A New Pipelined Adaptive-Group Communication for Large-Scale Subgraph Counting

Langshi Chen[1] Bo Peng[1] Zhao Zhao[2] Saliya Ekanayake [2] Anil Vullikanti[2] Madhav Marathe[2]
Shaojuan Zhu[3] Emily Mccallum[3] Lisa Smith[3] Lei Jiang[1] Judy Qiu[1]
[1]School of Informatics, Computing and Engineering, Indiana University
[2]Virginia Tech
[3]Intel Corporation
{lc37, pengb, jiang60, xqiu}@indiana.edu
{philofellow, vsakumar, mmarathe}@@vt.edu
{shaojuan.zhu, emily.l.mccallum, lisa.m.smith }@intel.com

***Abstract*—Subgraph counting involves comparing a subgraph template across a large input graph. Many domains have networks that can benefit from fast subgraph isomorphism for billion- or trillion- edge graphs generated by Internet of Things, social networks, and biological networks. However, it is computationally challenging with large template sizes since the time complexity and space grow exponentially. In this paper, we investigate parallelization and memory reducing strategies and propose a new pipelined adaptive-group communication for massive subgraph counting problems. In contrast to point-to-point MPI solutions for graph problems, we leverage model-centric parallelism and communication optimization. This includes 1) a fine-grained pipeline communication with regroup operation to significantly reduce memory footprint, 2) partition of big model such as neighbor list of subgraph for better in memory thread concurrency and load balance, and 3) interleaved computation and communication. We run experiments with tree-like subgraph counting based on color coding algorithm over an Intel Xeon cluster. The experimental results show that our implementation of Harp-DAAL subgraph counting achieves 5x speedups compared to related work and reduce memory utilization by a factor of 30 on large templates of 12 to 15 vertices and input graphs of 2 to 5 billions of edges.**

***Keywords*-Subgraph Counting, Big Data, HPC**

I. Introduction

Counting subgraphs in massive graphs is one of the fundamental tasks in graph analysis and has various applications, including analyzing the connectivity of social networks, uncovering network motifs (repetitive subgraphs) in gene regulatory networks of organism, indexing graph databases, task scheduling optimization in infrastructure monitoring, event detection in cyber security and other areas of national interest. Although these advanced graph analytics may provide a deep insight into the network's functional abilities, they will require the computing power to analyze the billion- and trillion-edge graphs generated by the Internet of Things, ever-expanding social networks, biological networks and future sensor networks.

In the Subgraph counting, we are given two graphs T and G as an input, and the question is whether $G = (V, E)$ contains a subgraph (also called graphlet) that is isomorphic to T (template) and subsequently counting the number of matching. These problems generalize subgraph isomorphism, which is NP-hard, even for very simple templates. The best algorithms for exact counting run in time $\Omega(n^{k/2})$ for trees [1], which motivates approximation algorithms. Color-coding [2] is a powerful technique to solve this problem using dynamic programming algorithm when the template has a bounded treewidth. This gives a *fixed parameter tractable* algorithm: for trees of size k, color coding runs in time exponential in k, but linear in m, the number of edges of the input graph. Treelet counting can also be used as a kernel to estimate the Graphlet frequency distribution (GFD), another widely used problem which estimates the relative frequency among all subgraphs of the same size. [3] shows that a careful implementation of treelet counting can push the limits of the state-of-the-art of GFD, both in terms of the size of the input graph and the template.

In this paper, we focus on treelet counting problem, and our main contributions in this paper are the following:

- We investigate computing capabilities that can solve subgraph counting problem with large tree-like templates (up to 15 vertices) and large graph (up to 5 billion edges and 0.66 billion vertices).
- A novel pipeline *Adaptive_Group* communication with regroup operation is developed to accelerate communication. The pipeline design reduces memory usage by a factor of 30 on large templates (12 to 15 vertices) and hides communication via overlapping with computation. Additional optimization reduces load imbalance by fine-grained parallelism at node level.
- We compare our results to state-of-the-art MPI Fascia implementation with 5X speedups.

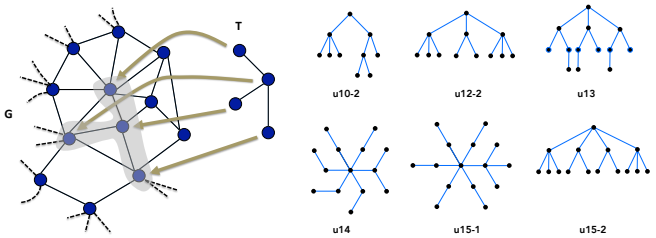
The rest of the paper is organized as follows. Section II introduces the problem, color coding algorithm and scaling challenges. Section III presents our approach on pipelined Ring-AlltoAll communication, as well as load

balance optimization for sparse input graphs. Section IV contains experimental analysis of our proposed methods and show performance improvements. After section V on related works, we conclude in Section VI.

II. Background and Preliminaries

A. Subgraph Counting Problem

Let $G = (V, E)$ denote a graph on the set V of nodes and set E of edges. We say that a graph $H = (V_H, E_H)$ is a *non-induced subgraph* of G if $V_H \subseteq V_G$ and $E_H \subseteq E_G$. We note that there may be other edges in $E_G - E_H$ among the nodes in V_H in an induced embedding. A template graph $T = (V_T, E_T)$ is said to be isomorphic to a non-induced subgraph $H = (V_H, E_H)$ of G if there exists a bijection $f : V_T \to V_H$ such that for each edge $(u, v) \in E_T$, we have $(f(u), f(v)) \in E_H$. In this case, we also say that H is a non-induced embedding of T.



(a) Embedding of Template (b) Tree-like Templates

Figure 1: (a) Example showing an embedding of a template T in the graph G. (b) Tree-like Templates with large number of vertices

Let $n(T, G)$ denote the number of all embeddings of template T in graph G. Our problem is to find an accurate estimate of $n(T, G)$.

B. A sequential algorithm: the color coding technique

Color coding is a randomized approximation algorithm which is able to approximately count the number of embeddings of trees of size k in time $O(c^k \text{poly}(n))$ time for a constant c—this is an example of a *fixed parameter tractable* algorithm, which is exponential in a suitable parameter (in this case, the size), but polynomial in the input size. We briefly describe the key ideas of the color coding technique here, since our algorithm involves a parallelization of it.

1. Counting colorful embeddings. The main idea is that if we assign a color $col(v) \in \{1, \ldots, k\}$ to each node v, "colorful" embeddings, namely those in which each node has a distinct color, can be counted easily in a bottom up manner. For tree of size T and root v, let $C(v, T, S)$ denote the number of colorful embeddings of it that use colors from the set S. With split the tree into two subtrees by an edge cut, we can compute $C(v, T, S)$ using the following recurrence.

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

where T_1 and T_2 denote the subtrees resulting from removing edge $(\rho(T), u)$ from T, and u is neighbour of v.

2. Random colorings. The second idea is that if the coloring is done randomly with $k = |V_T|$ colors, there is a reasonable probability that an embedding H of T is colorful with probability $\frac{k!}{k^k}$. Therefore, the expected number of colorful embeddings is $n(T, G)\frac{k!}{k^k}$. Alon et al. [2] show that this estimator has bounded variance, which can be used to estimate $n(T, G)$ efficiently. Algorithm 1 describes the sequential color coding algorithm.

Algorithm 1 The sequential color coding algorithm.

1: **Input:** Graph $G = (V, E)$, a template $T = (V_T, E_T)$, and parameters ϵ, δ
2: **Output:** A $(1 \pm \epsilon)$-approximation to $n(T, G)$ with probability at least $1 - \delta$
3: $N = O(\frac{e^k \log(1/\delta)}{\epsilon^2})$
4: **for** $j = 1$ to N **do**
5: For each $v \in V_G$, pick a color $c(v) \in S = \{1, \ldots, k\}$ uniformly at random, where $k = |V_T|$.
6: Pick a root $\rho(T)$ for T arbitrarily
7: Partition T into subtrees recursively to form $\mathcal{T}$.
8: For each $v \in V$, $T_i \in \mathcal{T}$ with root $\rho_i = \rho(T_i)$, and subset $S_i \subseteq S$, with $|S_i| = |T_i|$, we compute:

$$c(v, T_i, S_i) = \sum_u \sum c(v, T_i', S_i') \cdot c(u, T_i'', S_i''), \tag{1}$$

where T_i is partitioned into trees T_i' and T_i'' in $\mathcal{T}$.
9: Compute $C^{(j)}$, the number of colorful embeddings of T in G for the jth coloring as

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

where q denotes the number of vertices $\rho' \in V_T$ such that T is isomorphic to itself when ρ is mapped to ρ'.
10: **end for**
11: Partition the N estimates $C^{(1)}, ..., C^{(N)}$ into $t = O(\log(1/\delta))$ sets of equal size. Let Z_j be the average of set j. Output the median of $Z_1, ..., Z_t$.

C. Challenges to Scale out Color-Coding

1) Peak Memory Utilization: It's well known that color-coding has a main drawback of huge space requirement as the template size grows. The state-of-the-art FASCIA, although adopting an efficient compressed data structure, suffers from high demand of memory resource. If a graph G has n vertices and a template T contains k vertices, FASCIA implementation has an upper bound of peak memory utilization as

$$PeakMem_{G,T} = O(n(\sum_{T_i \in \mathcal{T}} C_k^{|T_i|})) \tag{3}$$

where T_i is a sub-template attached to the root $\rho(T)$. The Peak memory grows linearly with dataset size n, however, it grows exponentially, known as combinatorial explosion, with the template size k. Considering a middle-sized graph with 20 millions of vertices, a template u12-2 in Figure 1 used by [4] has a peak memory utilization of 96 GByte, which is likely beyond the single node memory capacity of commodity clusters.

To enable in-memory processing of much larger datasets and templates, it must scale out to use memory resource on other cluster nodes. Based on the vertices partitioning strategy, however, local vertices on each node still require counting information of subtrees from all the neighbour vertices located on remote nodes, and therefore curse large volume of data transfer.

MPI-Fascia, a distributed FASCIA in [4], uses *MPI_AlltoAll* or *MPI_Send/Recv* operations in a collective way. In that scenario, *MPI-Fascia* starts the computation after all of its remote data requests being satisfied, which requires it to temporarily store all the counting information from its neighbours. For datasets with billions of edges, this local copy of remote data immediately neutralizes the benefits of memory reduction by node scaling out. In fact, we find that the color coding algorithm does not need to wait for all the remote neighbour data copy before starting its local computation, which suggests an interleaving of communication by local computation and motivates our contributions.

2) Choice of Framework: Color coding algorithm has been implemented by big data frameworks, like Hadoop Mapreduce, Spark, GraphX, and HPC framework such as MPI. For instance, SAHAD [5] is built upon Hadoop and being able to run on a 300 node Cluster. FASICA [4] uses MPI and scales sub-template sizes up to 10 vertices.

We choose a new type of trending strategy named HPC-ABDS, which represents Cloud-HPC interoperable software with HPC like performance and rich functionalities of Big Data Stack. Harp-DAAL is our effort to bring HPC-ABDS into reality, where harp is a data analytics framework plugined into Hadoop ecosystem and invokes Intel's Intels Data Analytics Acceleration Library (DAAL)[1] to leverage highly optimized c/c++ kernels on Intel's Xeon and Xeon Phi architectures.

III. Method and Implementation

A. Design of Adaptive Group Communication

Adaptive_Group is motivated by group communication that is suitable for interactions between neighbor lists for graph applications. However this may require complicated routing patterns implemented with advanced group operation to accommodate for irregular graph computation, which is not implemented in standard *MPI_AlltoAll* and point-to-point MPI operations. To design a new communication mechanism, our basic idea is to break up a standard collective operation of *MPI_AlltoAll* into steps of group communications. *Adaptive_Group* shall meet the requirements as follows:

1) In each step, every process i sends data to k processes and receives data from k processes in a partial *alltoall* operation.
2) For every process, the k processes to send data and k processes to receive data are not required to be the same.
3) Each partial *alltoall* operation still benefits optimization from collective communication.
4) An internal routing algorithm ensures its effectiveness (the same result with a standard *alltoall* operation) and efficiency (no redundant data transfer).

Assuming that p processes attend the *Adaptive_Group* in N steps, and within each step, a process chooses k processes to send data and receive data. We have the time complexity, lower bound and optimal case, of *Adaptive_Group* in Equation 4

$$Time_{AG,low} = \sum_{s=0}^{N-1}((k-1)(t_s + t_w m)) \tag{4}$$

and in Equation 5

$$Time_{AdaptAlltoAll,opt} = \sum_{s=0}^{N-1}(\log_2 k t_s + m(k/2)(\log_2 k)t_w) \tag{5}$$

Accordingly, the memory footprint of buffer in *Adaptive_Group* is

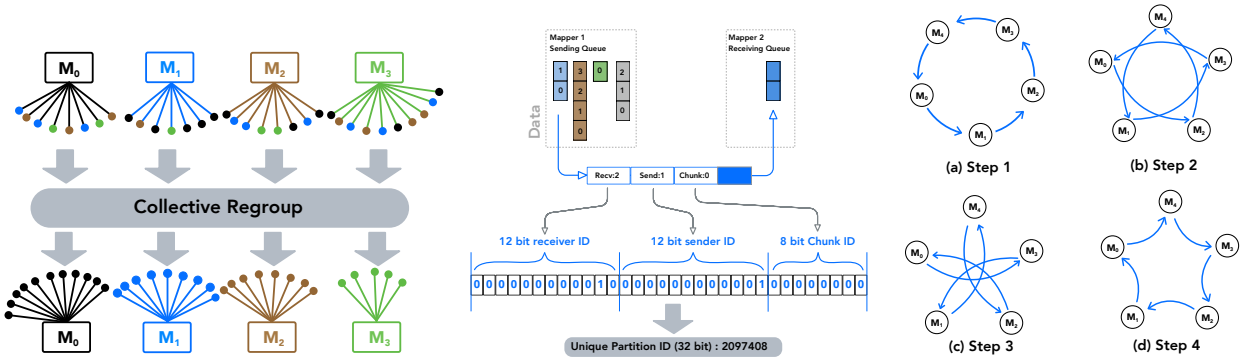
$$Mem_{AdaptAlltoAll} = Max_{s=0}^{N-1}(\sum_{i=0}^{k-1}(SendSize_i + RecvSize_i)) \tag{6}$$

What makes the group communication adaptive is the configuration of parameter k, which depends on application, hardware environment, and user demand. If the communication overhead is critical, e.g., small communication data with large number of processes, a large k value will lead to faster data transfer. Whereas, if peak memory consumption is critical, e.g., large communication data and/or limited hardware memory capacity, a small k value is usually beneficial. The characteristics of the application also affects the choice of k value. For instance, if an application is able to pipeline the work on the already received data with the rest of the communication, a small k value could still deliver a low overall execution time complexity supposing that communication is interleaved by computation.

B. Implementation wit Harp-DAAL

We choose *Harp-DAAL*, instead of MPI, as the framework to implement *Adaptive_Group* for our subgraph counting

[1]https://software.intel.com/en-us/intel-daal



(a) Harp Regroup Operation (b) *Adaptive-AlltoAll* and Message Queue (c) Pipeline Steps and Communication Order

Figure 2: (a) Harp Table partitions regrouped by their ID; (b)Each data chunk has a 32 bit meta ID, where the first 12 bits holds the receiver mapper ID, the middle 12 bits represents the sender mapper ID, and the last 8 bits represents the offset in the MQ. (c) A ring-like Send/Recv scheduling order among 5 mappers

application. *Harp*/*Harp-DAAL* provides MPI-like collective communications that are highly optimized for data-intensive applications and deliver comparable performance to MPI solutions [6] [7]. Furthermore, communication operations within *Harp*/*Harp-DAAL* have better easy-to-use interfaces than MPI operations. For instance, two MPI processes within the same *MPI_AlltoAll* operation must have a consistent data size in a pair-wised send/receive process, which requires users to communicate the information of data sizes before launching the *MPI_AlltoAll* operation. *Harp*/*Harp-DAAL* operations, however, use a Java object called Table to manage data, where users may transfer an arbitrary size of data without a prior notification of the data size to the receiver. In addition, *Harp*/*Harp-DAAL* allow users to customize their own communication operations from a variety of basic collective types, while MPI only provides customization on the data types.

Our *Adaptive_Group* is thus extended from a Harp collective operation called *regroup* illustrated in Figure 2 (a). In the regroup operation, each mapper send out its partitions and receive partitions from other mappers. The partitions are routed by their partition IDs, which are by default equal to the mapper IDs of receivers. By customizing the partition ID and the routing algorithm, we implement *Adaptive_Group* operation with the following features.

- Each mapper keeps a group of Message Queues (GMQ), with each of them labeled by the mapper ID of a receiver. A GMQ is implemented by a *Harp* Table
- Each Message Queue (MQ) contains a number of data chunks that correspondent to *Harp* table partitions.
- Each data chunk is associated with a 32 bits integer Meta ID, which is bit-wise packed from three parts: A Receiver mapper ID, A Sender mapper ID (self ID), and an offset of data chunk in its MQ.
- When the router receives a data chunk from a mapper, it routes it to the receiver by the decoded the receiver mapper ID from the chunk Meta ID.
- When the receiver obtain a chunk from the router, it also decodes the Meta ID to find the sender mapper ID and offset in MQ.

Figure 2 (b) explains the workflow of *Adaptive_Group*, and in each step s it executes the procedure as follow:

1) Selecting k receiver IDs and activate the labeled k MQs
2) Allocate memory and load data into active MQs.
3) Divide data in each MQ into chunks according to a pre-configured chunk size.
4) Generate a Meta ID for each chunk and push them to the router.
5) Router transfers data chunks to receiver mappers.
6) Receivers release data chunks to MQs labeled by sender mapper IDs, and keep them in the same offsets in the sending MQs.

By using aforementioned Meta ID and bit-wise encoding mechanism, we are able to map the three-level (GMQ, MQ, Data Chunk) data structure of *Adaptive_Group* to the two-level (Table, Partition) data structure of *Harp* operation, where MQ is abstracted as the set of data chunks with the same receiver or sender mapper IDs. From the procedure, we see that our implementation of *Adaptive_Group* within *Harp*/*Harp-DAAL* already satisfies the first and second requirements in Section III-A. Because regroup operation of *Harp* is a collective operation, thus our extension to *Adaptive_Group* inherits the benefits of collective communication. Even if there is only one MQ ($k = 1$) in a mapper, it still has multiple data chunks, which makes it a collective operation and meets the third requirement. To meet the last requirement for the effectiveness and efficiency of the scheduling algorithm, we provide a ring-like scheduling policy in Figure 2 (c), where we illustrate the case of $k = 1$.

This scheduling algorithm assumes that each mapper has the same communication distances to each other at the hardware level.

In practice, people can conceive much complicated scheduling policies according to the hardware specifications and application features.

C. Interleaving Computation in a Pipeline

A major motivation of using *Adaptive_Group* in color-coding is to create the pipeline for interleaving computation and communication and reducing peak memory usage. We implement a pipeline by using multi-threading programming model. The pipeline has two slots, each of them attached to a group of threads respectively. For communication slot, it uses a single thread; For computation slot, a multi-threaded library like OpenMP registers the rest of the physical threads resources. The pipeline has a cold start, where computation slot is waiting for data from communication slot. There is a cross-slot threads synchronization after each counting each sub-template to ensure the consistency of local data. The pipeline process is launched after the first step of *Adaptive_Group*, and each mapper computes the received data from last step while keeping in transferring data at the current step in a non-blocking way. Algorithm 2 illustrates the workflow of *Adaptive_Group* pipeline, where one group of threads are dedicated to communication task, the rest of threads are executing computation tasks concurrently. Except for the first step, the $threads\_compute$ will compute on the data received from the previous step by $threads\_comm$, and simultaneously, the $threads\_comm$ continues to transfer data.

Algorithm 2 *Adaptive_Group* pipeline workflow at Mapper i

```
1:  procedure RINGALLTOALL(p mappers, V_i vertices on i)
2:      create threads_comm, threads_compute
3:      initialize commOrder(i)
4:      objComm = commOrder(i).next()
5:      threads_comm.start(objComm)
6:      threads_comm.join()
7:      for j = 0; j < p − 1; j++ do
8:          objCompute = objComm
9:          objComm = commOrder(i).next()
10:         if is_threads_comm then
11:             threads_comm.start(objComm)
12:         else
13:             threads_compute.start(objCompute)
14:         end if
15:         threads_comm.join(), threads_compute.join()
16:     end for
17:     objCompute = objComm
18:     threads_compute.start(objCompute)
19:     threads_compute.join()
20: end procedure
```

D. Fine-grained Load Balance via Partitioning the Neighbour List

The efficiency of pipeline may be affected by the load imbalance among different mappers, which is due to the skewness of out degree distribution of the input graph. For graph data with large skewness, some vertices have several orders of magnitude more adjacent vertices in their neighbour list than by average. A standard multi-threaded counting kernel for local computation takes a vertex and the counting over all entries of its neighbour list as a single task. If the sub-template requires a substantial computation per vertex, this difference in neighbour list length is amplified significantly and curses a sever thread-level workload imbalance. Furthermore, the poor thread concurrency due to these "straggler" tasks finally cause certain mappers running much slower than the others and lead to node-level imbalance.

A traditional way to alleviate load imbalance is to pre-process the graph data in the partition stage for mappers. By regrouping vertices according to their total neighbour list lengths, each mapper shall get a comparable workload. However, this method could not resolve the intra-node thread-level load imbalance, because that there are still threads assigned by "straggler" tasks, which take much long time than the other threads.

We provide another fast and lightweight method, see Algorithm 3, to address this problem by partitioning the work on a vertex with long neighbour lists into multiple tasks instead of a single one. After choosing a task size t_{len}, we break up any neighbour list if its length is larger than t_{len}. In this way, it is guaranteed that each task has a neighbour list length no larger than t_{len}. Also, we can adjust the number of tasks by increasing or decreasing t_{len}, which is the granularity of a single task. This fine-grained tasks creation process is also adaptive to different hardware architectures.

IV. Experimentation and Results

A. Experimental Setup

We used 25 nodes of an Intel Xeon cluster. The node specification is described in Table I. In testing *MPI-Fascia*, all of the 120 GB Memory per node is used by the MPI process. However, in testing Harp-Subgraph and Harp-DAAL-Subgraph, we need to reserve 10 GB of memory for Hadoop daemons and JVM, which reduces the effective memory to 110 GB for a Harp mapper. The Java codes is compiled by JDK 8.0. The C/C++ codes of DAAL is compiled by Intel Compiler 2016 while the MPI/C++ codes of *MPI-Fascia* is compiled by OpenMPI 1.8.1. The *MPI-Fascia* directly uses InfiniBand setups via the runtime option *–mca btl openib,self*, however, Harp-Subgraph and Harp-DAAL-Subgraph requires TCP/IP emulation layers like IP over

Algorithm 3 Neighbour List Partitioning for Node i

```
1: procedure TASK CREATION(Task size s, vertices V_i)
2:     for all j ∈ V_i do
3:         if nbr(j).size < s then
4:             taskQueue.add(new Task(j, nbr(j)))
5:         else
6:             taskRemain = nbr(j).size
7:             nbrPos = 0
8:             while taskRemain > 0 do
9:                 newTaskLen = (taskRemain > s) ? s : taskRemain
10:                taskQueue.add(new Task(j, nbr(j, nbrPos, nbrPos+newTaskLen)))
11:                nbrPos += newTaskLen
12:                taskRemain -= newTaskLen
13:            end while
14:        end if
15:    end for
16:    shuffle(taskQueue)
17: end procedure
```

Table I: Specification of Xeon E5-2670v3 Node

Cores Specs		Uncore Specs	
Cores	12	DDR4	120 GB
Base Freq	2.3GHz	DDR4-Band	90 Gbps
L1 Cache	64 KB	Network	InfiniBand
L2 Cache	256 KB	Peak Port Band	100 Gbps
L3 Cache	30720 KB	Socket	2
Instruction Set	64 bit	Disk	1 TB
IS Extension	AVX256		
Max Threads	48		

InfiniBand (IPoIB) to use InfiniBand, which compromises communication speed in some extent.

B. Graph Datasets and Tree-like Templates

We used both of datasets collected from online repositories and synthetic generated datasets. In Table II, Miami and Orkut are social contact network in [8] [9] [10]; Twitter [11], sk-2005 [12], and friendster [9] are social network with billions of edges. Since color-coding works on undirected graph, we converted directed graph datasets to undirected, and the edge number in Table II is the number after conversion. Synthetic datasets are generated by PaRMAT [13], which uses RMAT recursive model [14] to create graphs with user-specified vertex number and edge number. Furthermore, the skewness of out-degree distribution is also configurable. A typical social network graph has a skewness around 3, but we also use graphs with skewness 1, 8, 10.

We use large tree-like templates in Figure 1 (b) for experiments. u10-2 and u12-2 are collected from [4], while u13 to u15 are beyond any previous work

Table II: Datasets in Experiment

Data	Vertex	Edges	Deg Avg	Deg Max	Source
Miami	2.1M	51M	49	9868	synthetic social network
Orkut	3M	230M	76	33K	social network
NYC	18M	480M	54	10K	synthetic social network
Twitter	44M	2B	37	750K	Twitter users
sk-2005	50M	3.8B	73	15M	UbiCrawler
Friendster	66M	5B	57	92K	social network

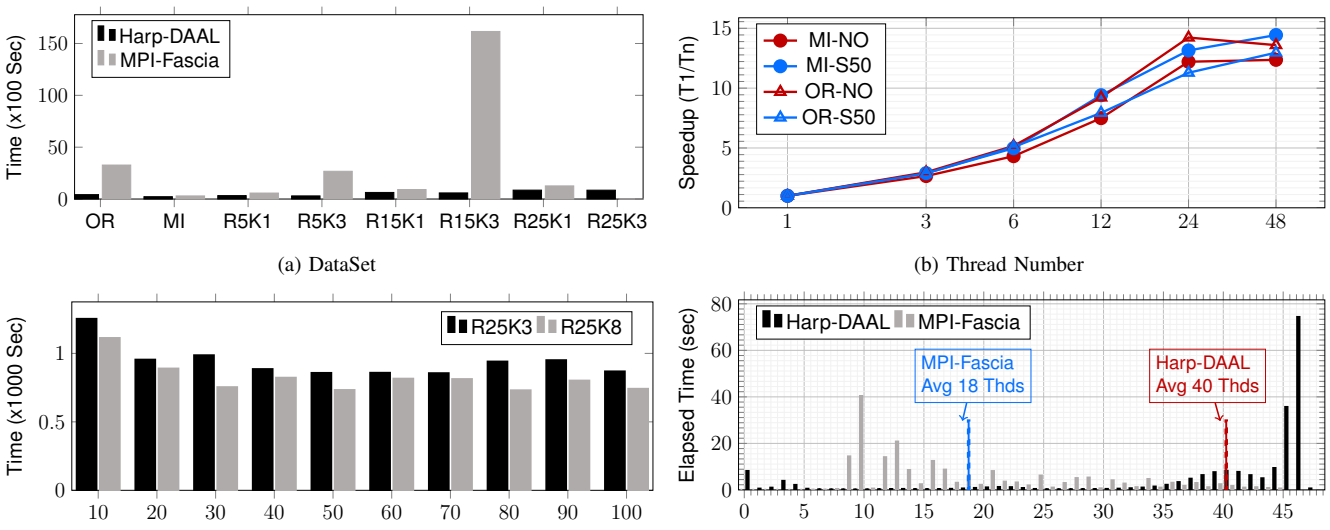
C. Performance on Single Node

We examine our neighbour list partitioning work in Section III-D by only running small datasets on a single Xeon E5-2670v3 node due to the limited memory capacity. The measured time is the time spent in effective counting work for a single iteration, which excludes the data loading time for both of *Harp-DAAL-Subgraph* and *MPI-Fascia*. In Figure 3 (a), the time in Miami (MI) is comparable between *Harp-DAAL-Subgraph* and *MPI-Fascia*, while *Harp-DAAL-Subgraph* is almost 8 times faster than *MPI-Fascia* in OR which has near 4 times more edges than MI, and 2 times higher skewness from Table II, which is likely to contain more long neighbour lists that is benefited from neighbour list partitioning. It is clear with RMAT data on skewness 1 and 3, where *Harp-DAAL-Subgraph* achieves 27x speedups on R15K3.

1) Thread Scaling: In Figure 3 (b), we examine the effect of neighbour list partitioning on the thread-level scaling. From 1 thread to 24 threads, the benefits of neighbour list partitioning is not significant; After 24 threads, when Xeon E5-2670v processor enabling hyper threading, the scalability of neighbour list partitioning is still growing while the other one drops. Hyper threading generally weakens single thread performance, which makes it more vulnerable to skewness in neighbour list workload and benefits from partitioning.

2) Variation of Task Size: Finally, we study the performance variation on partitioning task sizes in Figure 3 (c). With two skewness values 3 and 8, we observe that except for small task size 10, the other task sizes give a comparable performance, implying that the neighbour list partitioning strategies works well for wide range of task sizes. In the rest of the tests, we set 50 as the default task sizes.

3) Thread Concurrency: To demonstrate the effectiveness of our intra-node optimization at the hardware level, we explore the thread concurrency from using Intel VTune amplifier. In Figure 3 (d), we compare the statistics of concurrent running threads number from *Harp-DAAL-Subgraph* and *MPI-Fascia* on RMAT data with a high skewness of 8. The x-axis represents the different number of simultaneously running threads, the larger the better. The y-axis records the time spent in a certain concurrent running threads number. Given the total VTune profiling of 200 seconds, the major



(a) DataSet

(b) Thread Number

(c) Task Size in Neighbour List Partitioning

(d) Concurrent Running Threads

Figure 3: Single Node test on template u12-2, use neighbour list size 50 by default (a) Counting time on Miami, Orkut and synthetic data RMAT 5 million vertices with skewness 3, 8; (b) Speedup is defined to be T(1)/T(n), where n is the number of threads (c) Variation of performance on different neighbour list partitioning task size; (d) Thread concurrency test on RMAT 250 million edges with skewness 8 measured by Intel VTune

time of *Harp-DAAL-Subgraph* is more shifted toward high number of concurrent threads than that of *MPI-Fascia*. We also calculate the average concurrent thread usage, and *Harp-DAAL-Subgraph* outperforms *MPI-Fascia* by 40 to 18. This result is consistent with Figure 3 (a)

D. Peak Memory Utilization

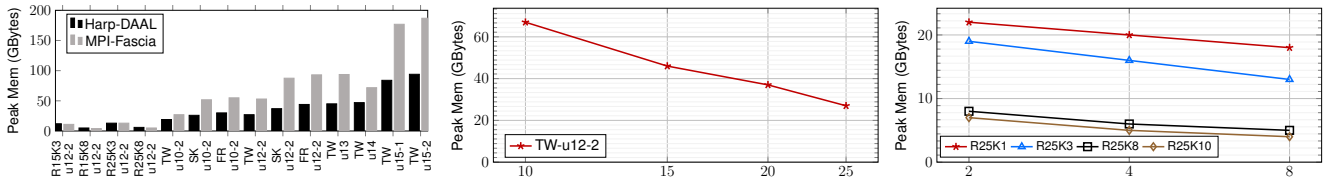
In Figure 4 (a), we examine the peak memory utilization by periodically querying the used physical memory flushed within system file "/proc/self/status". We use 8 nodes to test small datasets group from R12K3 to R25K8 and 25 nodes to test large datasets from Twitter to Friendster. When dataset size grows, e.g., from R25K3 to Twitter on template u12-2, *Harp-DAAL-Subgraph* has twice larger peak memory and *MPI-Fascia* has 4 times larger peak memory. When template size grows from u10-2 to u15-2 for the same Twitter datasets, peak memory on *Harp-DAAL-Subgraph* and *MPI-Fascia* both grows 3.5 times. Compared to *MPI-Fascia*, *Harp-DAAL-Subgraph* requires around half of the peak memory utilization in the cases of large datasets and templates on the same number of nodes.

In addition, we explore reduction of peak memory utilization on single node in distributed mode. In Figure 4 (b), we check *Harp-DAAL-Subgraph* with Twitter and template u12-2 from 10 nodes to 25 nodes, and it effectively reduces peak memory from 67 GBytes to 27 GBytes by 2.4 times, which gives almost a linear reduction by number of nodes. In contrast, the tests on *MPI-Fascia* proves a poor memory reduction in Figure 4 (c), which reflects the lack of pipeline design to save memory space in remote data copies analyzed in Section II-C1.

E. Communication Efficiency

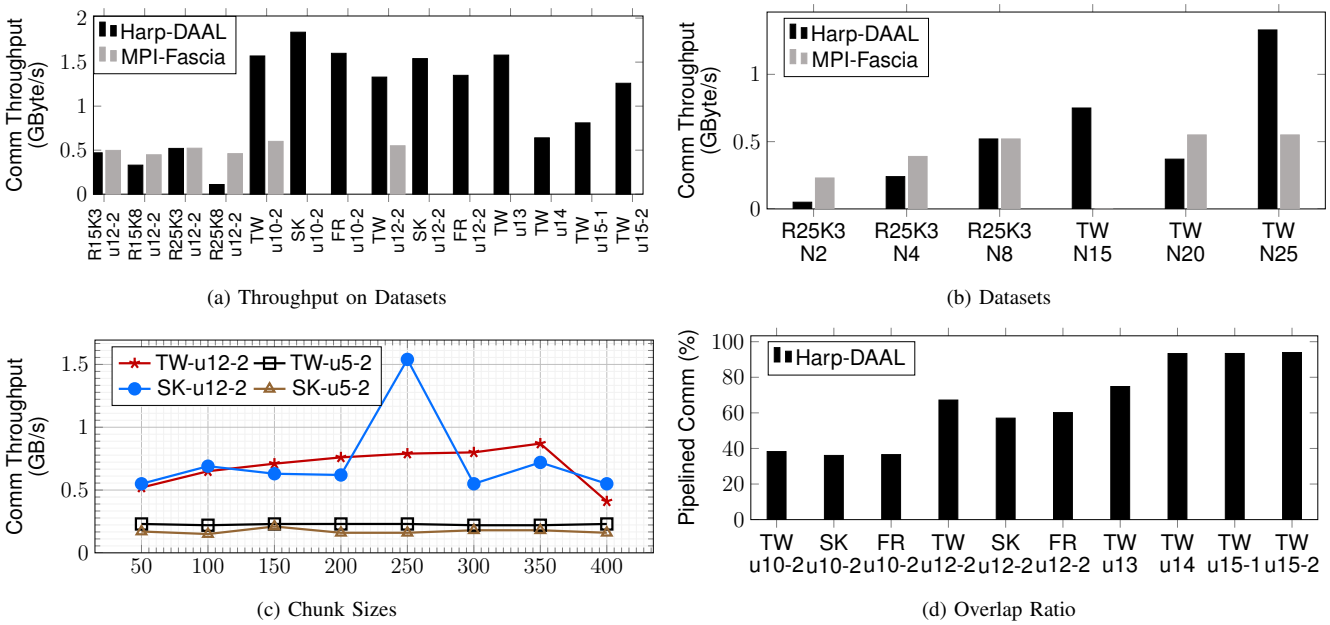
We investigate the communication from two aspects: 1) Data throughput is defined as the division of total transferred data by total communication time. Here, the communication also includes data compression and decompression of subgraph counts ahead of transferring, therefore it reflects the overall communication performance at application level instead of benchmarking like throughput on naive MPI operations. 2) Effectiveness of pipeline interleaving is the percentage of interleaved communication and waiting time in the total synchronization overhead.

1) Communication Data Throughput: In Figure 5 (a), we first check the throughput at RMAT datasets by using 8 nodes, where *MPI-Fascia* achieves comparable or even better throughput at RMAT of skewness 8. However, for large dataset Twitter and 25 nodes, *Harp-DAAL-Subgraph* achieves around 3 times higher throughput than *MPI-Fascia*, it also keeps this high throughput for even larger SK-2005 and Friendster. In Figure 5 (b), we notice that throughput increases with number of nodes at both of small datasets RMAT and large datasets Twitter. However, there is still some fluctuation, e.g., Twitter on 15 nodes has a higher throughput than on 20 nodes, which implies that *Adaptive_Group* may be affected by other factors than number of nodes. In Figure **??** (b), we focus on the chunk size defined in Section III-B. With small template u5-2, the throughput is insensitive to varied chunk size; With large template sizes, which brings in high communication data



(a) Peak Memory Usage (b) Memory Reduction,*Harp-DAAL-Subgraph* (c) Memory Reduction,*MPI-Fascia*

Figure 4: Tests on Peak memory utilization; (a) On dataset RMAT, Twitter, Sk-2005 and Friendster, templates from u10-2 to u15-2 (b) Measure the peak memory utilization along with increasing number of mappers in *Harp-DAAL-Subgraph* (c) Measure the peak memory utilization along with increasing number of processes in *MPI-Fascia*



(a) Throughput on Datasets (b) Datasets

(c) Chunk Sizes (d) Overlap Ratio

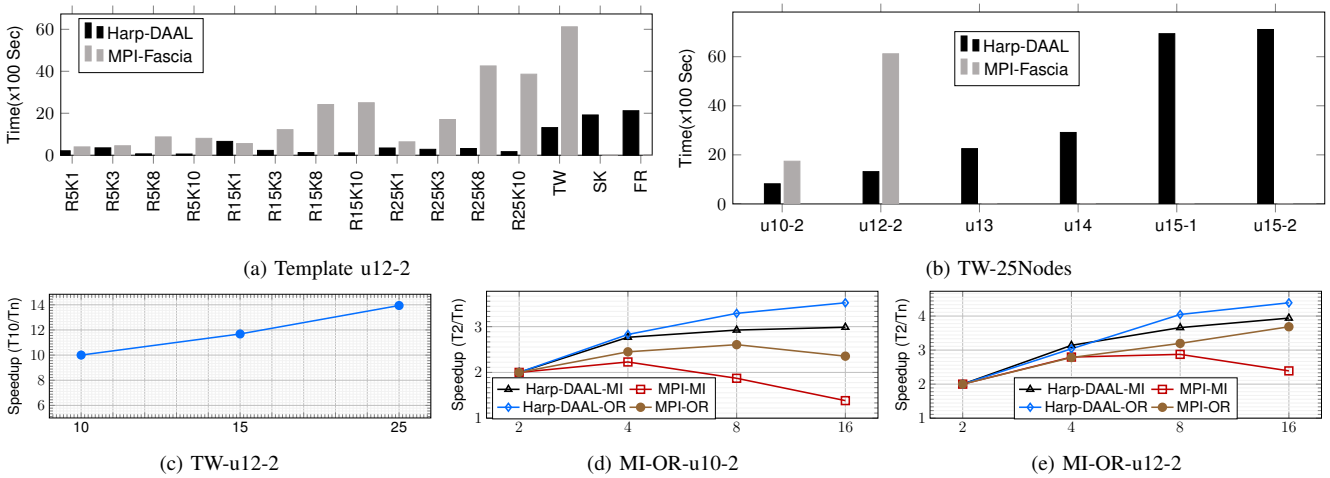
Figure 5: Communication Efficiency measured by communication throughput and ratio of pipelined overhead. (a) Throughput on RMAT datasets by 8 nodes, and Twitter, Sk-2005, Friendster by 25 nodes (b) Throughput variation with node number, tested on u12-2 and R25K3 from 2 to 8 nodes, Twitter from 15 to 25 nodes (c) Throughput variation with chunk sizes of Message Queue in *Adaptive_Group*, tested by 25 nodes on Twitter and Sk-2005 with templates u5-2 and u10-2; (d) Ratio (Percentage) of overlapped communication and waiting time from the whole synchronization overhead, tested by 25 nodes on Twitter, Sk-2005, Friendster with templates u10-2 to u15-2

volume, *Adaptive_Group* favors chunk sizes within a range. For instance, Sk-2005 datasets and u12-2 achieves 3x higher throughput at chunk size 250 than the other sizes. In general, a too small chunk size will increase the times of data copy between DAAL and harp, the compression and decompression time, and the communication latency overhead in *Adaptive_Group*. It is not optimal to use large chunk sizes since they will generate less number of chunks and make it hard to do fine-grained optimization for harp's collective operations.

2) Effectiveness of Pipeline Interleaving: The pipeline design does reduce peak memory utilization, however, it will cause more communication time if not effectively overlapped by computation. In Figure **??** (c), we examine the ratio of interleaved communication and waiting time caused by load imbalance. For template u10-2, all of the three datasets show a overlapping ratio less than 40%, which is due to the low computation intensity that provides insufficient computation time in pipeline to cover the communication overhead. With template u12-2, the overlapping ratio increases to more than 50%, and it achieves more than 90% with u13 to u15. Thus, for large templates, the pipeline design hides most of the communication and waiting time by computation, which benefits both of time and scalability.

F. Scaling for Large Problems

1) Increase Dataset and Template Size: From single node performance to communication efficiency, all the results imply a good time performance of *Harp-DAAL-Subgraph* with large datasets and templates in distributed systems compared



(a) Template u12-2

(b) TW-25Nodes

(c) TW-u12-2

(d) MI-OR-u10-2

(e) MI-OR-u12-2

Figure 6: Scaling tests: (a) The same dataset Twitter with increasing template sizes; (b) Same Twitter and template u12-2 but growing number of nodes; (c) The node-level scaling for Datasets Miami and Orkut with template u10-2; (d) The node-level scaling for Datasets Miami and Orkut with template u12-2

to *MPI-Fascia*. Figure **??** presents the time performance for small and large scale problems.

For small-scale tests on 8 nodes, *Harp-DAAL-Subgraph* has a comparable performance with *MPI-Fascia* on datasets with a small skewness of 1, however, with normal skewness 3 (real social network) and high skewness of 8, 10, the performance gap attains at least 5 times (R15K3) and as high as 22 times (R25K10). For large-scale tests on 25 nodes, we find a consistent performance gap of 5 times at Twitter, which is a social network with skewness of 3.

To confirm the time performance with different template sizes, we compare the results on Twitter on 25 nodes but with templates from u10-2 to u15. Because *MPI-Fascia* could not run Twitter beyond u12-2, we can only compare its performance on u10-2 and u12-2. Figure 6 (a) shows that the performance gap increases from 2x (u10-2) to 5x (u12-2).

2) Node-Level Scalability: In Figure 6 (b), we measure the strong scalability of *Harp-DAAL-Subgraph* on Twitter and template u12-2. From 10 nodes to 15 nodes, it attains a speedup of 12, which is close to the linear value (15). From 15 nodes to 25 nodes, the speedup still increase to 14 but much lower than the linear value of 25. This is due to the insufficient computation workload because the local computation time on 20 nodes is 636 seconds and on 25 nodes is 623 seconds. In Figure 6 (c) and (d), we compare strong scaling at small-scale. *Harp-DAAL-Subgraph* keeps a increasing scalability while that of *MPI-Fascia* drops after a certain number of nodes, where the communication overhead becomes dominant.

V. Related Work

ParSE [15] was the first distributed algorithm based on color coding that scale to graphs with millions of vertices, for template of size up to 10, within a few hours. It works on tree-like template that can be partitioned by a cut edge. SAHAD [5] further expand this algorithm up to 12 vertices labeled template on graph with 9 million vertices within less than an hour by using hadoop-based implementation. FASCIA [16], [17], [4] is the current state-of-the-art color coding treelet counting tool. By highly optimized data structure and MPI+OpenMP implementation, it supports tree-like template of size up to 10 vertices in billion-edge networks in a few minutes. Recent work [18] also explores the direction of more complex template with treewitdth 2, its solution scales up to 10 vertices for graphs of up to $2M$ vertices. Our work provides a color coding solution for treelet counting to a much larger scale, with template of 15 vertices and graph of 5 billion edges.

VI. Conclusion

Subgraph counting is a challenging computation problem, both computationally intensive and memory intensive. We proposed a pipelined adaptive-group communication for finding and counting large tree-based subgraph templates. We show that the new approach can scale up to 12 or 15 vertices of templates in Twitter input graphs of half a billion vertices and 2 billion edges. The experiments ran on a cluster of 25 nodes Intel Xeon (Haswell 24 core) architectures and achieved 5x speedups over state-of-the-art MPI solution in related work.

The pipelined *Adaptive_Group* is a novel collective communication technique that effectively reduces memory utilization, reduces load imbalance of sparse graphs, and hides

communication by overlapping with computation, thereby can scale to large datasets and templates. We have demonstrated its computing capability and run big data *Harp-DAAL-Subgraph* applications with 12 nodes of templates in massive input Friendster graph of 0.66 billion vertices and 5 billion edges, which is the largest graph size in related work. In future work, we can apply Harp *Adaptive_Group* to other data intensive sparse graph applications such as random subgraphs for scalable solutions to the computational and spacial challenges.

ACKNOWLEDGMENTS

We gratefully acknowledge generous support from the Intel Parallel Computing Center (IPCC) grant, NSF OCI-114932 (Career: Programming Environments and Runtime for Data Enabled Science), CIF-DIBBS 143054: Middleware and High Performance Analytics Libraries for Scalable Data Science. We appreciate the support from IU PHI, FutureSystems team and ISE Modelling and Simulation Lab.

REFERENCES

[1] V. Vassilevska and R. Williams, "Finding, minimizing, and counting weighted subgraphs," in *Proceedings of the 41st annual ACM symposium on Theory of computing*. ACM, 2009, pp. 455–464.

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

[3] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi, "Counting Graphlets: Space vs Time," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, 2017, pp. 557–566. [Online]. Available: http://dl.acm.org/citation.cfm?id=3018732

[4] G. M. Slota and K. Madduri, "Parallel color-coding," *Parallel Computing*, vol. 47, pp. 51–69, 2015.

[5] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. A. Kumar, and M. V. Marathe, "Sahad: Subgraph analysis in massive networks using hadoop," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, pp. 390–401. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6267876/

[6] B. Zhang, Y. Ruan, and J. Qiu, "Harp: Collective communication on Hadoop," *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, pp. 228–233, 2015.

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

[8] C. L. Barrett, R. J. Beckman, M. Khan, V. S. A. Kumar, M. V. Marathe, P. E. Stretz, T. Dutta, and B. Lewis, "Generation and analysis of large synthetic social contact networks," in *Proceedings of the 2009 Winter Simulation Conference (WSC)*, Dec. 2009, pp. 1003–1014.

[9] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[10] J. Yang and J. Leskovec, "Defining and Evaluating Network Communities Based on Ground-Truth," in *2012 IEEE 12th International Conference on Data Mining*, Dec. 2012, pp. 745–754.

[11] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring User Influence in Twitter: The Million Follower Fallacy." in *AAAI Conference on Weblogs and Social Media*, vol. 14, Jun. 2010.

[12] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[13] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-Efficient Graph Processing on GPUs," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct. 2015, pp. 39–50.

[14] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SIAM Proceedings Series*, vol. 6, Apr. 2004.

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

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

[17] ——, "Complex network analysis using parallel approximate motif counting," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 405–414.

[18] V. T. Chakaravarthy, M. Kapralov, P. Murali, F. Petrini, X. Que, Y. Sabharwal, and B. Schieber, "Subgraph Counting: Color Coding Beyond Trees," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 2–11.