High-Performance Massive Subgraph Counting using Pipelined Adaptive-Group Communication

Langshi Chen
School of Informatics, Computing and Engineering, Indiana University
lc37@indiana.edu

Bo Peng
School of Informatics, Computing and Engineering, Indiana University
pengb@indiana.edu

Sabra Ossen
School of Informatics, Computing and Engineering, Indiana University
sossen@iu.edu

Anil Vullikanti
Virginia Tech
vsakumar@vt.edu

Madhav Marathe
Virginia Tech
mmarathe@vt.edu

Lei Jiang
School of Informatics, Computing and Engineering, Indiana University
jiang60@indiana.edu

Judy Qiu
School of Informatics, Computing and Engineering, Indiana University
xqiu@indiana.edu

ABSTRACT

Subgraph counting involves comparing a subgraph template across a large input graph. Many domains have large networks (the Internet of Things, social and biological networks) that can benefit from fast subgraph isomorphism for billion- or trillion- edge graphs. However, it is a NP-hard problem that is computationally challenging. The time complexity and memory space grow exponentially with the increase in template size. In this paper, we investigate parallel efficiency and memory reduction strategies and propose a novel pipelined adaptive-group communication for massive subgraph counting problems. In contrast to MPI point-to-point solutions, we leverage fine-grained parallelism and communication optimization and develop a high-level communication abstraction that is suitable for irregular graph interactions. The proposed method also includes 1) an interleaved computation and group communication, 2) partitioning neighbor list of subgraph for better in memory thread concurrency and load balance, and 3) a fine-grained pipeline communication with regroup operation to significantly reduce memory footprint. Experimental results on an Intel Xeon E5 cluster show that our Harp-DAAL implementation of subgraph counting based on color-coding algorithm achieves 5x speedup compared to the current state-of-the-art work and reduces peak memory utilization by a factor of 2 on large templates of 12 to 15 vertices and input graphs of 2 to 5 billions of edges.

KEYWORDS

Subgraph Counting, Big Data, HPC, Communication Pattern

HPDC'18, June 2018, Tempe, Arizona, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Subgraph analysis in massive graphs is a fundamental task in numerous applications including analyzing the connectivity of social networks [9], uncovering network motifs (repetitive subgraphs) in gene regulatory networks in bioinformatics [19], indexing graph databases [16], optimizing task scheduling in infrastructure monitoring and detecting events in cybersecurity. Although these advanced graph analytics may provide a deep insight into the network's functional abilities, they require computing power to analyze the billion- and trillion-edge graphs generated by the Internet of Things, ever-expanding social networks, biological, and future sensor networks.

Given two graphs T on k vertices and G on n vertices as input, the broad questions of subgraph analysis include determining whether $G = (V, E)$ contains a subgraph that is isomorphic to template T, counting the number of such subgraphs, and finding the most significant ones. In this paper, we focus on non-induced subgraph isomorphism problem for which a formal definition is given in Section 2. Problems such as subgraph counting and finding the most significant ones generalize subgraph isomorphism, which is NP-hard even for very simple templates. Even the best algorithms for exact counting run in $\Omega(n^{k/2})$ time complexity which is exponential to the tree template size k [24]. This motivates the use of approximation algorithms.

Color-coding [2] is such an approximation algorithm to solve the subgraph isomorphism problem and gives a *fixed parameter tractable* algorithm with execution time exponential to template size k but ploynomial to vertices number n on tree-like template. Sequential fixed parametric algorithms other than color-coding to detect subgraphs can be found in [10, 12]. Tree template counting can also be used as a kernel to estimate the Graphlet Frequency Distribution (GFD), which is another widely accepted tool to estimate the relative frequency among all subgraphs of the same size. [4] shows that a well-implemented tree template counting kernel can push the limit of the state-of-the-art of GFD, both in terms of the size of the input graph and the template.

The current parallel algorithms with rigorous guarantees for counting trees involve parallelization of the color-coding technique, either implemented with MapReduce (SAHAD [29]) or with MPI (FASCIA [23]). However, both methods suffer from significant communication overhead and large memory footprints, which prevents them from scaling to templates with more than 12 vertices.

In this paper, we focus on tree template(treelet) counting problem and identify the bottlenecks of scaling, and design a new approach for parallelizing color-coding. Our main contributions in this paper aim to address the following computation challenges:

- **Communication:** Many graph applications are based on point-to-point communication. Thus the unavailability of high-level communication abstraction that is adaptive for irregular graph interactions.
- **Load Balance:** Sparsity of graph generates load imbalance for computation.
- **Memory:** High volume of intermediate data owning to large subgraph template (big model), which generates peak memory utilization at runtime.

We investigate computing capabilities to run subgraph counting at a very large scale, and we propose the following solutions:

- **Adaptive-group communication** with regroup operation developed to accelerate communication.
- **Partitioning neighbor list** for fine-grained task granularity and load balance in concurrent threading of a single node.
- **Model partition** with pipelined communication and data compression technique to reduce memory footprint.

We compare our results with the state-of-the-art MPI Fascia implementation [23] and show applicability of the proposed method, which can run large treelets (up to 15 vertices) and massive graphs (up to 5 billion edges and 0.66 billion vertices) for subgraph counting problems.

The rest of the paper is organized as follows. Section 2 introduces the problem, color-coding algorithm and scaling challenges. Section 3 presents our approach on Adaptive-Group communication as well as a fine-grained intra-node thread optimization. Section 4 contains experimental analysis of our proposed methods and show performance improvements. After Section 5 on related works, we conclude in Section 6.

2 BACKGROUND OF COLOR-CODING

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 $V_H \subseteq V$ and $E_H \subseteq E$. We note that there may be other edges in $E - E_H$ 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 $f : V_T \rightarrow V_H$ 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.

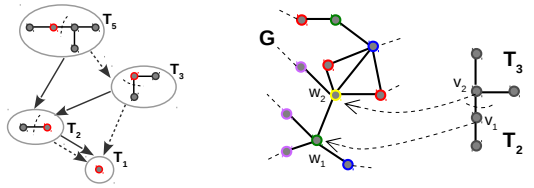
Color-coding is a randomized approximation algorithm, which estimates the number of tree-like embeddings in $O(c^k \text{poly}(n))$ with a tree size k and a constant c. We briefly describe the key ideas of the color-coding technique here, since our algorithm involves a parallelization of it.

Counting colorful embeddings. The main idea is that if we assign a color $col(v) \in \{1, \ldots, k\}$ to each node $v \in G$, "colorful" embeddings, namely those in which each node has a distinct color, can be counted easily in a bottom-up manner.

For a tree template $T = (V_T, E_T)$, let $\rho(T)$ denote its root, which can be picked arbitrarily. Then $T(v)$ denote a template T with root $v = \rho(T)$. Let T' and T'' denote the subtrees by cutting edge $(\rho(T), u)$ from T. We pick $\rho(T') = \rho(T)$ and $\rho(T'') = u$. Let $C(v, T, S)$ denote the number of colorful embeddings of T with vertex $v \in V_G$ mapped to the root $\rho(T)$, and using the color set S, where $|V_T| = |S|$. Then, we can compute $C(v, T, S)$ using dynamic programming with the following recurrence.

$$C(v, T, S) = \sum_{u \in N(v)} \sum_{S = S_1 \cup S_2} C(v, T', S_1) \cdot C(u, T'', S_2) \qquad (1)$$

Figure 1 (a) shows how the problem is decomposed into smaller sub-problems. In this partition process, one vertex arbitrarily is picked up as the root which is marked in red, then one edge of it is removed, splitting tree T into two small sub-trees. The arrow lines denote these split relationships, with the solid line pointing to the sub-tree with the root vertex and dotted line to the other. This process runs recursively until the tree template has only one vertex, T_1. Figure 1 (b) shows an example of the colorful embedding counting process which demonstrates the calculation on one neighbour of the root vertex. Here, tree template T_5 is split into sub templates T_2 and T_3, in order to count $C(w_1, T_5(v_1), S)$, or the number of embeddings of $T_5(v_1)$ rooted at w_1, using color set $S = \{red, yellow, blue, green, purple\}$, we enumerate over all valid combination of sub color sets on T_2 and T_3. For $S_1 = \{g, p\}, S_2 = \{y, r, b\}$, we have $C(w_1, T_2(v_1), \{g, p\}) = 2$ and $C(w_2, T_3(v_2), \{y, r, b\}) = 2$, and for $S_1 = \{g, b\}, S_2 = \{y, r, p\}$, we have $C(w_1, T_2(v_1), \{g, b\}) = 1$, $C(w_2, T_3(v_2), \{y, r, p\}) = 2$. As T_5 can be constructed by combinations of these sub trees, $C(w_1, T_5(v_1), S)$ equals to the summation of the multiplication of the count of the sub trees, and results $2 \times 2 + 1 \times 2 = 6$. In this example, the combination of two subtrees of T_5 uniquely locates a colorful embedding. But for some templates, some subtrees are isomorphic to each other when the root is removed. E.g., for T_3 in 1(a), the same embedding will be over counted for 2 times in this dynamic programming process.



(a) Template Partition (b) Colorful Embedding Counting

Figure 1: An example showing the two main steps of color-coding with tempate T_5.

Random colorings. The second idea is that if the coloring is done randomly with $k = |V_T|$ colors, there will be a reasonable probability that an embedding is colorful, i.e., each of its nodes is marked by a distinct color. Specifically, 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 the number of embeddings, denoted as $n(T,G)$, 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 of at least $1 - \delta$
3: $N = O(\frac{e^k \log(1/\delta)}{\varepsilon^2})$, δ and ε are parameters that control approximation quality.
4: **for** $j = 1$ to N **do**
5: For each $v \in V$, 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 $v \in V$, $T_i \in \mathcal{T}$ and subset $S_i \subseteq S$, with $|S_i| = |T_i|$, we compute:

$$c(v, T_i, S_i) = \frac{1}{d} \sum_u \sum c(v, T_i', S_i') \cdot c(u, T_i'', S_i''), \tag{2}$$

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

$$C^{(j)} = \frac{1}{q} \frac{k^k}{k!} \sum_{v \in V} c(v, T(\rho), S), \tag{3}$$

where q denotes the number of vertices that cause T to be isomorphic to itself when ρ is mapped to ρ' ($\rho = \rho(T)$ and $\rho' \in V_T$).
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 $Z_1, ..., Z_t$.

Distributed Color-Coding and Challenges. As color-coding runs N independent estimates in the outer loop at line 4 in the sequential Algorithm 1, it's straightforward to implement the outer loop at line 4 in a parallel way. However, if a large dataset cannot fit into the memory of a single node, the algorithm must partition the dataset over multiple nodes and parallelize the inner loop at line 8 of Algorithm 1 to exploit computation horsepower from more cluser nodes. Nevertheless, vertices partitioned on each local node requires count information of their neighbor u located on remote cluster nodes, which brings communication overhead that compromises scaling efficiency. Algorithm 2 uses a collective AlltoAll operation to communicate count information among processes and updates the counts of local vertices at line 12. This standard communication pattern ignores the impact of growing template size, which exponentially increases communication cost and reduces the parallel scaling efficiency. Moreover, skewed distribution of neighbor vertices on local cluster nodes will generally cause workload imbalance among processes and produce a "straggler" to slow down the collective communication operation. Finally, it requires a local node to hold all the transferred count information in memory before starting the computation stage on the remote data, resulting in high peak memory utilization on a single cluster node and become a bottleneck in scaling out the distributed color-coding algorithm.

Algorithm 2 Distributed Color-Coding Algorithm

1: **procedure** Distributed Color-Coding($G(V,E)$, T, P)
2: Input Graph $G(V,E)$ is randomly partitioned into P processes
Input Tree Template T is partitioned into subtemplates $T_i \in \mathcal{T}$
ρ is the root of T
3: **for** it=1 to Niter **do** ▷ Out-loop iterations
4: **for** Each process p **do** ▷ Process-level parallelism
5: Color local graph $G_p(V,E)$
6: **for all** $T_i \in \mathcal{T}$ in reverse order of partitioning **do**
7: **for all** $v \in G_p(V,E)$ **do** ▷ Thread-level parallelism
8: Compute $c_p(v, T_i, S_i)$ from neighbour vertices of v within process p
9: **end for**
10: Process p All-to-All exchanges local counts $c_p(, T_i, S_i)$ with other processes
11: **for all** $v \in G_p(V,E)$ **do** ▷ Thread-level parallelism
12: Update $c_p(v, T_i, S_i)$ by computing received neighbour vertices of v from other processes
13: **end for**
14: **end for**
15: **end for**
16: **end for**
17: $Counts \leftarrow Reduce(c_p(v, T_\rho, S))$
18: Scale $Counts$ based on Niter and embed colorful probability
19: **end procedure**

3 SCALING OF DISTRIBUTED COLOR-CODING

To address the challenges analyzed in Section 2, we propose a novel node-level communication scheme named Adaptive-Group in Section 3.1, and a fine-grained thread-level optimization called neighbor list partitioning in Section 3.2. Both of the approaches are implemented as a subgraph counting application to our open source project Harp-DAAL [8][14]. Harp-DAAL is part of the High Performance Computing Enhanced Apache Big Data Stack (HPC-ABDS) to run data intensive workload on HPC clusters. At the cluster node level, we use Harp [27][26][20] as an extension of Hadoop MapReduce to implement in-memory collective communication operations. At the intra-node level, we develop C/C++ computation kernels running on high-end HPC processors and contribute to Intel DAAL open source project [15].

3.1 Adaptive-Group Communication

Adaptive-Group is an interprocess communication scheme based on the concept of communication group. Given P parallel computing processes, each process p belongs to a communication group where it has data dependencies, i.e., sending/receiving data, with other processes in the group. In an AlltoAll operation, such as MPI_AlltoAll, each process p communicates data with all the other processes in a collective way, namely all processes are associated to

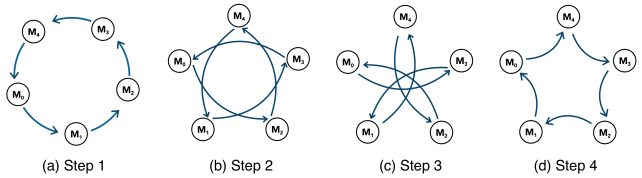


Figure 2: An example of ring-ordered steps in the adaptive-group communication

a single communication group with size P. In Adaptive-Group communication, the collective communication is divided into W steps, where each process p only communicates with processes belonging to a communication group of size m at each step w. The size m and step number W are both configurable on-the-fly and adaptive to computation overhead, load balance, and memory utilization of irregular problems like subgraph counting.

A routing method is required to guarantee that no missing and redundant data transfer occurs during all the W steps. Figure 2 illustrates such a routing method, where the AlltoAll operation among 5 processes is decoupled into 4 steps, and each process only communicates with two other processes within a communication group of size 3 at each step. Line 4 to 13 of Algorithm 3 gives out the pseudo code of Adaptive-Group communication that implements the routing method in Figure 2. Here the communication is adaptive to the template size $|T|$. With a large template size $|T|$, the algorithm adopts the routing method in Figure 2 with a communication group size of 3, while it switches to the traditional AlltoAll operation if the template size is small.

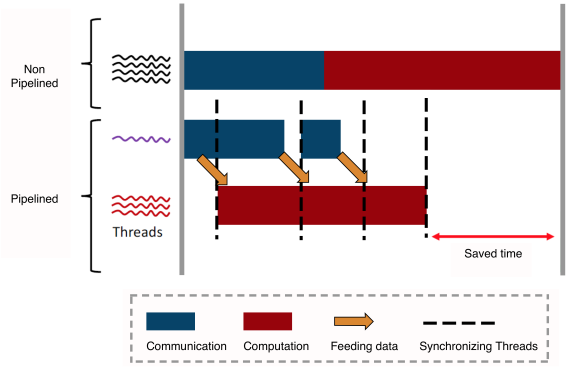


Figure 3: Pipelined adaptive-group communication

3.1.1 Pipeline Design. When adding up all W steps in Adaptive-Group, we apply a pipeline design shown in Figure 3, which includes a computation pipeline (red) and a communication pipeline (blue). Given an Adaptive-Group communication in W steps, each pipeline follows $W + 1$ stages to finish all the work. The first stage is a cold start, where no previous received data exists in the computation pipeline and only the communication pipeline is transferring data. For the following W stage, the work in communication pipeline can be interleaved by the work in computation pipeline. This interleaving can be achieved by using multi-threading programming model, where a single thread is in charge of the communication pipeline and the other threads are assigned to the computation

Algorithm 3 Adaptive-Grouping in Distributed Color-Coding

1: **procedure** Adpative-Grouping in Distributed Color-Coding($G_p(V,E)$, T_i, P)
2: $G_p(V,E)$ is partition of the input graph at process p
T_i is the subtemplate to compute
P is the total number of processes to communicate
T is the template
3: **if** ($|T|$ is large) **then** ▷ Adaptive to large T
4: **for** $r = p+1, p+2, \ldots, P-1, 0, \ldots, p-1$ **do**
5: **if** threadIdx = 0 **then** ▷ Communication Pipeline
6: Compress and Send $< I, C >_{p,r}$ to process r
7: Receive $< I, C >_{2p-r,p}$ from process $2p-r$
8: **else** ▷ Computation Pipeline
9: **for all** $v \in G_p(V,E)$ **do** ▷ Thread-level parallelism
10: Update $c_p(v, T_i, S_i)$ by computing received neighbour vertices of v from process $2p-r-1$
11: **end for**
12: **end if**
13: **end for**
14: **else** ▷ Adaptive to small T
15: Process p All-to-All exchanges local counts $c_p(v, T_i, S_i)$ with other processes
16: **for all** $v \in G_p(V,E)$ **do** ▷ Thread-level parallelism
17: Update $c_p(v, T_i, S_i)$ by computing received neighbour vertices of v from other processes
18: **end for**
19: **end if**
20: **end procedure**

pipeline (See Algorithm 3 line 5 to 12). Since at each stage, the computation pipeline relies on the data received at the previous stage of the communication pipeline, a synchronization of two pipelines at the end of each stage is required (shown as a dashed line in Figure 3). The saved time by using pipeline depends on the ratio of overlapping computation and communication in each stage of two pipelines. We will estimate the bounds of computation and communication in pipeline design for large templates through an analysis of complexity.

3.1.2 Complexity Analysis. When computing subtree T_i, we estimate the computation complexity on remote neighbors at step w as:

$$Comp_{w,p} = O(\binom{k}{|T_i|}\binom{|T_i|}{|T_i'|}\sum_{v \in V_p} N_{r,w}(v)) \quad (4)$$

where k is the number of colors, $|T_i|$ is the size of subtree T_i in template T, and T_i' is a subtree partitioned from T_i according to Algorithm 1.

We divide the neighbors of v into local neighbors $N_l(v)$ and remote neighbors $N_r(v)$. The $N_r(v)$ is made up of neighbors received in each step, $N_r(v) = \sum_{w=1}^{W} N_{r,w}(v)$. With the assumption

of random partitioning G(V,E) by vertices across P processes,

$$\begin{aligned} E[N_{r,w}(V_p)] &= E[\sum_{v \in V_p} N_{r,w}(v)] \\ &= \sum_{v \in V} E[N_{r,w}(v)] \Pr[v \in V_p] \\ &= \sum_{(u,v) \in E} \Pr[v \in V_p, u \in N_{r,w}(v)] = |E|/P^2 \end{aligned} \quad (5)$$

where $|E|$ is the edge number. Further by applying Chernoff bound, we have $N_{r,w}(V_p) = \Theta(|E|/P^2)$ with probability at least $1 - 1/n^2$.

Therefore, we get the bound of computation as

$$Comp_{w,p} = \binom{k}{|T_i|}\binom{|T_i|}{|T_i'|}\Theta(N_{r,w}(V_p)) = \Theta(\binom{k}{|T_i|}\binom{|T_i|}{|T_i'|}|E|/P^2) \quad (6)$$

Similarly, the expectation of peak memory utilization at step w is

$$\begin{aligned} PeakMem_{w,p} &= O(\sum_{v \in V_p} \Big[c(v,T_i) + \sum_{u \in N_{r,w}(v)} c(u,T_i)\Big]) \\ &= O(\binom{k}{|T_i|}(|V|/P + |E|/P^2) \end{aligned} \quad (7)$$

where $c(u,T_i)$ is the length of array (memory space) that holds the combination of color counts for each u, and its complexity is bounded by $O(\binom{k}{|T_i|})$ (refer to line 8 of Algorithm 1).

The communication complexity at step w by Hockney model [13] is

$$\begin{aligned} Com_{w,p} &= O(\alpha + \delta_{w,p} + \beta \sum_{v \in V_p} \sum_{u \in N_{r,w}(v)} c(u,T_i)) \\ &= O(\alpha + \delta_{w,p} + \beta\binom{k}{|T_i|}|E|/P^2) \end{aligned} \quad (8)$$

where α is the latency associated to the operations in step w, β is the data transfer time per byte, and $\delta_{w,p}$ is the time spent by process p in waiting for other processes because of the load imbalance among P processes at step w, which is bounded by

$$\begin{aligned} \delta_{w,p} &= O(Max_{q \neq p}(Time_{w-1,q} - Time_{w-1,p})) \\ &= O(Max_{q \neq p}(Time_{w-1,q})) \end{aligned} \quad (9)$$

where $Time_{w-1,q}$ is the execution time of process q at step $w-1$ which is expressed as

$$Time_{w-1,q} = Max(Comp_{w-1,q}, Com_{w-1,q}). \quad (10)$$

When it comes to the total complexity of all W steps, we assume a routing algorithm described in Figure 2 is used, where $W = P - 1$. We obtain the bound for computation as

$$\begin{aligned} Comp^{pip}_{total,p} &= \sum_{w=1}^{W} Comp_{w,p} \\ &= \Theta(\binom{k}{|T_i|}\binom{|T_i|}{|T_i'|}|E|(P-1)/P^2) \end{aligned} \quad (11)$$

While the peak memory utilization is

$$\begin{aligned} PeakMem^{pip}_{total,p} &= O(Max_w(PeakMem_{w,p})) \\ &= O(\binom{k}{|T_i|}(|V|/P + |E|/P^2) \end{aligned} \quad (12)$$

The total communication overhead in the pipeline design of all steps W is calculated by,

$$Com^{pip}_{total,p} = Com_{w=1,p} + \sum_{w=2}^{W}(1 - \rho_w)Com_{w,p} \quad (13)$$

where ρ_w is defined as the ratio of effectively overlapped communication time by computation in a pipeline step w

$$\rho_w = \frac{Min(Comp_{w-1,p}, Com_{w,p})}{Com_{w,p}}, (w > 1) \quad (14)$$

As the computation per neighbor $u \in N_{r,w}(v)$ for T_i is bounded by $\binom{k}{|T_i|}\binom{|T_i|}{|T_i'|}$ and communication data volume per u bounded by its memory space complexity $\binom{k}{|T_i|}$, $Comp_{w,p}$ increases faster than $Com_{w,p}$ with respect to the template size $|T_i|$. Therefore, for large templates, the computation term $Comp_{w,p}$ is generally larger than the communication overhead $Com_{w,p}$ at each step, and we have $\rho_w \approx 1$. Equation 13 is bounded by

$$\begin{aligned} Com^{largeT,pip}_{total,p} &= O(Com_{w=1,p}) \\ &= O(\alpha + \delta_{w=1,p} + \frac{\beta}{P}\binom{k}{|T_i|}|E|/P^2) \end{aligned} \quad (15)$$

With large $|T_i|$, we have

$$\begin{aligned} \delta_{w=1,p} &= O(Max_{q \neq p}(Comp_{w=1,q})) \\ &= O(\frac{1}{P^2}\binom{k}{|T_i|}\binom{|T_i|}{|T_i'|}|E|) \end{aligned} \quad (16)$$

which is inversely proportional to P^2. The third term in Equation 15 is also inversely proportional to P. Therefore $Com^{largeT,pip}_{total,p}$ shall decrease with an increasing P, which implies that the algorithm is scalable with large templates by bounding the communication overhead.

For small templates, there is usually no sufficient workload to interleave communication overhead, which gives a relatively small ρ_w value in Equation 13 and compromises the effectiveness of pipeline interleaving. Even worse, as the transferred data at each step w is small, it cannot well leverage the bandwidth of interconnect when compared to the AlltoAll operation. In such cases, the Adaptive-Group is able to switch back to AlltoAll mode and ensure a good performance.

3.1.3 Implementation. We implement the pipelined Adaptive-Group communication with Harp. In Harp, a mapper plays the role of a parallel process, and mappers can do various collective communications that are optimized for big data problems. In implementation like MPI_AlltoAll, each process p out of P prepares a slot Slot(q) for any other process q that it communicates with, and pushes data required by q to Slot(q) prior to the collective communication. The ID label of sender and receiver are attached to the slots in a static way, and the program must choose a type of collective operation (E.g., AlltoAll, Allgather) in the compilation stage.

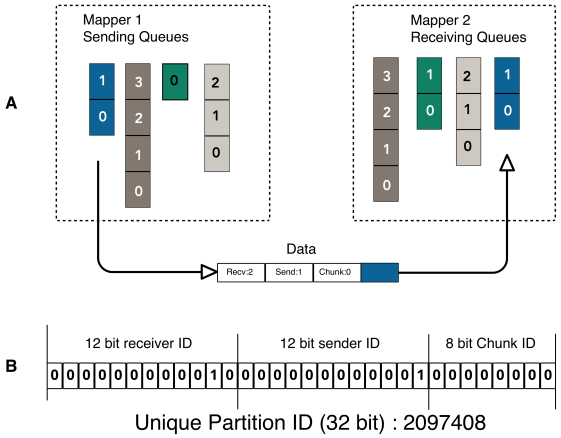


Figure 4: Adaptive-Group tags each data packet with a meta ID, which is used by a routing algorithm for data transfer. Both the meta ID and the routing algorithm are re-configurable on-the-fly

In contrast, each Harp mapper keeps a sender queue and a receiving queue, while it is the data packet that has been labeled by a meta ID as shown in Figure 4. For Adaptive-Group, the meta ID for each packet consists of three parts (bit-wise packed to a 32 bit Integer): The sender mapper ID, the receiver mapper ID and the offset position in the sending queue. A user-defined routing algorithm then decodes the meta ID and delivers the packet in a dynamically-configurable way. The routing algorithm is able to detect template and workload sizes, and switch on-the-fly between pipeline and AlltoAll modes.

3.2 Fine-grained Load Balance

For an input graph with high degree skewness (distribution of vertex out-degree), it imposes a load imbalance issue at the thread-level. In Algorithm 1 and 2, the task of computing the counts of a certain vertex by looping all entries of its neighbor list is assigned to a single thread. If the max degree of an input graph is several orders of magnitudes larger than the average degree, one thread may take orders of magnitude workload than average. For large templates, this imbalance is amplified by the exponential increase of computing counts for a single vertex in line 9 of Algorithm 1.

To address the issue of workload skewness, we propose a neighbor list partitioning technique, which is implemented by the multi-thread programming library OpenMP. Algorithm 4 illustrates the process of creating the fine-grained tasks assigned to threads. Given maximal task size s, the process detects the neighbor list length n of a vertex v. If n is beyond s, it extracts a sub-list sized s out of the n neighbors and creates a task including neighbors in the sub-list associated to vertex v. The same process applies to the remaining part of the truncated list until all neighbors are partitioned. If n is already smaller than s, it creates a task with all the n neighbors associated to vertex v.

The neighbor list partitioning ensures that no extremely large task is assigned to a thread by bounding the task size to s, which improves the workload balance at thread-level. However, it comes with a race condition if two threads are updating tasks associated to the same vertex v. We use atomic operations of OpenMP to resolve the race condition and shuffle the created task queue at line 23 of Algorithm 4 to mitigate the chance of conflict.

Algorithm 4 Create Parallel Tasks via Neighbour List Partitioning

```
1:  procedure TASK CREATION(s)
2:      s is user-defined maximal task size
3:      V is local vertices
4:      N_v is neighbour list of v ∈ V
5:      n is the number of neighbours
6:      l is the length of new task
7:      pos is the offset of sub-list
8:      Q stores created tasks
9:      for all v ∈ V do
10:         if |N_v| < s then
11:             Q add task(v, N_v)
12:         else
13:             n ← |N_v|
14:             pos ← 0
15:             while n > 0 do
16:                 l ← Min(n, s)
17:                 Q add Task(v, N_v(pos : pos + l))
18:                 pos+ = l
19:                 n− = l
20:             end while
21:         end if
22:     end for
23:     shuffle tasks in Q
24: end procedure
```

4 EVALUATION OF PERFORMANCE AND ANALYSIS OF RESULTS

4.1 Experimentation Setup

We conduct a set of experiments by implementing 4 code versions of distributed color-coding algorithm with Harp-DAAL: Naive, Pipeline, Adaptive and AdaptiveLB (Load Balance). Table 1 lists individual optimization technique for experiments. They aim to systematically investigate the impact of each optimization, which addresses the sparse irregularity, the low computation to communication ratio or the high memory footprint issues of subgraph counting.

Table 1: Harp-DAAL code version in experiments

Harp-DAAL version	Communication Mode	Adaptive Switch	Neighbor list partitioning
Naive	AlltoAll	Off	Off
Pipeline	Pipeline	Off	Off
Adaptive	AlltoAll/pipeline	On	Off
AdaptiveLB	AlltoAll/pipeline	On	On

We use synthetic and real datasets in our experiments which are summarized in Table 2. Miami, Orkut [3][18][25], Twitter [5], SK-2005 [11], and Friendster [18] are datasets generated by real applications. RMAT synthetic datasets are generated by the RMAT model [7] by specifying the size and skewness. Specifying a higher skewness generates a highly imbalanced distribution of out-degree

Table 2: Datasets in Experiments (K=10^3, M=10^6,B=10^9)

Data	Vertices	Edges	Avg Deg	Max Deg	Source	Abbreviation
Miami	2.1M	51M	49	9868	social network	MI
Orkut	3M	230M	76	33K	social network	OR
NYC	18M	480M	54	429	social network	NY
Twitter	44M	2B	50	3M	Twitter users	TW
Sk-2005	50M	3.8B	73	8M	UbiCrawler	SK
Friendster	66M	5B	57	5214	social network	FR
RMAT-250M(k=1,3,8)	5M	250M	100,102,217	170,40K,433K	PaRMAT	R250K1,3,8
RMAT-500M(k=3)	5M	500M	202	75K	PaRMAT	R500K3

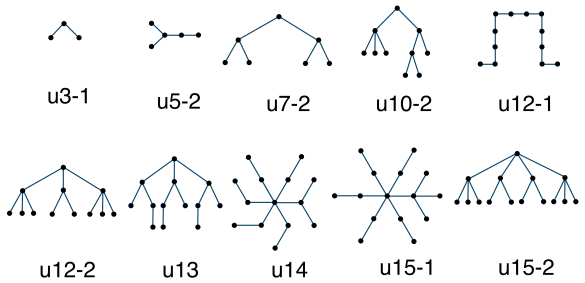


Figure 5: Tree Templates used in experimentation with growing sizes and different shapes

Table 3: Computation Intensity of Templates

Template	Memory Space Complexity	Computation Complexity	Computation Intensity
u3-1	3	6	2
u5-2	25	70	2.8
u7-2	147	434	2.9
u10-2	1047	5610	5.3
u12-1	4082	24552	6.0
u12-2	3135	38016	12
u13	4823	109603	22
u14	7371	242515	32
u15-1	12383	753375	60
u15-2	15773	617820	39

for input graph datasets. Therefore we can use different skewness of RMAT datasets to study the impact of unbalanced workload on the performance. The different sizes and structures of the tree templates used in the experiments are shown in Figure 5, where templates from u3-1 to u12-2 are collected from [23], while u13 to u15 are the largest tree subgraphs being tested to date.

We observe that the size and shape of sub-templates affect the ratio of computation and communication in our experiments. This corresponds to code line 8 of Algorithm 1, where each sub-template T_i is partitioned into trees T_i' and T_i''. The space complexity for each neighbor $u \in N(v)$ is bounded by $\binom{k}{|T_i|}$ when computing sub-template T_i, and is proportional to the communication data volume. The computation, which depends on the shape of the template, is bounded by $\binom{k}{|T_i|}\binom{|T_i|}{|T_i'|}$. In Table 3, the memory space complexity is denoted as $\sum_i \binom{k}{|T_i|}$, and the computation complexity is $\sum_i \binom{k}{|T_i|}\binom{|T_i|}{|T_i'|}$. In this paper, we define the *computation intensity* as the ratio of computation versus communication (or space) for a template in Figure 5. For example, the computation intensity generally increases along with the template size from u3-1 to u15-2, However, for the same template size, template u12-2 has a computation intensity of 12 while u12-1 only has 6. We will use these definitions and refer to their values when analyzing the experiment results in the rest of sections.

All experiments run on an Intel Xeon E5 cluster with 25 nodes. Each node is equipped with two sockets of Xeon E5 2670v3 (2×12 cores), and 120 GB of DDR4 memory. We use all 48 threads from by default in our tests, and InfiniBand is enabled in either Harp or the MPI communication library. Our Harp-DAAL codes are compiled by JDK 8.0 and Intel ICC Compiler 2016 as recommended by Intel. The MPI-Fascia [23] codes are compiled by OpenMPI 1.8.1 as recommended by its developers.

4.2 Scaling with Adaptive Communication

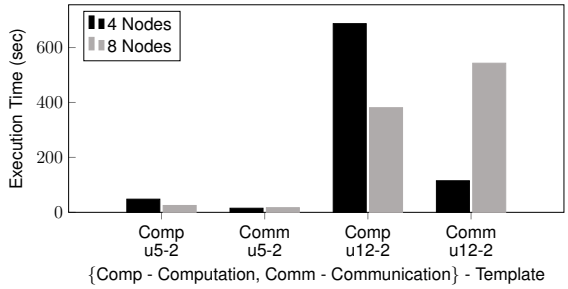


Figure 6: Scaling up template sizes on dataset R500K3 for Harp-DAAL Naive implementation from 4 cluster nodes to 8 cluster nodes

We first conduct a baseline test with the naive implementation of distributed color-coding. When the subgraph template size is scaled up as shown in Figure 6, we have the following observations: 1) For small template u5-2, computation decreases by 2x when scaling from 4 to 8 nodes while communication only increases by 13%. 2) For large template u12-2, doubling cluster nodes only reduces computation time by 1.5x but communication grows by 5x. It implies that the AlltoAll communication within the naive implementation does not scale well on large templates.

To clarify the effectiveness of Harp-DAAL Pipeline on large templates, Figure 7 compares strong scaling speedup, total execution time, and ratio of communication/computation time between the Naive and Pipeline implementation versions on Dataset R500K3, which has skewness similar to real application datasets such as

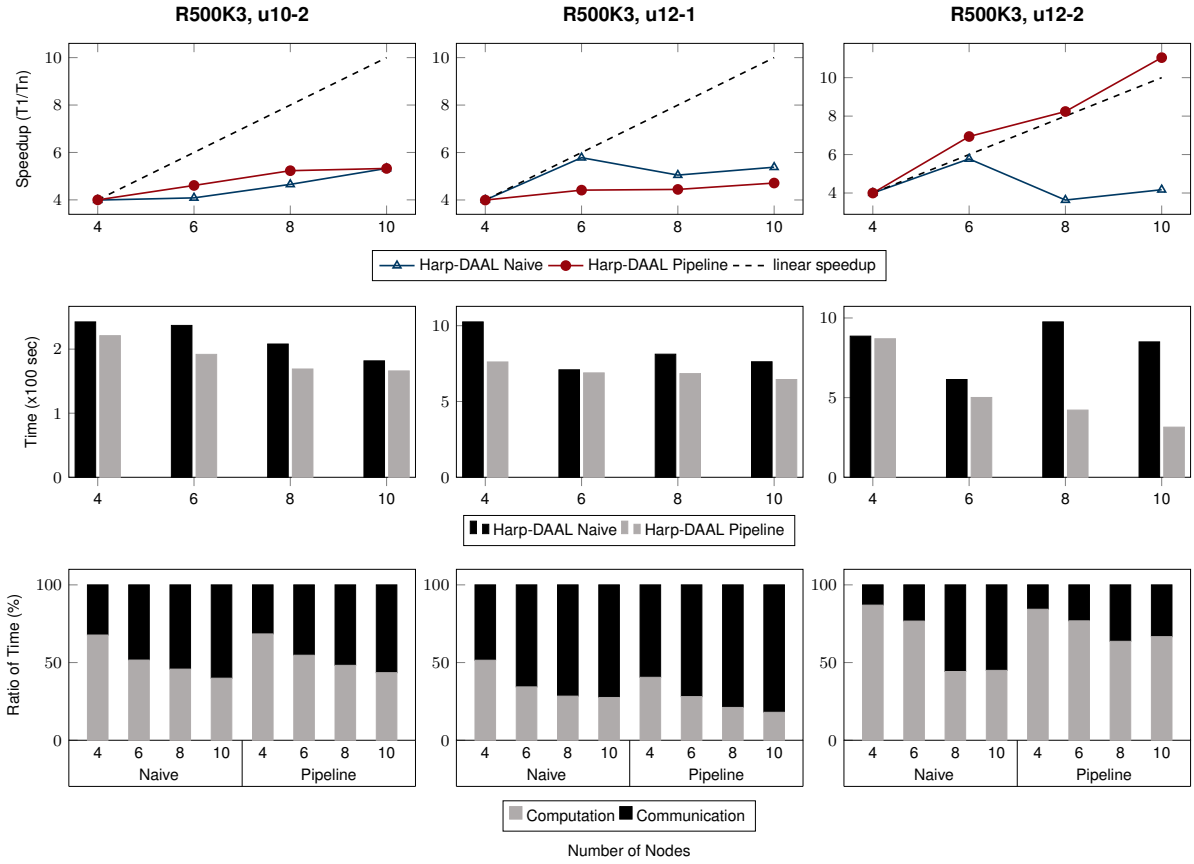


Figure 7: Strong Scaling Tests on dataset R500K3 from 4 to 10 cluster nodes with large templates (u10-2, u12-1, u12-2). First row gives the speedup starting from 4 cluster nodes since a single node cannot hold the dataset; The second row compares the total execution time from two implementations; The third row is the ratio of compute/communicate time in the total execution time

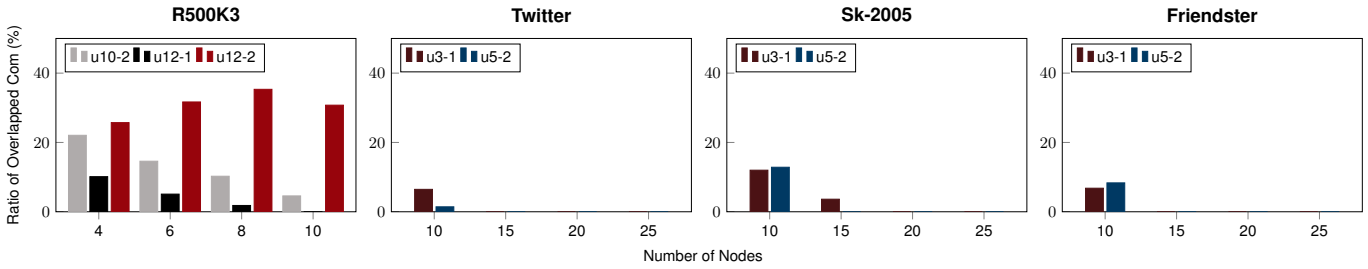


Figure 8: The ratio ρ of overlapped communication/total communication by Harp-DAAL pipeline, tests on R500K3 for large templates (u10-2, u12-1, u12-2), and Twitter, Sk-2005, Friendster for small templates u3-1, u5-2

Orkut. For template u10-2, Harp-DAAL Pipeline only slightly outperforms Harp-DAAL Naive in terms of speedup and total execution time. However, for u12-2, this performance gap increases to 2.3x (8 nodes) and 2.7x (10 nodes) in execution time, and the speedup is significantly improved starting from 8 nodes. The result is consistent with the Table 3, where u12-2 has 2 times higher computation intensity than u10-2, which provides the pipeline design of sufficient workload to interleave the communication overhead. The ratio charts of Figure 7 also confirm this result that Harp-DAAL Pipeline has more than 65% of computation on 8 and 10 nodes, while the computation ratio for Harp-DAAL Naive is below 50% when scaling on 8 and 10 nodes. Although template u12-1 has the same size as template u12-2, it only has half of the computation intensity as shown in Table 3. According to Equation 13, the low computation intensity on u12-1 reduces the overlapping ratio ρ, and we find in Figure 8 that Harp-DAAL Pipeline has less than 10% of overlapping ratio for u12-1, while u12-2 keeps around 30% when scaling up to 10 cluster nodes.

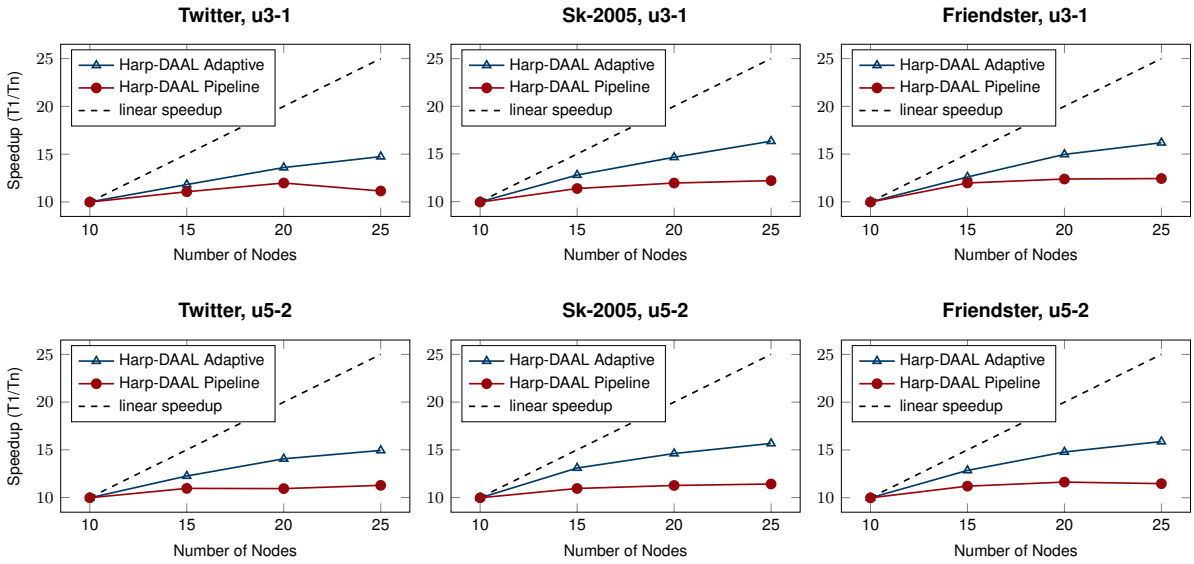


Figure 9: Strong Scaling Tests on large dataset Twitter, SK-2005, Friendster from 10 cluster nodes to 25 cluster nodes with small templates (u3-1, u5-2). Harp-DAAL Adaptive switches to AlltoAll mode and outperforms pipeline.

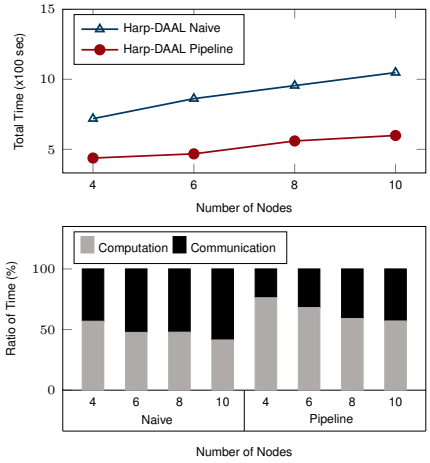


Figure 10: Weak Scaling of RMAT with skeweness 3; The workload is proportional to the cluster nodes: e.g., 5 million vertices with 250 million edges on 4 cluster nodes, and 7.5 million vertices with 375 million edges on 6 cluster nodes.

For small templates similar to u3-1 and u5-2 which have low computation intensities, we shall examine the effectiveness of adaptability in Harp-DAAL Adaptive, where the code switches to AlltoAll mode. In Figure 9, we did the strong scaling tests with small templates u3-1 and u5-2. Results show that when compared to Harp-DAAL Pipeline, Harp-DAAL Adaptive has a better speedup for tests of both u3-1 and u5-2 on three large datasets Twitter, Sk-2005, and Friendster. Also, the poor performance of Harp-DAAL Pipeline is due to the low overlapping ratio in Figure 8 for Twitter, Sk-2005, and Friendster, where ρ drops to near zero quickly after scaling to more than 15 nodes.

In addition to strong scaling, we present weak scaling tests in Figure 10 for template u12-2. We generate a group of RMAT datasets with skewness 3 and an increasing number of vertices and edges proportional to the running cluster nodes. By fixing the workload on each cluster node, the weak scaling on Harp-DAAL Pipeline reflects the additional communication overhead when more cluster nodes are used. For Harp-DAAL Pipeline, execution time grows only by 20% with cluster nodes growing by 2 (from 4 nodes to 8 nodes). From the ratio chart in Figure 10, it is also clear that the Naive implementation has its communication ratio increased to more than 50% by using 8 cluster nodes while that communication ratio of Pipeline implementation keeps under 40%.

4.3 Fine-grained Load Balance

Although Adaptive-Group communication and pipeline design mitigate the node-level load imbalance caused by skewness of neighbor list length for each vertex in input graph, it can not resolve fine-grained workload imbalance at thread-level inside a node. By applying our neighbor list partitioning technique, we compare the performance of Harp-DAAL AdaptiveLB with Harp-DAAL Adaptive on datasets with different skewness. In Figure 11, we first compare the datasets with increasing skewness shown in Table 2. With R250K1 and MI having small skewness, the neighbor list partitioning barely gains any advantage, and its benefit starts to appear from dataset OR by 2x improvement of the execution time. For dataset with high skewness such as R250K8 with u12-2 template, this acceleration achieves up to 9x of the execution time as shown in Figure 11.

When scaling threads from 6 to 48, for dataset MI having small skewness, the execution time does not improve much. While for R250K8, Harp-DAAL AdaptiveLB keeps a good performance compared to Naive implementation. In particular, the thread-level performance of Harp-DAAL Naive drops down after using more than physical core number (24) of threads, which implies a suffering from hyper threading. However, Harp-DAAL AdaptiveLB is able to keep the performance unaffected by hyper threading. To further justify

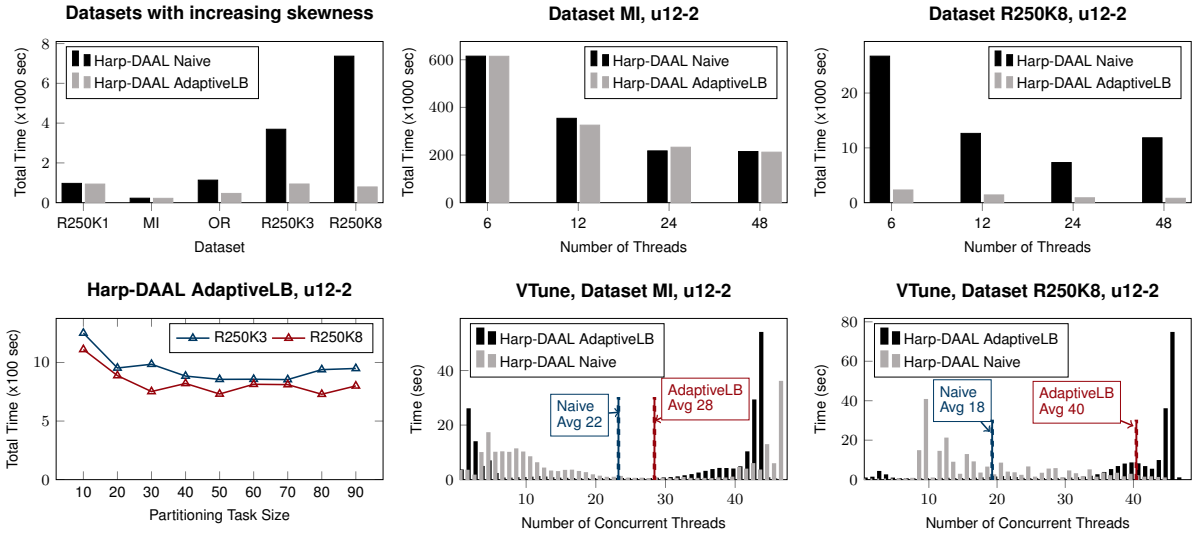


Figure 11: Execution details on a single Xeon E5 node (x2 sockets, and a total of 24 physical cores). The default thread number in test is 48 and partitioned neighbor list is 50.

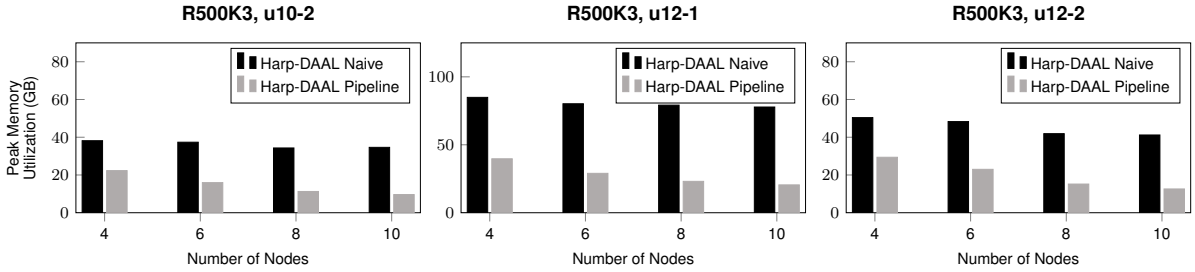


Figure 12: Peak memory utilization for Harp-DAAL Naive and Harp-DAAL Adaptive on dataset R500K3 with templates u10-2,u12-1, u12-2 from 4 to 10 nodes

the thread efficiency of Harp-DAAL AdaptiveLB, we measure the thread concurrency by VTune. The histograms show the distribution of execution time by the different numbers of concurrently running threads. For dataset MI, the number of average concurrent threads of Harp-DAAL Naive and AdaptiveLB are close (22 versus 28) because the dataset MI does not have sever load imbalance caused by skewness. For dataset R250K8, the number of average concurrent threads of Harp-DAAL AdaptiveLB outperforms that of Harp-DAAL Naive by around 2x (40 versus 18).

Finally, we study the granularity of task size and how it affects partitioning of the neighbor list. In Algorithm 4, each task of updating neighbor list is bounded by a selected size s. If s is too small, there will be a substantial number of created tasks, which adds additional thread scheduling and synchronization overhead. If s is too large, it can not fully exploit the benefits of partitioning neighbor list. There exists a range of task granularity which can be observed in the experiments on R250K3 and R250K8. To fully leverage the neighbor list partitioning, a task size between 40 and 60 gives better performance than the other values.

4.4 Peak Memory Utilization

Adaptive-group communication and pipeline design also reduce the peak memory utilization at each node. According to Equation 12, peak memory utilization depends on two terms: the $c(v, T)$ from local vertices V_p and $c(u, T)$ from remote neighbors $u \in N_{r,w}(v)$. When total $|V|$ of dataset is fixed, $|V_p|$ decreases with increasing process number P and thus reduces the first peak memory term. The second term associated with u at step w is also decreasing along with P because more steps ($W = P - 1$) leads to small data volume involved in each step. In Figure 12, we observe this reduction of peak memory utilization along with the growing number of cluster nodes from 4 to 10. Compared to Harp-DAAL Naive, Harp-DAAL Pipeline reduces the peak memory utilization by 2x on 4 nodes, and this saving grows to around 5x for large templates u10-2, u12-1, and u12-2.

4.5 Overall Performance

The overall performance combines the optimization for scaling, load imbalance, and peak memory utilization. Figure 13 shows a comparison of Harp-DAAL AdaptiveLB versus MPI-Fasica in total execution time with growing templates on Twitter Dataset. For

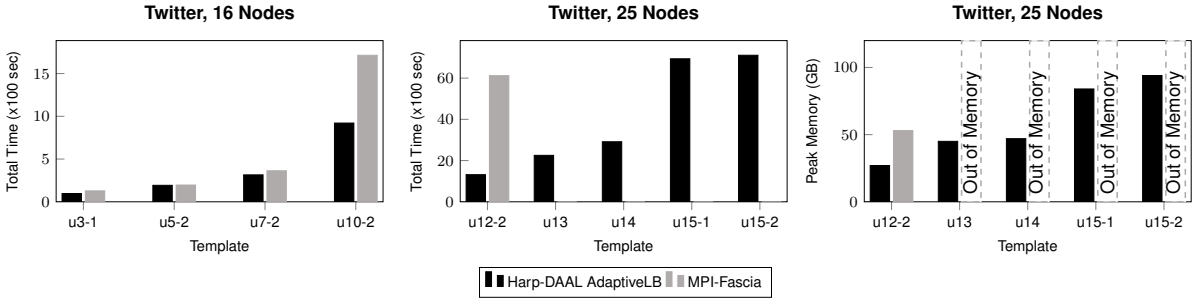


Figure 13: Overall Performance of Harp-DAAL AdaptiveLB vs. MPI-Fascia with increasing template sizes from u3-1 to u15-2

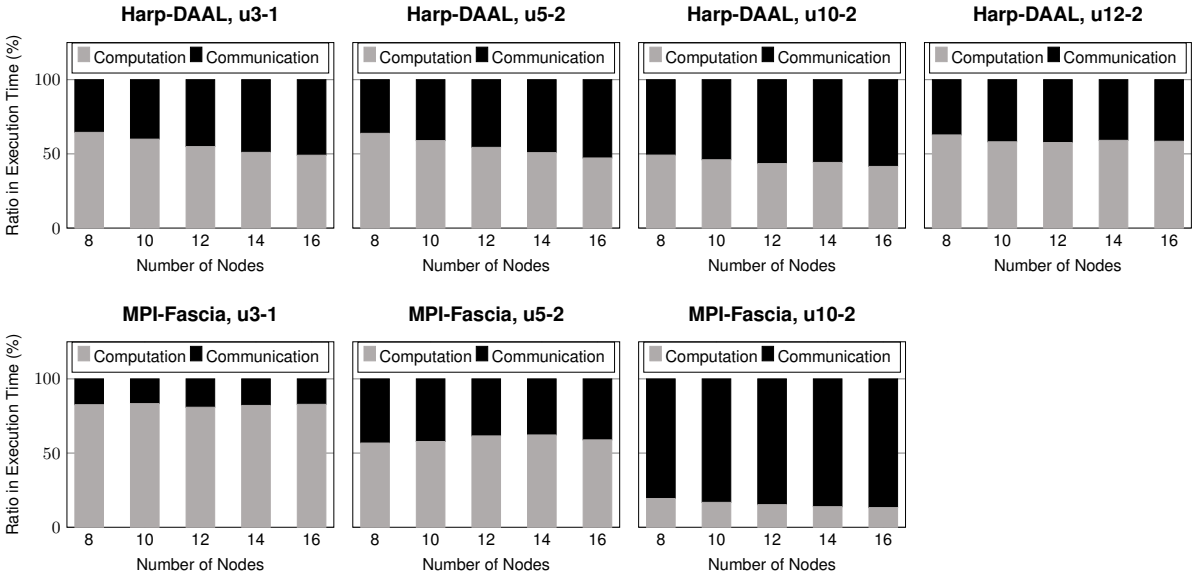


Figure 14: The ratio of computation versus communication in total execution time for Harp-DAAL AdaptiveLB and MPI-Fascia

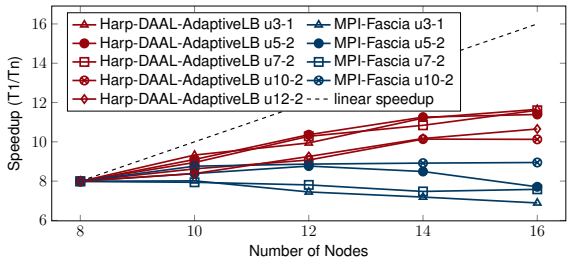


Figure 15: Strong scaling of Harp-DAAL AdaptiveLB vs. MPI-Fascia on Twitter with template sizes from u3-1 to u12-2

small templates u3-1, u5-2, and u7-2, Harp-DAAL AdaptiveLB performs comparably or slightly better. Small templates can not fully exploit the efficiency of pipeline due to low computation intensity. For large template u10-2, Harp-DAAL AdaptiveLB achieves 2x better performance than MPI-Fascia, and it continues to gain by 5x better performance for u12-2. Beyond u12-2, Harp-DAAL AdaptiveLB can still scale templates up to u15-2. MPI-Fascia can not run templates larger than u12-2 on Twitter because of high peak memory utilization over the 120 GB memory limitation per node.

Figures 14 and 15 further compare the strong scaling results between Harp-DAAL AdaptiveLB and MPI-Fascia. Scaling from 8 nodes to 16 nodes, Harp-DAAL AdaptiveLB achieves better speedup than MPI-Fascia for templates growing from u3-1 to u12-2. MPI-Fascia even can not run Twitter on 8 nodes due to its high peak memory utilization. The ratio charts in Figure 14 give more details about the speedup, where MPI-Fascia has a comparable communication overhead ratio in execution time for small tempaltes u3-1 and u5-2, however, the communication ratio increases to 80% at template u10-2 while Harp-DAAL AdaptiveLB keeps communication ratio around 50%. At template u12-2, Harp-DAAL AdaptiveLB further reduces the communication overhead to around 40% because the adaptive-group and pipeline favors large templates with high computation intensity.

5 RELATED WORK

Subgraphs of size k with an independent set of size s can be counted in time roughly $O(n^{k-s}\text{poly}(n))$ through matrix multiplication based methods [17, 24]. There is substantial work on parallelizing the color-coding technique. ParSE[28] is the first distributed algorithm

based on color-coding that scales to graphs with millions of vertices with tree-like template size up to 10 by a few hours. SAHAD [29] expands this algorithm up to 12 vertices labeled template on a graph with 9 million vertices within less than an hour by using a Hadoop-based implementation. FASCIA [21–23] is the state-of-the-art color-coding treelet counting tool. By highly optimized data structure and MPI+OpenMP implementation, it supports tree template of size up to 10 vertices in billion-edge networks in a few minutes. Recent work [6] also explores the topic of a more complex template with treewidth 2, which scales up to 10 vertices for graphs of up to $2M$ vertices. The original color-coding technique has been extended in various ways, e.g., a derandomized version [1], and to other kinds of subgraphs.

6 CONCLUSION

Subgraph counting is a NP-hard problem with many important applications on large networks. We propose a novel pipelined communication scheme for finding and counting large tree templates. The proposed approach simultaneously addresses the sparse irregularity, the low computation to communication ratio and high memory footprint, which are difficult issues for scaling of complex graph algorithms. The methods are aimed at large subgraph cases and use approaches that make the method effective as graph size, subgraph size, and parallelism increase. Our implementation leverages the Harp-DAAL framework adaptively improves the scalability by switching the communication modes based on the size of subgraph templates. Fine-grained load balancing is achieved at runtime with thread level parallelism. We demonstrate that our proposed approach is effective in particular on irregular subgraph counting problems and problems with large subgraph templates. For example, it can scale up to the template size of 15 vertices on Twitter datasets (half a billion vertices and 2 billion edges) while achieving 5x speedup over the state-of-art MPI solution. For datasets with high skewness, the performance improves up to 9x in execution time. The peak memory utilization is reduced by a factor of 2 on large templates (12 to 15 vertices) compared to existing work. Another successful application has templates of 12 vertices and a massive input Friendster graph with 0.66 billion vertices and 5 billion edges. All experiments ran on a 25 node cluster of Intel Xeon (Haswell 24 core) processors. Our source code of subgraph counting is available in the public github domain of Harp project[14].

In future work, we can apply this Harp-DAAL subgraph counting approach to other data-intensive irregular graph applications such as random subgraphs and obtain scalable solutions to the computational, communication and load balancing 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] Noga Alon and Shai Gutner. 2010. Balanced Families of Perfect Hash Functions and Their Applications. *ACM Trans. Algorithms* 6, 3, Article 54 (July 2010), 54:1–54:12 pages.
[2] Noga Alon, Raphael Yuster, and Uri Zwick. 1995. Color-Coding. *J. ACM* 42, 4 (July 1995), 844–856.
[3] C. L. Barrett, R. J. Beckman, M. Khan, V. S. A. Kumar, M. V. Marathe, P. E. Stretz, T. Dutta, and B. Lewis. 2009. Generation and Analysis of Large Synthetic Social Contact Networks. In *WSC*. 1003–1014.
[4] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2017. Counting Graphlets: Space vs Time. In *WSDM*. 557–566.
[5] Meeyoung Cha, Hamed Haddadi, Fabrício Benevenuto, and Krishna P. Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy.. In *ICWSM*, Vol. 14.
[6] V. T. Chakaravarthy, M. Kapralov, P. Murali, F. Petrini, X. Que, Y. Sabharwal, and B. Schieber. 2016. Subgraph Counting: Color Coding Beyond Trees. In *IPDPS*. 2–11.
[7] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *SIAM*, Vol. 6.
[8] Langshi Chen, Bo Peng, Bingjing Zhang, Tony Liu, Yiming Zou, Lei Jiang, Robert Henschel, Craig Stewart, Zhang Zhang, Emily Mccallum, Tom Zahniser, Omer Jon, and Judy Qiu. 2017. Benchmarking Harp-DAAL: High Performance Hadoop on KNL Clusters. In *IEEE Cloud*. Honolulu, Hawaii, US.
[9] X. Chen and J. C. S. Lui. 2016. Mining Graphlet Counts in Online Social Networks. In *ICDM*. 71–80.
[10] Radu Curticapean and Dániel Marx. 2014. Complexity of counting subgraphs: Only the boundedness of the vertex-cover number counts. In *FOCS*. IEEE, 130–139.
[11] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (Dec. 2011), 1:1–1:25.
[12] Jörg Flum and Martin Grohe. 2004. The parameterized complexity of counting problems. *SIAM J. Comput.* 33, 4 (2004), 892–922.
[13] Roger W. Hockney. [n. d.]. The communication challenge for MPP: Intel Paragon and Meiko CS-2. 20, 3 ([n. d.]), 389–398.
[14] Indiana University. 2018. Harp-DAAL official website. https://dsc-spidal.github.io/harp. (2018). Online; Accessed: 2018-01-21.
[15] Intel Corporation. 2018. The Intel Data Analytics Acceleration Library (Intel DAAL). https://github.com/intel/daal. (2018). Online; accessed 2018-01-21.
[16] Arijit Khan, Nan Li, Xifeng Yan, Ziyu Guan, Supriyo Chakraborty, and Shu Tao. 2011. Neighborhood Based Fast Graph Search in Large Networks. In *SIGMOD*. New York, NY, USA, 901–912.
[17] Miros Kowaluk, Andrzej Lingas, and Eva-Marta Lundell. 2011. Counting and Detecting Small Subgraphs via Equations and Matrix Multiplication. In *SODA*. 1468–1476.
[18] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).
[19] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824.
[20] B. Peng, B. Zhang, L. Chen, M. Avram, R. Henschel, C. Stewart, S. Zhu, E. Mccallum, L. Smith, T. Zahniser, J. Omer, and J. Qiu. 2017. HarpLDA+: Optimizing latent dirichlet allocation for parallel efficiency. In *2017 IEEE International Conference on Big Data (Big Data)*. 243–252.
[21] George M. Slota and Kamesh Madduri. 2013. Fast approximate subgraph counting and enumeration. In *ICPP*. 210–219.
[22] George M. Slota and Kamesh Madduri. 2014. Complex network analysis using parallel approximate motif counting. In *IPDPS*. 405–414.
[23] George M. Slota and Kamesh Madduri. 2015. Parallel Color-Coding. *Parallel Comput.* 47 (2015), 51–69.
[24] V. Vassilevska and R. Williams. 2009. Finding, minimizing, and counting weighted subgraphs. In *STOC*. 455–464.
[25] J. Yang and J. Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *ICDM*. 745–754.
[26] Bingjing Zhang, Bo Peng, and Judy Qiu. 2016. High Performance LDA through Collective Model Communication Optimization. *Procedia Computer Science* 80 (2016), 86–97.
[27] Bingjing Zhang, Yang Ruan, and Judy Qiu. 2015. Harp: Collective Communication on Hadoop. In *IC2E*. 228–233.
[28] Zhao Zhao, Maleq Khan, VS Anil Kumar, and Madhav V. Marathe. 2010. Subgraph enumeration in large social contact networks using parallel color coding and streaming. In *ICPP*. 594–603.
[29] Zhao Zhao, Guanying Wang, Ali R. Butt, Maleq Khan, VS Anil Kumar, and Madhav V. Marathe. 2012. Sahad: Subgraph analysis in massive networks using hadoop. In *IPDPS*. 390–401.