Parallelizing Recursive Backtracking Based Subgraph Matching on a Single Machine

Shixuan Sun
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Hong Kong, China
ssunah@cse.ust.hk

Qiong Luo
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Hong Kong, China
luo@cse.ust.hk

***Abstract*—We propose PSM, an algorithmic framework to parallelize a common class of subgraph matching algorithms, which are based on recursive backtracking. Specifically, we abstract the matching process as a tree search in the state space and different matching algorithms as different orders in the search. Subsequently, we parallelize such subgraph matching by dividing up the state space search tree and exploring it in parallel. Different from traditional approaches that parallelize the search by each individual state, we dynamically split the state tree into search regions each of which consist of a subtree. We further optimize PSM for load balance and communication efficiency. As case studies, we have parallelized three representative recursive backtracking based subgraph matching algorithms in PSM and studied their performance in comparison with their sequential counterparts. Our results show that the PSM-style parallel algorithms achieved a speedup of 15X-19X on the in-memory execution time on a twenty-core machine.**

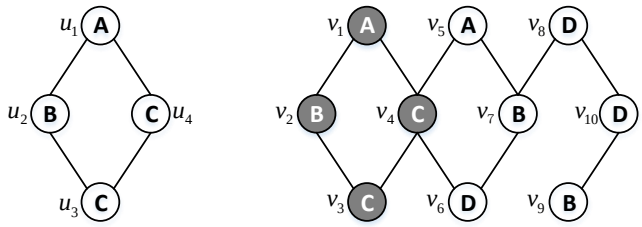
***Index Terms*—graph, subgraph isomorphism, recursive backtracking, parallel subgraph matching, multi-core CPUs**

I. Introduction

Given a labeled data graph G and a labeled query graph q, subgraph matching finds all subgraph isomorphisms from q to G. For example, given q and G in Figure 1, $\{(u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4)\}$ is a subgraph isomorphism. As a basic type of graph queries, subgraph matching is widely used in real world applications such as computer aided design [7], protein interaction relationship detection [5], social network analysis [27] and RDF queries [29].

The subgraph matching problem is NP-hard [16], and a variety of sequential algorithms [4], [6], [8], [9], [22], [24], [28] have been proposed to solve this problem. All these algorithms adopt a recursive backtracking strategy proposed by Ullmann [26] in 1976, which recursively expands partial results along an order of query vertices by mapping a query vertex to a data vertex at each step to find all subgraph isomorphisms, and focus on designing powerful filtering rules to eliminate invalid candidate data vertices of each query vertex as many as possible and generating effective matching orders to reduce the search space. However, due to the hardness of the subgraph matching problem, these algorithms often take a long time to process big data graphs and complex queries.

In order to further improve the performance of subgraph matching, a natural idea is to accelerate the processing by parallelization, because a commodity machine nowadays has considerable parallel computation capabilities. Recently, some parallel algorithms working on a single machine, such as PGX.ISO [18] (denoted PGX in short) and parallel RI [11] (denoted pRI in short), have been proposed to utilize the multi-threading technique provided by multiple cores in modern CPUs. Both algorithms take partial results as the basic task units. However, due to the exponential number of partial results in the search space, PGX easily runs out of memory, whereas pRI's speedup over the sequential RI [5] is limited to less than 10 times on a machine of 16 CPU cores. Moreover, several performance studies find that there is no single winner among existing sequential subgraph matching algorithms [10], [14], because the matching orders generated by existing algorithms are based on different heuristics. Therefore, we explore a parallelization approach that is orthogonal to the heuristics.



(a) Query graph q. **(b)** Data graph G.

Fig. 1: Subgraph isomorphism.

In this paper, we propose to parallelize the backtracking based subgraph matching algorithms in a shared-memory environment, such as a multicore machine. Specifically, we abstract these algorithms into a uniform model and present a generic **P**arallel **S**ubgraph **M**atching (PSM) algorithm. PSM serves as a template to parallelize existing sequential subgraph matching algorithms. Furthermore, users can develop new subgraph matching strategies in PSM without worrying about the parallelization. The resulting algorithms can exploit the parallel processing capability of the multicore machines as well as benefit from the state-of-the-art subgraph matching strategies.

We consider three challenges to design such a framework. The first challenge is to abstract different kinds of backtracking subgraph matching algorithms to a uniform model. The second is to determine a suitable granularity of parallelism.

Existing parallel approaches for a single machine [11], [18] or a distributed environment [23] utilize the independence between partial results to achieve a high parallelism. However, the exponential number of partial results can take up all memory available on a single machine. The last challenge is to achieve load balance and reduce overhead introduced by parallelization, as these factors have a great performance impact, especially with various data sets and different algorithms.

Addressing the three challenges, we make the following contributions in this paper.

- We abstract the exploration procedure of existing backtracking subgraph matching algorithms into a depth-first search (DFS) of a state space tree generated on the fly. We further analyze the properties of the search space.
- We propose a generic parallel subgraph matching framework, named PSM, based on the state space tree search to parallelize existing backtracking subgraph matching algorithms.
- We design a search region based dynamic task split strategy and an action replay technique, to achieve load balance and reduce overhead caused by parallelization in PSM.
- We parallelize three state-of-the-art subgraph matching algorithms following the PSM framework and conduct detailed experiments on a variety of real datasets. The experimental results show that the PSM algorithms achieves a speedup of 15X-19X over their sequential versions on a twenty-core machine.

Paper Organization. Section II presents preliminaries and related work. Section III introduces the state space tree model. Section IV presents the parallel subgraph matching (PSM) framework. We evaluate the performance of PSM in Section V and conclude in Section VI.

II. Background

In this section, we first present the preliminaries used in this paper and then introduce the related work.

A. Preliminaries

In this paper, we focus on the vertex-labeled undirected graph $g = (V, E, \Sigma, L)$, where V is a set of vertices, E is a set of edges, Σ is a set of labels, and L is a function that associates a vertex v with a label $L(v) \in \Sigma$. Moreover, the query graph q is connected and the data graph G is much bigger than q (i.e., $|V(G)| \gg |V(q)|$). Next, we give a formal definition of subgraph matching and related preliminaries used in this paper, and summarize the frequently used notations in Table I.

Definition 1: Subgraph Isomorphism: Given a query graph $q = (V, E, \Sigma, L)$ and a data graph $G = (V^{'}, E^{'}, \Sigma^{'}, L^{'})$, a subgraph isomorphism is an injective function $f : V \rightarrow V^{'}$ that satisfies:

(1). $\forall u \in V, L(u) = L^{'}(f(u))$;

(2). $\forall e(u, v) \in E, \exists e(f(u), f(v)) \in E^{'}$.

Definition 2: Subgraph Matching: Given q and G, find all subgraph isomorphisms from q to G.

TABLE I: Notations.

Notations	Descriptions
g, q, G	graph, query graph and data graph
$V(g)$, $E(g)$	vertex set and edge set of g
$d(u)$, $L(u)$, $N(u)$	degree, label and neighbors of u
$e(u, v)$	edge between u and v
S	partial result (i.e., state)
H, $H(S)$	state space tree and subtree rooted at S
π	matching order
$C(S, u)$	candidate set of u given S
$H(S, [i : j])$	search region
$N^{\pi}_{+}(u)$	backward neighbors of u in π
n	number of workers
q^{π}_{i}	vertex induced subgraph of q on $\pi[1 : i]$
α, β	cutoff depth and cutoff width

Definition 3: Matching Order: Given q, a matching order π is a permutation of the vertices in $V(q)$. $\pi[i]$ is the ith vertex in π and $\pi[i : j]$ is the set of vertices from index i to j in π.

Definition 4: Backward Neighbors: Given q and π, suppose that $u \in \pi$. The backward neighbors of u, denoted as $N^{\pi}_{+}(u)$, is the neighbors of u positioned before u in π.

Given q and π, q^{π}_{i} denotes the vertex induced subgraph of q constructed on $\pi[1 : i]$. In order to reduce the search space, existing subgraph matching algorithms require that the matching order π satisfies that $\forall 1 \leq i \leq |V(q)|$, q^{π}_{i} is a connected graph. Equivalently, given any vertices u in π except $\pi[1]$, $N^{\pi}_{+}(u) \neq \emptyset$. Therefore, we also assume that π satisfies such a constraint in this paper.

Assumptions. In summary, we have the following three assumptions in this paper.

1) The query graph q is connected.
2) The matching order π is connected (i.e., given any vertices u in π except $\pi[1]$, $N^{\pi}_{+}(u) \neq \emptyset$).
3) $|V(G)| \gg |V(q)|$.

B. Related Work

As a fundamental graph querying operation, subgraph matching receives a lot of research interests.

1) Sequential Algorithms: Existing sequential subgraph matching algorithms follow the same enumeration process that expands partial results recursively along an order of query vertices, but have different strategies to prune the invalid candidate data vertices and optimize the matching order. Ullmann [26], VF2 [6], QuickSI [22], SPath [28] and RI [5] obtain the candidate data vertices of each query vertex individually based on filters such as the label/degree filter and the neighborhood signature. In contrast, GraphQL [9], TurboIso [8] and CFL [4] build an auxiliary data structure before the enumeration and conduct the enumeration process based on the auxiliary data structure instead of the original data graph. These algorithms also adopt different ordering techniques to generate matching orders. Specifically, QuickSI designs an infrequent-label first strategy and GraphQL proposes the left-deep join based method. SPath, TurboIso and CFL utilize the path based ordering method, whereas RI uses the constraint based approach. For brevity, we omit the details of these algorithms, because our focus is to parallelize

the enumeration process instead of designing new pruning strategies and ordering methods. Interested readers can refer to the recent performance studies [10], [14] for the details of these algorithms.

In addition, some researchers boost subgraph matching by exploiting the vertex relationship in the data graphs [15], [20] and utilizing the matching results among multiple queries [21].

2) Parallel Algorithms: Compared with the extensive research on the sequential algorithms, the research on the parallel subgraph matching algorithms working on multicore CPUs is rather limited. PGX [18] takes each partial result as the basic task unit and expands partial results iteratively along an order of query vertices, which is a parallel breadth-first search approach with the bulk synchronous parallel model. As a result, it requires to store all intermediate results at each step. On a single machine, this method easily blows up memory. pRI [11] parallelizes the RI algorithm [5], which is a sequential algorithm developed in 2013. pRI also uses partial results as the task units. Each worker (i.e., thread) maintains a private deque to store its partial results (i.e., tasks). During the execution, the worker first fetches a partial result from the head of its own deque, then expands the partial result, and finally adds the generated partial results into the head of its own deque. If a worker is idle, then it will steal tasks from the tail of the deque of other workers. Fetching and adding tasks from the deque requires lock operations to avoid the race condition. Because of the exponential number of partial results, the frequent lock operations incur significant overhead.

Additionally, STwig [24] works on the distributed environment, whereas GpSM [25] runs on GPUs. Both of them convert the subgraph matching problem into the join problem.

3) Other Related Work: Subgraph enumeration is to find all subgraph isomorphisms in unlabeled graphs. Due to the lack of label, the search space of subgraph enumeration is very large and subgraph enumeration is more challenging than subgraph matching. The latest research on this problem focuses on designing parallel distributed approaches such as Afrati [3], TwinTwig [12], SEED [13], PSgL [23], Crystal [17]. In addition to the difference on the presence of labels, subgraph matching is generally integrated as a query operation in a graph database such as Neo4j, whereas subgraph enumeration is generally an offline analytic task [23].

III. The State Space Tree

In this section, we first abstract backtracking subgraph matching algorithms into an exploration of a state space tree, and then analyze the properties of the tree.

A. State Space Tree Exploration

Algorithm 1 presents the enumeration procedure proposed by Ullmann [26], which is used in existing sequential subgraph matching algorithms. It takes q and G as input and outputs all subgraph isomorphisms from q to G. Line 2 generates a matching order π, and S records mappings from query vertices to data vertices, called the *partial result* in this paper. In particular, $S.keys$ and $S.values$ denote the query vertices and data vertices in S respectively. Line 7 generates a candidate set $C(S, u)$, which stores the data vertices that can be mapped to u to extend S. Existing subgraph matching algorithms propose a variety of approaches to minimize the size of $C(S, u)$. For simplicity, in this paper, we assume that a data vertex $v \in C(S, u)$ at least satisfies that $L(v) = L(u)$ and $d(v) \geq d(u)$. Lines 8-12 loop over $C(S, u)$ to extend S. If v has not been mapped and there are edges between v and the data vertices mapped to the backward neighbors of u (Lines 13-15), then we extend S by mapping u to v. Otherwise, we skip v. If all query vertices have been mapped, then line 11 outputs S'. Otherwise, line 12 invokes the $Enumerate$ procedure recursively.

Algorithm 1: Sequential Subgraph Matching

Input: a query graph q and a data graph G
Output: all subgraph isomorphisms from q to G

```
1  begin
2      π ← GenerateMatchingOrder(q, G);
3      S ← {}, i ← 1;
4      Enumerate(S, i, π, q, G);
5  Procedure Enumerate(S, i, π, q, G)
6      u ← π[i];
7      C(S, u) ← GenerateCandidateSet(S, u, q, G);
8      foreach v ∈ C(S, u) do
9          if IsFeasible(S, u, v, G) = true then
10             S' ← S ∪ {(u, v)};
11             if i = |π| then Output S';
12             else Enumerate(S', i + 1, π, q, G);
13 Function IsFeasible(S, u, v, G)
14     if v ∉ S.values and ∀u' ∈ N+_π(u), e(S[u'], v) ∈ E(G)
       then return true;
15     return false;
```

Given a partial result S generated in Algorithm 1 that contains i mappings, S is a subgraph isomorphism from q_i^π to G because the candidate set ensures that $\forall u \in S.keys, L(u) = L(S[u])$ and the $IsFeasible$ function guarantees that $\forall e(u, u') \in E(q_i^\pi), e(S[u], S[u']) \in E(G)$. In particular, we denote S as S_r when S is empty. The $Enumerate$ procedure conceptually constructs a state space tree H on the fly. It starts from S_r and always extends the most recently generated partial results for the following step. In other words, it explores H by the depth-first search.

The initial state of H is S_r and the internal states are the partial results generated during the enumeration. The edges of H correspond to mappings between query vertices and data vertices. The leaves are terminations of search paths originated from the initial state, which can be categorized into two classes: the success leaves with $i = |\pi|$ at line 11, and the failure leaves with $IsFeasible$ returning $false$. All the success leaves are the solutions of subgraph matching. There are one to one relationships from internal states and success leaves to partial results. Therefore, we also denote the state in H as S. In contrast, the failure leaves are not states.

Example 1: Figure 2 shows the state space tree H generated by Algorithm 1 on the graphs in Figure 1. Suppose that $\pi =$

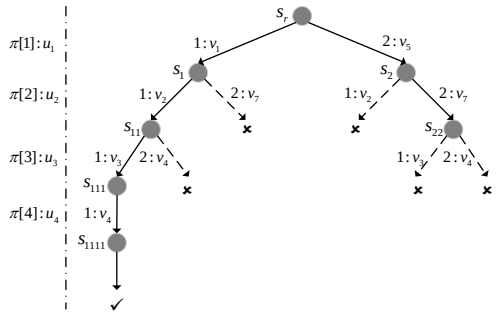


Fig. 2: H generated by Algorithm 1 on graphs in Figure 1.

(u_1, u_2, u_3, u_4). Take the state $S_{22} : \{(u_1, v_5), (u_2, v_7)\}$ as an example. The next query vertex in π is u_3. Thus, $C(S_{22}, u_3) = \{v_3, v_4\}$. The *Enumerate* procedure explores H in the DFS order, and finds all solutions, which is the following in this example: $S_{1111} = \{(u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4)\}$.

Remark. Existing recursive backtracking subgraph matching algorithms adopt the same enumeration process as Algorithm 1 with differences in the strategies to generate the matching order (i.e., the $GenerateMatchingOrder$ function) and the methods to obtain the candidate sets (i.e., the $GenerateCandidateSet$ function).

B. Properties of the State Space Tree

Given q and G, search paths are terminated if $IsFeasible$ returns $false$ or all query vertices in π have been mapped. Suppose that the depth of S_r in H is 0. Thus, the depth of H is at most $|\pi|$. Moreover, suppose that H_i contains all the states at depth i $(0 \leq i < \pi)$ and the average branching factor of states in H_i is b_i. Then, we have the following equation.

$$|H_i| = \begin{cases} 1 & i = 0. \\ \prod_{j=0}^{i-1} b_j & 0 < i \leq |\pi|. \end{cases} \quad (1)$$

In the following, we discuss the value of the branching factor. Given $S \in H_i$ $(0 < i < \pi)$ and $u = \pi[i+1]$, the $Enumerate$ procedure loops over $C(S, u)$ to extend S. Suppose that $\Phi(S)$ is the set of partial results derived from S by mapping u to $v \in C(S, u)$. Then, the branching factor of S denoted as b_S is equal to $|\Phi(S)|$. Recall that S is a subgraph isomorphism from q_i^π to G and $S' \in \Phi(S)$ is a subgraph isomorphism from q_{i+1}^π to G. According to Definition 1 and 4, b_S is less than or equal to $\min_{u' \in N_+^\pi(u)} d(S[u'])$. As a result, b_S is affected by the degree of data vertices mapped to the backward neighbors of u. Given the various degrees of data vertices, the branching factors of states can be very different. Therefore, H is **irregular**. Moreover, the multiplication over the average branching factors in Equation 1 indicates that H can contain an **exponential** number of states. b_0 is the branching factor of S_r. As S_r contains no mappings and $N_+^\pi(\pi[1]) = \emptyset$, each data vertex v that satisfies $L(v) = L(\pi[1])$ and $d(v) \geq d(\pi[1])$ can be mapped to $\pi[1]$. Therefore, the size of the data graph has an important effect on b_0, which is different from the branching factors of the other states. Additionally, $|V(q)|$ generally scales from 3 to tens of vertices, whereas $|V(G)|$ ranges from thousands to millions even billions. Let $|H_{max}| = \max_{0 \leq i \leq \pi} \{|H_i|\}$ and we have $|\pi| \ll |H_{max}|$, which means that the state space tree H is **flat**.

In summary, given q and G, Algorithm 1 constructs an irregular and flat state space tree H that can contain an exponential number of states. Because H is constructed on the fly and branching factors depend on the properties of q and G as well as the methods generating matching orders and candidate sets, it is hard to quantify the exact value of branching factors and the size of H in advance.

IV. Parallel Subgraph Matching Framework

In this section, we present the design of our parallel subgraph matching (PSM) framework.

A. Parallel Task

In order to parallelize subgrpah matching, the first step is to identify the parallelism in the sequential algorithm. Based on the state space tree model, the states can be expanded independently. Therefore, a natural way is to regard states as the basic task units, which is also the strategy used in existing parallel algorithms such as PGX and pRI. We call this method the **fine-grained parallelism**. Given a state S, suppose that $W(S)$ represents the workload of expanding S. Then, $W(S)$ can be estimated as b_S, which is small. Since H contains an exponential number of states, the fine-grained parallel method results in a large number of lightweight tasks. Consequently, this approach can incur a high communication overhead.

In order to reduce the communication overhead, we need to increase the workload of parallel tasks to reduce the number of generated tasks [1]. Reconsidering H, we cannot only expand the immediate successors of given states S simultaneously, but also the subtrees rooted at different states S, denoted as $H(S)$. Furthermore, $H(S)$ can be divided into more fine-grained ones by taking part of the candidate set of S or separating a subtree rooted at a successor of S. In the following, we define the search region.

Definition 5: Search Region: Given a matching order π, a state space tree H, a state S and the candidate set $C(S, u)$, a search region $H(S, [i : j])$ is the subtree rooted at S constrained with the candidate set ranged $[i : j]$ in H.

Based on Equation 1, the workload of exploring $H(S, [i : j])$, denoted as $W(H(S, [i : j]))$, is much more than $W(S)$. Moreover, search regions can be explored independently as well as divided into more fine-grained ones. Inspired by this observation, PSM takes a search region as a parallel task instead of a state in the fine-grained parallel approach. We say PSM is **coarse-grained parallel**, where each worker expands assigned search regions in a depth-first search order independently.

Example 2: Figure 3 shows a state space subtree H rooted at S in which circles and triangles represent states and subtrees respectively. Suppose that the next vertex in the matching order of S is u and that of S' is u'. By only taking $C(S, u)[1 : 2]$, we can get the subtree H' in the area surrounded by the dashed line. Let the part outside the area be H''. We have

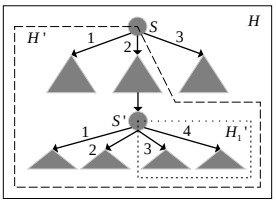


Fig. 3: Search region.

$H = H' \cup H''$ and $H' \cap H'' = \emptyset$. H' can be divided into more fine-grained ones by taking the subtree H'_1 rooted at S' with $C(S', u')[3:4]$. Let the remaining part of subtree after split be H'_2. If we have S and S', then H'', H'_1 and H'_2 can be explored concurrently.

B. Load Balancing

As H is constructed on the fly and irregular, it is hard to assign equal amounts of work to workers at the beginning. Therefore, we require a dynamic load balancing approach to resolve the load imbalance problem. Specifically, we need to determine when and how to create new tasks. In this subsection, we mainly focus on how to create new tasks. The problem that decides when to create tasks is discussed in Section IV-C.

Suppose that a busy worker is required to give part of its task to an idle one. Then, the goal of splitting a task is to divide the task into two subtasks with nearly equal workloads. When performing a task split, we only know the range of unvisited vertices in candidate sets of expanded states at each depth, called the *action range*.

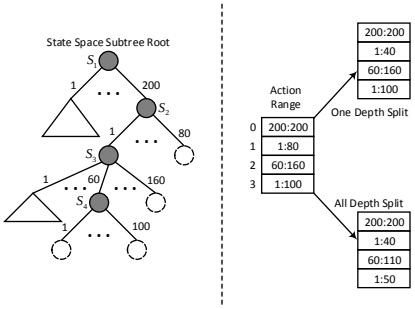


Fig. 4: Task split example.

Example 3: Figure 4 shows a task split example. The subtree rooted at S_1 is the search region of the busy worker. The expanded states at each depth are marked in gray, while the white circles represent the search space that is not explored. Suppose that when expanding S_4, the busy worker is required to donate part of its remaining work. The right part of Figure 4 shows the action range.

Since the size of the unexplored search space is unknown, we need to estimate it based on the action range at each depth. A heuristic rule is that the subtrees rooted at states of the same depth contains a similar number of states. According to this rule, we present two task split strategies.

The first strategy is to divide the action range at each depth into two equal halves. We name it **all-depth split**. As shown in Figure 4, after an all-depth split, we keep half of the elements at each depth and get a new task containing three search regions $H(S_2, [41:80])$, $H(S_3, [111:160])$ and $H(S_4, [51:100])$. The second strategy is to only split the action range of the expanded state close to the subtree root, named as **one-depth split**. For example, in Figure 4, we only partition the action range of S_2 equally and get the search region $H(S_2, [41:80])$.

To perform the all-depth split, we need to loop over the entire action range array to find all candidate sets that can be partitioned. Moreover, the task generated by the all-depth split consumes more memory than that of the one-depth split. In contrast, as the all-depth split partitions the action range of the expanded states at each depth, the workload of generated tasks tends to be more uniform than that of the one-depth split. However, because H grows exponentially with the depth, the workload of a search region rooted at a shallow depth is much more than that of one rooted at a deep depth. In other words, the search region rooted at the shallowest depth dominates the workload in a all-depth split. Therefore, compared with its cost, the gain of the all-depth split is thin, so PSM adopts the one-depth split in its implementation.

Another problem of the task split is to avoid generating the lightweight tasks, whose benefit offsets its overhead. More specifically, we need to determine when to stop splitting a task and keep the remaining task executing in a worker. A classic idea is to set the maximum depth, denoted as α, at which the action range is allowed to split [19], which is called the **cutoff depth**. This idea works as the size of the subtree rooted at a deep state is so small that the time saved by exploring it with many workers can not offset the cost of task split. However, as H is flat, the number of states of a subtree rooted at a great depth can be still very large.

In order to address this problem, we introduce another metric called the **cutoff width**, denoted as β. When the depth of the selected state S is greater than α, if the action range is greater than β, we still split its action range to generate a new task.

C. Communication Model

PSM adopts a decentralized communication model that has no master responsible for assigning tasks. As PSM is designed for a single machine which has a limited number of cores with a shared memory environment, PSM adopts a sender-initiated communication method with a global concurrent queue to deliver tasks among workers. The benefit of this approach is that idle workers are able to almost immediately acquire work by the donation of busy workers [2]. Specifically, the busy workers will frequently check the status of the queue and the number of idle workers. If a busy worker finds that the queue is empty and there are idle workers, then it will donate part of its task, push the generated task into the queue, and wake up the idle workers to fetch the task.

PSM utilizes a search region $H(S, [i:j])$ as a parallel task. The recovery of the search region root has an important effect on the communication cost. Hence, the design of the data structure storing the search region information affects both the

communication cost and the memory cost. The action range $[i:j]$ is easy to be represented as an index pair. The difficulty is in how to record the search region root S.

Because S only records the mapping relationship from query vertices to data vertices, a simple idea is to clone S during delivery, which we call a **root clone**. An idle worker starts the search from S directly without any recomputation. However, a variety of backtracking subgraph matching algorithms maintain auxiliary data structures to improve performance. The auxiliary data structure is updated dynamically as the state is updated. For example, the VF2 algorithm [6] maintains vectors to record neighbors of a matched subgraph. Under this circumstance, the root clone strategy does not work due to the lack of the auxiliary data structure information. Even if we could clone the state and the auxiliary data structure together, both communication and memory costs are expensive.

Recall that the state space tree H is flat. Namely, the search path is very short and the time required to go through the path is polynomial to $|V(q)|$. Furthermore, we generate S from S_r by mapping query vertices to data vertices along π at each step, while the auxiliary data structure is also updated based on the mapping. Thus, if we replay the mappings in order, we can recover both S and its auxiliary data structure without copying them. We name this strategy **action replay**.

By action replay, a task can be stored as mappings and an index pair, which consumes $O(|V(q)|)$ memory. Moreover, as PSM generates tasks as necessary (i.e., the queue is empty and there are idle workers), PSM can generate at most n tasks simultaneously where n is the number of workers. Therefore, the size of the queue can be fixed to n. Thus, the concurrent queue consumes $O(n \times |V(q)|)$ memory.

D. Initial Distribution and Termination Detection

In PSM, dynamic load balancing is key to the high parallel efficiency. Therefore, at the beginning of the execution, we simply assign vertices in $C(S_r, \pi[1])$ evenly to workers as their initial tasks. As the number of cores (i.e., workers) on a single machine is limited, PSM implements a simple termination detection method: when a busy worker becomes idle, it will check the number of idle workers and the status of the queue. If there are n idle workers and the queue is empty, it will first wake up other idle workers to exit and then terminate itself.

V. Experiments

In this section, we conduct detailed experiments on a variety of real datasets to evaluate the performance of PSM.

A. Experimental Setup

Algorithms Under Study. In order to demonstrate the generality of PSM, we implement three state-of-the-art subgraph matching algorithms with different properties in PSM: QSI (QuickSI [22]), GQL (GraphQL [9]) and CFL [4], whose parallel versions are denoted as pQSI, pGQL and pCFL respectively. Given q and G, the size of the state space tree is determined by the matching order and the candidate sets (i.e., the integrated subgraph matching algorithms). Existing parallel algorithms PGX [18] and pRI [11] implement specific subgraph matching algorithms. In contrast, the goal of PSM is to parallelize a common class of subgraph matching algorithms with a generic framework. Hence, we evaluate the efficiency of PSM through comparing the performance of the parallel algorithms in PSM with their corresponding sequential algorithms. Additionally, we compare PSM with both PGX and pRI when applicable.

Experimental Environment. All of our algorithms are implemented in C++ and the parallelization is based on the Pthreads library. The source code is compiled by g++ 4.9.3 with -O3 flag. We conduct experiments on a 64-bit Linux machine equipped with 64GB RAM and two Intel Xeon E5-2650 v3 CPUs each of which has ten 2.30GHz physical cores. Therefore, we set the number of workers as 20 by default in experiments (i.e., one thread per physical core).

Parameters Configuration. PSM requires two parameters to control the minimum workload of a task: the cutoff depth α and the cutoff width β. Through experiments, we set α as $|V(q)| - 3$ and β as 8 by default from the configurations $\alpha \in \{1, 2, ..., |V(q)| - 1\}$ and $\beta \in \{1, 2, 4, 8, 16, 32\}$ for the best performance.

TABLE II: Properties of real datasets.

Dataset	$\|V\|$	$\|E\|$	$\|\Sigma\|$	Avg. Degree
Yeast	3,112	12,519	71	8.04
WordNet	76,853	120,399	5	3.13
Youtube	1,134,890	2,987,624	25	5.27
US Patents	3,774,768	16,518,948	20	8.75

Graph Datasets. We select four real datasets, which have been widely used in previous work [4], [20], [24]. Yeast and WordNet originally contain labels. The other two datasets have no labels, to each of which we randomly assign distinct labels. Table II lists the detailed information. As shown in Table II, $|V|$ scales from thousands to millions, $|\Sigma|$ ranges from 5 to 71 and the average degree varies from 3.13 to 8.75. Thus, the real datasets have various properties to evaluate the performance of PSM.

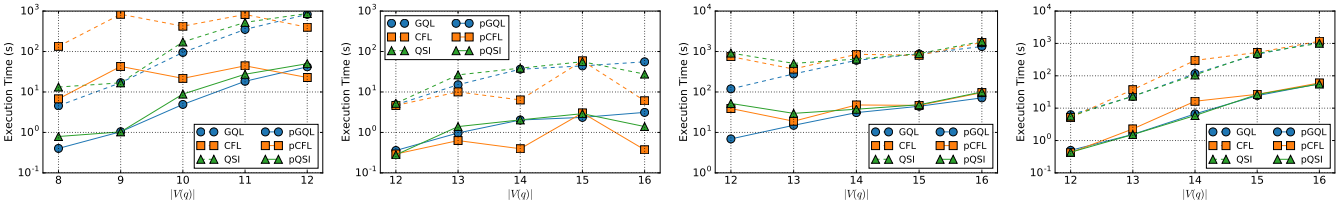
Query Sets. Following the previous research [4], [20], [24], we generate query graphs by selecting subgraphs from a data graph randomly to guarantee that at least one subgraph isomorphism exists. We generate five query sets for each data graph, and each query set contains ten query graphs with the same number of vertices. Each query graph is connected. The query sets of each data graph are shown in Table III, where q_i represents a set of i-vertex query graphs. The query graphs for WordNet are smaller, because 63,098 of the 76,853 vertices (i.e., $\geq 80\%$) in WordNet have the same label. As a result, subgraph matching on WordNet is harder than the other three graphs.

TABLE III: Query sets information.

Dataset	Query Set
Yeast, Youtube, US Patents	$q_{12}, q_{13}, q_{14}, q_{15}, q_{16}$
WordNet	$q_8, q_9, q_{10}, q_{11}, q_{12}$

TABLE IV: Speedup on the real datasets.

	Short Queries			Long Queries			Overall		
	$gSpeedup$	$iSpeedup$	$\#queries$	$gSpeedup$	$iSpeedup$	$\#queries$	$gSpeedup$	$iSpeedup$	$\#queries$
pGQL	17.32	15.32	124	19.25	19.18	76	19.10	16.79	200
pCFL	17.68	15.70	114	18.50	18.49	77	18.46	16.82	191
pQSI	17.65	16.60	117	17.94	17.93	79	17.92	17.14	196



(a) WordNet **(b)** Yeast **(c)** Youtube **(d)** US Patents

Fig. 5: Execution time on the real datasets with the number of query vertices varied.

Metrics. We examine two metrics in our experiments: the execution time and the speedup. The execution time is the average time of processing a query graph in a query set, which excludes the time of loading the data from the disk. A challenging problem is that given query graphs with the same number of vertices, the execution time of answering them has a large variance. As a result, the long queries dominate the average execution time. To solve this problem, we use two kinds of speedup, the global speedup (gSpeedup) and the individual speedup (iSpeedup), to evaluate the gain of PSM. Equation 2 and 3 define gSpeedup and iSpeedup respectively, in which Q is a query set, $T_S(q)$ is the execution time answering q in sequential and $T_P(q)$ is the execution time in parallel. The gSpeedup metric is designed from the system perspective, because the system focuses on the total speedup of answering a collection of queries with PSM, while iSpeedup is for an individual user, since the user is interested in the expected speedup of answering the individual query.

$$gSpeedup = \frac{\sum_{q \in Q} T_S(q)}{\sum_{q \in Q} T_P(q)}. \quad (2)$$

$$iSpeedup = \frac{1}{|Q|} \sum_{q \in Q} \frac{T_S(q)}{T_P(q)}. \quad (3)$$

Additionally, in order to complete the experiments in reasonable time, the time limit for processing a query graph is 90 minutes (i.e., $5,400$ seconds). If a sequential algorithm cannot finish within the time limit, then we terminate the query of omit the result of this query.

B. Speedups of Individual Algorithms on PSM

In this subsection, we evaluate the execution time of PSM and the speedup obtained with PSM compared with the sequential counterparts.

Figure 5 illustrates the execution time on the real datasets with the number of query vertices varied. The dashed lines denote the execution time of original sequential algorithms, while the solid lines represent the execution time of these algorithms parallelized with PSM. Overall, there is no single winner on all test cases among the sequential algorithms. As our goal is to evaluate PSM rather than comparing the performance of the sequential algorithms, we next focus on the performance improvement with PSM. As shown in the figure, the parallel algorithms perform much better than their original sequential ones on all test cases. For example, GQL on average spends 1,327 seconds to answer a query with 16 vertices on youtube, while pGQL only takes 71 seconds.

Next, we examine the speedup achieved with PSM, which is presented in Table IV. Generally, the benefit of processing a long-running task in parallel is greater than that of a short task, because the overhead incurred by parallelization compared with the execution time of the long task is low. Therefore, we categorize the queries into two classes by the processing time. Specifically, we regard a query as a short query if the sequential algorithm takes less than 10 seconds to answer it. Otherwise, it is a long query. Note that in our experiments, we examine each algorithm with 200 queries (i.e., 4 datasets $\times$ 50 query graphs) in total. The gSpeedup of short queries is greater than iSpeedup, because among the short queries, there are many instances taking very small execution time that dominate the value of iSpeedup. Furthermore, all the three algorithms achieve speedups (both gSpeedups and iSpeedups) of more than 15X on short queries. In comparison, the gSpeedup and iSpeedup values of long queries are similar, since the execution time of each long query is much greater than the overhead incurred by parallelization. Furthermore, the parallel algorithms achieve higher speedups on long queries than on short ones. More specifically, on long queries the gSpeedup and iSpeedup values are 17.92 and 16.79 respectively. Note that, regardless of parallelization, CFL and QSI fail to finish 9 and 4 queries respectively due to the ineffective matching orders.

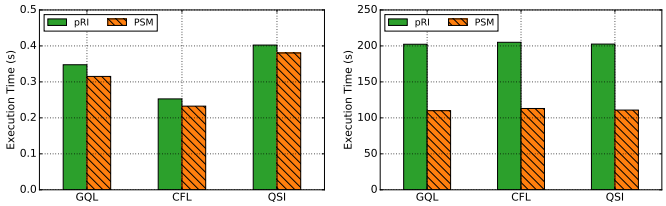
Through the detailed experimental results, we show that all the three integrated algorithms achieve significant performance improvement on all the four datasets.

C. Evaluate PSM efficiency

In this subsection, we first compare PSM with existing parallel algorithms PGX and pRI, and then evaluate PSM efficiency, including the dynamic load balancing, the scalability, the overhead incurred by communication and the memory cost. Specifically, instead of examining the average value of a

query set, we pick a short query on Yeast and a long query on Youtube as case studies for detailed evaluation.

1) Compare PSM with PGX and pRI: Given q and G, the size of the search space is determined by the matching order and the candidate sets. PSM, PGX and pRI explore the search space in parallel, but not reduce its size. Therefore, it is unfair to compare the parallel algorithms directly, as they parallelize different sequential algorithms, which can generate various search spaces even with the same q and G. In order to resolve this problem, we first extract the parallel strategies such as the parallel task representation, the load balancing method and the communication model from PGX and pRI, and then adopt the strategies to parallelize GQL, QSI and CFL respectively. This way, we can make a fair comparison among PSM, PGX and pRI. In particular, because PGX explores the state space tree in a breadth-first search order, it has to maintain all the intermediate results at each iteration. Consequently, algorithms with PGX parallelization fail to complete the query because they run out of memory. For example, given the long query on Youtube, there are at most 3.2×10^{10} partial results at an iteration, which cannot fit entirely in the memory. In contrast, both PSM and pRI explore the search tree in a depth-first search order, and consume a small amount of memory to store the partial results. In the following, we omit the results of PGX and mainly compare PSM with pRI.

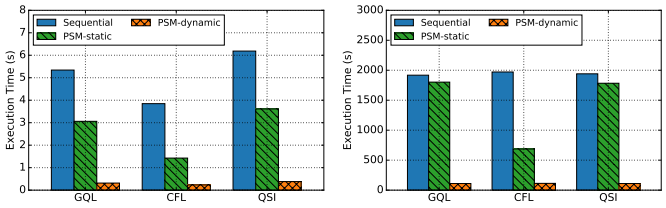


(a) Yeast **(b)** Youtube

Fig. 6: Compare PSM with pRI.

Figure 6 presents the execution time of the algorithms parallelized with the strategies of PSM and pRI. All the algorithms run with 20 workers. For the short query on Yeast, the performance of the pRI-style parallel algorithms is close to that of the PSM-style ones. In contrast, for the long query on Youtube, the PSM-style parallel algorithms run around 1.85 times faster than the pRI-style. This difference is because the search space of the short query is small, whereas that of the long query is large. Because of the exponential number of partial results in the large search space, the fine-grained parallel strategy of pRI incurs expensive overhead. Overall, the PSM-style algorithms achieve speedups of more than 16X on the short query and 17.4X on the long query, whereas the speedups of the pRI-style algorithms are around 15X and 9.5X. Specifically, because GQL, CFL and QSI generate similar matching orders for the long query on Youtube, the execution time of the competing algorithms is similar on this query.

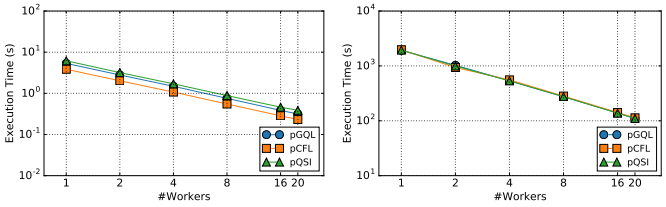
2) Compare the dynamic load balancing with the static load balancing: In order to demonstrate the impact of the dynamic load balancing in PSM, we compare PSM with the dynamic load balancing method enabled, denoted as PSM-dynamic, with that only with the initial distribution of tasks, called the *static load balancing* and denoted as PSM-static. Figure 7 presents the execution time of the algorithms parallelized with PSM-dynamic and PSM-static respectively as well as the sequential counterparts. We can see that the algorithms parallelized with PSM-static achieve limited speedups compare with the sequential counterparts due to the irregular search space. In contrast, the algorithms with the dynamic load balancing strategy achieve significant speedups.



(a) Yeast **(b)** Youtube

Fig. 7: Compare the dynamic load balancing with the static load balancing.

3) Evaluate the scalability of PSM: Figure 8 presents the execution time of the parallel algorithms with #workers varied from 1 to 20. As shown in the figure, all the three algorithms achieve almost linear speedups on the two queries. Specifically, the speedups with 20 workers are about 16X on Yeast and 17.4X on Youtube datasets. In Figure 8b, the three lines coincide with each other, because the three algorithms have similar matching orders and candidate sets.



(a) Yeast **(b)** Youtube

Fig. 8: Execution time with #workers varied.

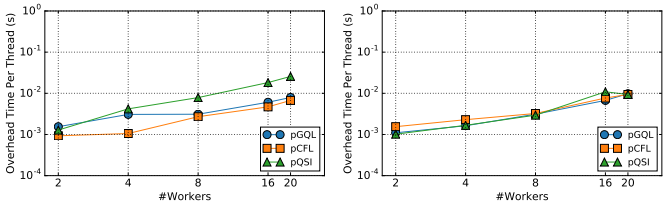
4) Evaluate the time overhead of PSM: Figure 9 presents the time overhead of each worker, which is incurred by parallelization. In particular, the time overhead consists of the time on creating tasks, delivering tasks, and performing the action replay. There is no time overhead on the 1-worker setup, as the sequential algorithms have no parallelization. We can see that the time overhead generally increases with the number of workers. Subsequently, we further examine the number of tasks executed per worker with the total number of workers varied. As shown in Figure 10, the number of tasks per worker grows with the number of workers, and this curve has a similar trend to that of the time overhead. Moreover, by comparing Figure 10a with 10b, we find that the number of tasks in the long query is greater than that in the short query. This is because a greater workload and more workers require more tasks to keep the load balance in PSM. Nevertheless, the time overhead incurred by PSM

TABLE V: Memory cost.

	Yeast		Youtube	
	Task Queue	Candidates	Task Queue	Candidates
pGQL	2.969 KB	0.0387 MB	2.969 KB	0.2257 MB
pCFL	2.969 KB	0.0443 MB	2.969 KB	0.2538 MB
pQSI	2.969 KB	0.0366 MB	2.969 KB	0.2309 MB

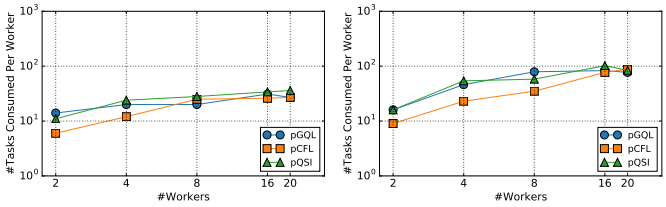
only accounts for a small proportion of the execution time, especially for the long queries.

Recall that the speedups of the two queries are 16X and 17.4X in the 20-worker setup, which are slightly lower than the ideal speedup of 20. After an investigation with a performance profiling tool Intel VTune, we find that this difference is mainly because (1) The memory accesses of the multi-threaded graph algorithms are frequently irregular; and (2) Context switches between multiple threads take time.



(a) Yeast **(b)** Youtube

Fig. 9: Time overhead per worker with #workers varied.



(a) Yeast **(b)** Youtube

Fig. 10: #Tasks consumed per worker with #workers varied.

5) Evaluate the memory cost of PSM: Except the data graph, PSM consumes extra memory space for the queue containing tasks (denoted task queue in short) and the candidate sets generated during the enumeration of each worker (denoted candidates in short). Table V lists the memory cost of the two queries, which excludes the cost of the data graph. As a task consumes $O(|V(q)|)$ memory and the queue has a fixed size to the number of workers, the task queue of the six cases consumes the same memory space, which is around 3 KB and negligible. Given a query vertex, it has at most $|V(G)|$ candidate data vertices. Therefore, PSM consumes $O(n{\times}|V(q)|{\times}|V(G)|)$ memory to hold the candidate sets (n is #workers). In practice, the memory cost is very small, because most of data vertices can be ruled out by filtering rules used in the $GenerateCandidateSet$ function, for example the degree and label filters. As illustrated in the table, PSM consumes less than 0.3 MB to store the candidates in the experiments, which shows that PSM consumes small extra memory space to parallelize the sequential algorithms.

VI. Conclusion

In this paper, we propose a generic parallel subgraph matching framework called PSM to accelerate backtracking subgraph matching algorithms. We integrate three representative subgraph matching algorithms into PSM to show its generality. Through extensive experiments on a variety of real datasets, we demonstrate the efficiency and robustness of PSM.

References

[1] F. N. Abu-Khzam, K. Daudjee, A. E. Mouawad, and N. Nishimura. On scalable parallel recursive backtracking. *Journal of Parallel and Distributed Computing*, 2015.

[2] U. A. Acar, A. Charguéraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. *SIGPLAN*, 2013.

[3] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. *ICDE*, 2013.

[4] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. *SIGMOD*, 2016.

[5] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 2013.

[6] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *TPAMI*, 2004.

[7] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS*, 2006.

[8] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. *SIGMOD*, 2013.

[9] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. *SIGMOD*, 2008.

[10] F. Katsarou, N. Ntarmos, and P. Triantafillou. Subgraph querying with parallel use of query rewritings and alternative algorithms. *EDBT*, 2017.

[11] R. Kimmig, H. Meyerhenke, and D. Strash. Shared memory parallel subgraph enumeration. *IPDPS Workshop*, 2017.

[12] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 2015.

[13] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. In *PVLDB*, 2017.

[14] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 2012.

[15] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. *TODS*, 2014.

[16] R. G. Michael and S. J. David. Computers and intractability: a guide to the theory of np-completeness. *WH Free. Co.*, 1979.

[17] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: on compression and computation. *PVLDB*, 2017.

[18] R. Raman, O. van Rest, S. Hong, Z. Wu, H. Chafi, and J. Banerjee. Pgx. iso: parallel and efficient in-memory engine for subgraph isomorphism. *GRADES*, 2014.

[19] V. N. Rao and V. Kumar. Parallel depth first search. part i. implementation. *IJPP*, 1987.

[20] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, 2015.

[21] X. Ren and J. Wang. Multi-query optimization for subgraph isomorphism search. *PVLDB*, 2016.

[22] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 2008.

[23] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. *SIGMOD*, 2014.

[24] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 2012.

[25] H.-N. Tran, J.-j. Kim, and B. He. Fast subgraph matching on large graphs using graphics processors. *DASFAA*, 2015.

[26] J. R. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 1976.

[27] S. Zhang, S. Li, and J. Yang. Summa: subgraph matching in massive graphs. *CIKM*, 2010.

[28] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 2010.

[29] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: answering sparql queries via subgraph matching. *PVLDB*, 2011.