ (n is the number of vertices in the graph, and h is the number of vertices in the template, k is the number of colors). For $k = 12, h = 12$, and $n = 2,000,000$, this would mean that we require 32 GB of memory to determine a subgraph count using this algorithm. We have thus implemented a number of different memory-saving techniques to reduce the table size.

As previously mentioned, 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 Fig. 1. We can reorganize the parent/child relationships in the partitioning tree so that only one of the isomorphic 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 color set value. 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 avoid initializations for a given vertex in a graph G of n vertices and m edges when there are no embeddings of a subtemplate rooted at that vertex. To give a sense of memory savings possible with this simple change, we analytically determine the expected memory savings for some synthetic random graphs and test template instances. If we determine the expected number of embeddings given the degree of a vertex, we can determine the number of vertices that are expected to have at least one embedding. Because the memory savings can be dependent on template topology, for simplicity of analysis, we assume the initial template is a star. For a star template, it follows that all subtemplates are also stars. By using stars, we only need to consider a vertex and its immediate neighborhood.

We first assume an R-MAT graph for our calculations [31,32], with parameters $a = 0.75$ and $b = 0.10$ and $n = 2^z$. An R-MAT graph follows a degree distribution such that the expected number of vertices C with an out-degree of d and $p = a + b$ is as follows:

$$C(d) = \binom{m}{d} \sum_{i=0}^{z} \left(p^{(z-i)}(1-p)^{i}\right)^{z} \left(1 - p^{(z-i)}(1-p)^{i}\right)^{(m-z)}$$

We consider only directed out-edges to determine a bound on the memory savings. The expected number of embeddings of a non-induced star subtemplate with h vertices for a given degree is the number of total embeddings multiplied by the probability that any given embedding is colorful with k colors:

$$E(d) = \binom{d}{h-1} \frac{h! \binom{k}{h}}{k^h}$$

If we solve for $E(d) = 1$, we get the minimum degree d_0 at which a vertex in G is expected to have at least one subtemplate embedding. We can get the ratio of the expected uninitialized vertices to total vertices by integrating from 0 to the calculated d_0. As the calculation of degree distribution for R-MAT graphs is unwieldy, we can also just sum over all $C(d)$ from $0 \cdots \lfloor d_0 \rfloor$ and divide by n for the desired effect. We use the floor of d_0 to simply establish a lower bound for our calculation.

$$C(d = 0 \cdots \lfloor d_0 \rfloor) = \sum_{d=0}^{\lfloor d_0 \rfloor} \binom{m}{d} \sum_{i=0}^{z} \binom{z}{i} \left(p^{(z-i)}(1-p)^{i}\right)^{z} \left(1 - p^{(z-i)}(1-p)^{i}\right)^{(m-z)}$$

To get the total estimated reduction, we calculate the ratios for all subtemplates scaled by the number of possible color sets for each subtemplate, $\binom{k}{h}$. We now explicitly consider a 4-star embedded on an R-MAT graph of $n = 1024, z = 10$ vertices and $m = 32768$ edges colored with $k = 5$ colors. We calculate the expected number of vertices without embeddings for $h = 1$ through $h = 5$ as 0, 133, 623, 690, and 757, respectively. Scaling each value relative to the number of possible color sets for each subtemplate and comparing to the expected total, we therefore might observe an expected 37% reduction in memory utilization just from our array-based approach.

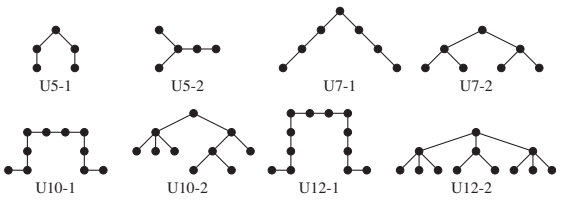

Fig. 1. Select templates used in performance analysis.

Similarly, we can calculate the expected memory savings for a $G(n,p)$ random graph. Using the same d_0, we can get the estimated reduction ratios by taking the integral over the degree distribution from $0 \cdots d_0$:

$$\int_0^{d_0} \binom{n-1}{d} \left(\frac{m}{n^2}\right)^d \left(1-\frac{m}{n^2}\right)^{(n-1-d)} dx$$

With the same parameters of n and m, we observe less than a 1% reduction for memory utilization. This is due to the relatively even degree distribution centered around an average degree much higher than what is needed to embed a 4-star.

For high-selectivity templates, we have also developed a hashing scheme that can be used in the place of a three-dimensional array. 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 use 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.

The memory requirements for the hashing approach is dependent on the number of total embeddings rather than the number of vertices with at least one embedding. As such, this hashing scheme will generally save memory over the simpler array-based scheme 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. As we will demonstrate in our results, this approach is quite effective with regular templates on regular graphs (such as road networks).

3.2.3. Template partitioning

We also explore various template partitioning strategies. When possible, we employ a one-at-a-time approach, where we partition a given subtemplate so that either the active or passive child is a single vertex. There are two reasons why we do this. The running time of the two innermost loops of the algorithm are dependent on $\binom{k}{h_i} \cdot \binom{h_i}{a_i}$, where k is the number of colors, h_i is the number of vertices in the subtemplate we are getting the count for, and a_i is the number of vertices in the active child of that subtemplate (note that $\binom{h_i}{a_i} = \binom{h_i}{p_i}$, where p_i is the number of vertices in the passive child). The running time of the algorithm grows as the sum over all $\binom{k}{h_i} \cdot \binom{h_i}{a_i}$, for every pair of h_i and a_i, 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 observe faster performance with a one-at-a-time partitioning approach over the symmetry-based template partitioning. This is due to the fact that by 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 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.

Also, note that the root selection can impact how the template can be partitioned using the one-at-a-time approach. Our strategy is to randomly select a leaf vertex as the initial root. After the first cut, we continue to greedily prune leaf vertices whenever possible. We have not yet explored other ways of determining the root. This might make for interesting future work.

3.3. Shared-memory parallelism

We support shared-memory parallelism in Fascia using the OpenMP programming model and have two modes of multithreaded parallelism. The choice is left to the user and is dependent on graph and template size. For large graphs, we parallelize the loop that calculates counts for all vertices $v \in G$. Each thread is assigned a unique set of vertices, for which it calculates and stores the next level of counts. Because vertices are partitioned among threads, and given the tabular layout of the counts table, there is no concurrent writes to shared locations.

However, for small graphs and small templates, the ratio of available parallel work to the necessary serial computational portion is low, and multithreaded performance suffers. Therefore, for this instance, we perform multiple outer loop iterations concurrently, where each thread independently computes the full counts for a subset of the total number of iterations. Each thread necessarily has its own dynamic table. 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.

While both inner and outer loop parallelism offer speedups over serial code, the choice is dependent on graph and subgraph topology as well as the runtime system. A hybrid strategy that combines both levels of parallelism is additionally possible. A dynamic scheduler that determines the optimal parallel strategy for a given input would make for interesting future work, but our current version of Fascia leaves the choice as an input parameter to be given by the user.

3.4. Distributed memory parallelism

3.4.1. Distributed counting

There are several avenues for distributed-memory parallelization of color-coding subgraph counting. Just as we implemented an outer loop method in shared-memory, we can extend this to distributed memory. A chunk of iterations of the outer loop can be assigned to a task. The inner-loop shared-memory parallelization can be complementarily performed. We refer to this hybrid parallelization strategy as *distributed counting*, and pseudocode is given in Algorithm 3.

Algorithm 3. Dynamic programming routine with distributed counting

```
for it = 1 to Niter do in parallel          ▷ MPI task-level parallelism
    Color G(V, E) with k colors
    Initialize 3D count table
    for all S_i in reverse order of partitioning do
      for all v ∈ V do in parallel          ▷ Thread-level parallelism
        Update count table for template S_i
        using child subtemplate counts
```

3.4.2. Partitioned counting algorithm

For modest-sized graphs (more than 2 million vertices) and large templates ($k > 10$), memory utilization quickly becomes problematic with distributed counting. We have therefore also implemented a distributed graph partitioning-based approach, where each task performs counts of a subset of all vertices, to reduce per-task memory requirements further. A description of the algorithm is given in Algorithm 4.

Algorithm 4. FASCIA Fully partitioned counting approach

```
Partition subgraph S using single edge cuts
for it = 1 to Niter do
    Color G(V, E) with k colors
    for all S_i in reverse order of partitioning do
      Init Table_{i,r} for V_r (vertex partition on task r)
      for all v ∈ V_r do                    ▷ Thread-level parallelism
        for all c ∈ C_i do
          Compute all Count_{S_i,c,v}
      ⟨N, I, B⟩ ← Compress(Table_{i,r})
      All-to-all exchange of ⟨N, I, B⟩
      Update Table_{i,r} based on information received
    Count_r ← Count_r + Σ_v^{V_r} Σ_c^{C_T} Count_{T,c,v}
Count ← Reduce(Count_r)
Scale Count based on Niter and colorful embed prob.
```

The graph is partitioned in a one-dimensional manner among the MPI tasks, with each task storing V_r vertices and their adjacencies. For every S_i, we only initialize the current table for the task's specific subset of vertices V_r. We compute all the counts for the subset of vertices for the current subtemplate. We then compress the table in the Compressed Sparse Row (CSR) format (details in the next subsection), with N denoting the array of count values, I the color mapping indexes, and B containing the start offsets for each vertex. The compressed table is ordered according to the ordering of tasks that have $v \in V_r$ in their one-hop neighborhood, as only these vertices are required in calculating the counts for each S_i, and we want to reduce communication costs. We distribute the counts in an all-to-all fashion among all r nodes, so that each node now has the child counts required to compute counts for the new parent template.

For the final S_i, i.e., the original template, each task computes the final count for the template for their subset of vertices. We simply keep a running sum of the counts for each task, for every iteration. After all iterations are completed, we perform a global reduction of the sum from all nodes, scale the value by the number of iterations and probability that the template is colorful, to get the final count estimate. Note that no additional approximations are introduced during this procedure, and so a count produced with say, 15 MPI tasks, will be the same as the count generated by the serial algorithm (assuming the random graph colorings are seeded with the same value).

3.4.3. Table compression

Due to the large memory footprint of the dynamic programming-based arrays, the partitioned approach also incurs a substantial inter-node communication cost. We reduce the total volume of data transferred by using a compressed sparse row (CSR) representation for storing the non-zero counts. The CSR format is commonly used in numerical analysis for the storage

of sparse matrices. Storage using this format consists of three arrays. One array stores all values held in the matrix in row-major ordering. This array would be organized as $[(\text{row}_1)(\text{row}_2)\cdots(\text{row}_n)]$, where (row_i) is a list of all nonzero values in that row. A second array of the same length as this first array is used to hold the column indexes at each of the nonzero values stored in the first array. The final array is of length n, or the number of rows, and it holds indexes to the start of the sequence of values for each row.

By considering a table for each discrete subtemplate S_i as a matrix of size $n \times c_i$, where n is the number of vertices in G and c_i is the number of possible color sets for S_i, we can apply the CSR format to our table, in order to compress it for faster transfer across tasks running on our cluster. The first array N stores all non-zero counts for all vertices and color mappings. The second array I is the color mapping indexes for each count value, as computed using the combinatorial number system approach. The final array B denotes the indexes for the start of count values for each vertex.

Due to the large graph and template sizes considered in our study, the overall per-vertex and per-color set count magnitudes can be quite massive in scale. This requires the N array to be of type 64-bit `double` to avoid overflow. Similarly, because the N array length can exceed the limits provided by 32-bit `unsigned int` for array indexing, the B array is of type 64-bit `unsigned long`. We use a 16-bit integer to store the color set index array I, which will allow unique indexes up to templates of 18 vertices in size. Because the lookup for any specific (x, y) index can be slow using this format and the color-coding approach requires a significant number of such lookups, we ideally want to re-expand the compressed values. However, in order to further reduce memory footprint, we only re-expand for each vertex when they are needed to compute the count of the new parent subtemplate. The overhead for this decompression step is minimal in practice.

As with the hash table, the memory savings for the CSR compression is greatest when there is a low number of total counts stored in the table, regardless of counts stored per-vertex. As we will show in the results, even on network with a high rate of per-vertex template embeddings, the CSR format will still reduce memory usage relative to the improved table by up to three quarters.

3.5. Enumerating low-weight simple paths with FastPath

We now present a color-coding based scheme to enumerate simple paths of length L in a graph with positive edge weights. Finding the minimum-weight path is an NP-hard problem, but color-coding gives us an approximation algorithm whose cost is linear in the number of edges in the graph, but exponential in the value L. The main idea is the same as the subgraph counting case: instead of enumerating all paths of length L and looking for a simple path with the minimum weight, we instead only search for *colorful paths* by randomly coloring vertices. There are prior approaches and tools that implement this strategy. But prior work has primarily focused on reducing running time by limiting the required number of iterations for a given confidence bound [25,12], with the exception of the approach of Hüffner et al. [24]. Here, we will only consider minimizing per-iteration costs through the previously-described optimizations (combinatorial table indexing, memory-reducing optimizations, partitioning, multithreading). Our approach can be combined with the graph topology-aware coloring methods [25,12] to further reduce end-to-end running time.

Algorithm 5. FastPath: Enumerating low-weight simple paths using color-coding

```
Initialize all entries of a min heap H of size n_L to ∞
for it = 1 to Niter do                          ▷ Outer loop parallelism
    Color G(V, E) with k colors
    Initialize all Weights [1] [v ∈ V_1][1 ··· c_1] ← ∞
    for i = 2 to L + 1 do
      for all v ∈ V_i                           ▷ Inner loop parallelism
        for all color sets C do
          min_w ← ∞
          for all C_a, C_p ∈ C do
            for all u ∈ N(v) do
              w_a ← GetEdgeWeight(u, v)
              w_p ← Weights[i − 1][u][C_p]
              if w_a + w_p ⩽ min_w then
                min_w ← w_a + w_p
        if min_w < H.max then                   ▷ Critical section
          if i = L + 1 then
            insert min_w into H
          else
            Weights[i][v][C] ← min_w
Return H as output.
```

Algorithm 5 gives an overview of the general approach for finding low-weight simple paths. The algorithm is similar to the general color-coding template for subgraph counting. Since this implementation only considers simple paths (which can be considered a tree template) rather than a more complex template, we can simplify the partitioning phase. We avoid partitioning completely by assuming that we already performed a one-at-a-time partitioning, and have set the active child as the single cut vertex at each step in the partitioning tree.

To simplify the description of the algorithm, we only show weights of the n_L least-weight colorful paths being stored in Algorithm 5. In our actual implementation, we also store the corresponding vertices in the low-weight path as an array of integers. In prior work, paths were stored using compressed representations [12,24]. We use a min heap of size n_L to store the best weights and the corresponding paths.

Algorithm 5 has L inner loop iterations. At each step, we are attempting to find over all $v \in V_i$ the least-weight colorful path that ends at v, for every possible color set C. Initially, weights for all vertices and color sets are set to ∞ for a single vertex path. For succeeding steps, we look at the sum of weights of all previously discovered paths ending on neighbors u of v, while considering adding the weight of the edge between u and v. For each color set C, we take the minimum and store the summed weight of the path in Weight$[i][v][C]$.

We can also compare the weights found during each step to the current highest value in the min heap, and store the path only if it is one the current n_L lowest-weight paths. These paths are inserted into the heap in the final step of the inner loop ($i = L + 1$). We update the heap over subsequent iterations, storing better paths if we find them. This decreases memory requirements for subsequent iterations by avoiding unnecessary storage of heavy paths in the Weights array.

There are FastPath-specific issues to note with regards to memory utilization. Storing the actual paths for all color sets for all vertices can increase memory costs considerably. However, the biological networks and path lengths examined are usually both small enough that memory is not a concern. Additionally, there is usually a predefined directivity in the input paths (e.g. finding a path between membrane proteins and transcription factors), and this allows us to restrict the size of the table for each step i by only placing a subset of possible vertices into V_i with per-vertex initializations. Using a min heap with a small n_L value will also substantially decrease memory requirements after the first few iterations.

Finally, note that we implement both inner-loop and outer-loop parallelism here, similar to Fascia. For the size of biological networks commonly considered for the minimum-weight path problem, outer loop parallelism performs considerably better. If every outer loop thread maintains its own min heap, we can avoid the synchronized heap insertions that inner loop parallelism requires. After all iterations are complete, we can simply examine all heaps and return the n_L-best unique paths.

4. Results and analysis

4.1. Experimental setup

Experiments were performed on various parallel platforms and interactive systems, including Stampede at the Texas Advanced Computing Center, and the Hammer and Cyberstar systems at Penn State University. For experiments where execution times are reported, we used the Compton system at Sandia National Laboratories. Each Compton node has 2 Intel Xeon E5-2670 (Sandy Bridge) processors with 64 GB DDR3 memory running RHEL 6.1. We use up to 16 nodes for our experiments. The MPI libraries used were from Intel (version 4.1) and we used OpenMP for shared-memory parallelism. Code was compiled with `icc` using the `-O3` optimization flag, and `KMP_AFFINITY` was used to control thread to core pinning.

We evaluate performance of our implementations on a collection of several large-scale low diameter graphs, listed in Table 1. Orkut and Twitter (follower network) are crawls of online social networks obtained from the SNAP Database and the Max Planck Institute for Software Systems [34,38,37]. Also from the SNAP database is the Pennsylvania Road network

Table 1
Network sizes and average and maximum degrees and approximate diameter for all networks used in our analysis.

Network	n	m	d_{avg}	d_{max}	$\widetilde{D}$	Source
Enron email	34 K	180 K	11	1.4 K	9	[33,34]
PA roads	1.1 M	1.5 M	2.8	9	430	[35,34]
Portland	1.6 M	31 M	39	275	16	[36]
Orkut	3.1 M	117 M	76	33 K	9	[37,34]
Twitter	44 M	2.0 B	37	750 K	36	[38]
sk-2005	44 M	1.6 B	73	15 M	308	[39,40]
H. pylori	710	1.4 K	4.0	54	10	[41]
S. cerevisiae	5.1 K	22 K	8.7	290	11	[41]
H. sapiens	9.1 K	41 K	9.0	250	10	[42]
Human	9.0 K	22 K	5.0	322	14	[43]
Caenorhabditis	3.2 K	5.5 K	3.4	186	14	[43]
Drosophila	7.2 K	21 K	5.9	176	12	[43]
Mammalia	8.8 K	19 K	4.4	323	18	[43]

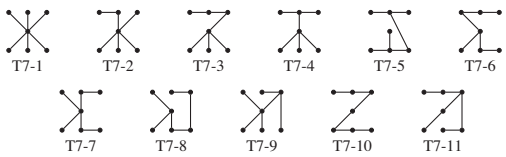

Fig. 2. All possible 7 vertex undirected tree-structured templates.

[35] and the Enron Email Corpus [33]. sk-2005 is a crawl of the Slovakian (.sk) domain performed in 2005 using UbiCrawler and downloaded from the University of Florida Sparse Matrix Collection [39,40]. Portland is a large synthetic social contact network modeled after the city of Portland, from the Virginia Tech Network Dynamics and Simulations Science Laboratory (NDSSL) [36]. The *Helicobacter pylori* (intestinal bacteria) and *Saccharomyces cerevisiae* (yeast) networks where obtained from the Database of Interacting Proteins [41], and the *Homo sapiens* (human) network is from Radivojac et al. [42]. For analyzing our weighted pathways algorithm, we considered the weighted Human, Caenorhabditis (*Caenorhabditis elegans* genus), Drosophila (fruit flies), and Mammalia protein interaction networks from the Molecular INTeraction database [43].

All the networks considered are undirected. The originally-directed Twitter and sk-2005 graphs were preprocessed to ignore edge directivity, remove multiple edges and self loops, and extract only the largest connected component. Table 1 lists the properties of the graphs after this preprocessing.

While analyzing execution times and scaling on the larger networks, we considered two different templates with 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 U5-1, U7-1, U10-1, and U12-1. These templates and their labels are shown in Fig. 1. Other templates are used in the analysis that are not listed follow the same naming convention, with UX-1 implying a simple path and UX-2 being a more complex tree. For motif finding, we looked at *all possible treelets* of size 7, 10, and 12. $k = 7, 10$, and 12 would imply 11, 106, and 551 possible tree topologies, respectively. The treelets for $k = 7$ are given in Fig. 2.

4.2. Single-node performance

To assess single node FASCIA performance, we will examine running times of a single iteration on moderate-sized networks across varying template sizes, running times for several iterations across all motifs on smaller PPI networks, parallel scaling, memory utilization with the various strategies, as well as an analysis of approximation error.

4.2.1. Running times vs. template size

Fig. 3 gives the absolute single-node running times for all templates listed in Fig. 1 on the Portland and Orkut networks. These results are from running the inner loop-parallel version on 16 cores. We observe minimal to no performance improvement when using hyperthreading, qnd so most tests were performed with only a single thread per core despite two hardware thread contexts.

As can be observed on Fig. 3, the single-iteration time for smaller templates is extremely low, making it feasible to obtain realtime count estimates for 7 vertex templates on both networks. Even for the largest template, the total running time was still less than 20 min on both networks. The U12-2 template took the longest time as expected. This template was explicitly designed to stress subtemplate partitioning and therefore gives a practical upper bound for our running times across all template of size 12 and smaller. 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 3× variation in running time.

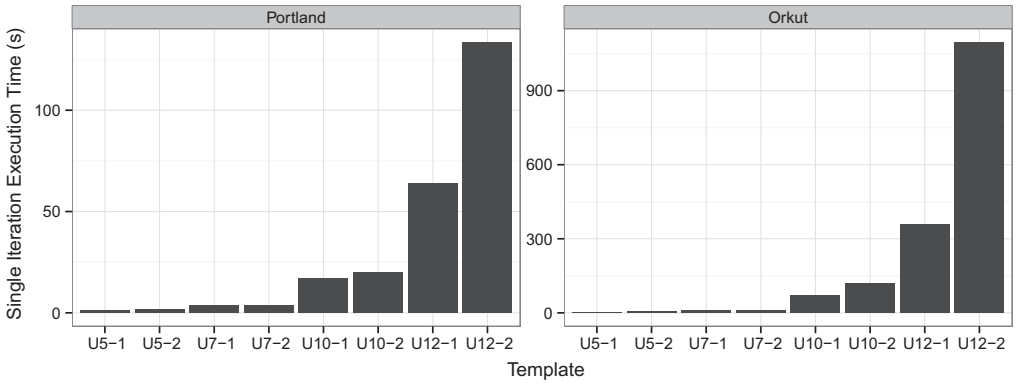

Fig. 3. FASCIA running times on templates of size 5, 7, 10, and 12 vertices, on the Portland and Orkut networks, for a single iteration, with inner loop parallelism.

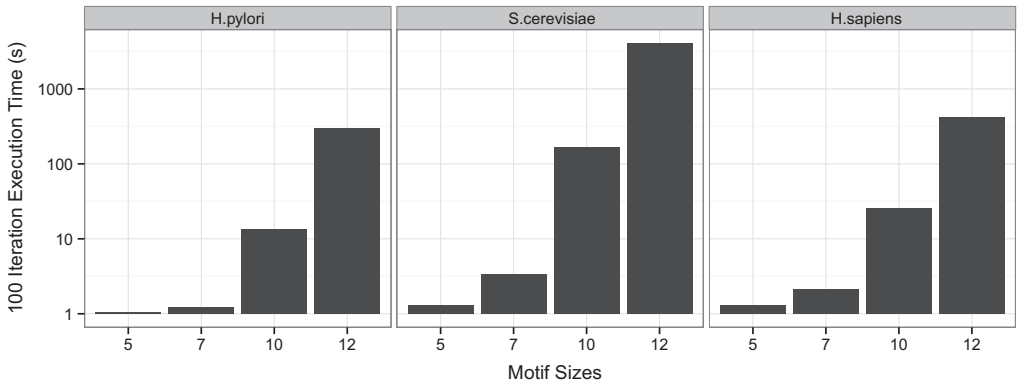

Fig. 4. FASCIA running times on templates of size 5, 7, 10, and 12 on the *H. pylori*, *S. cerevisiae*, and *H. sapiens* PPI networks for 100 iterations with outer loop parallelism.

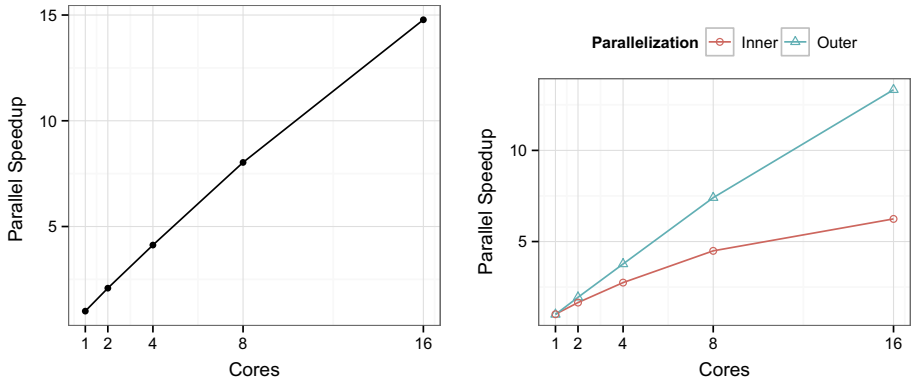

Fig. 5. Parallel scaling from 1 to 16 cores of the U12-2 template on the Portland network for a single iteration with inner-loop parallelism (left) and parallel scaling for 100 iterations of all 10 vertex templates on the *H. pylori* network with both inner and outer scaling (right).

Fig. 4 gives the running times for 100 iterations across all possible 5, 7, 10, and 12-vertex templates on the *H. pylori*, *S. cerevisiae*, and *H. sapiens* protein interaction networks. For these tests, we use outer loop parallelism. It demonstrates superior running times on smaller networks with larger iteration counts, as there is less parallel overhead on a per-iteration basis. Note the log scale in Fig. 4. Both the running time of FASCIA and the total number of possible templates increase exponentially with increasing network size. This demonstrates the importance of implementing fast serial algorithms for subgraph counting when analyzing motifs of larger sizes.

4.2.2. Parallel scaling

We now observe how our approaches scale when increasing the number of processing cores. Fig. 5 gives the parallel speedups for 1–16 cores on the Portland network with inner-loop parallelism, as well as the *H. pylori* network with both outer loop and inner loop parallelism. As was mentioned previously, we observe better speedups with outer loop parallelism on the smaller networks, as the per-iteration parallelization overhead is reduced. However, on larger networks, we still observe very good speedups and near-linear scaling with the inner-loop parallelism, since the computational requirements overshadow the parallel overheads for these instance. Overall, our implementation scales quite well, demonstrating about 15× and 14× speedups on the larger Portland and smaller *H. pylori* networks, respectively.

4.2.3. Reduction in memory use

Fig. 6 demonstrates impact of the memory-reducing optimizations over the baseline naïve approach. The peak memory footprint is given in both the networks for various template sizes. For the Portland network, we see a 20% savings over baseline with the improved dynamic programming table representation. Further, if we consider the case of per-vertex labels in the graph (all vertices randomly initialized with one of 8 labels), the memory requirements drop considerably, due to the much higher selectivity that the label restriction imposes.

In Fig. 6, we also see how a hash table representation can improve memory usage dramatically over using the three-dimensional table, on certain networks. The PA road network is quite regular and nearly planar, and so it is expected that for

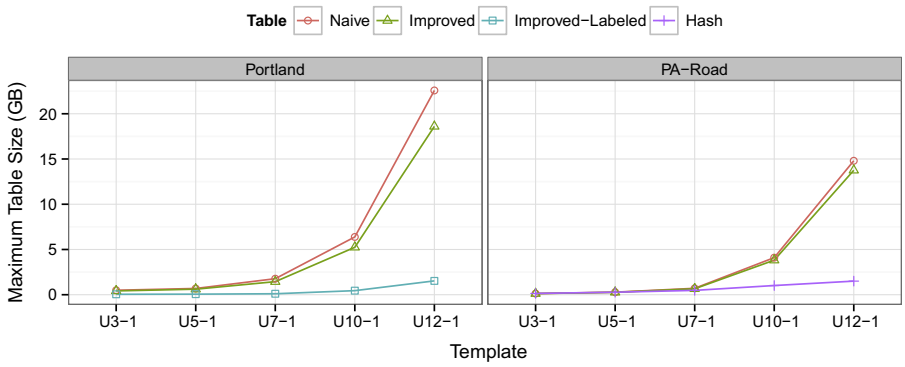

Fig. 6. Peak memory use reduction on the unlabeled and labeled Portland network with the improved table (left), and memory use reduction that results from using an improved table and hash table on the PA Road network (right).

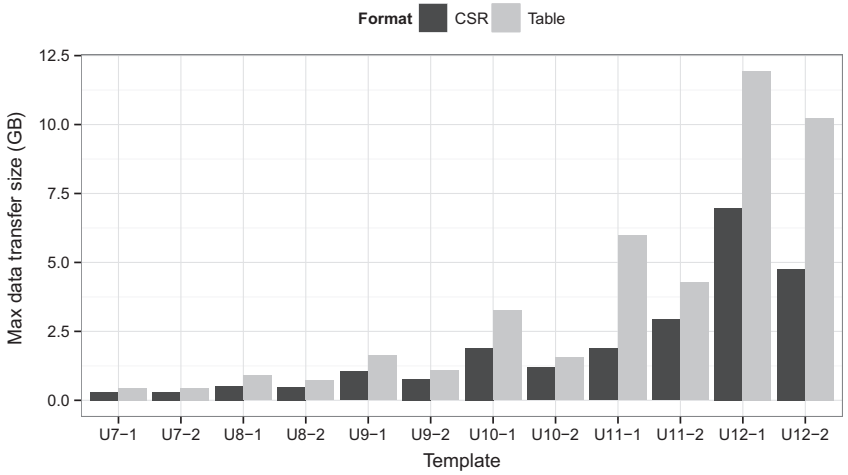

Fig. 7. Maximum data transfer required in GB on Portland using various templates from 7 to 12 vertices with the improved table and CSR storage.

any given template, a vertex will have an embedding rooted at it. This results in the table having to initialize storage for every vertex. However, since the network is sparse, it is unlikely that every color set will have an embedding, which is why we see such a significant memory use reduction. On a denser network like Portland, it is unlikely there would be much improvement with the hash table, due to the relatively higher number of embeddings relative to the number of vertices in the network.

Fig. 7 gives the maximum data transfer during a single iteration for the UX-1 and UX-2 templates from 7 to 12 vertices on the Portland network with the improved table and its CSR compression. On average, a 35% reduction is observed, with a reduction of over 77% occurring for select templates. As it was noted, the Portland network has a relatively high average degree, which correspondingly results in a relatively high rate of template embeddings across the network. An even higher compression ratio is observed on lower-density networks.

4.2.4. Error analysis

We next analyze the error in the subgraph counts produced on small and moderate-sized networks. We report the magnitude of relative error (difference in counts divided by true count) in the two figures. In Fig. 8 (left), we observe the error produced when counting 3 and 5 vertex chain templates on the Enron network. We note that the error falls under 1% after three iterations for both templates. We observe higher error with the larger template. The extremely small number of iterations necessary for low error, on modest-sized networks, mirrors the results seen in prior work [11,15].

For the smaller *H. pylori* network, we note that it takes about 100 iterations to reach about 1% average error across all 7 vertex templates. This network is very small and sparse, and so for large templates, a relatively larger number of iterations are required. Generally, we observe per-iteration error increasing with template size, but decreasing with network size. Also, the greater the number of template embeddings that exist within the graph, the lower the error. This is due to the fact that the random coloring of the graph and the subsequent count scaling has a relatively-smaller impact on the final count

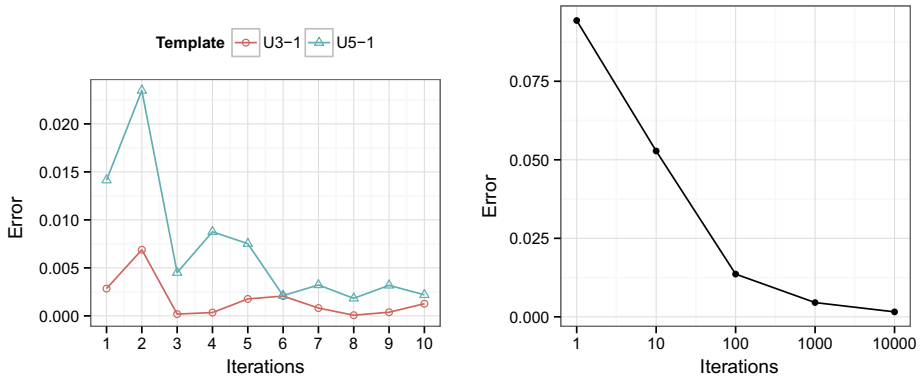

Fig. 8. Error obtained with the 3 and 5 vertex path templates on the Enron network after a small number of iterations (left) and the average error over all possible 7 vertex templates on the *H. pylori* network after 1–10 K iterations (right).

estimate. We observe low inter-iteration variation between produced counts on large scale networks. A dynamic stopping criteria based on the variance of produced per-iteration counts is left for future work.

Note that Alon et al. [1] proved that to ensure a computed count is within $\mathcal{C}(1 \pm \epsilon)$ with probability $(1 - 2\delta)$, where $\mathcal{C}$ is the true count, at most $\frac{e^k \log 1/\delta}{\epsilon^2}$ iterations of the dynamic programming scheme are required. However, this upper bound is very loose in practice, as it ignores network size and topology. For example, we require only 100 iterations to compute counts with an error less than 1% for a 7 vertex template on *H. pylori* ($\epsilon = 0.01$).

4.2.5. Analytic capabilities

We demonstrate an application of our shared-memory Fascia in bioinformatics, through a global comparison of various protein interaction networks from the DIP, a previously-used human protein interaction network, a randomly-generated small-world network, and the PA Road network. Fig. 9 gives the subgraph frequency distances [3] using all possible templates of sizes 4 and 9 vertices. The heatmap indicates agreement values normalized between 0 and 1. A darker color indicates a higher agreement value between the networks. We order the biological networks in perceived complexity of the organism from yeast (*S. cerevisiae*) to human.

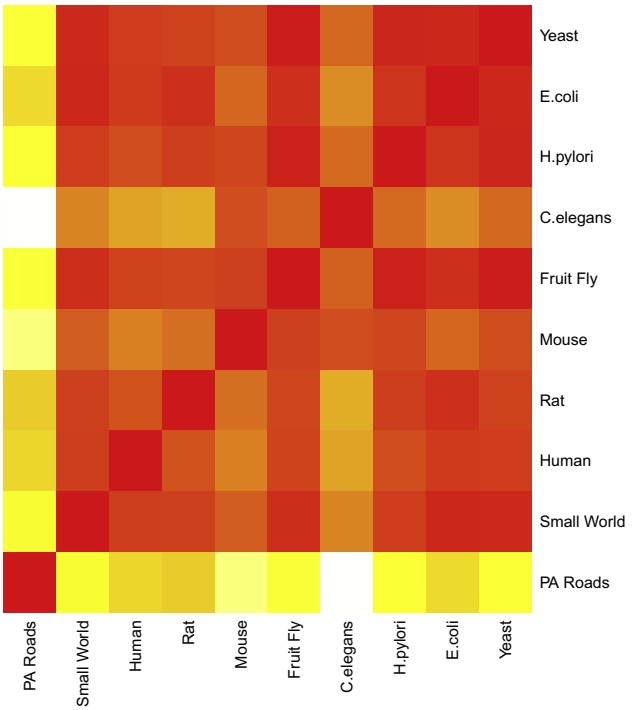

Fig. 9. Heatmap generated by calculating the treelet frequency distance between all possible 3–8 templates on several biological PPI networks.

We observe high agreement between the three unicellular organisms, Yeast, *Escherichia coli*, and *H. pylori*. We also observe that the small-world network demonstrates a good fit for this relatively-simple comparative metric. The regularly-structured, high-diameter, and near-planar PA Road network shows obvious and strong dissimilarity with the other networks, as would be expected.

4.3. Multi-node performance

For the distributed-memory FASCIA implementation, we again analyze performance with regards to running times and parallel scaling. We also demonstrate the inter-node communication reduction with CSR table compression, which also translates to memory savings during the subsequent dynamic programming step. For these experiments, we employ a label propagation-based graph partitioning [44] with random intra-partition vertex reordering to balance overall computation and communication costs.

4.3.1. Running times vs. template size

Fig. 10 gives the running times on the large sk-2005 and Twitter networks for templates between 5 and 10 vertices on 16 nodes of Compton. Due to the large scale of the networks and restricted parallelism on our modest cluster, the memory requirements for the 12-vertex template was too high for us to run to completion. Using more compute nodes would reduce per-node memory requirements and allow us to scale to larger network and template sizes. This is left for future work.

From Fig. 10, we observe that count times for 5-vertex templates complete in seconds, and the larger templates in minutes on these networks. Note that the running times of the binary tree-structured templates are lower than that of the path-like templates in these instances. This is because the computational requirements for the tree templates are higher than the path templates, but they have lower communication and memory costs. This leads to lower performance in shared memory, but faster performance in distributed memory.

4.3.2. Parallel scaling

Fig. 11 gives the scaling of the single-iteration running times on the Orkut network from 1 to 16 nodes, and the scaling of the sk-2005 from 2 to 16 nodes. The Orkut network calculates the counts for the U12–2 templates and sk-2005 network uses the U7-2 network. We show scaling for the total execution time, the portion of time spent in the counting computation phase, and the total time spent in the communication phase.

We observe about a 4× overall speedup on the Orkut network, with about 10× speedup of just computation. Communication increases about 3× from 2 to 16 nodes. We observe higher communication costs with this approach compared to our previous implementation [23], due to a more complex communication phase. However, this higher complexity reduces memory utilization and is necessary in order to calculate the counts on networks as large as sk-2005. On the sk-2005 network, we observe about 2.5× speedup from 2 to 16 nodes. We observe that there is relatively good scaling with computational time, but at the cost of increasing communication time. Due to the larger-scale and higher overall computation costs, even with a smaller template, the communication costs are slightly less than the computation costs at 16 nodes. With greater parallelism, it is likely that communication costs will dominate.

We note that the increasing communication cost is inherent to the nature of the graphs being analyzed. Small-world graphs tend to partition very poorly, with the number of cut edges output from a state-of-the-art partitioner being of similar magnitude to that resulting from a random partitioning, especially for a high task count. To explicitly quantify the issue, we assume a random vertex partitioning. The number of cut edges approximately follows $m(1-\frac{1}{nt})$, where nt is the number of tasks and m is total edges. For 16 tasks, we expect communication to be necessary on up to 94% of edges.

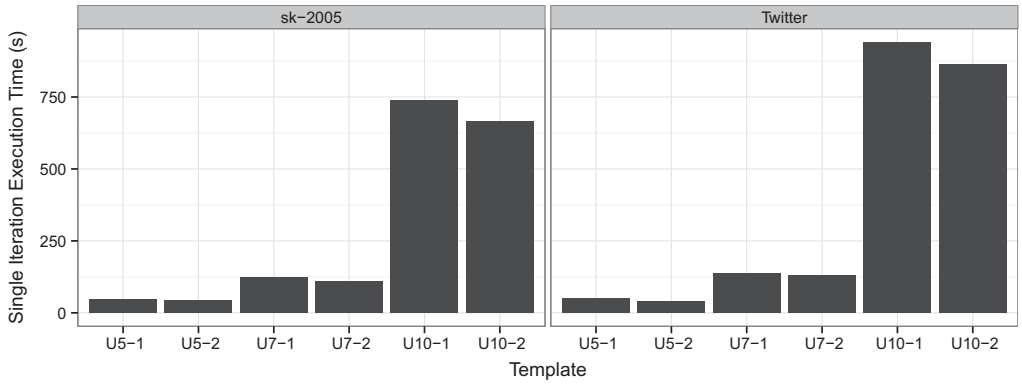

Fig. 10. Running times on 16 nodes of Compton of tested 5, 7, and 10 vertex templates on the sk-2005 and Twitter networks for a single iteration with partitioned counting and inner loop parallelism.

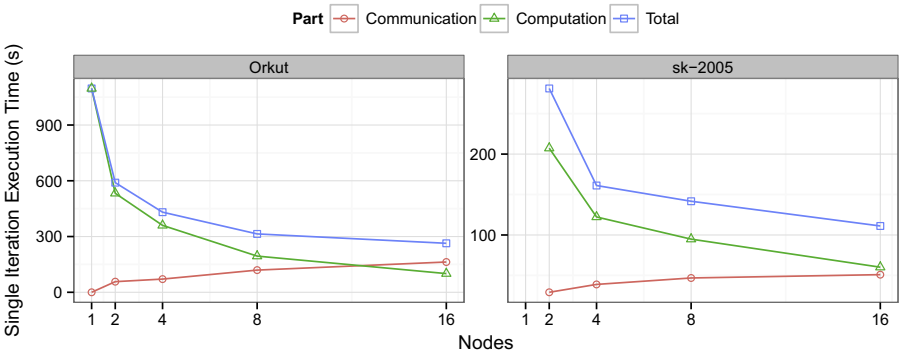

Fig. 11. Parallel scaling from 1 to 16 nodes of the U12-2 template on Orkut network and the U7-2 template on the sk-2005 network for a single iteration with partitioned counting and inner loop parallelism.

A specific vertex v needs to communicate its counts table information to all tasks containing its neighbors. If v is of high or even moderate degree (greater than the number of tasks), this could result in communication of v's data to all other tasks. Essentially, as communication costs are proportional to the number of inter-task (cut) edges, we expect the plot of communication times to follow a similar curve proportional to $(1 - \frac{1}{nt})$. From Fig. 11, this is what we observe. In future work, we will attempt to optimize the All-to-all exchange in order to minimize these costs for larger networks, and achieve better parallel scaling.

4.3.3. Analytic capabilities

We use our distributed counting implementation to find motifs [4,2] on several large networks. We compare the relative counts produced for all 7 vertex templates, shown in Fig. 2, on the Portland, Orkut, sk-2005, and Twitter networks. The relative counts are given by Fig. 12.

On Orkut, sk-2005, and Twitter, we observe very high selectivity for the T7-1 and T7-4 templates. We also note very low selectivity and extremely large embedding counts for T7-5 and T7-6 templates on these networks. The Portland network shows the least variation in relative count magnitudes for different templates. We note that the Portland network is randomly generated, while the Orkut, sk-2005, and Twitter networks all represent real-world datasets.

4.4. FastPath performance

We now analyze our FastPath implementation for finding low-weight paths in the weighted Human, Drosophila, Caenorhabditis, and Mammalia protein interaction networks from the MINT database. We compare our per-iteration running times to the current state-of-the-art serial code of Hüffner et al. (HWZ) [24]. We do not directly compare to other work, as most other work has focused on reducing the number of color-coding iterations, rather than improving per-iteration performance.

Fig. 13 gives the running times for FastPath and HWZ to determine the 100-best 4–9 length paths over 500 search iterations. We report serial and parallel performance (inner-loop parallel) of FastPath, with our parallel code run across 16 cores on Compton. We include both the Hüffner et al. baseline color-coding approach (HWZ), as well as their dynamic programming heuristic technique (HWZ-Heuristic). We also analyzed outputs, and noticed that all four approaches find paths with the same minimum weight.

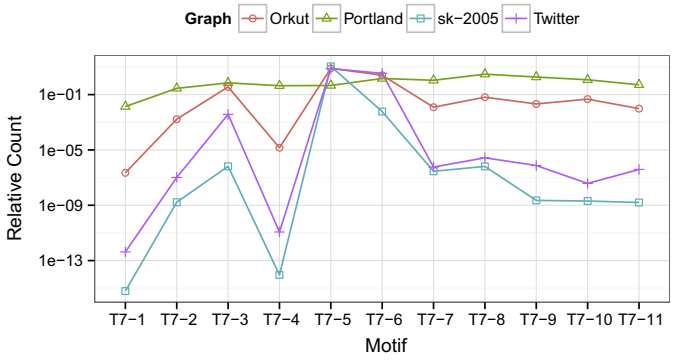

Fig. 12. Motif finding by comparing the relative counts obtained on Portland, Orkut, sk-2005, and Twitter for all possible 7 vertex templates.

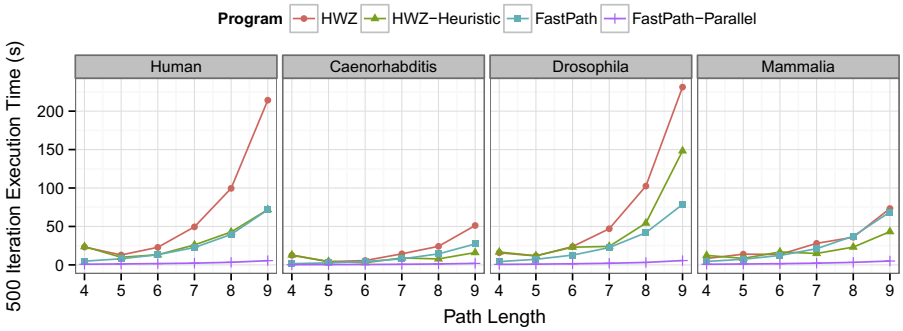

Fig. 13. Absolute running times for 500 iterations of finding path lengths 4 through 9 using the Hüffner et al. baseline and heuristic methods, as well as FastPath in serial and on 16 cores.

From Fig. 13, we observe that our parallel FastPath implementation demonstrates considerable speedup across all test instances. We note that the running times of serial FastPath are close to that of HWZ-Heuristic on most tests, and that the heuristic offers improved speedup to the HWZ baseline. Our current version of FastPath does not implement the HWZ heuristic, but future work combining both our and Hüffner et al.'s optimizations could lead to improved performance.

Fig. 14 gives the parallel speedup from 1 to 16 cores for all the tested path lengths and networks. We observe that 500 iterations of path length 9 takes about 5 s on the Human network, with a parallel speedup of about 12.5×. As expected, we see that we obtain better speedup for larger values of L. This is because the total per-iteration work performed scales in proportion to $(m+n)2^L$, and so for larger graphs, there is more work to be parallelized. Also, note that L barrier synchronizations are required for each color-coding iteration. There is some overhead because of barrier synchronization, but it is insignificant compared to the parallelized work.

We also employ FastPath to see if we can determine the minimum path weight in the Human-MINT network for different values of L. We performed a search with paths of length 5 using FastPath and FASPAD [45], a tool based on the Hüffner et al. method. We found several minimum weight paths in the network in just a few iterations. Two paths are shown in Fig. 15, one generated using FastPath and the other using FASPAD. The simple path detected is shown with black edges, and other edges connecting proteins in the path are shown in grey. Further analysis using DAVID [46,47] reveals that proteins in the high-scoring paths appear in the well-studied chronic myeloid leukemia KEGG pathway [48]. We also compared our results with those presented by Gabr et al. [25] for different path lengths, and despite the fact that we do not explicitly restrict the search space from membrane proteins to transcription factors in our test, we notice the same proteins appearing in both our works.

Finally, we demonstrate the statistical significance of the paths found using FastPath. We use the standard score metric, also known as a z-score, which gives the number of standard deviations a given path weight is from the mean path weight determined over some sample of paths. The z-score is calculated as $z = \frac{x-\mu}{\sigma}$, where x is a given single path weight and μ and σ are the mean and standard deviation path weights over the sample, respectively.

Using all four networks and path lengths from 3 to 8, we take the mean (μ) and standard deviation (σ) of weights from 1000 randomly-selected paths, and calculate the z-scores (z) using the lowest weight (x) returned by FastPath after 500 iterations. Table 2 gives these results.The statistical significance of the paths we find is apparent. The z-scores obtained on the Drosophila and Caenorhabditis networks are especially high. As was similarly noted by Gabr et al., we observe that z-score

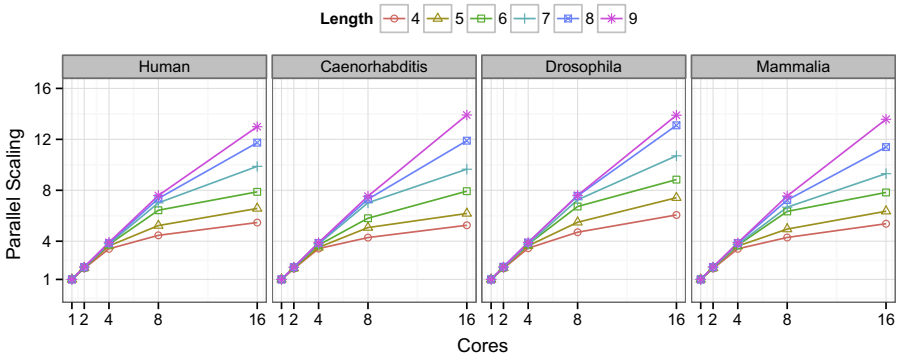

Fig. 14. Speedup for FastPath from 1 to 16 cores for path lengths 4 through 9.

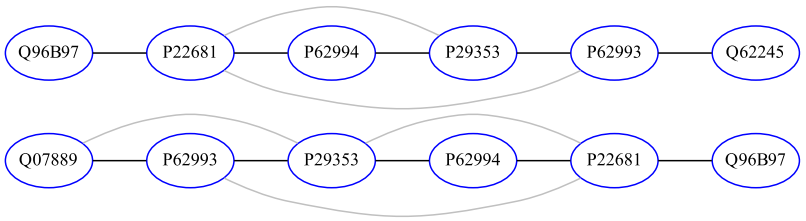

Fig. 15. Sample minimum-weight paths of path length five found in the MINT Human PIN using FastPath (top) and FASPAD (bottom). The path weight is 0.0211329 in both cases.

Table 2
The lowest weight paths obtained with FastPath for several networks and path lengths, along with its z-score calculated using the mean and standard deviation of a random sample of paths.

Network	Path length	x	μ	σ	z
Human	3	0.003	3.37	0.87	3.80
	4	0.012	4.55	1.07	4.22
	5	0.021	5.67	1.22	4.61
	6	0.045	6.76	1.45	4.61
	7	0.065	7.92	1.59	4.92
	8	0.086	9.03	1.72	5.19
Caenorhabditis	3	0.65	3.73	0.27	11.1
	4	1.49	4.98	0.32	10.6
	5	2.51	6.23	0.33	11.2
	6	3.27	7.46	0.37	11.2
	7	4.03	8.71	0.38	12.2
	8	4.86	9.95	0.41	12.2
Drosophila	3	1.12	3.76	0.25	10.3
	4	1.80	5.02	0.24	13.1
	5	2.65	6.27	0.29	12.3
	6	3.64	7.53	0.23	16.6
	7	4.37	8.80	0.23	19.0
	8	4.81	10.0	0.26	19.7
Mammalia	3	0.046	3.42	0.98	3.44
	4	0.063	4.62	1.20	3.80
	5	0.116	5.70	1.48	3.77
	6	0.158	6.85	1.56	4.29
	7	0.178	8.04	1.68	4.68
	8	0.286	9.34	1.88	4.82

consistently increases with path length. This demonstrates the importance of scalable methods for finding low-weight paths of large L. In future work, we will perform a detailed exploration of the biological significance of the paths found, and see if they correlate with the apparent statistical significance.

4.5. Comparisons to other tools

The color-coding subgraph counting tools SAHAD and PARSE, both by Zhao et al. [15,11], perform partitioned counts in distributed-memory environments. A single iteration of PARSE for counting a 6-vertex chain on a network with 2 million vertices and 50 million edges, using 400 cores across 50 nodes, takes about an hour. A single iteration of SAHAD takes 25 min for counting a 10-vertex tree in the same 50 million edge graph, using 1344 cores across 42 compute nodes. With Fascia, we show that a single color-coding iteration for a 10 vertex tree template on a 44 million vertex, 2 billion edge network takes about 15 min with 256 cores (16 nodes). We are not aware of any other work for counting occurrences of large templates in large networks.

In the previous section, we have performed a direct comparison of the per-iteration running times of FastPath and the Hüffner et al. approach. The focus of other work in this area, such as Gabr et al. [25] and Scott et al. [12], has been to reduce the total number of color-coding iterations required for a given confidence and error bound. A combination of their techniques with our algorithmic improvements can possibly produce a faster tool.

5. Conclusions and future work

This work presents several new optimizations for implementations of color-coding based graph algorithms. Using these optimizations, we create shared- and distributed-memory parallelized Fascia, a fast and memory-efficient tool for subgraph

counting on both small and large networks. Future work will further extend Fascia to even larger networks and for scaling to larger compute platforms. The all-to-all exchange step of Fascia is currently a bottleneck in the distributed-memory implementation, and we intend to investigate alternatives.

The optimization techniques we describe for Fascia can be applied to other graph computations that use the color-coding method. To demonstrate this, we present FastPath, which is a tool for enumerating low-weight simple paths in a weighted graph with positive edge weights.

Acknowledgments

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070, ACI-1238993, and ACI-1444747) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This work is also supported by NSF grants ACI-1253881, CCF-1439057, and the DOE Office of Science through the FASTMath SciDAC Institute. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04–94AL85000.

References

[1] N. Alon, R. Yuster, U. Zwick, Color-coding, J. ACM 42 (4) (1995) 844–856.
[2] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, U. Alon, Network motifs: simple building blocks of complex networks, Science 298 (5594) (2002) 824–827.
[3] N. Pržulj, Modeling interactome, scale-free or geometric?, Bioinformatics 20 (18) (2004) 3508–3515
[4] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, S. Sahinalp, Biomolecular network motif counting and discovery by color coding, Bioinformatics 24 (13) (2008) i241–i249.
[5] J. Huan, W. Wang, 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.
[6] M. Kuramochi, G. Karypis, Frequent subgraph discovery, in: Proc. IEEE Int'l. Conf. on Data Mining (ICDM), 2001, pp. 313–320.
[7] N. Kashtan, S. Itzkovitz, R. Milo, U. Alon, Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs, Bioinformatics 20 (11) (2004) 1746–1758.
[8] S. Wernicke, Efficient detection of network motifs, IEEE/ACM Trans. Comput. Biol. Bioinf. 3 (4) (2004) 347–359.
[9] M. Rahman, M. Bhuiyan, M. Hasan, GRAFT: an efficient graphlet counting method for large graph analysis, IEEE Trans. Knowl. Data Eng. (TKDE) 26 (10) (2014) 2466–2478.
[10] J. Chen, W. Hsu, M.L. Lee, S.-K. Ng, NeMoFinder: dissecting genome-wide protein-protein interactions with meso-scale network motifs, in: Proc.ACM Int'l. Conf. on Knowledge Discovery and Data mining (KDD), 2006, pp. 106–115.
[11] Z. Zhao, G. Wang, A.R. Butt, M. Khan, V.S.A. Kumar, M.V. Marathe, SAHAD: subgraph analysis in massive networks using Hadoop, in: Proc. 26th Int'l. Parallel and Distributed Processing Symp. (IPDPS), 2012, pp. 390–401.
[12] J. Scott, T. Ideker, R.M. Karp, R. Sharan, Efficient algorithms for detecting signaling pathways in protein interaction networks, J. Comput. Biol. 13 (2) (2006) 133–144.
[13] B.P. Kelley, R. Sharan, R.M. Karp, T. Sittler, D.E. Root, B.R. Stockwell, T. Ideker, Conserved pathways within bacteria and yeast as revealed by global protein network alignment, Proc. Natl. Acad. Sci. 100 (20) (2003) 11394–11399.
[14] M. Steffen, A. Petti, J. Aach, P. D'haeseleer, G. Church, Automated modelling of signal transduction networks, BMC Bioinf. 3 (1) (2002) 34.
[15] Z. Zhao, M. Khan, V.S.A. Kumar, M.V. Marathe, Subgraph enumeration in large social contact networks using parallel color coding and streaming, in: Proc. 39th Int'l. Conf. on Parallel Processing (ICPP), 2010, pp. 594–603.
[16] G.M. Slota, K. Madduri, Fascia: parallel subgraph counting, <http://fascia-psu.sourceforge.net/>, (last accessed March 2015).
[17] G.M. Slota, K. Madduri, FastPath: fast parallel pathway enumeration, <http://fastpath-psu.sourceforge.net/>, (last accessed March 2015).
[18] N. Pržulj, Biological network comparison using graphlet degree distribution, Bioinformatics 23 (2) (2007) e177–e183.
[19] N. Pržulj, D. Corneil, I. Jurisica, Efficient estimation of graphlet frequency distributions in protein-protein interaction networks, Bioinformatics 22 (8) (2006) 974–980.
[20] T. Milenković, N. Pržulj, Uncovering biological network function via graphlet degree signatures, Cancer Inf. 6 (2008) 257–273.
[21] I. Bordino, D. Donata, A. Gionis, S. Leonardi, Mining large networks with subgraph counting, in: Proc. 8th IEEE Int'l. Conf. on Data Mining (ICDM), 2008, pp. 737–742.
[22] G.M. Slota, K. Madduri, Fast approximate subgraph counting and enumeration, in: Proc. 42nd Int'l. Conf. on Parallel Processing (ICPP), 2013, pp. 210–219.
[23] G.M. Slota, K. Madduri, Complex network analysis using parallel approximate motif counting, in: Proc. Int'l. Parallel and Distributed Processing Symp. (IPDPS), 2014, pp. 405–414.
[24] F. Hüffner, S. Wernicke, T. Zichner, Algorithm engineering for color-coding with applications to signaling pathway detection, Algorithmica 52 (2) (2008) 114–132.
[25] H. Gabr, A. Dobra, T. Kahveci, From uncertain protein interaction networks to signaling pathways through intensive color coding, in: Proc. Pacific Symp. on Biocomputing, 2012, pp. 111–122.
[26] T. Shlomi, D. Segal, E. Ruppin, R. Sharan, Qpath: a method for querying pathways in a protein-protein interaction network, BMC Bioinf. 7 (1) (2006) 199.
[27] B. Dost, T. Shlomi, N. Gupta, E. Ruppin, V. Bafna, R. Sharan, Qnet: a tool for querying protein interaction networks, J. Comput. Biol. 15 (7) (2008) 913–925.
[28] G. Gülsoy, B. Gandhi, T. Kahveci, Topology aware coloring of gene regulatory networks, in: Proc. 2nd ACM Conf. on Bioinformatics, Computational Biology and Biomedicine (BICoB), 2011, pp. 435–440.
[29] G. Guelsoy, B. Gandhi, T. Kahveci, Topac: alignment of gene regulatory networks using topology-aware coloring, J. Bioinf. Comput. Biol. 10 (1) (2012) 1240001.
[30] E. Beckenbach, Applied Combinatorial Mathematics, Krieger Pub Co, 1981.
[31] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-MAT: a recursive model for graph mining, in: 4th SIAM Int'l. Conf. on Data Mining (SDM), 2004, pp. 442–446.
[32] C. Groër, B.D. Sullivan, S. Poole, A mathematical analysis of the R-MAT random graph generator, Networks 58 (3) (2011) 159–170.
[33] B. Klimmt, Y. Yang, Introducing the Enron corpus, in: Proc. 1st Conf. on Email and Anti-Spam (CEAS), 2004, pp. 1–2.
[34] J. Leskovec, SNAP: stanford network analysis project, <http://snap.stanford.edu/index.html>, (last accessed Feb 2014).
[35] J. Leskovec, K. Lang, A. Dasgupta, M. Mahoney, Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters, Internet Math. 6 (1) (2009) 29–123.

[36] Network Dynamics and Simulation and Science Laboratory, Synthetic data products for societal infrastructures and proto-populations: Data set 1.0, Tech. Rep. NDSSL-TR-06-006, Virginia Polytechnic Institute and State University, 2006.
[37] J. Yang, J. Leskovec, Defining and evaluating network communities based on ground-truth, in: Proc. 12th IEEE Int'l. Conf. on Data Mining (ICDMs), 2012, pp. 745–754.
[38] M. Cha, H. Haddadi, F. Benevenuto, K.P. Gummadi, Measuring user influence in Twitter: the million follower fallacy, in: Proc. Int'l. Conf. on Weblogs and Social Media (ICWSM), 2010, pp. 1–8.
[39] P. Boldi, B. Codenotti, M. Santini, S. Vigna, UbiCrawler: a scalable fully distributed web crawler, Software Pract. Exp. 34 (8) (2004) 711–726.
[40] T.A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Software 38 (1) (2011) 1–25.
[41] I. Xenarios, L. Salwinski, X.J. Duan, P. Higney, S.M. Kim, D. Eisenberg, DIP, the database of interacting proteins: a research tool for studying cellular networks of protein interactions, Nucleic Acids Res. 30 (1) (2002) 303–305.
[42] P. Radivojac, K. Page, W.T. Clark, B.J. Peters, A. Mohan, S.M. Boyle, S.D. Mooney, An integrated approach to inferring gene-disease associations in humans, Proteins 72 (3) (2008) 1030–1037.
[43] A. Chatr-Aryamontri, A. Ceol, L.M. Palazzi, G. Nardelli, M.V. Schneider, L. Castagnoli, G. Cesareni, Mint: the molecular interaction database, Nucleic Acids Res. 35 (suppl.1) (2007) D572–D574.
[44] G.M. Slota, K. Madduri, S. Rajamanickam, PULP: scalable multi-objective multi-constraint partitioning for small-world networks, in: Proc. IEEE Int'l. Conference on Big Data (BigData), 2014, pp. 1–10.
[45] F. Hüffner, S. Wernicke, T. Zichner, Faspad: fast signaling pathway detection, Bioinformatics 23 (13) (2007) 1708–1709.
[46] D.W. Huang, B.T. Sherman, R.A. Lempicki, Systematic and integrative analysis of large gene lists using DAVID bioinformatics resources, Nat. Protoc. 4 (1) (2009) 44–57.
[47] D.W. Huang, B.T. Sherman, R.A. Lempicki, Bioinformatics enrichment tools: paths toward the comprehensive functional analysis of large gene lists, Nucleic Acids Res. 37 (1) (2009) 1–13.
[48] M.W.N. Deininger, J.M. Goldman, J.V. Melo, The molecular biology of chronic myeloid leukemia, Blood 96 (10) (2000) 3343–3356.