MESM: A Query-Agnostic and Memory-Efficient Parallel Subgraph Matching Algorithm

Shubhashish Kar
Department of Computer Science
University of Nevada, Las Vegas
Las Vegas, USA
kars1@unlv.nevada.edu

Shaikh Arifuzzaman
Department of Computer Science
University of Nevada, Las Vegas
Las Vegas, USA
shaikh.arifuzzaman@unlv.edu

***Abstract*—Subgraph matching, also known as motif finding, is a fundamental problem in graph analysis with extensive applications. However, identifying subgraphs in large-scale graphs is challenging due to its NP-Hard complexity. In addition to the time complexity, previous solutions often suffer from excessive memory usage when dealing with large-scale graphs. This issue is exacerbated in shared-memory systems, where memory is more limited compared to distributed settings. Therefore, achieving a balance between execution time and memory efficiency is vital in such environments. In this paper, we present a query-agnostic shared-memory parallel algorithm that incorporates ordering in set intersection, resulting in an 8% reduction in enumeration time for large graphs. Our approach also achieves memory usage reductions ranging from 2× to 8.2× compared to state-of-the-art techniques, while maintaining comparable runtime performance on large datasets. Extensive experiments with various query and graph datasets demonstrate improved scalability and effective workload balancing of our approach compared to other methods.**

***Index Terms*—Graph Algorithms, Subgraph Matching, High Performance Computing, Shared-Memory System, Parallel Algorithms.**

I. Introduction

Graphs are a versatile framework to model the problems of different application domains and systems such as marketing [19], bioinformatics [7], [22], social networks [12], [24], [28], and cybersecurity [23]. Recently, the rapid increase in data and system complexity have led to a significant growth in the size of corresponding graphs [5], [9], [12], [13], [24]. One effective method for extracting insights from the vast graph data across various domains is through subgraph matching, which involves searching a large graph for all instances of a specified query pattern while taking into account the labels and edges among the vertices. For example, in Fig. 1, the four matches found in the data graph for the query graph based on the labels are {(u0, v0), (u1, v1), (u2, v2)}, {(u0, v3), (u1, v8), (u2, v2)}, {(u0, v3), (u1, v4), (u2, v5)} and {(u0, v6), (u1, v4), (u2, v5)}.

Subgraph matching has numerous applications, including identifying diamond subgraphs in social networks [15], detecting rumor patterns in message transmission graphs [30], performing sub-compound searches in chemical compounds [18], and finding similar patterns in genomic networks [7], [8]. Despite its wide-ranging applications in graph mining, subgraph matching is highly computationally complex and is classified as an NP-hard problem [16]. Consequently, there is an urgent need to identify patterns in large-scale graphs more quickly than previous solutions have allowed. However, the massive size of modern graphs, combined with the sparsity of degree distribution, vast search space, poor memory locality, and unpredictable memory access patterns, complicates the development of efficient solutions. These challenges are further exacerbated by limited memory availability [3], [4], [6], [25].

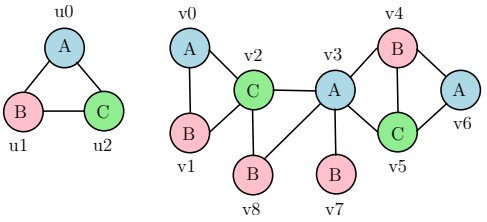


Fig. 1: Illustration of a query graph and a data graph.

Subgraph matching algorithms are classified into three categories in [27]: *direct-enumeration*, *indexing-based-enumeration* and *preprocessing-based-enumeration*. The preprocessing-enumeration approach consists of two main phases: preprocessing and enumeration. In the preprocessing phase, unpromising candidates are filtered out, auxiliary data structures are maintained, and an optimal order is generated to facilitate faster computation in the subsequent phases. Existing solutions predominantly use vertex-extension techniques, which rely on incrementally mapping the vertices of the query graph to the data graph. Within vertex-extension methods, approaches such as backtracking and query compilation are common. The backtracking approach, in particular, is query-agnostic and explores the data graph recursively to identify the query pattern.

Our implementation, **MESM** (Memory-Efficient Subgraph Matching), combines the preprocessing-enumeration approach with the query-agnostic method for a shared-memory system to achieve memory efficiency, improved runtime, and scalability. The contributions of this paper can be summarized as follows.

- We propose a memory-efficient solution for generic

queries that achieves competitive runtime, addressing the limitations in memory efficiency found in many previous solutions.
- We introduce an ordering mechanism to the set-intersection method during the enumeration phase, demonstrating its positive impact on performance using real-world datasets.
- We perform extensive experiments on datasets from various application domains, showcasing results that highlight parallel efficiency, including improved load balancing and scalability.

II. Preliminaries

This section defines subgraph isomorphism and describes the backtracking algorithm to support our discussion.

Problem Definition: Given a labeled query graph and a labeled data graph, this study focuses on identifying subgraphs within the data graph that are isomorphic to the query graph using a shared-memory parallel approach. Subgraph isomorphism is the core concept at the heart of the subgraph matching problem.

To define subgraph isomorphism, a graph can be represented as $G = (V, E, l, \sigma)$, where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and σ is the set of labels. Consider two graphs, $G = (V, E)$ and $G_0 = (V_0, E_0)$. The subgraph isomorphism between G and G_0 is a bijection between the vertex sets V and V_0, i.e., $f : V \to V_0$ such that $l(v_i) = l(f(v_i))$ and $(v_i, v_j) \in E$ if and only if $(f(v_i), f(v_j)) \in E_0$ for all $v_i, v_j \in V$.

Algorithm 1: Subgraph Matching Algorithm

Input: query graph Q and data graph G;
Output: all the matches of Q in G;

```
1  C ← generate candidate vertex sets;
2  A ← build auxiliary data structure;
3  φ ← generate a matching order;
4  Match (Q, G, C, A, φ, {}, 1);
5  Match (Q, G, C, A, φ, M, i) :
6      if i = |φ| + 1 then
7          Output M;
8          return;
9      end
10     u ← select i-th node in φ;
11     LC(u, M) ← ValidCandidates (Q, G, C, A, φ, M, u, i);
12     foreach v ∈ LC(u, M) do
13         if v ∉ M then
14             Map u to v in M;
15             Match (Q, G, C, A, φ, M, i + 1);
16             Remove (u, v) mapping from M;
17         end
18     end
19 end
```

Backtracking based Graph Pattern Matching: Firstly, Ullman [29] proposed the practical algorithm of subgraph isomorphism, which is specifically a backtracking algorithm. Algorithm 1 shows the core structure of the preprocessing-based-enumeration approach. The enumeration process leverages the backtracking technique, which involves exploring all possible solutions by the generated candidates previously and backtracking as soon as it is determined that a solution cannot possibly be valid. After generating candidate sets, an auxiliary data structure which catalyzes the faster computation, stores the query graph (line 2). Then the matching order is generated for the nodes in the query graph (line 3). To enumerate the subgraph isomorphism, a recursive procedure is invoked following the matching order. The subgraph is initiated as empty and gradually it grows until the cardinality of the query graph. It calculates the valid candidates in G that match the i-th node of the pattern graph for step i (line 11). Then, for each vertex v in the computed valid local candidate set, it maps i-th query node to v the in the embedding (line 14) and makes a recursive call to match the following node in the matching order (line 15). Finishing the recursive call, it removes v from M.

III. Related Work

Several prior studies have proposed efficient solutions using parallelization to identify specific query patterns, such as triangles, k-trusses, and squares, in input graphs, demonstrating improved scalability for graphs with billions of edges. In [9], an approach for counting triangles uses a serial implementation that reduces computation through a divide-and-conquer strategy. This method divides the large graph into two subgraphs based on the BFS levels of the vertices of each edge: one subgraph groups edges whose vertices are on the same level, while the other groups edges whose vertices are on different levels. In [14], the implementation exhibits strong scalability for counting triangles in billion-edge graphs within distributed-memory settings. He et al. [17] compared various filtering strategies, including statistics-based, walking-based, and learning-based filtering. They also utilized a graph cache to minimize redundant communications by selecting and storing remote vertices for subgraph matching in a distributed-memory implementation.

Bhattarai et al. [10], in their work on CECI, proposed a method for dividing the target graph into multiple embedding clusters called compact embedding cluster indices. These clusters are then distributed across multiple computing nodes to facilitate subgraph matching. They implemented a pull-based dynamic workload balancing scheme, which provides better scalability for certain datasets. In CFL-Match [11], the authors introduced a framework that decomposes a query graph into a core and a forest to aid in subgraph matching. They developed a compact path-based auxiliary data structure aimed at reducing false positives and redundant candidates. This framework post-

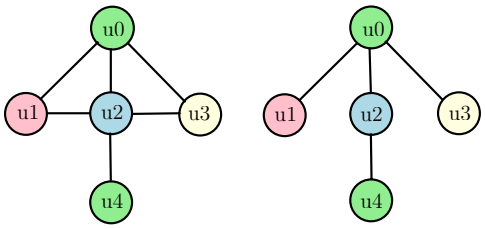


Fig. 2: Illustration of a query graph and its BFS tree.

pones Cartesian products and avoids generating redundant ones, enhancing efficiency. Among parallelized approaches focusing on the query compilation phase, both Dryadic [21] and STMatch [31] optimized query processing using code motion techniques to reduce the repetition of loop-invariant computations during set intersection in subgraph matching. Wei et al. [31] also proposed a GPU-based subgraph matching solution. To address load imbalance and synchronization challenges, they introduced a two-level dynamic load balancing technique and utilized loop unrolling to reduce thread underutilization.

Graph algorithms often struggle to fully leverage the computational power and memory bandwidth of GPUs due to irregular memory access patterns. A key challenge in subgraph isomorphism for large graphs is the extensive memory consumption. In cuTS [32], the authors tackled this issue by building a trie-based data structure to reduce intermediate storage needs.

IV. Methodology

This section outlines the methods used in the solution, which are broadly divided into two main steps: preprocessing and enumeration.

A. Preprocessing

Fig. 2 represents the query graph and the BFS tree created after the BFS traversal of the query graph where the start vertex for the traversal is $u0$. All the edges in the BFS tree are the tree edges and the edges other than the tree edges are the non-tree edges. After the BFS traversal in the query graph, all the vertices $u0, u1, u2, u3, u4$ have their designated levels as $0, 1, 1, 1, 2$ respectively. Some terminologies related to different types of neighbors in the graph by BFS traversal, applied filters in the data graph, and the sequence of steps are as follows.

Three types of neighbors are **Back Neighbor**, **Front Neighbor**, and **Under Level Neighbor**. First, if two vertices are neighbors and they are in the two consecutive levels of the BFS tree, then the vertex in the smaller level is the **back neighbor** to the vertex of the greater level. Additionally, in case of two vertices of the same level and they are connected through a non-tree edge, then the vertex which precedes the other vertex in the BFS order is the **back neighbor** of the following vertex. As per the illustration in Fig. 2, $u0, u1$ are the back neighbors of vertex $u2$, of which $u0$ is in the previous level and $u1$ is in the same level. Second, if the two vertices are in the same level and connected by a non-tree edge, then the vertex which follows the other vertex in the BFS order is a **front neighbor** of the preceding vertex. As illustrated, $u2$ is a front neighbor of vertex $u1$. Third, if two vertices are in the two consecutive levels and connected by tree or non-tree edge, then the vertex in the greater level is the **under level neighbor** of the vertex of the previous level. As projected in Fig. 2, $u4$ will be counted as the under level neighbor of vertex $u2$.

For the filters, **Label and Degree Filter (LDF)** finds a data vertex eligible depending on having greater or equal degree than the corresponding query vertex and the same label. For $u \in Q$ and $v \in G$, $l(u) = l(v)$ and $d(u) \leq d(v)$. Whereas **Neighborhood Label Frequency Filter (NLF)** filters a data vertex if the cardinality of neighbors of the specific label is less than that of the corresponding query vertex. For $u \in Q$ and $v \in G$, $|N(u,l)| \leq |N(v,l)|$ where $N(v,l)$ means the neighbors with label l of vertex v.

1) Matching Order Generation: Matching order is mainly the order of the vertices in the query graph in which the vertices will be matched with the data graph vertices. After filtering by two filters, the start vertex is chosen based on the number of candidates and degree of that vertex, $\text{argmin}_u |C(u)|/d(u)$. Specifically, we applied BFS in the query graph using the start vertex as the source of the graph.

2) Auxiliary Data Structure: After BFS traversal of the query graph, the tree and non-tree edges are identified. Based on the two types of edges, the different types of neighbors are identified for every query vertex and stored in the auxiliary data structure.

3) Candidate Generation & Pruning: Maintaining the matching order, the candidates of any query vertex u are generated based on the candidates of the back neighbors of that query vertex u. The neighbors of the candidates of the back neighbors participate in set intersection for generating the candidates for the query vertex u and are then filtered by NLF and LDF. For every step, the candidates for the query vertices are generated for one BFS level. After generating candidates for one particular level of the query graph, the generated candidates by the back neighbors of the query vertex u are pruned through validating appropriate edge connections with the candidates of the front neighbors. This process is repeated for all the next levels.

4) Reversed BFS based Refinement: Just after completing the level-wise combined generation and pruning of the candidates for all the levels of the query graph, the refinement of candidates is started in the reverse direction of BFS traversal. In the refinement phase, the candidates of the under level neighbors of the query vertex u check the validity of the candidates of u.

B. Enumeration

1) Set Intersection: For mapping a query vertex to the corresponding vertex of the data graph, it is mandatory to generate the valid candidates for the particular query vertex based on the previously mapped vertices in the

particular embedding. This valid candidate generation for the selected query vertex, specifically, the set intersection method among the neighbors of the previously mapped vertices for validating the connections, takes a considerable amount of time compared to other operations. Different optimizations for set intersection can be applied. Malithody et al. [20] proposed a two pointer set intersection and it can be applied when the neighbor lists are sorted. The time complexity of two pointer set intersection method is $\mathcal{O}(n_1 + n_2)$ where n_1 and n_2 are the cardinality of the two sets participating in intersection. Hash based intersection between two sets can be used relaxing the requirements of the adjacency lists to be sorted. For more than two sets in set intersection, the consecutive two sets perform set intersection, their result with the next set and so on. The ordering based on the cardinality of the neighbor sets plays a vital role in reducing the number of computations. The set with minimum cardinality is put into the first position in the order to perform set intersection. This directly reduces the effective size of the result sets for the subsequent intersections and ultimately reduces computation. If there are n numbers of sorted sets denoting n_1, n_2,..., n_{small}, ..., n_n, of which n_{small} is of the smallest length and the term $I_{i,j,k..}$ represents the resultant set by the intersection of the sets $n_i, n_j, n_k, \ldots$. The entire intersection of the n sets can be represented as I and $|I|$ will be of at most $|n_{small}|$.

$$I = n_1 \cap n_2 \cap \ldots \cap n_{small} \cap \ldots \cap n_n \tag{1}$$

$$T = \mathcal{O}(|n_1| + |n_2| + \left|I_{1,2}\right| + |n_3| + \left|I_{1,2,3}\right| + \ldots) \tag{2}$$

$$I_{order} = n_{small} \cap n_1 \cap n2 \ldots \cap n_n \tag{3}$$

$$T_{order} = \mathcal{O}(|n_{small}| + |n_1| + \left|I_{small,1}\right| + |n_2| + \left|I_{small,1,2}\right| + \ldots) \tag{4}$$

According to the two pointer set intersection method, Equations 2 and 4 show the time complexity for Equations 1 and 3, respectively. It is computationally expensive, $\mathcal{O}(n^2)$, to find the combinations of two sets for which the intersection length will be minimum. For Equation 4, the length of any term $I_{small,i,j..}$ is at most $|n_{small}|$, whereas, for Equation 2, the length of any term $I_{i,j,k..}$ is equal as the expected value of the intersection length of the participating sets. The empirical analysis for social network in Table III shows the performance gain achieved by adopting the approach of putting the set of minimum length in the first position of the sequence.

2) Enumeration: The intermediate results are extended by mapping the query vertices along the matching order of query vertices and after finding a match, it backtracks and finds the other eligible embeddings.

V. Experimental Evaluation

This section provides a detailed overview of our extensive experiments.

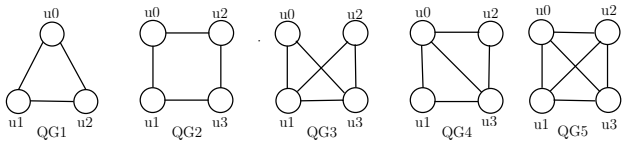


Fig. 3: Query Graphs

A. Dataset Description

Query Graph & Data Graph: The query graphs are illustrated in Fig. 3 which are the representatives of the dense and sparse structures. These query graphs are also used by the projects such as PsgL [26] and CECI [10]. For data graphs, we use six real world graphs and synthetic graphs, spanning from social networks to e-commerce, having different characteristics and hence, providing a better ground to run the extensive experiments. For unlabeled real-world datasets, the vertices of the query graph and the data graph are labeled with the same label. All real graphs are collected from the Stanford Network Analysis Project (SNAP) repository [1].

Category	Dataset	$\|V\|$	$\|E\|$	d_{avg}	d_{max}
Ecomm.	Amazon	0.3M	0.9M	2.03	154
Social	Youtube	1.1M	2.9M	5.21	28752
	DBLP	0.3M	1.0M	4.93	264
	LiveJournal	3.9M	34.6M	17.316	14762
	Orkut	3M	11.7M	76.28	33313
RoadNet	Road-CA	1.9M	2.7M	2.80	12

B. Experimental Environment & Implementation

The code is compiled by **g++ 11.4.0**. We conduct experiments on a Linux machine with **13th Gen Intel Core i7-13700 processor (16 cores)** and **32GB RAM**, also including cache L1: 640 KB, L2: 24 MB, and L3: 30 MB.

Compressed Sparse Row (CSR) format is used to represent both the query graph and the data graph. Our implementation also covers a data structure to efficiently track the list of the nodes in the query graph whose a particular node in the data graph is candidate. The filtering, refinement, and enumeration phase has been parallelized using OpenMP. The execution time results are averaged over three runs.

C. Triangle Enumeration

Triangle pattern is one of the most frequent patterns in most of the real-world datasets [5], [6]. We compare our implementation with TriC [14] for Graph Challenge Kronecker Dataset. Our tool gives correct results for counting the number of triangles. Additionally, this generic subgraph matching can not outperform the specialized triangle counting methods on the GraphChallenge datasets. Because of breaking the automorphism in counting the triangles, the techniques for specialized wedge checking for the triangles, the specialized triangle counting algorithms take less time compared to our implementation. We compare the execution time of the serial implementation of different

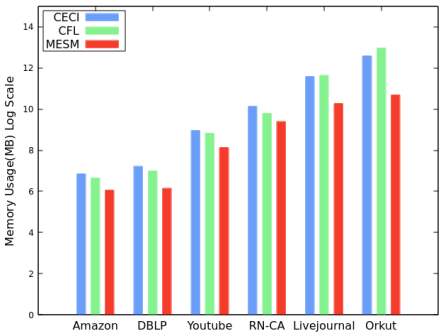


Fig. 4: Memory Usage for QG1

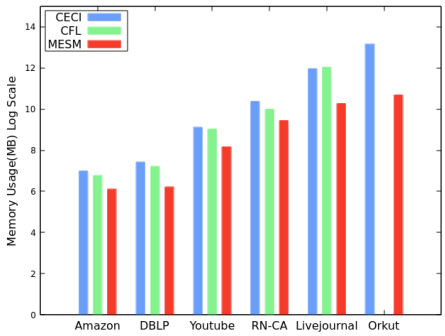


Fig. 5: Memory Usage for QG3

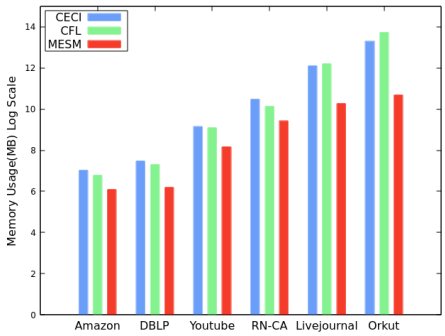


Fig. 6: Memory Usage for QG5

techniques for **QG1** in Table I. For Orkut dataset, CECI outperforms MESM by 1.5× in terms of enumeration time but uses 4× more memory than MESM.

TABLE I: Enumeration Time (second) Comparison

Dataset	CECI	CFL-Match	MESM
Amazon	0.08	0.07	0.057
DBLP	0.23	0.51	0.22
Youtube	1.59	29.38	2.89
LiveJournal	30.26	186.93	36.12
Orkut	217.67	2642.77	322.61

D. Other Queries

For evaluating the adaptability of our implementation with diverse nature of query graphs, we run experiments using comparatively dense query structures k-truss, cliques, and spare query structures, for example, squares. Considering all the vertices with the same label, we demonstrate the residual data graph size in terms of remaining vertex count for the query graph **QG4** after applying the filters, for example, NLF, LDF mentioned above and the techniques, for example, Candidate Generation and Pruning (CGP), and Reversed BFS based Refinement (RBR) of the candidates in Table II. Additionally, in the triangle pattern, there is only a set intersection operation among the neighbors of the previous two vertices to get the valid candidates of the last vertex in the matching order. For other query graphs with increased number of vertices and edges, there are multiple set intersections among two or more sets. For **QG4** in Fig. 3, to generate the valid candidates for query vertex $u3$, the set intersection among the neighbors of mapped vertices for query vertices $u0, u1$ and $u2$ is computed. For the set intersection operation among more than two sets, ordering among the sets causes a reduction in the enumeration time. The reduction in enumeration time for the serial implementation of MESM involving the set intersection ordering is shown in the Table III for the query graph **QG4**.

E. Memory Efficiency

We characterize the memory usage of two competitor techniques for the entire computation portion of subgraph matching and their comparison with our implementation shown in Fig. 4, 5, and 6. The comparison uses logarithmic scale with base two due to the significant memory usage

TABLE II: Residual Data Graph Size (Remaining Unique Vertex Count) After Applying NLF, LDF, CGP, and RBR

Dataset	Number of Nodes	NLF + LDF	NLF+LDF+ CGP+RBR
Amazon	334,863	151,253	144,825
DBLP	317,080	201,628	201,066
Youtube	1,134,890	527,184	519,265
LiveJournal	3,997,962	3,195,231	3,192,145
Orkut	3,072,441	3,004,605	3,004,605

TABLE III: Enumeration Time (second) Comparison of Set intersection Operation with and without ordering

Dataset	Enumeration Time without ordering	Enumeration Time with ordering
Amazon	0.1253	0.1238
DBLP	1.97	1.80
Youtube	52.24	46.32
LiveJournal	1256.43	1131.12
Orkut	12822.7	11844.5

gap among the implementations. We conduct experiments using the query graphs mentioned in Fig. 3 and real-world datasets for memory efficiency analysis. The experimental analysis is conducted using the same graph representation strategies and the same ground of comparison for all the representative implementations. The space complexity of CECI and CFL-Match are $\mathcal{O}(|E_q| \times |E_g|)$ and $\mathcal{O}(|V_q| \times |E_g|)$. To accommodate NTE candidates, the space requirement is comparatively high in CECI. Conversely, our implementation has space complexity of $\mathcal{O}(|V_q| \times |V_g|)$. We use **Valgrind** [2] with the heap profiler **Massif** to track the amount of heap memory the program used. As per the experiments, MESM outperforms all other implementations irrespective of the query graphs and the data graphs due to its low memory usage. Through the empirical analysis, it is seen that the difference in the memory usage grows with the size of the data graph. For the Orkut dataset and the query graph **QG5**, our implementation achieves 5.5× reduction in terms of memory usage as compared to CECI and 8.2× against CFL-Match. CFL-Match can not handle the Orkut dataset for the query **QG3** due to larger search space. Compared with CECI and CFL-Match from the perspective of memory usage, our implementation achieves $1.9-8.2\times$ reduction in memory usage.

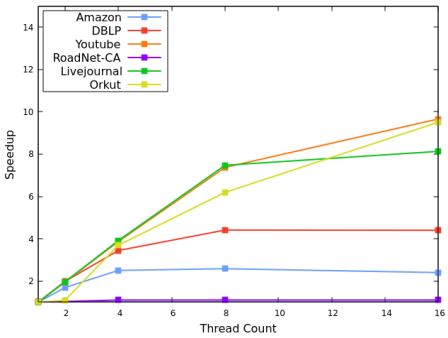


Fig. 7: Scalability for QG1

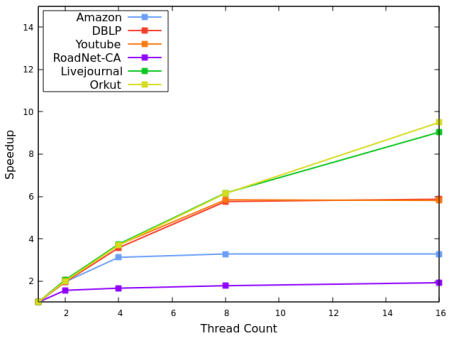


Fig. 8: Scalability for QG3

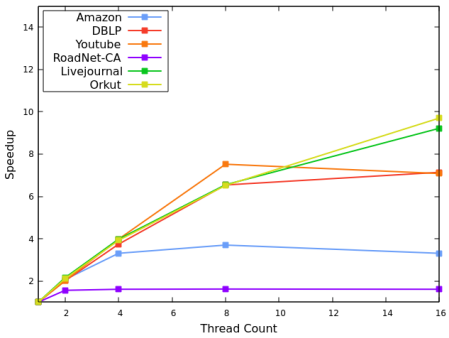


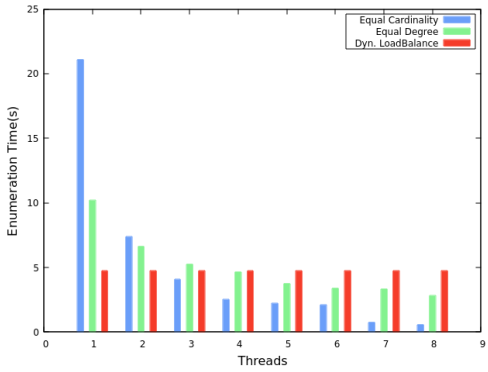
Fig. 9: Scalability for QG5

F. Workload Balancing

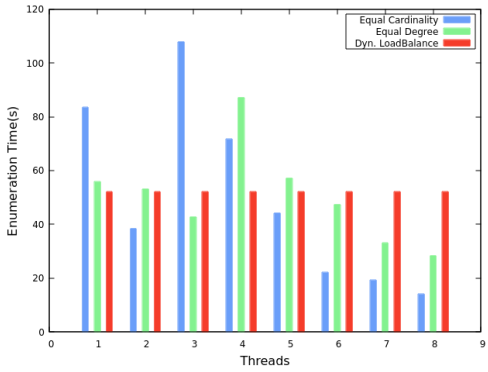
The skewed-degree distribution for the social network datasets is one of the responsible factors for workload imbalance. The workload for a particular thread mostly depends on the backtracking depth and the number of valid candidates generated in each depth. We compare three approaches, for example, cardinality, degree and dynamic load balancing to investigate the workload imbalance shown in Fig. 10. First, we divide the equal number of candidates of the start query vertex into groups and they are handed over to the involved threads. Though the number of vertices allocated per thread is approximately the same, the amount of workload among the threads is not evenly distributed and causes workload imbalance. Second, we apply degree based division of the candidates. The candidate vertices having approximately the same summation of degrees are grouped together and we get a significant speedup and reduction in the enumeration time as compared to the first approach. But workload imbalance is clearly visible in this approach. Third, we implement the dynamic scheduling to distribute workload among the threads at the cost of a thread coordination overhead. This workload distributing technique outperforms all the previous approaches of workload distribution for all the query graphs. In Fig. 10(a) and 10(b), the comparison of three approaches is demonstrated for the Livejournal and Orkut dataset for triangle enumeration using 8 threads.

G. Scalability

We evaluate the strong scalability of our implementation for different query graphs demonstrated in Fig. 7, 8, and 9. In the case of finding the triangular pattern in the Amazon and DBLP dataset, after involving 8 workers, the speedup trend gets flat out because of inadequate workload for the threads. But for the other datasets, the scalability plot maintains approximately a linear speedup. For the Youtube and Livejournal datasets, CECI shows poor scalability with increased number of workers due to the communication overhead in the distributed setting, whereas our implementation maintains a decent scaling for an increased number of threads. The result for the Orkut dataset shows better scalability with the increasing number of threads. Conversely, the road-network does not show power-law



(a)



(b)

Fig. 10: Workload imbalance for (a) Livejournal (b) Orkut

characteristic and also has long diameter compared to other datasets. The scalability for the road-network datasets is limited due to poor locality in vertex sorting, inadequate workload distribution, and other previously mentioned factors.

VI. Conclusion

We present an approach that leverages parallelization in a shared-memory system to efficiently enumerate pattern graphs within diverse real-world graph datasets. Our method prioritizes memory efficiency, an area often overlooked by previous solutions. The results demonstrate a significant improvement in memory usage over state-of-the-art methods, while maintaining comparable execution times.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation (NSF) under Award Number 2323533.

REFERENCES

[1] Stanford network analysis project repository. https://snap.stanford.edu/data/.

[2] Valgrind instrumentation framework. https://valgrind.org/.

[3] Sherif Abdelhamid, Md. Maksudul Alam, Richard Alo, Shaikh Arifuzzaman, et al. {CINET} 2.0: {A} cyberinfrastructure for network science. In *10th {IEEE} International Conference on e-Science, eScience 2014, Sao Paulo, Brazil, October 20-24, 2014*, pages 324–331, 2014.

[4] S. Arifuzzaman, H. S. Arikan, M. Faysal, M. Bremer, J. Shalf, and D. Popovici. Unlocking the potential: Performance portability of graph algorithms on kokkos framework. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 526–529, Los Alamitos, CA, USA, May 2024. IEEE Computer Society.

[5] S. Arifuzzaman, M. Khan, and M. Marathe. Fast parallel algorithms for counting and listing triangles in big graphs. *ACM Trans. Knowl. Discov. Data (TKDD)*, 14(1):5:1–5:34, 2019.

[6] S Arifuzzaman, Maleq Khan, and Madhav Marathe. A space-efficient parallel algorithm for counting exact triangles in massive networks. In *Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications*, August 2015.

[7] Shaikh Arifuzzaman and Bikesh Pandey. Scalable mining, analysis, and visualization of protein-protein interaction networks. In *International Journal of Big Data Intelligence (IJBDI)*, volume 6, pages 176–187. Inderscience, 2019.

[8] A. Azad, A. Buluc, and A. Pothen. Computing maximum cardinality matchings in parallel on bipartite graphs via tree-grafting. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):44–59, Jan 2017.

[9] David A Bader. Fast triangle counting. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2023.

[10] Bibek Bhattarai, Hang Liu, and H Howie Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1447–1462, 2019.

[11] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1199–1214, 2016.

[12] Md Abdul M. Faysal, Shaikh Arifuzzaman, Cy Chan, Maximilian Bremer, Doru Popovici, and John Shalf. Hypc-map: A hybrid parallel community detection algorithm using information-theoretic approach. In *2021 IEEE High Performance Extreme Computing Conference (HPEC 2021)*, 2021.

[13] Md Abdul Motaleb Faysal, Maximilian Bremer, Cy Chan, John Shalf, and Shaikh Arifuzzaman. Fast parallel index construction for efficient k-truss-based local community detection in large graphs. In *In Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 132–141, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Sayan Ghosh and Mahantesh Halappanavar. Tric: Distributed-memory triangle counting by exploiting the graph structure. In *2020 IEEE high performance extreme computing conference (HPEC)*, pages 1–6. IEEE, 2020.

[15] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 7(13):1379–1380, 2014.

[16] Juris Hartmanis. Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). *Siam Review*, 24(1):90, 1982.

[17] Jiezhong He, Zhouyang Liu, Yixin Chen, Hengyue Pan, Zhen Huang, and Dongsheng Li. Fast: A scalable subgraph matching framework over large graphs. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2022.

[18] Yaohui Lei. A survey of graph and subgraph isomorphism problems. 2004.

[19] Xiao Liang, Chenxu Wang, and Guoshuai Zhao. Enhancing content marketing article detection with graph analysis. *IEEE Access*, 7:94869–94881, 2019.

[20] Vikram S Mailthody, Ketan Date, Zaid Qureshi, Carl Pearson, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. Collaborative (cpu+ gpu) algorithms for triangle counting and truss decomposition. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[21] Daniel Mawhirter, Samuel Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. Dryadic: Flexible and fast graph pattern matching at scale. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 289–303. IEEE, 2021.

[22] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[23] Steven Noel. A review of graph approaches to network security analytics. *From Database to Cyber Security: Essays Dedicated to Sushil Jajodia on the Occasion of His 70th Birthday*, pages 300–323, 2018.

[24] Naw Safrin Sattar and Shaikh Arifuzzaman. Community detection using semi-supervised learning with graph convolutional network on gpus. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 5237–5246, 2020.

[25] Naw Safrin Sattar and Shaikh Arifuzzaman. Scalable distributed louvain algorithm for community detection in large graphs. *Journal of Supercomputing*, 78, 2022.

[26] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of Data*, pages 625–636, 2014.

[27] Shixuan Sun and Qiong Luo. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1083–1098, 2020.

[28] Johan Ugander, Lars Backstrom, and Jon Kleinberg. Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections. In *Proceedings of the 22nd international conference on World Wide Web*, pages 1307–1318, 2013.

[29] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[30] Shihan Wang and Takao Terano. Detecting rumor patterns in streaming social media. In *2015 IEEE international conference on big data (big data)*, pages 2709–2715. IEEE, 2015.

[31] Yihua Wei and Peng Jiang. Stmatch: accelerating graph pattern matching on gpu with stack-based loop optimizations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13. IEEE, 2022.

[32] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. cuts: scaling subgraph isomorphism on distributed multi-gpu systems using trie based data structure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.