2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)

MPMatch: A Multi-Core Parallel Subgraph Matching Algorithm

1st Xin Jin
East China Normal University
Shanghai, China
xinjin@stu.ecnu.edu.cn

2nd Longbin Lai
The University of New South Wales
Sydney, Australia
llai@cse.unsw.edu.au

***Abstract*—Subgraph Matching is a fundamental problem in graph analysis, and is widely used in many application scenarios in biology, chemistry and social network. Given a data graph and a query graph, subgraph matching aims to compute all subgraphs of the data graph that are isomorphic to the query graph. The problem is computationally expensive as the core operation it depends on, known as subgraph isomorphism, is NP-complete. In recent years, graph is increasing extensively and it is hard for to compute subgraph matching on massive graph data using existing serial algorithm. Meanwhile, there exist distributed solutions, but they are mostly limited to the case where the graphs are unlabelled. In response to this gap, we study the subgraph matching problem in the multi-core environment. From the algorithm level, we propose a multi-core parallel subgraph matching algorithm called MPMatch. From the research level, we explore the concurrent allocation of subgraph matching search space to approach load balancing. We conduct extensive empirical studies on real and synthetic graphs to demonstrate that our techniques improve the performance of serial subgraph matching algorithm via parallelization and well-developed load balancing schema.**

***Index Terms*—subgraph isomorphism, parallelism of multi-core, load balance**

I. INTRODUCTION

Giving a labelled query graph q and a labelled data graph G, subgraph matching aims to compute all subgraphs of G that are isomorphic to q. Subgraph matching is one of the most fundamental problems in graph analysis, and is widely used in many application scenarios such as protein interaction network analysis [11], chemical compound search [19] and social network analysis [16]. Despite the wide range of applications, subgraph matching is computationally intensive as the core operation it depends on, known as *subgraph isomorphism*, is NP-complete. Therefore, researchers have been exploring efficient algorithms for subgraph matching in the past decades.

State-of-the-arts. Existing solutions on subgraph matching are mainly categorized into serial algorithms and distributed algorithms. The serial algorithms follow Ullmann's backtracking approach [18], which recursively matches query vertices to data vertices following a given matching order of query vertices. VF2 [4] proposes to generate a matching order by selecting a vertex connected to already selected ones so that false-positive candidates can be pruned at an early stage, especially those caused by redundant Cartesian products. Infrequence is used in candidates reduction in QuickSI [14] that proposes to first process infrequent query vertices and edges in data graph. GraphQL [6] and SPath [20] exploit neighborhood-based filter to reduce the candidates of query vertices. Turbo$_{ISO}$ [5] proposes to merge similar vertices with the aim to provide a more effective matching order. An efficient matching order can significantly reduce the number of infeasible intermediate results, especially the ones caused by unpromising Cartesian products. The most state-of-the-art work `CFLMatch` [3] features: (1) Core-forest-leaf matching order that aims to postpone Cartesian products. The framework decomposes a query graph into a core part, a forest part and the remaining leaf part, and then processes the matching following the order of core, forest and leaves. The leaf part is the main culprit of the costly Cartesian products. (2) A compact auxiliary path-based data structure called CPI with size $O(E|G| \times |V(q)|)$.

The distributed algorithms can be categorized into three classes, where the join paradigm has been demonstrated as the most popular strategy. The join paradigm strategy treats the matches of the query's certain subgraphs as the base relations to join and expands the result from an initial base relation by joining one base relation each round. StarJoin [17], PSgL [15], TwinTwigJoin [7] and CliqueJoin [8] belong to the join paradigm. The algorithms in join paradigm mainly differ in the base relation. For example, PSgL uses a single edge as the base relation while CliqueJoin uses clique and star. The algorithms using the second strategy, such as BigJoin [13] and CrystalJoin [12], expand the result by matching one query vertex each round. The algorithms using the third strategy, such as MultiwayJoin [2], attempt to partition the searching space into different workers, which allows each worker to compute the task locally without further communication.

Motivations. Graph is growing massively these days, and it is challenging to compute subgraph matching on such massive graph data using serial algorithms. Existing solutions, however, fail to fully address this issue due to their own constraints. The serial algorithms, especially `CFLMatch`, are efficient for small to medium scale data graphs, but they are developed and designed in a stand-alone environment, and thus cannot fully utilize ever-developing hardware (e.g. multi-core) to improve the scalability while handling large graph. The distributed algorithms are mainly developed for the unlabelled case (both

2473-3490/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDEW.2019.000-6

query graph and data graph are unlabelled), and can not utilize the large filtering power of the label. Motivated by this, we are developing a multi-core parallel subgraph matching algorithm in this paper. The reasons that we target a multi-core context are twofold. Firstly, researchers have explored compact storage scheme for graph data that enables trillion-scale processing power in one single machine [10]. Secondly, there are normally not many results of labelled subgraph matching (much fewer than the unlabelled case), thus it may be unwise to pay the expensive communication cost in the distributed context.

Challenges and Our approaches. As `CFLMatch` is so far the most advanced serial subgraph matching algorithm, we will develop our multi-core algorithm based on `CFLMatch`. It is non-trivial to achieve this due to the following two challenges.

Challenge 1: Concurrent Allocation of Subgraph Matching Search Spaces. `CFLMatch` recursively matches query vertices to a data graph by following a certain matching order of query vertices. In the sequential design, we can not map a query vertex without knowing the mapping results of previous query vertices in the matching order, which increases the difficulty and complexity of the parallel design.

Our Approach: Decompose Matching process. We decompose matching process into two phases. In the first phase, we map first few query vertices in the matching order to generate pre-mapping results in a stand-alone environment. Theses pre-mapping results are considered as subtasks. In the second phase, we distribute these subtasks to different workers to complete the full matching process.

Challenge 2: Load Balancing. When distributing pre-mapping results to different workers, we should balance each worker's workload for scalability consideration.

Our Approach: Estimate Subtask Workload. Given a pre-mapping result (a subtask), we propose a method of estimating its number of subgraph isomorphic embeddings as its workload. The subtask whose workload exceeds limitation will be further decomposed by mapping more query vertices.

Contributions. Our main contributions are summarized as follows.

- We develop an efficient and load-balancing algorithm `MPMatch` for parallel conducting subgraph matching.
- We explore the concurrent allocation of subgraph matching search space by decomposing matching process into two phases for subtask generation and subtask processing, respectively. We design a load balancing scheme by estimating subtask workload. The subtask whose workload exceeds limitation will be further decomposed into finer-grained subtasks. Moreover, a greedy algorithm is adopted in subtask distribution for load balancing.
- Our extensive performance studies on large real and synthetic graphs demonstrate that our techniques greatly improve the performance of `CFLMatch`. The experiment results also show the effectiveness of our load balancing scheme.

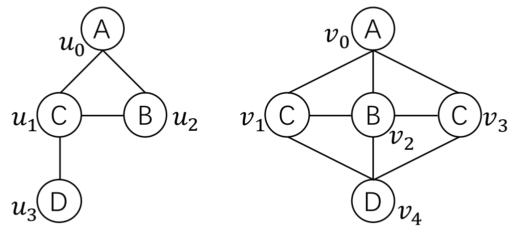
Organization. The rest of the paper is organized as follows. Section 2 defines the problem of subgraph matching and introduces preliminary knowledge. Section 3 presents our algorithms for conducting subgraph matching in multi-core environments. Specifically, our design for concurrent allocation of searching space is in Section 3.2, followed by our load balancing schema in Section 3.3. Experimental results are reported in Section 4, and Section 5 concludes the whole paper.

II. PRELIMINARIES

A. Problem Definition

A graph G is represented as a tuple $G = (V, E, l, \Sigma)$, where $V(G)$ is the set of vertices, $E(G) \subset V \times V$ is the set of edges in G, Σ is the set of labels, and l is a labelling function that assigns each vertex $v \in V$ a label in Σ, denoted $l_G(v)$. We focus on *undirected, labelled, connected and simple* graphs in this paper. Here, a simple graph is a graph with no self-loops and no parallel edges. Due to extendability of `CFLMatch`, our techniques can also be extended to handle edge-labelled and directed graphs. We denote the number of vertices and edges in G by $|V(G)|$ and $|E(G)|$, respectively. The set of neighbors of $v \in V(G)$ in G is denoted by $N_G(v) = \{v' \in V(G) | (v, v') \in E(G)\}$ and the degree of v, denoted by $d_G(v)$, is the number of neighbors of v, i.e., $d_G(v) = |N_G(v)|$. A subgraph G' of G induced by a subset V_s of V, is $G[V_s] = (V_s, \{(v, v') \in E | v, v' \in V_s\}, l, \Sigma)$. We denote the data graph as G and the query graph as q.

Definition 2.1: (Subgraph Isomorphism) Given graphs $q = (V(q), E(q), l, \Sigma)$ and $G = (V(G), E(G), l, \Sigma)$, q is subgraph isomorphism to G iff there is an *injective* mapping M from $V(q)$ to $V(G)$ such that $\forall u \in V(q)$, $l_q(u) = l_G(M(u))$ and $\forall (u, u') \in E(q)$, $(M(u), M(u')) \in E(G)$, where $M(u)$ is the vertex to which u is mapped. If q is subgraph isomorphism to G, q is called a subgraph of G. □



(a) Query q (b) Data graph G

Fig. 1. Subgraph Matching

We call an injective mapping from vertices of q to vertices in G as a *subgraph isomorphism embedding* of q in G. For example, consider the query graph q in Fig. 1(a) and the data graph G in Fig. 1(b) where $\{A, B, C, D\}$ is the set of vertex-labels. Since there is a subgraph isomorphism embedding $M(u_0 \rightarrow v_0, u_1 \rightarrow v_1, u_2 \rightarrow v_2, u_3 \rightarrow v_4)$, q is subgraph isomorphism to G.

Given a query graph q and a large data graph G, in this paper we study the *Subgraph Matching* problem which aims to compute all subgraphs of G that are isomorphic to q. For example, consider the query graph q in Fig. 1(a) and the data graph G in Fig. 1(b), there are two subgraph isomorphism embeddings of q in G, which maps (u_0, u_1, u_2, u_3) to (v_0, v_1, v_2, v_4) and (v_0, v_3, v_2, v_4), respectively.

Definition 2.2: (Pre-mapping) We use pre-mapping result m^+ to denote the partial subgraph isomorphism embedding which only maps first few vertices from q to G in the matching order. Given a pre-mapping result m^+, the worker will map the remaining vertices from q to G in the matching order. □

Example 2.1 Consider the query graph q in Fig. 1(a) and the data graph G in Fig. 1(b), given a pre-mapping result $m^+(u_0 \rightarrow v_0, u_2 \rightarrow v_2)$ and the matching order (u_0, u_2, u_1, u_3), the worker will map the remaining query vertices $\{u_1, u_3\}$ to G to get the full subgraph isomorphism embeddings $M(u_0 \rightarrow v_0, u_2 \rightarrow v_2, u_1 \rightarrow v_1, u_3 \rightarrow v_4)$ and $M(u_0 \rightarrow v_0, u_2 \rightarrow v_2, u_1 \rightarrow v_3, u_3 \rightarrow v_4)$. □

B. CPI Structure

we will use the compact auxiliary path-based data structure CPI first addressed in [3] for accurately estimating the number of embeddings of pre-mapping results and for generating subgraph isomorphic embeddings. CPI is defined regarding a *BFS tree* q_T of q and has the same structure as q_T. To differentiate the vertices of CPI from the vertices of q and G, we call vertices of CPI as *nodes*. Similar to the parent-child relationships in q_T, any two adjacent nodes in CPI also have a parent-child relationship. We use $u.Child$ to denote the set of child nodes of u in CPI. The structure of CPI is as follows.

- Each node u of CPI has a candidate set, denote $u.C$, which stores all vertices of G to which u can be mapped.
- There is an edge between $v \in u.C$ and $v' \in u'.C$ for adjacent nodes u and u' in CPI if and only if (v, v') exists in G.

We use $N_u^{u'}(v)$ to denote the adjacency list of v regarding (u, u') in CPI. For example, Fig. 2 shows the CPI constructed for the query graph q in Fig. 1(a) over the data graph G in Fig. 1(b). We have $N_{u_0}^{u_1}(v_0) = \{v_1, v_3\}$. The candidate set of u_1 is $u_1.C = \{v_1, v_3\}$. The child node set of u_0 is $u_0.Child = \{u_1, u_2\}$

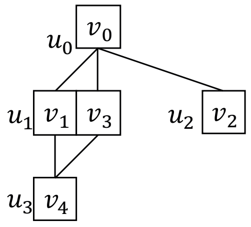


Fig. 2. Example CPI

III. OUR APPROACH

In this section, we propose a new multi-core parallel subgraph matching algorithm `MPMatch`. In the following, we first introduce a naive solution. Then we show our method of decomposing matching process in Section 3.2 and discuss how to estimate the workload of a pre-mapping result in Section 3.3. In Section 3.4, we give our method of generating pre-mapping results.

A. A Naive Solution

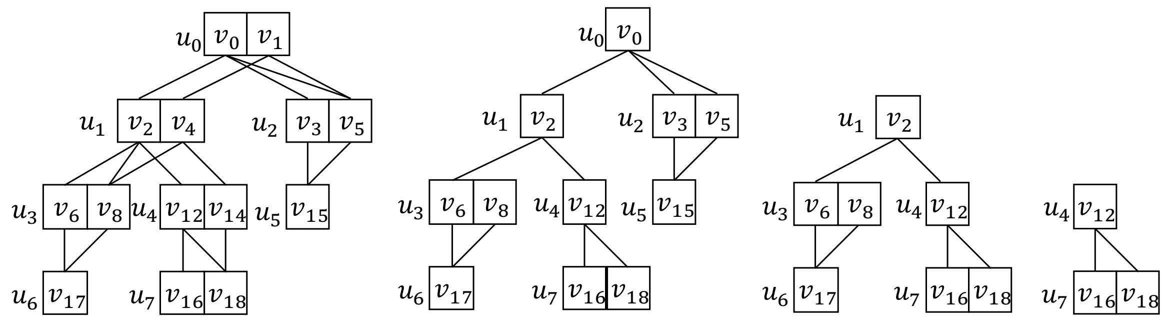
One naive solution is mapping the first query vertex to G to generate a set of pre-mapping results. Then distribute these pre-mapping results to different workers to finish the matching process. However, this solution suffers from two drawbacks. First, it can not guarantee the sufficient pre-mapping results. Some workers may have no subtask to process. Second, the workload of different pre-mapping results may vary a lot, which leads to load imbalance. For example, given the CPI structure in Fig. 3(a) and three workers, if we use the naive solution, there are two pre-mapping results $m_1^+(u_0 \leftarrow v_0)$ and $m_2^+(u_0 \leftarrow v_1)$. At least one worker has no subtask to process and the workload of m_1^+ is obviously larger than m_2^+.

B. Matching Process Decomposition

Algorithm 1 MPMatch(query q, data Graph G, heap $\mathcal{H}_w$)

```
1: CPI ← CPI-Construct($q, G$)
2: $\mathcal{H}^+ \leftarrow$ Pre-Match($q, G$, CPI)
3: $\mathcal{M} \leftarrow \emptyset$;
4: while $\mathcal{H}^+ \neq \emptyset$ do
5:     $m^+ \leftarrow \mathcal{H}^+$.Pop(); $w \leftarrow \mathcal{H}_w$.Pop();
6:     $w.M^+ \leftarrow w.M^+ \cup \{m^+\}$;
7:     $w.load \leftarrow w.load + m^+.load$;
8:     $\mathcal{H}_w$.Push($w$);
9: end while
10: for each worker $w$ in $\mathcal{H}_w$ par do
11:     for each $m^+$ in $w.M^+$ do
12:         $n \leftarrow |V_q| - |m^+|$
13:         $\mathcal{M} \leftarrow \mathcal{M} \cup$ CFLMatch($m^+, n$, CPI);
14:     end for
15: end for
16: return $M$;
```

In this subsection, we explore the concurrent allocation of subgraph matching search space. We decompose matching process into two phases. In the first phase, we map first few query vertices in the matching order to generate pre-mapping results that we view as subtasks in a stand-alone environment. We will push all the pre-mapping results into a max-heap $\mathcal{H}^+$ and each pre-mapping result m^+ has workload value, denoted $m^+.load$. In the second phase, we distribute these subtasks to different workers to finish the matching process. We push all the workers into a min-heap $\mathcal{H}_w$ and each worker w has a set of pre-mapping results and total workload value, denoted $w.M^+$ and $w.load$ respectively. The pseudocode is shown in



(a) CPI Structure (b) $C^{v_0}_{u_0}$ (c) $C^{v_2}_{u_1}$ (d) $C^{v_{12}}_{u_4}$

Fig. 3. Example CPI for Workload Estimation

Algorithm 1, denoted `MPMatch`.

Given a query graph q and a data graph G, we first compute *CPI Structure* using the method CPI-Construct(q, G) addressed in [3] (line 1). Then we generate a set of pre-mapping results and push them into $\mathcal{H}^+$ (line 2). We will introduce how to compute $\mathcal{H}^+$ using Pre-Match(q, G, CPI) later. Each entry $m^+ \in \mathcal{H}^+$ represents a pre-mapping result. In line 4-9, we distribute pre-mapping results to workers in a greedy way.

In each iteration (line 4-9), we distribute the pre-mapping result m^+ with the maximum workload value in $\mathcal{H}^+$ to the worker w with the minimum workload value and update $w.load$ by adding m^+, until all pre-mapping results in $\mathcal{H}^+$ has been distributed. Each entry w in the min-heap $\mathcal{H}_w$ represents a worker.

In line 10-15, each worker conduct subgraph isomorphism embeddings in parallel. Given a pre-mapping result m^+, the worker uses CFLMatch(m^+, n, CPI) first addressed in [3] to get the subgraph isomorphism embeddings (line 13). We use a simple modification of the original CFLMatch method which maps all query vertices in the matching order. We use two arguments m^+ and n to control the query vertices to be mapped: we map n query vertices from the first unmapped one in the matching order.

C. CPI-based Workload Estimation

In this subsection, we discuss how to estimate the workload of a pre-mapping result using the CPI structure. The workload is the cost of conducting subgraph isomorphism embeddings based on the given pre-mapping result.

Recall that CPI is defined regarding a *BFS tree* q_T of q and has the same structure as q_T. We use C^v_u to denote a partial CPI structure whose root vertex is u and u has been mapped to v. For example, given the CPI structure in Fig. 3(a), we show the partial CPI structure $C^{v_0}_{u_0}$ and $C^{v_2}_{u_1}$ in Fig. 3(b) and Fig. 3(c) respectively.

We use $C^v_u.load$ to denote the workload of mapping all query vertices in C^v_u except u. We use the number of embeddings of the C^v_u to estimate $C^v_u.load$. If u is a leaf node in CPI, we have $C^v_u.load = 1$. Although the embeddings is not explicitly stored in C^v_u, we can compute $C^v_u.load$ in a bottom-up way using a dynamic programming algorithm. We can define $C^v_u.load$ as:

$$C^v_u.load = \prod_{u' \in u.Child} \sum_{v' \in N^{u'}_u(v)} (C^{v'}_{u'}.load) \tag{1}$$

Note that we compute all $C^v_u.load$ immediately after the CPI structure has been constructed, not in real-time. Having defined the partial CPI structure, we show how to compute the workload of a given pre-mapping result in Algorithm 2.

Algorithm 2 Workload-Estimate(pre-mapping result m^+, CPI)

```
 1: $load \leftarrow 1$
 2: for (query vertex $u$, data vertex $v$) $\in m^+$ do
 3:     for child node $u' \in u.Child$ and $u'$ is not mapped do
 4:         $load_{tmp} \leftarrow 0$
 5:         for child candidate node $v' \in N^{u'}_u(v)$ do
 6:             $load_{tmp} \leftarrow load_{tmp} + C^{v'}_{u'}.load$
 7:         end for
 8:         $load \leftarrow load \times load_{tmp}$
 9:     end for
10: end for
11: return $load$
```

Given a pre-mapping result m^+ and the CPI structure, intuitively, $m^+.load$ should be multiplication of all $C^v_u.load$ for each mapping pair $(u \leftarrow v)$ in m^+. But for a child node u' of u, u' may also be mapped and the workload of partial CPI structure whose root node is u' has been counted in the workload of partial CPI structure whose root node is u. So for each mapping pair $(u \leftarrow v)$, we first add up all $C^{v'}_{u'}.load$ where u' is an unmapped child node of u and $v' \in N^{u'}_u(v)$ (line 6) and then multiply all results (line 8).

Example 3.1 Given the CPI structure and partial CPI structures in Fig. 3, $C^{v_{12}}_{u_4}.load = 2$, $C^{v_2}_{u_1}.load = 4$ and $C^{v_0}_{u_0}.load =$

8. Given a pre-mapping result $m^+(u_0 \leftarrow v_0, u_1 \leftarrow v_2)$, $m^+.load = 6$. □

D. Pre-mapping Generation

In this subsection, we will introduce our method of generating pre-mapping results. We use a max-heap $\mathcal{H}^+$ to maintain our pre-mapping results. We ensure that the workload of each pre-mapping result not exceeds the given limitation δ. The algorithm to generate pre-mapping results is shown in Algorithm 3, denoted Pre-Match.

Algorithm 3 Pre-Match(query q, data Graph G, CPI)

```
1:  $\mathcal{H}^+ \leftarrow \emptyset$
2:  $load^+ \leftarrow 0$
3:  for pre-mapping $m^+ \in$ CFLMatch($\emptyset$, 1, CPI) do
4:      $m^+.load \leftarrow$Workload-Estimate($m^+$, CPI);
5:      $load^+ \leftarrow load^+ + m^+.load$
6:      $\mathcal{H}^+$.Push($m^+$);
7:  end for
8:  $m^* \leftarrow \mathcal{H}^+$.Pop();
9:  while $m^*.load > \delta$ do
10:     for pre-mapping $m^+ \in$ CFLMatch($m^*$, 1, CPI) do
11:         $m^+.load \leftarrow$Workload-Estimate($m^+$, CPI);
12:         $\mathcal{H}^+$.Push($m^+$);
13:     end for
14:     $m^* \leftarrow \mathcal{H}^+$.Pop();
15: end while
16: $\mathcal{H}^+$.Push($m^*$);
17: return $\mathcal{H}$;
```

Given a query graph q, a data graph G and their CPI structure, we first initial the heap $\mathcal{H}^+$ (line 3-7). To generate first few pre-mapping results, we map the root query vertex to G by using CFLMatch($\emptyset$, 1, CPI). Then for each pre-mapping result m^+, compute its workload by using Workload-Estimate(m^+, CPI). We use $load^+$ to denote the total workload.

In each iteration (line 9-15), we first get the pre-mapping result m^* with the maximum workload (line 14) and expand it by mapping one more query vertex (line 10-13), if its workload exceeds the given limitation δ, i.e., $m^+.load > \delta$ (line 9). Each expanded pre-mapping results, we update its workload and push it into $\mathcal{H}^+$. The iteration will stop if the workload of m^* is less than δ.

Workload Limitation. Intuitively, to ensure that each worker has at least one subtask, we simply define δ as:

$$\delta = \frac{load^+}{|\mathcal{H}_w|} \tag{2}$$

Note that If δ is too small, it will cost a lot to generate pre-mapping results in a stand-alone environment and total processing time may increase. If δ is too big, it may lead to load-imbalance. So how to choose an appropriate δ might be an interesting issue to be further investigated.

Example 3.2 Given the CPI structure in Fig. 3, the matching order $(u_0, u_2, u_1, u_3, u_4, u_5, u_6, u_7)$ and $\delta = 4$, then $load^+ = C^{v_0}_{u_0}.load + C^{v_1}_{u_0}.load = 9$. In the first iteration, the pre-mapping result m^* with the maximum workload is $(u_0 \leftarrow v_0)$ and $m^*.load = 8 > \delta$, we map query vertex u_2 and get two new pre-mapping results $(u_0 \leftarrow v_0, u_2 \leftarrow v_3)$ and $(u_0 \leftarrow v_0, u_2 \leftarrow v_5)$. In the second iteration, the pre-mapping result m^* with the maximum workload is $(u_0 \leftarrow v_0, u_2 \leftarrow v_3)$ and $m^*.load = 4 \leqslant \delta$, so we return $\mathcal{H}^+$ with three pre-mapping results: $(u_0 \leftarrow v_1)$, $(u_0 \leftarrow v_0, u_2 \leftarrow v_3)$ and $(u_0 \leftarrow v_0, u_2 \leftarrow v_5)$.

IV. EXPERIMENTS

We conduct extensive performance studies to evaluate the efficiency of our multi-core parallel subgraph matching algorithm and our load-balancing schema. Specifically, we evaluate the following two algorithms.

- `CFLMatch`: the most state-of-the-art algorithm which we extend to the multi-core environment.
- `MPMatch`: our multi-core parallel subgraph matching algorithm (see Section 3.2).

Both algorithms are implemented in Rust. Experiments are conducted on a machine with an Intel i7 3.60GHz CPU and 64GB memory.

#Workers. We vary the number of workers in `MPMatch` from 3 to 6, 12, 18, 24 and 30, with #workers = 30 being the default.

Metrics. For each testing, we run an algorithm three times, and *report the avertage CPU time in milliseconds for processing each query graph.* Note that, we set the time limit for processing a query set to 3 hours (i.e., 1.08×10^7 ms). If an algorithm cannot finish within the time limit, then we plot its processing time as ”INF”.

A. Comparing with CFLMatch

In this subsection, we evaluate `MPMatch` against `CFLMatch`.

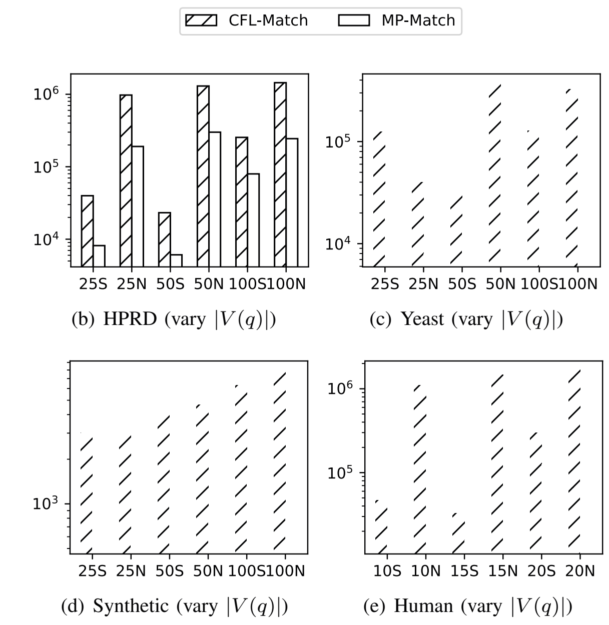
Datasets. We evaluate the performance of the tested algorithms on both real and synthetic graphs as follows.

Real Graphs. We evaluate the algorithms on three real graphs, HPRD, Yeast and Human, which are widely used in existing works [3], [9], [13], [20]. All the three graphs are protein interaction networks where vertex labels are generated under the *Gene Ontology Term. HPRD* contains 37081 edges, 9460 vertices with an average degree 7.8, and 307 distinct labels. *Yeast* contains 12, 519 edges, 3, 112 vertices with an average degree 7.8 and 71 distinct labels. *Human* is a dense graph of human protein interactions, which contains 86, 282 edges, 4, 674 vertices with an average degree 36.9, and 44 distinct labels.

Synthetic Graphs. We also use synthetic data graph *Synthetic* in [3] to evaluate the algorithms. *Synthetic* contains 10^5 vertices with an average degree 8.0, and 50 distinct labels.

Query Graphs. A query graph is generated as a connected subgraph of the data graph, by conducting random walk on the data graph. For HPRD, Yeast, Synthetic graphs, we generate 6 query sets, each containing 100 query graphs of the same size. In specific, we generate query sets $q_{25S}, q_{25N}, q_{50S}, q_{50N}, q_{100S}$ and q_{100N}, where q_{iS} and q_{iN} denote query sets with i vertices and, respectively, average degree $\leqslant 3$ (i.e. **S**parse) and > 3 (i.e. **N**pn-sparse). For Human which is a harder data graph for subgraph matching due to higher average degree and fewer distinct labels, we generate smaller query sets $q_{10S}, q_{10N}, q_{15S}, q_{15N}, q_{20S}$ and q_{20N}.

Eval-I: Against CFLMatch by Varying $|V(q)|$. We evaluate MPMatch against CFLMatch by varying $|V(q)|$ regarding the *total processing time* and *embedding enumeration time*. Embedding enumeration time is the time to enumerate embeddings after computing auxiliary data structures CPI and a matching order of query vertices.

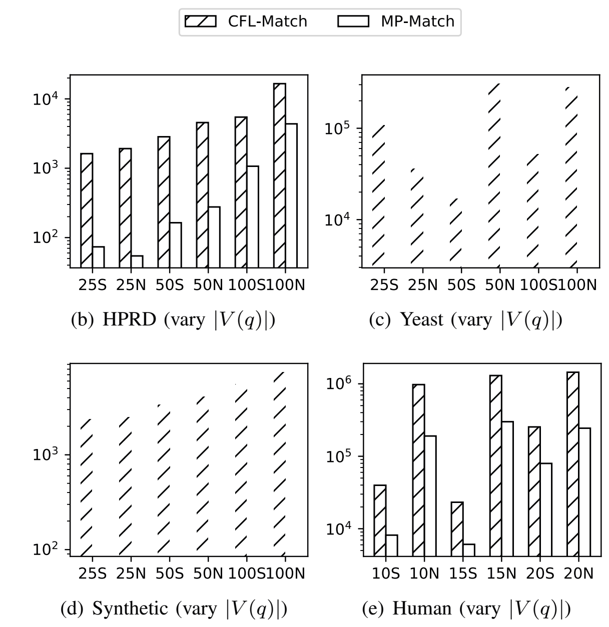


(b) HPRD (vary $|V(q)|$) (c) Yeast (vary $|V(q)|$) (d) Synthetic (vary $|V(q)|$) (e) Human (vary $|V(q)|$)

Fig. 4. Against Existing Algorithms (total processing time)

Total Processing Time. Fig. 4 shows the average total processing time for each query graph. In general, both two algorithms run slower for larger and denser queries. MPMatch consistently outperforms CFLMatch. This is due to parallelization with well-designed load balancing schema.

Embedding enumeration time. Fig. 5 shows the embedding enumeration time for each query graph. For some data graphs, the time to compute CPI and a matching order dominates the total processing time. MPMatch and CFLMatch shares the same CPI construction process, so the embedding enumeration time shows the performance improvement of MPMatch more obviously.



(b) HPRD (vary $|V(q)|$) (c) Yeast (vary $|V(q)|$) (d) Synthetic (vary $|V(q)|$) (e) Human (vary $|V(q)|$)

Fig. 5. Against CFLMatch (enumeration time)

B. Effectiveness of Multi-core Parallel Algorithm

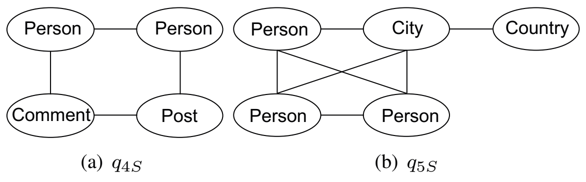
In this subsection, we evaluate the effectiveness of multi-core parallel algorithm and the scalability of our MPMatch algorithm.

Datasets. We evaluate the scalability of MPMatch on synthetic graphs only, due to the lack of labelled big graphs in the public. We use a data generator provided by the LDBC social network benchmarking [1] to generate large synthetic data graphs. These datasets are generated using the "Facebook" mode with a duration of 3 years. The default settings of synthetic graphs are: $|V(G)| = 30m$ (i.e. 3×10^7 vertices), $d(G) = 12$ (i.e. the average degree is 12), and $|\Sigma| = 11$ (i.e. the number of distinct labels is 11). Note that the smaller the number of distinct labels, the more challenging. The following synthetic data graphs are generated to test the scalability of our algorithms.

- *Vary* $|V(G)|$: We generate 5 data graphs denoted by $G_{3m}, G_{9m}, G_{30m}, G_{90m}, DG_{180m}$, where each G_{im} has $im(= i \times 10^6)$ vertices with the default settings of $d(G)$ and $|\Sigma|$.
- *Vary* $|\Sigma|$: We generate 4 data graphs denoted by $G_{L=5}, G_{L=10}, G_{L=15}, G_{L=20}$, where each $G_{L=i}$ contains i distinct vertex labels with the default settings of $|V(G)|$ and $d(G)$.

Query Graphs. For these synthetic graphs, which are harder data graphs for subgraph matching, we select two queries from LDBC Graphalytics workloads, denoted q_{4S} and q_{5S}, as shown in Fig. 6.

Eval-II: Evaluating Our Algorithm. We evaluate the effectiveness of our method for concurrent allocation of subgraph



(a) q_{4S} (b) q_{5S}

Fig. 6. Query graphs

matching search space and our load-balancing schema by varying the number of workers. Fig. 7 shows the maximum and minimum total processing time for q_{4S} and q_{5S} when using 1, 3, 6, 12, 18, 24 and 30 workers. As expected, the processing time decreases when using more workers. Moreover, MPMatch improves upon CFLMatch by over 1 order of magnitude when using 30 workers. For both q_{4S} and q_{5S}, maximum total processing time and minimum total processing time are very close, which confirms our well-designed load balancing schema.

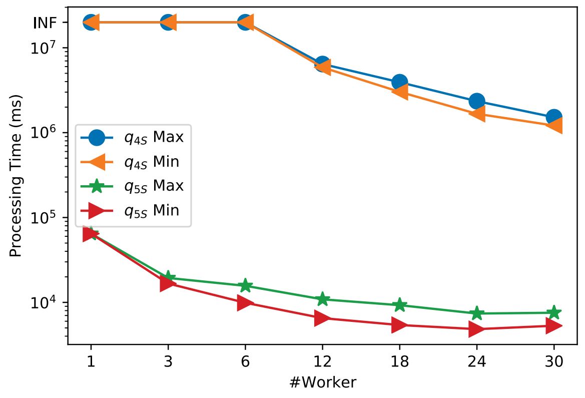
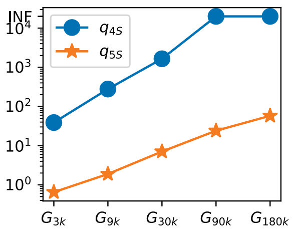
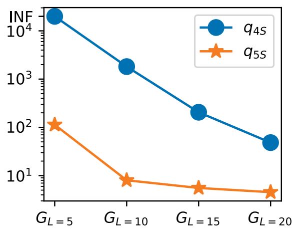


Fig. 7. Evaluating Our Algorithm vary ma

Eval-III: Scalability Testing. We test the scalability of MPMatch on Synthetic graphs by varying $V|G|$ and $|\Sigma|$.



(a) Vary $|V(G)|$



(b) Vary $|\Sigma|$

Fig. 8. Scalability Testing of MPMatch

Varying $|V(G)|$. The results of varying $V|G|$ are shown in Fig. 8(a). As expected, the processing time of MPMatch increases when $|V(G)|$ becomes larger, due to more candidates for each query vertex.

Varying $|\Sigma|$. The results of varying the number $|\Sigma|$ of distinct labels are shown in Fig. 8(b). The processing time of MPMatch decreases when $|\Sigma|$ becomes larger due to fewer candidates for each query vertex.

The experiment results show the scalability of MPMatch are consistent with CFLMatch.

V. CONCLUSION

In this paper, we develop an efficient and load-balancing algorithm MPMatch for parallel conducting subgraph matching. We explore the concurrent allocation of subgraph matching search space by decomposing matching process into two phases for subtask generation and subtask processing, respectively. We design a load balancing scheme by estimating subtask workload. The subtask whose workload exceeds limitation will be further decomposed into finer-grained subtasks. Moreover, a greedy algorithm is adopted in subtask distribution for load balancing. Our extensive performance studies on large real and synthetic graphs demonstrate that our techniques greatly improve the performance of CFLMatch. The experiment results also show the effectiveness of our load balancing scheme. As a possible future work, extending the process of CPI construction to multi-core concurrent environment might be an interesting issue to be investigated.

REFERENCES

[1] Ldbc benchmarks. http://ldbcouncil.org/benchmarks.

[2] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 62–73. IEEE, 2013.

[3] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1199–1214. ACM, 2016.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[5] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 337–348. ACM, 2013.

[6] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418. ACM, 2008.

[7] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment*, 8(10):974–985, 2015.

[8] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 10(3):217–228, 2016.

[9] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the VLDB Endowment*, volume 6, pages 133–144. VLDB Endowment, 2012.

[10] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 527–543, New York, NY, USA, 2017. ACM.

[11] N. Pržulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics*, 22(8):974–980, 2006.

[12] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: on compression and computation. *Proceedings of the VLDB Endowment*, 11(2):176–188, 2017.

[13] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment*, 8(5):617–628, 2015.

[14] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 2008.

[15] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. 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. ACM, 2014.

[16] T. A. Snijders, P. E. Pattison, G. L. Robins, and M. S. Handcock. New specifications for exponential random graph models. *Sociological methodology*, 36(1):99–153, 2006.

[17] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, 2012.

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

[19] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 335–346. ACM, 2004.

[20] P. Zhao and J. Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.