ParaCOSM: A Parallel Framework for Continuous Subgraph Matching

Haibin Lai
Southern University of Science and Technology
Shenzhen, China
12211612@mail.sustech.edu.cn

Sicheng Zhou
Southern University of Science and Technology
Shenzhen, China
zhousc2021@mail.sustech.edu.cn

Site Fan
Southern University of Science and Technology
Shenzhen, China
fanst2021@mail.sustech.edu.cn

Zhuozhao Li*
Southern University of Science and Technology
Shenzhen, China
lizz@sustech.edu.cn

Abstract

Continuous Subgraph Matching (CSM) has been widely studied, yet most single-threaded algorithms struggle with large query graphs. Existing CSM algorithms on CPU suffer from load imbalance in searching and sequential updates to the index structure.

In this paper, we present ParaCOSM (Parallel COntinuous Subgraph Matching), an efficient parallel framework for existing CSM algorithms on CPU. ParaCOSM leverages two levels of parallelism: inner-update parallelism and inter-update parallelism. Inner-update parallelism uses a fine-grain parallelism approach to decompose the search tree during each CSM query, enabling efficient search for large queries under load balancing. In inter-update parallelism, we introduce an innovative safe-update mechanism that uses multi-threading to verify the safety of multiple updates, thereby enhancing the overall throughput of the system under large-scale update scenarios. ParaCOSM achieves 1.2× to 30.2× speedups across datasets and up to two orders of magnitude faster execution, with up to 71% higher success rates on large query graphs.

Keywords

Continuous Subgraph Matching, Parallel Computing, Graph Algorithms, Dynamic Graphs, Real-time Analysis

ACM Reference Format:
Haibin Lai, Sicheng Zhou, Site Fan, and Zhuozhao Li. 2025. ParaCOSM: A Parallel Framework for Continuous Subgraph Matching. In *54th International Conference on Parallel Processing (ICPP '25), September 08–11, 2025, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3754598.3754603

1 Introduction

Subgraph matching (SM) aims to find all instances of a query pattern Q within a data graph G [34], and is widely used in social network analysis [17] and bioinformatics [5]. While significant progress has been made in subgraph matching over static graphs [19, 27, 31], many real-world applications, such as pattern matching for financial risk control [33] and recommendation systems [11] involve evolving data, leading to the need for *continuous subgraph matching (CSM)*.

CSM focuses on detecting newly emerged or expired matches as the graph evolves [26]. As illustrated in Figure 1, given a query graph Q, a dynamic data graph G, and a graph update stream $\Delta\mathcal{G}$, CSM incrementally identifies matches of Q within G in response to each update $\Delta G \in \Delta\mathcal{G}$.

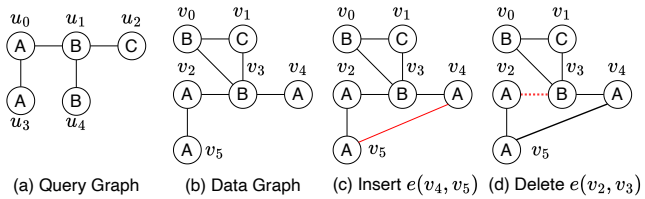

Figure 1: Running example of continuous subgraph matching

As summarized in Table 1, many CSM algorithms follow a common two-stage process: i) updating the data graph and auxiliary structures to prune the search space and construct a search tree; ii) traversing this search tree to find matching embeddings.

Table 1: Existing CSM solutions in recent research. Para: Parallelism. Srch: search method (✓ = backtrack, ✗ = join-based).

System	Para	index $\mathcal{A}$ update	Find Matches	Srch
		CPU Algorithms		
IncIsoMatch [9]	✗	Recomputation	N/A	✓
SJ-Tree [6]	✓	$O(\lvert E(G)\rvert^{\lvert E(Q)\rvert})$	$O(\lvert E(G)\rvert^{\lvert E(Q)\rvert})$	✗
Graphflow [15]	✓	O(1)	$O(d(G)^{\lvert V(Q)\rvert})$	✗
TurboFlux [16]	✗	$O(\lvert E(G)\rvert\lvert V(Q)\rvert)$	$O(d(G)^{\lvert V(Q)\rvert})$	✓
IEDyn [14]	✗	$O(\lvert E(G)\rvert\lvert V(Q)\rvert)$	$O(d(G)^{\lvert V(Q)\rvert})$	✓
Symbi [20]	✗	$O(\lvert E(G)\rvert\lvert E(Q)\rvert)$	$O(d(G)^{\lvert V(Q)\rvert})$	✓
RapidFlow [25]	✓[1]	$O(\lvert E(G)\rvert\lvert E(Q)\rvert)$	$O(d(G)^{\lvert V(Q)\rvert})$	✓
Mnemonic [1]	✓	O(1)	$O(d(G)^{\lvert V(Q)\rvert})$	✓
CaLiG [32]	✗	$O(\lvert E(G)\rvert\lvert E(Q)\rvert)$	$O(\lvert V(G)\rvert^{K})$[2]	✓
NewSP [18]	✗	O(1)	$O(d(G)^{\lvert V(Q)\rvert})$	✓
		GPU Algorithms		
GAMMA [23]	✓	O(1)	$O(d(G)^{V(Q)-1}$ $d(Q)\log(V))$	✗
GCSM [30]	✓	N/A	N/A	✗

[1] RapidFlow has been parallelized in [30].

[2] K is the number of kernel vertices.

*Corresponding author.

ICPP '25, San Diego, CA, USA

ACM ISBN 979-8-4007-2074-1/25/09
https://doi.org/10.1145/3754598.3754603

Although many efficient CSM algorithms have been proposed, most are implemented in a single-threaded fashion, which limits their scalability on large and dynamic graphs. While Mnemonic [1] introduces coarse-grained parallelism at the batch level, it may suffer from load imbalance. GPU-based solutions like GCSM [30] incur high hardware costs and complex programming burdens. Moreover, dynamic search trees and irregular graph topologies in CSM make parallelization fundamentally more challenging than in static subgraph matching. Additionally, most CSM algorithms rely on auxiliary indexing structures for pruning, but these structures typically do not support concurrent updates, which becomes a performance bottleneck.

We identify two key opportunities to accelerate CSM algorithms. First, **parallel decomposition within a single graph update** ΔG: Since search tree exploration dominates the runtime of CSM, and each node in the tree can be treated as an independent computation unit, the search can be decomposed into parallel subtasks. Second, **parallel filtering across updates**: Many real-world updates do not affect query results. This observation enables the design of efficient filtering strategies to safely parallelize update processing.

Based on these insights, we propose ParaCOSM, a general-purpose framework that automatically parallelizes single-threaded CSM algorithms. With minimal user input—specifically, a traversal routine and a filtering rule—ParaCOSM manages parallel execution across both the search and update dimensions, accelerating CSM without requiring modifications to the core algorithm logic.

ParaCOSM adopts a two-level parallelism strategy: i) *Inner-update parallelism* exploits the structure of the search tree by decomposing it into independent subtrees via BFS traversal. These are enqueued into a concurrent task queue and dispatched to worker threads, which execute local enumeration logic and dynamically rebalance the workload. ii) *Inter-update parallelism* takes advantage of the high proportion of *safe updates*—updates that do not affect query matches. ParaCOSM introduces an *update type classifier* (based on label, degree, and index filtering) with a *batch executor* to process safe updates in parallel, while deferring unsafe updates to sequential execution to guarantee correctness.

In summary, the key contributions of our work are as follows:

- We introduce ParaCOSM, a general-purpose parallel framework that automatically parallelizes existing single-threaded CSM algorithms via **two-level parallelism**.
- We propose **inner-update parallelism**, which decomposes the dynamic search tree into independent subtrees, enabling fine-grained parallel exploration without altering the core logic of CSM algorithms.
- We present **inter-update parallelism**, which improves throughput by processing *safe updates* in parallel through an **update type classifier** and a **batch executor**.

Extensive experiments on real-world and synthetic datasets demonstrate the effectiveness of ParaCOSM. Specifically, ParaCOSM achieves significant speedups ranging from 1.2× to 30.2× across multiple datasets. When handling large query graphs, ParaCOSM reduces runtime by up to two orders of magnitude and improves success rates by as much as 71%. It also exhibits strong scalability, maintaining performance as the number of threads increases. Furthermore, ParaCOSM demonstrates excellent load balancing and filtering effectiveness, ensuring efficient resource utilization throughout the matching process.

The rest of the paper is organized as follows. We formulate our problem and review prior work in §2. §3 discusses the motivation, challenges, and opportunities for parallelism. §4 details our system design. §5 presents our performance evaluation. Finally, §6 covers additional related works, and §7 concludes the paper.

2 Preliminaries

2.1 CSM Problem Definition

In this section, we present a formal definition of the CSM problem. Commonly used notations are summarized in Table 2.

Table 2: Frequently used notations

Symbol	Description
g	A labeled undirected graph (V, E)
G	Data graph
Q	Query graph
$V(g), E(g)$	Vertex set and edge set of graph g
L_V, L_E	Vertex and edge labeling functions, L for short
Σ_V, Σ_E	Vertex and edge label sets
$N(u)$	Neighbor set of vertex u
$d(u)$	Degree of vertex u
M	A subgraph isomorphism mapping
$\mathcal{M}$	Set of matches of Q in G
$\Delta\mathcal{G}$	Sequence of graph updates
ΔG	A single graph update
$\Delta\mathcal{M}$	Incremental matching result
$\mathcal{T}$	Search-Tree for each of the enumeration process
$\mathcal{A}$	Auxiliary data structure used during matching

Definition 2.1. Labeled Graph for CSM. Let $g = (V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges. Each vertex and edge is associated with a label through the labeling functions $L_V : V \to \Sigma_V$ and $L_E : E \to \Sigma_E$, respectively. For convenience, we denote both mappings as L. For a vertex $u \in V$, let $N(u)$ be its set of neighbors, and $d(u) = |N(u)|$ its degree.

Definition 2.2. Subgraph Matching. Let Q denote the query graph and G the data graph. The subgraph matching task aims to find all matchings $\mathcal{M}$ of Q in G. A match is a mapping $M : V(Q) \to V(G)$ satisfying the following conditions:

(1) $\forall u \in V(Q), L(u) = L(M(u))$;
(2) $\forall e(u, u') \in E(Q), L(e(u, u')) = L(e(M(u), M(u')))$;
(3) $\forall e(u, u') \in E(Q), e(M(u), M(u')) \in E(G)$.

Example 2.1. The query graph Q and Data Graph G in Figure 1 displays one scenario example. $\{(u_2, v_1), (u_1, v_3), (u_4, v_0), (u_0, v_2), (u_3, v_5)\}$ is a match for Q and G.

Definition 2.3. Graph Stream. We represent the evolution of the data graph as a sequence of updates $\Delta\mathcal{G} = (\Delta G_1, \Delta G_2, \dots)$, where each ΔG is a single edge or node insertion or deletion: $\Delta G = (+/-, e/v)$. Applying ΔG to G yields an updated graph G'. Let $\mathcal{M}$ and $\mathcal{M}'$ be the sets of matches of Q in G and G', respectively. The set of incremental matches $\Delta\mathcal{M}$ is defined as the difference between $\mathcal{M}$ and $\mathcal{M}'$.

Definition 2.4. CSM Problem Statement. Given a data graph G, a query graph Q, and a sequence of updates $\Delta\mathcal{G}$, the goal of CSM is to report the newly or expired matches $\Delta\mathcal{M}$ for each update $\Delta G \in \Delta\mathcal{G}$.

Example 2.2. When the edge $e(v_4, v_5)$ is inserted to G in Figure 1, a positive match $\{(u_2, v_1), (u_1, v_3), (u_4, v_0), (u_0, v_4), (u_3, v_5))\}$ occurs in G'. After that, $e(v_0, v_4)$ is deleted from G'. A negative match $\{(u_2, v_1), (u_1, v_3), (u_4, v_0)\ , (u_0, v_2), (u_3, v_5)\}$ disappears in G''.

Definition 2.5. Compatible Set. Given a query graph Q and a data graph G, let $\phi : V'_Q \to V'_G$ be a partial mapping from a subset of query vertices $V'_Q \subset V_Q$ to data vertices $V'_G \subset V_G$ that satisfies the matching constraints.

For a query vertex $u \in V_Q \setminus V'_Q$ to be matched next, its *compatible set* $C(u, \phi)$ is defined as the set of all such feasible candidate $v \in V_G \setminus V'_G$.

2.2 General CSM Models

Most existing CSM algorithms follow a two-stage design, as illustrated in Figure 2 and Algorithm 1: *offline* and *online* stages.

In the offline stage, CSM algorithms preprocess the query graph Q by computing search indices and determining a matching order (e.g., deciding which query vertex to match first for each incoming update). Many algorithms [26] also construct an index or auxiliary data structure $\mathcal{A}$ to assist in filtering candidate vertices or edges during matching.

In the online stage, CSM algorithms process streaming updates to the data graph. For each incoming edge insertion, the update is first applied to the data graph, followed by an update to the auxiliary structure $\mathcal{A}$. The updated structure is then used to filter vertices or edges and generate candidate sets relevant to the query, which are subsequently used to identify potential matches.

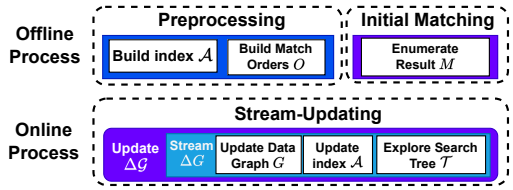

Figure 2: The general process of CSM algorithms.

Then, CSM algorithms trigger a local search process over an abstract search tree $\mathcal{T}$ as Figure 3 shows. This search tree encodes the matching state of the query graph, with the root node as the beginning, and the first layer nodes corresponding to the updated vertices (i.e., the source and target of the modified edge). The search proceeds as illustrated in the function `Find_Matches` in Algorithm 1: starting from the root, it incrementally extends partial matches toward the leaf nodes. Once a complete match is found at a leaf node, the match count is incremented. After all possible search paths have been explored, the processing of the current update is considered complete.

In contrast, deletions are processed in reverse order: positive matches are formed after insertions, whereas negative matches only exist before deletions. Updates involving isolated vertex insertions or deletions are typically trivial in the context of CSM, as they do not affect any valid matches.

Algorithm 1: General CSM Framework

Input: data graph G, query graph Q, update stream $\Delta\mathcal{G}$
Output: incremental matches $\Delta\mathcal{M}$ for each update $\Delta G \in \Delta\mathcal{G}$
/* **Offline process** */
$\mathcal{A} \leftarrow$ *Build_ADS*(G, Q) ;
$O \leftarrow$ Build_Match_Order(Q) ;
$\mathcal{M}_0 \leftarrow$ Find_Initial_Matches$(G, Q, \mathcal{A})$;
/* **Online process** */
foreach $\Delta G \in \Delta\mathcal{G}$ **do**
  if $\Delta G.operation$ *is insertion* **then**
    $G \leftarrow G \cup \{e\}$;
    $\mathcal{A} \leftarrow$ *Update_ADS*$(Q, \mathcal{A}, \Delta G)$;
    $\Delta\mathcal{M} \leftarrow$ *Find_Matches*(Search_Tree(e, O)) ;
  else
    $\Delta\mathcal{M} \leftarrow$ *Find_Matches*(Search_Tree(e, O)) ;
    $\mathcal{A} \leftarrow$ *Update_ADS*$(Q, \mathcal{A}, \Delta G)$;
    $G \leftarrow G \setminus \{e\}$;
Function `Find_Matches`$(\mathcal{T})$:
  $\Delta\mathcal{M} \leftarrow \emptyset$;
  Function `Traverse`(M):
    if $|M| = |Q|$ **then**
      $\Delta\mathcal{M} \leftarrow \Delta\mathcal{M} \cup \{M\}$; **return**
    $u \leftarrow$ `SelectNext`$(Q \setminus M)$;
    $C(u) \leftarrow$ `Compatible_Set_Enum`(u, M) ;
    foreach $v \in C(u)$ ***where*** `Valid`(u, v, M) **do**
      `Traverse`$(M \cup \{(u, v)\})$;
  foreach *task* $t \in \mathcal{T}$ **do**
    `Traverse`$(\{t\})$;
  return $\Delta\mathcal{M}$;

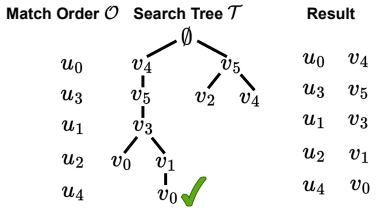

Figure 3: CSM search tree for Figure 1(c), take $O\{u_0, u_3, u_1, u_2, u_4\}$ as an example.

To accelerate online enumeration, existing CSM algorithms typically adopt one or both of the following strategies: *auxiliary data structure (ADS) filtering* and/or *efficient search-tree traversal*. ADS filtering efficiently narrows down candidate vertices or edges before search begins, and traversal aims to reduce redundant exploration during enumeration.

Auxiliary Data Structure Filtering. To reduce the cost of online enumeration, many CSM algorithms utilize ADS that maintain intermediate metadata to efficiently prune the search space. These structures typically encode candidate constraints or dynamic state

Table 3: The proportion of time consumption (%) of auxiliary data structure update (ADS Upd) and Find Matches of each algorithm in the incremental matching process, and algorithm success rate (%), all by query size. GraphFlow does not have an auxiliary data structure.

Algorithm	Query Size 6			Query Size 7			Query Size 8			Query Size 9			Query Size 10		
	ADS Upd	Find Matches	Succ Rate	Aux Upd	Find Matches	Succ Rate	Aux Upd	Find Matches	Succ Rate	Aux Upd	Find Matches	Succ Rate	Aux Upd	Find Matches	Succ Rate
CaLiG	0.49	99.33	100	3.69	95.28	100	0.64	99.25	99	0.07	99.91	94	0.68	99.30	24
GraphFlow	N/A	N/A	92	N/A	N/A	83	N/A	N/A	68	N/A	N/A	0	N/A	N/A	0
NewSP	0.75	53.64	98	0.17	85.59	94	0.02	98.13	90	0.01	98.62	50	0.01	98.85	30
Symbi	2.57	80.78	93	1.16	91.71	85	0.15	98.94	84	0.13	99.36	1	0.09	99.22	1
TurboFlux	1.80	86.01	87	0.72	94.45	83	0.13	98.99	70	0.09	99.30	6	0.28	97.83	1

transitions to eliminate infeasible matches early. While effective in reducing redundant computation, most existing ADS designs are inherently single-threaded and thus difficult to parallelize. We defer a detailed comparison of representative ADS-based methods (e.g., [16, 20, 32]) to §6.

Efficient search-tree traversal. Traditional CSM algorithms typically adopt either *join-based* or *backtracking-based* search strategies. Join-based approaches, such as those in **Graphflow** [15] and **SJ-Tree** [6], traverse the search tree in a BFS manner, while backtracking-based methods like **RapidFlow** [25] employ DFS traversal. The **NewSP** algorithm [18] refines this process by decoupling traversal into two operations: CPT (compatible set computation) and EXP (expansion). CPT adheres to the matching order to retain DFS-style pruning, whereas EXP defers expansion to reduce redundancy and introduces BFS-like flexibility. This hybrid model improves efficiency by avoiding premature exploration and balancing pruning with exploration breadth.

3 Motivation and Challenges

In this section, we discuss the limitations of existing CSM algorithms, highlight the challenges, and our paper's motivation.

3.1 Limitations of Existing CSM Algorithms

CSM has become a critical capability in many real-world applications, where both the data graph G and the query graph Q are typically large and complex. For instance, ByteGraph [33], deployed for financial risk control at ByteDance, performs subgraph matching over dynamic graphs using queries with search depths ranging from 5 to 10 hops, requiring real-time responsiveness. Similarly, GeaFlow [21] handles streaming updates on large graphs, as shown in example experiments on a 1.47-billion-edge graph with up to 10-hop traversals per update.

However, existing CSM algorithms, such as Symbi [20], TurboFlux [16], GraphFlow [15], NewSP [18], and CaLiG [32], struggle to scale effectively under such demanding conditions. To evaluate this limitation, we conducted a preliminary study using the LiveJournal dataset. For each query size (i.e., the number of vertices in the query graph, ranging from 6 to 10), we generated 100 query graphs by extracting subgraphs from the data graph. Each algorithm was executed with a one-hour timeout per query, and a run was considered *successful* if it completed without errors within the time limit.

Figure 4 and Table 3 show two key observations: i) Incremental matching time grows exponentially across all algorithms. For example, CaLiG's average processing time increases from 1.68×10^4ms at size 6 to 2.82×10^6ms at size 10, a 168× increase. ii) Success rates degrade substantially. While algorithms perform well at size 6 (e.g., CaLiG: 100%, NewSP: 98%), performance drops markedly at size 10 (e.g., CaLiG: 24%, GraphFlow: 0%). This inverse relationship between query size and performance highlights the fundamental scalability challenge in CSM, underscoring the need for more efficient and parallelizable solutions.

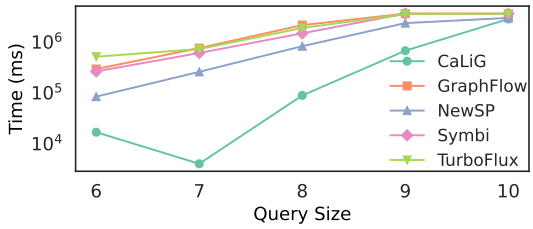

Figure 4: Computing time of different single-threaded CSM algorithms running on different query sizes.

3.2 Challenges of Parallelization

Most existing CSM algorithms are designed for single-threaded execution on CPUs. This observation motivates exploring parallelism to improve scalability and throughput. However, parallelizing subgraph matching in dynamic settings introduces two key challenges:

Challenge 1: Load imbalance caused by coarse-grained parallelism. In static subgraph matching, the search space is derived from a fixed data and query graph. This allows the full search tree to be decomposed into a few large, independent tasks, enabling effective coarse-grained parallelism with good load balance [34].

However, CSM processes high-frequency update streams, where each update triggers a localized search over a dynamically generated search tree. These trees vary in size and structure across updates, and are often highly asymmetric due to aggressive pruning by auxiliary data structures (e.g., CaLiG [32], DCS [20]). As a result, tasks differ significantly in complexity: some terminate early due to pruning, while others require deep exploration. This leads to *fragmented* and *unbalanced* workloads, where some threads finish early

while others remain active, leaving CPU resources underutilized. Coarse-grained parallelism, which assumes large and predictable task sizes, fails to adapt to this dynamic and irregular workload, resulting in poor real-time load distribution.

Challenge 2: Bottleneck caused by sequential auxiliary data structure updates. Auxiliary data structures such as DCG [16] and DCS [20] play a central role in pruning the search space and improving efficiency in CSM algorithms. However, these structures are typically designed for single-threaded use and do not support concurrent updates. In a multi-threaded environment, this design becomes a bottleneck: threads must serialize access to the shared structure, which stalls progress and limits scalability.

3.3 Opportunity

We identify two key opportunities for parallelization in CSM.

The first opportunity lies within a single graph update ΔG, where the local search tree can be explored in parallel—referred to as *inner-update* parallelism. The second lies across multiple graph updates, where independent updates can be processed concurrently in batches—termed *inter-update* parallelism. These two dimensions naturally align with the structure of CSM: each update triggers an independent search process (inner-update), while many such updates can be handled concurrently (inter-update). Leveraging both forms of parallelism can substantially improve system throughput.

Inner-update parallelism. Within each update, the structure of the search tree $\mathcal{T}$ in CSM offers unique opportunities for load balancing. Our analysis in Table 3 shows that the *find match* phase dominates the total computation time (often exceeding 90%), whereas auxiliary data structure updates contribute minimally.

Each node in $\mathcal{T}$ represents a computational unit, forming modular subtrees that can be executed independently. This property enables decomposition of $\mathcal{T}$ into subtasks that can be dynamically assigned to threads, ensuring balanced workload distribution.

Inter-update parallelism. We observe a key statistical property of real-world graph workloads: the vast majority of graph updates do not affect the matching results. We refer to such updates as *safe* updates, in contrast to *unsafe* updates, which may alter the match set $\Delta\mathcal{M}$.

To quantify this property, we analyzed four widely used datasets, LSBench, LiveJournal, Orkut, and Amazon, and generated 100 random query graphs per query size. As shown in Table 4, over 98.4% of updates are classified as safe, indicating that unsafe updates are exceedingly rare in practice. This presents a significant opportunity for parallelism: by dynamically classifying updates as safe or unsafe at runtime, we can process safe updates in parallel without compromising correctness. Since unsafe updates are infrequent, they can be processed sequentially to ensure result consistency. This hybrid strategy enables efficient inter-update parallelism and achieves substantial speedups in high-throughput CSM systems.

4 ParaCOSM Design

To accelerate CSM, we propose ParaCOSM, a general-purpose framework that automatically parallelizes existing single-threaded CSM algorithms conforming to the model described in §2.2, including algorithms such as NewSP, CaLiG, and Symbi.

Table 4: Average of unsafe update percentage (%).

Dataset	Query Size				
	6	7	8	9	10
LSBench	1.0794	1.5688	1.1622	0.4257	0.3192
LiveJournal	0.3028	0.3618	0.3848	0.3266	0.3057
Orkut	0.0010	0.0010	0.0010	0.0012	0.0011
Amazon	0.5356	0.6759	0.5249	0.5254	0.5724

As illustrated in Figure 5, ParaCOSM allows users to easily integrate their own CSM algorithms by providing two key functions: i) a search-tree traversal procedure, which is embedded into the inner-update executor; ii) a custom filtering method, which is embedded into the inter-update executor.

Once these functions are provided, ParaCOSM manages all parallelization details automatically. Given a data graph G, query graph Q, and a stream of updates $\Delta\mathcal{G}$, ParaCOSM orchestrates parallel execution and maintains the corresponding enumeration results alongside the evolving graph.

ParaCOSM adopts a two-level parallelism strategy with two dedicated executors: an **inner-update executor** for parallelizing search tree exploration within each update, and an **inter-update executor** for batch processing of multiple updates.

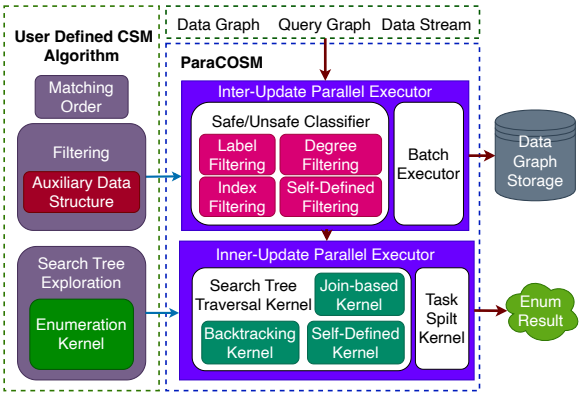

Figure 5: ParaCOSM architecture.

4.1 Inner-update Parallelism

To address the load imbalance challenge due to irregular match orders and aggressive pruning by auxiliary structures $\mathcal{A}$, we introduce **inner-update parallelism**—a fine-grained, task-based parallel mechanism that accelerates single-threaded CSM algorithms without altering their core logic. As illustrated in Algorithm 2, the inner-update executor operates in two phases:

Initialization Phase: The executor performs a breadth-first traversal of the search tree $\mathcal{T}$ to identify root-level candidate sets and decompose the search space into independent subtrees. Each subtree is treated as a subtask and enqueued into a concurrent task queue CQ. Once enough subtasks are generated, the main thread dispatches them to worker threads and transitions to the next phase.

Parallel Execution Phase: Each worker processes its assigned subtask by performing local enumeration, directly reusing the logic of existing single-threaded CSM algorithms. To ensure load balance,

workers monitor the system and dynamically offload remaining subtasks to CQ when idle threads are detected. This adaptive task-sharing mechanism enables fine-grained parallelism, even under skewed or unpredictable workloads.

Algorithm 2: Inner-update Executor

Input: Data graph G, root of search tree $root$, maximum depth max_depth
Output: Incremental matching results
Global Variables:
SPILT_DEPTH `// defined depth for spilting`
$CQ \leftarrow \emptyset$ `// Concurrent task queue`
$T \leftarrow \{thread_1, thread_2, \ldots, thread_{num}\}$ `// Thread pool`
Initialization (main thread):
CQ.push($Search\_Tree(e, O)$) ;
while $|CQ| < num\_threads$ **do**
  foreach $task \in CQ$ **do**
    $children \leftarrow$ Traverse_Next_Layer($task$) ;
    CQ.push($children$) ;
Parallel Execution:
foreach $thread \in T$ ***in parallel*** **do**
  while $CQ \neq \emptyset$ **do**
    $task \leftarrow CQ$.pop() ;
    `Parallel_Find_Matches(task)` ;
    Report($result$) ;
Function `Parallel_Find_Matches(task)`**:**
  if $task.depth < max\_depth$ **then**
    $subtasks \leftarrow$ Traverse_Next_Layer($task$) ;
    if `HasIdleThreads()` ***and*** $CQ.is\_empty()$ ***and*** $task.depth < SPILT\_DEPTH$ **then**
      CQ.push($subtasks$) ;
    else
      Perform the same search as the single-threaded function *Traverse*
      foreach $subtask \in subtasks$ **do**
        $result \leftarrow$ `Parallel_Find_Matches(subtask)` ;

Our method effectively addresses load imbalance by distributing search tree subtasks dynamically based on runtime thread availability. Unlike static task allocation, which struggles with unpredictable workloads in CSM, our fine-grained task splitting ensures that computational resources are fully utilized.

4.2 Inter-update Parallelism

As mentioned earlier, a large proportion of updates in real-world workloads are *safe*, meaning they do not affect the matching results when processed in isolation. To leverage this, we propose **inter-update parallelism**, which enables batch processing of update streams. However, implementing inter-update parallelism faces two key challenges: i) efficiently classifying updates as safe or unsafe, and ii) safely executing updates in parallel without compromising correctness.

To address these challenges, ParaCOSM introduces two components: an *update type classifier* and a *batch executor*. The classifier determines whether an update can be safely processed in parallel, while the executor schedules and applies updates accordingly.

Update type classifier. The update type classifier determines whether a given update ΔG can be processed in parallel without interfering with others in the same batch. It applies a three-stage filtering process:

- *Label filtering:* The labels of the two endpoints of the updated edge v_1 and v_2 must match the corresponding query vertices u_1 and u_2, respectively.
- *Degree filtering:* The degrees of v_1 and v_2 must not be smaller than those of u_1 and u_2 to preserve the matching feasibility.
- *Candidate filtering:*. The update must not impact the auxiliary data structure $\mathcal{A}$. Since $\mathcal{A}$ typically indexes neighborhood structures, candidate sets, or summaries to accelerate pruning, updates that modify it may introduce side effects and invalidate concurrent matching. Thus, such updates are classified as *unsafe* and deferred.

As a result, the classifier returns `true` for safe updates and `false` for unsafe ones. Safe updates are forwarded directly to the batch executor for parallel execution, while unsafe ones are isolated for sequential handling.

Batch executor. The batch executor applies a sequence of updates $\Delta\mathcal{G}$ to the dynamic graph G while maximizing parallelism and preserving correctness. Its key principle is to decouple update classification from application, enabling aggressive parallelism for safe updates and controlled sequential fallback for unsafe ones. The process comprises three stages:

Parallel classification and execution: As illustrated in Figure 6, each update $\Delta G \in \Delta\mathcal{G}$ is concurrently passed to the classifier by k worker threads. Safe updates are immediately applied to the graph G in place, with no need for locking or synchronization, since they do not modify $\mathcal{A}$ or affect other ongoing updates.

Detecting unsafe updates: When an update is classified as unsafe, it is excluded from the parallel phase and handled in a sequential pass. Importantly, the appearance of an unsafe update invalidates the safety assumptions of all subsequent updates in the current batch. Therefore, once an unsafe update is detected, all remaining updates are deferred to the next batch and re-evaluated after G and $\mathcal{A}$ are updated. Unsafe updates are then processed sequentially by the inner-update executor to ensure correctness and consistency.

Batch coordination: The main loop maintains a pool of unprocessed updates and dispatches them in batches of size k. After each round, the executor checks the classifier flags. If all updates are safe, the batch is completed. Otherwise, the appearance of an unsafe update triggers a switch to sequential mode, and all subsequent updates are deferred to preserve consistency.

4.3 Theoretical Analysis

Overall speedup analysis. We analyze the theoretical speedup of ParaCOSM by modeling the time cost of CSM under parallel execution. Let $|\Delta\mathcal{G}|$ be the number of updates, and let γ denote the ratio of *safe* updates. Denote the time for auxiliary data structure

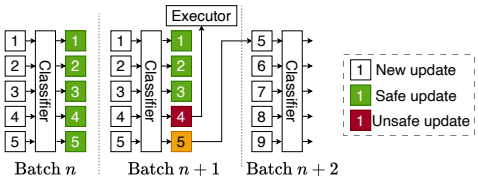

Figure 6: Safe/unsafe classifier. The executor is responsible for updating G and $\mathcal{A}$ and finding incremental matches. Batch n: all updates are classified as safe, so the add edge executor is not invoked. Batch $n+1$: updates 1-3 are safe and 4 is unsafe. So update 4 invokes the add edge executor, and update 5, which is after an unsafe update, is moved to batch $n+2$.

maintenance as T_{ADS}, and the time for match enumeration as T_{FM}. Let M and N be the number of threads for ADS update maintenance and match searching, respectively. Assuming ideal linear scalability (i.e., perfect speedup proportional to thread count), the total runtime of our parallel framework is:

$$T_{csm} = |\Delta\mathcal{G}| \left[(1-\gamma) \left(T_{ADS} + \frac{T_{FM}}{N} \right) + \gamma \frac{T_{ADS}}{M} \right] \tag{1}$$

$$= |\Delta\mathcal{G}| \left[\left(1 + \gamma \left(\frac{1}{M} - 1 \right) \right) T_{ADS} + \frac{1-\gamma}{N} T_{FM} \right] \tag{2}$$

This expression shows the influence of the proportion of safe updates on the total computing time: unsafe updates incur both T_{ADS} and T_{FM} costs, while safe updates only require parallelized auxiliary updates.

Reference values. For different CSM algorithms, the values of T_{ADS} and T_{FM} are listed in Table 1. The following example shows how to compute the theoretical optimal speedup ratio using the above expression. Suppose $N = 10$, $M = 10$, and $\gamma = 0.4$. The runtime simplifies to:

$$T_{csm} = |\Delta G| \, (0.64\, T_{ADS} + 0.06\, T_{FM}) \tag{3}$$

As the proportion of safe updates γ increases, the framework benefits more from update parallelism (M), while the impact of matching (T_{FM}) diminishes through matching parallelism (N).

Safe update ratio analysis. Here we use label filtering in three-stage filtering to theoretically estimate the number of safe updates. Consider an edge insertion into the dynamic graph G. For a *safe update*, the inserted edge does not match any edge in Q. Both graphs have labeled vertices and edges, with vertex label set L_V and edge label set L_E. Consider inserting an edge $e_1 = (v_1, v_2)$ into G. The update is *unsafe* if the label triple $(L(v_1), L(v_2), L(e_1))$ matches that of some edge in $E(Q)$. Assuming uniform label distribution, the probability that e_1 matches a specific edge in $E(Q)$ is:

$$P(\text{match}) = \frac{1}{|L(E)|} \times \frac{1}{|L(V)|} \times \frac{1}{|L(V)|} = \frac{1}{|L(E)||L(V)|^2}$$

Thus, the probabilities of an unsafe and safe update are:

$$P(\text{unsafe}) = |E(Q)| \times \frac{1}{|L(E)||L(V)|^2} = \frac{|E(Q)|}{|L(E)||L(V)|^2}$$

$$P(\text{safe}) = 1 - \frac{|E(Q)|}{|L(E)||L(V)|^2}$$

We can calculate it by substituting the metadata of the LiveJournal dataset into this expression using an query graph with 6 edges:

$$P(\text{unsafe}) = \frac{6}{30^2 \cdot 1} = 0.677\%, \quad P(\text{safe}) = 99.33\%.$$

Hence, the probability of a safe update is at least 99.33%, indicating that safe updates are highly likely in practical scenarios.

5 Experiments

We implement five CSM algorithms: CaLiG, GraphFlow, NewSP, Symbi, and TurboFlux, into parallel versions using ParaCOSM[1] and evaluate their performance. All algorithms are implemented in C++, compiled with the Intel `icpx` compiler.

5.1 Experimental Setup

Testbed. We conducted the evaluation on a machine with an Intel Xeon Platinum 8380 CPU (80 physical cores / 160 threads), 250 GB of memory, running Ubuntu 22.04.

Datasets. We selected four representative datasets, ranging in size from KB to GB, and with different label and degree characteristics. The *Amazon* dataset is a product co-purchasing network collected from the Amazon website. The *LiveJournal* is a large-scale online community network. The *LSBench* is a synthetic dynamic social graph generated by the Linked Stream Benchmark. The *Orkut* dataset is a social network from a former Google service. Previous CSM study [26] has sampled insertion graphs by randomly sampling 10% edges from the original graphs for Amazon, LiveJournal, and LSBench. So we followed this way to sample from Orkut. Detailed graph information is summarized in Table 5.

Table 5: Summary of Datasets

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert L(V)\rvert$	$\lvert L(E)\rvert$	$d(G) = \frac{2\lvert E\rvert}{\lvert V\rvert}$
Amazon	403,394	2,433,408	6	1	12.06
LiveJournal	4,847,571	42,841,237	30	1	17.68
LSBench	5,210,099	20,270,676	1	44	7.78
Orkut	3,072,441	117,185,083	20	20	20

Query Generation. Following previous studies [16, 26], we generate query graphs by extracting subgraphs from the data graph randomly. Specifically, for each dataset, we construct query graphs with vertex counts of 6, 7, 8, 9, and 10. For each query size, we generate 100 query graphs by initiating random walks from randomly selected seed nodes.

Metrics. All reported *times* in this section measure incremental matching time. *Success rate* is defined as the percentage of queries completed within one hour, with queries taking longer being marked as timeouts. For fairness, when comparing with CaLiG, which does not support edge labels, we remove edge labels from all datasets during CaLiG evaluation. Unless otherwise specified, all experiments were tested multiple times, and the average runtime was reported.

[1] open source in https://github.com/SUSTech-HPCLab/ParaCOSM.git

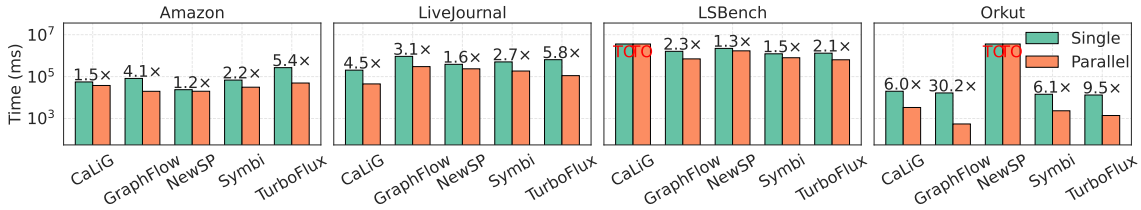

Figure 7: The speedup ratio of ParaCOSM algorithms with 32 threads to single-threaded algorithms on different datasets. TO means timeouts.

5.2 Overall Comparison

Overall time efficiency. As shown in Figure 7, we report the speedup achieved by ParaCOSM with 32 threads, compared to their original single-threaded implementations. Experiments were conducted on four datasets.

The results demonstrate that ParaCOSM provides significant acceleration across all tested algorithms and datasets. Notably, GraphFlow and TurboFlux show the highest improvements, achieving speedups of 3× to 6× on the Amazon and LiveJournal datasets, over 2× on LSBench, and up to 30.2× and 9.5× on the Orkut dataset, respectively. Other algorithms also have benefits, with speedups ranging from 1× to 7× depending on the dataset.

It is worth noting that the acceleration on LSBench is generally less pronounced compared to other datasets. This may be attributed to LSBench having the lowest average degree $d(G)$, which leads to frequent expansion of the search tree and higher overhead from managing the concurrent task queue. Additionally, CaLiG consistently fails to complete within the time limit on LSBench. Beyond the structural properties of the graph, a key contributing factor is that the original CaLiG implementation does not support edge label matching—an essential feature for LSBench, which contains a large number of edge-labeled edges in a sizable graph.

Time efficiency on large query graphs. We evaluate the performance of ParaCOSM with 32 threads on query graphs of varying sizes using the LiveJournal dataset. Figure 8 shows the speedup computed for successful queries versus the query size for different parallelized algorithms . Our parallel implementation exhibits substantial speedups across all algorithms. Notably, our proposed three-stage filtering technique proves especially effective for query sizes 6 and 7, achieving significant speedup in these cases. For larger query sizes, our parallel algorithm still delivers consistent and effective acceleration.

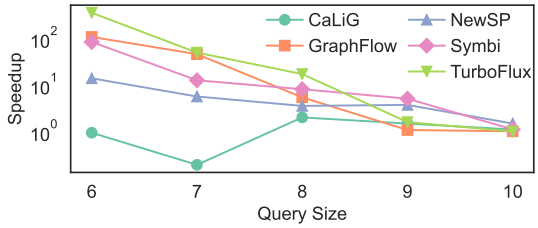

Figure 8: ParaCOSM with 32 threads speedup on big query graphs under the time limit.

In addition to reducing runtime, ParaCOSM also significantly improves the success rate for large queries. As shown in Table 6, we report the percentage of successful runs for each algorithm with ParaCOSM and query size. When the query size is below 8, nearly all algorithms reach close to 100% success with ParaCOSM. Even for more complex queries, ParaCOSM consistently improves success rates. Notably, Symbi sees a 71% improvement at query size 9 after adopting our approach.

Table 6: Success rate of parallel CSM algorithms on LiveJournal with 32 threads. "+" and "−" indicate the change in success rate compared to their single-threaded results in Table 3.

Alg. (Parallel)	Query Size				
	6	7	8	9	10
CaLiG	100 (+0)	99 (-1)	96 (-3)	84 (-10)	40 (+16)
GraphFlow	100 (+8)	97 (+14)	92 (+24)	11 (+11)	0 (+0)
NewSP	99 (+1)	95 (+1)	98 (+8)	55 (+5)	45 (+15)
Symbi	100 (+7)	99 (+14)	92 (+6)	72 (+71)	13 (+12)
TurboFlux	100 (+13)	98 (+15)	94 (+24)	32 (+26)	0 (-1)

Scalability. We evaluate the scalability of ParaCOSM on the LiveJournal. For each thread configuration (8, 16, 32, 64, and 128 threads), we randomly select 10 distinct query graphs. Speedup is computed relative to the single-threaded baseline. The results are presented in Figure 9.

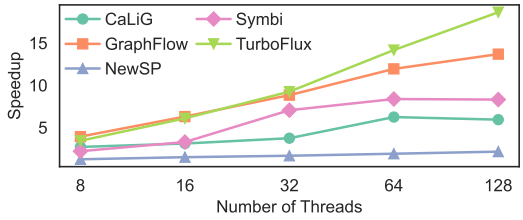

Figure 9: Speedup of ParaCOSM with different number of threads.

The experiments demonstrate strong scaling characteristics across all evaluated algorithms. TurboFlux achieves the highest improvement, scaling from 3.4× (8 threads) to 18.6× (128 threads), with especially large gains at higher thread counts. GraphFlow shows steady scalability, rising from 3.9× to 13.7× as threads increase. Symbi and CaLiG show nonlinear scaling behavior: both achieve their peak speedups (7.0× and 3.2×, respectively) at 32 threads, but the performance gain plateaus or declines beyond that point. NewSP demonstrates more limited scalability, reaching up to 2.1× speedup at 128 threads. Overall, these results confirm that ParaCOSM maintains high parallel efficiency as the number of threads increases.

The particularly strong performance in TurboFlux and GraphFlow underscores the effectiveness of our parallelization strategies and their suitability for highly concurrent environments.

5.3 Breakdown Comparison

Inner update load balancing. To evaluate the effectiveness of the load balancing strategy in inner-update parallelism, we run experiments on the LiveJournal with 32 threads. For each run, we recorded the execution time of each thread under both load-balanced and unbalanced conditions. Figure 10 presents the cumulative distribution function (CDF) of the per-thread execution times.

The results reveal that load balancing greatly improves workload distribution. Without it, some threads finish in about 10 seconds while others take up to 50 seconds, indicating severe task skew and underutilized resources. In contrast, with load balancing enabled, most threads complete in roughly 20 seconds, reflecting a more uniform distribution. This leads to better resource utilization and shorter total search time.

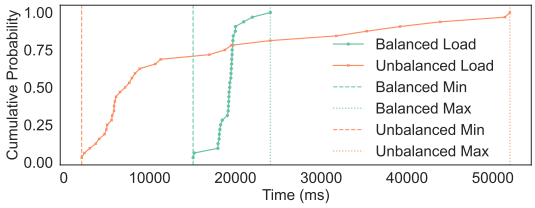

Figure 10: CDF of thread execution time for load-balanced vs. unbalanced cases. Algorithm is GraphFlow.

Inter update efficiency. Figure 11 illustrates the effectiveness of the inter-update parallelism strategy described in §4.2. We conducted experiments on the Orkut and measured the execution times before and after applying the inter-update parallel mechanism with 32 threads. The results show that this mechanism provides substantial acceleration across all tested algorithms, with speedups exceeding 3.47× in every case. Notably, Symbi achieves a 7.39× improvement, demonstrating particularly strong responsiveness to inter-update parallelism.

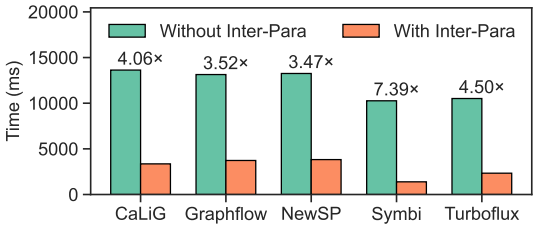

Figure 11: Inter-update mechanism speedup.

Using the same experimental setup, we further evaluated the efficiency of the proposed three-stage filtering strategy. The results are shown in Figure 12. Most CSM algorithms typically apply label and degree matching as basic pruning steps. By using these two filters to classify edges as safe or unsafe for parallel processing, we find that more than 99.6% of edges can be classified as safe and processed in parallel.

The remaining candidate edges, those that satisfy both label and degree filter, are then passed through the auxiliary data structure

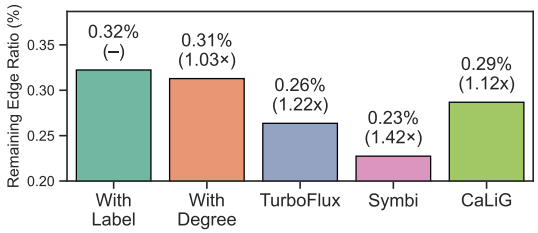

Figure 12: Three-stage filtering pruning effectiveness.

(ADS) filters. Across all three evaluated algorithms (TurboFlux, Symbi, and CaLiG), the ADS filters prune more than 99.7% of the remaining edges. These results validate the high effectiveness of our three-stage filtering strategy in enabling efficient and safe inter-update parallelism.

6 Related Work

Streaming Graph Computation. Many real-world applications require processing dynamically evolving graphs. RisGraph [10] is a real-time streaming system that supports low-latency analysis (e.g., BFS, SSSP) per update with high throughput. It achieves inter-update parallelism via domain-specific concurrency control, classifying updates as safe or unsafe for parallel execution. Tesseract [4] performs pattern mining on evolving graphs by efficiently distributing updates to worker nodes. KickStarter [29] accelerates convergence of monotonic algorithms via trimming and approximation. LSGraph [22] improves memory efficiency with hierarchical indexing and locality-aware updates. However, these systems do not target complex pattern queries in continuous subgraph matching, which remains the focus of our work.

Static subgraph matching. Subgraph matching has been widely studied since Ullmann's algorithm [28]. Later methods, including VF2 [7], GraphQL [13], SPATH [24], TurboISO [12], CFL [3], and CECI [2], improve performance through optimized matching orders and pruning. However, these approaches target static graphs, while our focus is dynamic graph scenarios.

Continuous subgraph matching. Early CSM approaches like IncIsoMat [8], GraphFlow [15], and TurboFlux [16] focus on incremental computation via affected regions, worst-case joins, and auxiliary data structures. RapidFlow [25], newSP [18], and CaLiG [32] improve pruning or decomposition strategies but remain single-threaded. Mnemonic [1] assigns a single thread to each update, achieving batch-level but coarse-grained parallelism. GPU-based methods [23, 30] offer speedups at high hardware and development cost. In contrast, ParaCOSM enables efficient CPU-based intra-update parallelism with minimal algorithm modification.

7 Conclusion

In this paper, we proposed ParaCOSM, a parallel framework for accelerating Continuous Subgraph Matching (CSM). ParaCOSM exploits both inner-update and inter-update parallelism to improve performance. The inner-update executor employs fine-grained task decomposition to achieve effective load balancing, while the inter-update executor uses a three-stage filtering strategy to reduce redundant computation. Extensive experiments show that ParaCOSM consistently outperforms single-threaded baselines across diverse

datasets and query sizes. A detailed breakdown analysis confirms the contribution of each component to overall performance gains.

Acknowledgments

We thank Dr. Shengxin Liu for his valuable suggestions on this work. This work was supported in part by the National Natural Science Foundation of China Grant No. 62202216, the Guangdong Basic and Applied Basic Research Foundation Grant No. 2023A1515010244, and the Shenzhen Science and Technology Program Grant 20231121101752002. This work was also supported by Center for Computational Science and Engineering at Southern University of Science and Technology.

References

[1] Bibek Bhattarai and Howie Huang. 2022. Mnemonic: A Parallel Subgraph Matching System for Streaming Graphs. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 313–323.

[2] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, 1447–1462.

[3] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. 1199–1214.

[4] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. 2021. Tesseract: distributed, general graph pattern mining on evolving graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. ACM, 458–473.

[5] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14, Suppl 7 (2013), S13.

[6] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. arXiv:1503.00849 [cs.DB] https://arxiv.org/abs/1503.00849

[7] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE PAMI* 26, 10 (2004), 1367–1372.

[8] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. 2011. Incremental graph pattern matching. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 925–936.

[9] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Trans. Database Syst.* 38, 3, Article 18 (Sept. 2013), 47 pages.

[10] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. ACM, 513–527.

[11] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-time twitter recommendation: online motif detection in large dynamic graphs. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1379–1380.

[12] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. 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*. 337–348.

[13] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*. 405–418.

[14] M. Idris, M. Ugarte, S. Vansummeren, et al. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *The VLDB Journal* 29, 4 (2020), 619–653.

[15] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, 1695–1698.

[16] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. ACM, 411–426.

[17] Sofiane Lagraa, Martin Husák, Hamida Seba, Satyanarayana Vuppala, Radu State, and Moussa Ouedraogo. 2024. A review on graph-based approaches for network security monitoring and botnet detection. *International Journal of Information Security* 23, 1 (2024), 119–140.

[18] Ziming Li, Youhuan Li, Xinhuan Chen, Lei Zou, Yang Li, Xiaofeng Yang, and Hongbo Jiang. 2024. NewSP: A New Search Process for Continuous Subgraph Matching over Dynamic Graphs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3324–3337.

[19] Zhiheng Lin, Ke Meng, Changjie Xu, Weichen Cao, and Guangming Tan. 2025. Jupiter: Pushing Speed and Scalability Limitations for Subgraph Matching on Multi-GPUs. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. ACM, 558–572.

[20] Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F. Italiano, and Wook-Shin Han. 2021. Symmetric continuous subgraph matching with bidirectional dynamic programming. *Proc. VLDB Endow.* 14, 8 (April 2021), 1298–1310.

[21] Zhenxuan Pan, Tao Wu, Qingwen Zhao, Qiang Zhou, Zhiwei Peng, Jiefeng Li, Qi Zhang, Guanyu Feng, and Xiaowei Zhu. 2023. GeaFlow: A Graph Extended and Accelerated Dataflow System. *Proc. ACM Manag. Data* 1, 2, Article 191 (June 2023), 27 pages.

[22] Hao Qi, Yiyang Wu, Ligang He, Yu Zhang, Kang Luo, Minzhi Cai, Hai Jin, Zhan Zhang, and Jin Zhao. 2024. LSGraph: A Locality-centric High-performance Streaming Graph Engine. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. ACM, 33–49.

[23] Linshan Qiu, Lu Chen, Hailiang Jie, Xiangyu Ke, Yunjun Gao, Yang Liu, and Zetao Zhang. 2024. GPU-Accelerated Batch-Dynamic Subgraph Matching . In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 3204–3216.

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

[25] Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. 2022. RapidFlow: an efficient approach to continuous subgraph matching. *Proc. VLDB Endow.* 15, 11 (July 2022), 2415–2427.

[26] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. 2022. An in-depth study of continuous subgraph matching. *Proceedings of the VLDB Endowment* 15, 7 (2022), 1403–1416.

[27] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.* 5, 9 (May 2012), 788–799.

[28] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1, 31–42.

[29] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, 237–251.

[30] Yihua Wei and Peng Jiang. 2024. GCSM: GPU-Accelerated Continuous Subgraph Matching for Large Graphs. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1046–1057.

[31] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: Scaling Subgraph Isomorphism on Distributed Multi-GPU Systems Using Trie Based Data Structure. In *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[32] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction. *Proc. ACM Manag. Data* 1, 1, Article 15 (May 2023), 26 pages.

[33] Wei Zhang, Cheng Chen, Qiange Wang, Wei Wang, Shijiao Yang, Bingyu Zhou, Huiming Zhu, Chao Chen, Yongjun Zhao, Yingqian Hu, Miaomiao Cheng, Meng Li, Hongfei Tan, Mengjin Liu, Hexiang Lin, Shuai Zhang, and Lei Zhang. 2024. BG3: A Cost Effective and I/O Efficient Graph Database in Bytedance. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. ACM, 360–372.

[34] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proc. ACM Manag. Data* 2, 1, Article 60 (March 2024), 29 pages.