Distributed Exact Subgraph Matching in Small Diameter Dynamic Graphs

Charith Wickramaarachchi[1], Rajgopal Kannan[1], Charalampos Chelmis[2], and Viktor K. Prasanna[1]

[1]University of Southern California, Los Angeles, USA
[2]University at Albany - SUNY, NY, USA
[1]{*cwickram,rajgopak,prasanna*}*@usc.edu*
[2]*cchelmis@albany.edu*

***Abstract*—Subgraph isomorphism is a fundamental graph problem with many applications. Due to its NP-Hard nature, subgraph isomorphism in large dynamic graphs is considered as a challenging problem. In this paper, we present a distributed graph pruning algorithm (D-IDS) for dynamic graphs to enable efficient subgraph isomorphism. D-IDS continuously maintains the maximum dual simulation match in a dynamic graph. We develop D-ISI, a distributed incremental algorithm for subgraph isomorphism that utilizes D-IDS. We evaluated our algorithms on a commodity cluster in Amazon EC2 using real world graph datasets. Our evaluation results show that the graph pruning technique is highly effective on graphs with small diameter where it achieves over 60% reduction in graph size.**

***Index Terms*—subgraph isomorphism, dynamic graphs, distributed, small diameter graphs**

I. Introduction

Graphs are fundamental in representing and analyzing many modern datasets in diverse application domains, including online social networks, communication networks, road networks and cyber-physical systems. Many of these graph data sets have a small diameter where nodes are separated only by few hops [1]. The massive scale and computational complexity of processing such graphs, accompanied by the recent and the enormous growth of data center technologies and cloud infrastructures have reignited interest in the development of *highly scalable* and *distributed* graph processing platforms [2], [3]. These distributed systems consist of a large number of commodity machines with small individual memory and CPU footprints compared to traditional super-computing systems (e.g. IBM Blue Gene, Cray XMT, etc.). Distributed graph algorithms that can make use of these systems/infrastructures are gaining traction as a result [4], [5], [6].

Subgraph isomorphism (SIM) is a fundamental graph problem with wide and varied applications such as substructure matching in chemical components, analysis of social network structures, threat detection in social networks and cyber systems, cryptography and security [7], [8], [9], [10], [11]. For the case of static graphs, several algorithms have been developed for SIM [12], [13], [14]. These fall into two main categories ([15], [16]) 1) exploratory methods or 2) partial query match-join methods. Exploratory methods start from a single vertex in the data graph and explore the rest of the vertices, evaluating them against structural properties and matching constraints to determine whether they can be matched to the query. Partial query match-join methods find partial candidates for different query vertices and try to join them incrementally to create the complete pattern.

The fast changing nature of modern graphs and low latency application requirements [10] demand low latency solutions for *dynamic* graphs. Research on incremental algorithms that work on streams of *edge updates* has gained a new thrust as result. While there has been a lot of work on parallel and distributed incremental algorithms for dynamic graphs in general [17], [18], [16], not much effort has been devoted to the SIM problem. Due to its NP-Hard nature, exact SIM in large dynamic graphs is considered challenging [16] and thus much of the existing work has focused on approximate algorithms [16]. However, we assert that exact SIM is an essential requirement in several mission critical application domains, for example, fully automated cyber intrusion detection and prevention systems [19] and thus the exact SIM problem is of independent interest.

Driven by the need to develop low latency and exact solutions to the SIM problem in dynamic graphs, in this paper, we take a practical approach towards developing incremental distributed/parallel algorithms. Our main technical contributions are summarized below:

- We identify the limitations of incremental SIM based on neighborhood search for small diameter graphs. To address these limitations, we present a novel distributed graph pruning technique for dynamic graphs (D-IDS) that preserves subgraph isomorphism matches.
- We present a distributed incremental algorithm for exact SIM that utilizes the above mentioned graph pruning technique.
- Via experimental evaluations, we demonstrate the effectiveness of our algorithms using real world graph data sets on a commodity cluster environment in Amazon EC2.

The rest of the paper is organized as follows. In Section II, we introduce necessary definitions of concepts used in this paper. We present related work in Section III. Section IV and V presents our proposed algorithms. Section VI provides a summary of our experimental evaluation results. Section VII draws our conclusions and discuss future research directions.

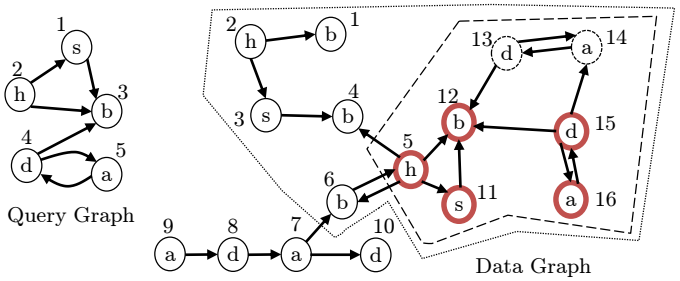


Fig. 1: Vertices $\{5, 11, 12, 15, 16\}$ in data graph match the query graph via subgraph isomorphism. Vertices $\{5, 11, 12, 13, 14, 15, 16\}$ match the query graph via dual simulation and vertices $\{1, 2, 3, 4, 5, 6, 11, 12, 13, 14, 15, 16\}$ match via graph simulation.

II. Background and Problem Statement

The *subgraph matching* problem takes as input a data graph $G = (V, E, l)$, query graph $Q = (V^q, E^q, l^q)$ where both G and Q consists of a set of vertices V/V^q, set of edges E/E^q and a vertex label function label function l/l^q that associates each vertex with a label from label set L. It finds all the subgraphs of G that match Q based on a given matching criteria. There are three main criteria of interest to us in this paper:

1. *Subgraph Isomorphism [20]*: Q matches data graph G under subgraph isomorphism iff there exists subgraph $G^s \subseteq G$ and bijective function $f : V^q \rightarrow V^s$ such that for any two nodes $v_i, v_j \in V^q$, $(v_i, v_j) \in E^q \Rightarrow (f(v_i), f(v_j)) \in E^s$, $l^q(v_i) = l^s(f(v_i))$ and $l^q(v_j) = l^s(f(v_j))$.

2. *Subgraph Simulation [20]*: Q matches data graph G via graph simulation if there exists a binary relation $R \subseteq V^q \times V$ such that 1) $\forall u \in V^q, \exists u' \in V : (u, u') \in R$; 2) if $(u, u') \in R$ then $l^q(u) = l(u')$; 3) $\forall (u, v) \in E^q, \exists (u', v') \in E : (u, u') \in R$.

3. *Subgraph Dual Simulation [20]*: Q matches data graph G via dual simulation if 1) Q matches G via graph simulation under a match relation $R_D \subset V^q \times V$ and 2) $\forall (u, u') \in R_D[(w, u) \in E^q \Rightarrow \exists w' \in V : (w, w') \in R_D \bigwedge (w', u') \in E]$

Dual simulation possess the useful property of unique maximum match. For a given query graph Q, there exists a unique maximum dual simulation match G_{Dsim} with G such that every other dual simulation match $M_{Dsim} \subseteq G_{Dsim}$ [20]. Fig. 1 provides illustrative examples of these matching criteria. Intuitively graph simulation preserves the child relationships of the query graph while dual simulation preserves both parent and child relationships.

A dynamic graph $G_T = \{..., G_{t-2}, G_{t-1}, G_t, ...\}$, $T \subset \mathbb{Z}^+$ is a graph that changes over discrete time steps. At any time t, directed graph G_t denotes a snapshot of G_T. At time $t+1$, graph G_{t+1} is obtained from G_t based on a set of edge updates Δe_u. In this paper, we assume updates consist only of edge additions or removals, not label updates. An edge stream is a sequence of edge updates. Fig. 2 illustrates five graph snapshots of a dynamic graph, each snapshot G_{t+1} is obtained by applying an edge update to the previous snapshot G_t.

For dynamic graphs, let M_t be the set of subgraphs in G_t that match a query graph Q via SIM. An incremental subgraph matching algorithm takes G_t, Δe_u and M_t as the input to produce M_{t+1} for G_{t+1} by computing the changes ΔM to match set M_t. In Fig. 2, vertices $\{1, 2, 3, 5\}$ in G_4 match the given query graph via SIM. This match happens as the graph changes from G_3 to G_4 with the addition of an edge. Also, the matched pattern dissolves as the graph changes from G_4 to G_5 with an edge removal. A distributed incremental subgraph matching algorithm partitions the input among worker nodes and reuses already computed results to minimize unnecessary re-computations, thus enabling low latency high throughput computation. The algorithms presented in this paper assume a distributed memory computing environment with a set of worker nodes W. Each worker $w \in W$ maintains its own *vertex disjoint* partition G_t^w of dynamic graph G_t at time t.

Table I provides a summary of symbols and definitions used in the rest of the paper.

Symbol	Definition
d_G	Diameter (longest undirected shortest path) of the graph G.
$s_{(v_i,v_j)}$	Shortest path between two vertices v_i and v_j
$V(\Delta M)$	Set of vertices in ΔM
$l(v)$	Label associated with vertex v
G_t^w	Partition of G_t in worker $w \in W$
$G_{t,DSim}$	Maximum dual simulation match of G_t
$G_{t,DSim}^w$	Partition of $G_{t,DSim}$ in worker $w \in W$
$MS[v]$	Match set of vertex v
$P[v]$	Parent set of vertex v
$C[v]$	Child set of vertex v
$MS_t^W[v]$	Match set of each vertex v in G_t^w
$P_t^W[v]$	Match set of parents of vertex v in G_t^w
$P_t^W[v][u]$	Match set of parent u of vertex v in G_t^w
$C_t^W[v]$	Match set of children of vertex v in G_t^w
$C_t^W[v][u]$	Match set of child u of vertex v in G_t^w
Δe_u^w	Set of edge updates assigned to worker $w \in W$
e_u^w+	Set of edge additions assigned to worker $w \in W$
e_u^w-	Set of edge removals assigned to worker $w \in W$
$match[v]$	Match status (true/false) of vertex v
$L[v]$	Labels of parents and children of vertex v

TABLE I: Symbols and their definitions

III. Related Work

Most approaches for subgraph matching in graph data in the last few decades focused on small graphs [12], [13], [14]. This was mainly due to the NP-Hard nature of the SIM problem [21]. A detailed comparison of existing algorithms for SIM matching in static graphs can be found in [22].

In [23] Fan et al. introduced a class of relaxed polynomial time subgraph matching methods called graph simulation matching to enable subgraph pattern matching in larger graphs. This work was motivated mainly by the applications in the social network domain where relaxed matching methods are acceptable. Improving on that work, in [20] Ma et al. introduced much strict matching criteria to capture the topology of the query graphs. These works focused more on providing

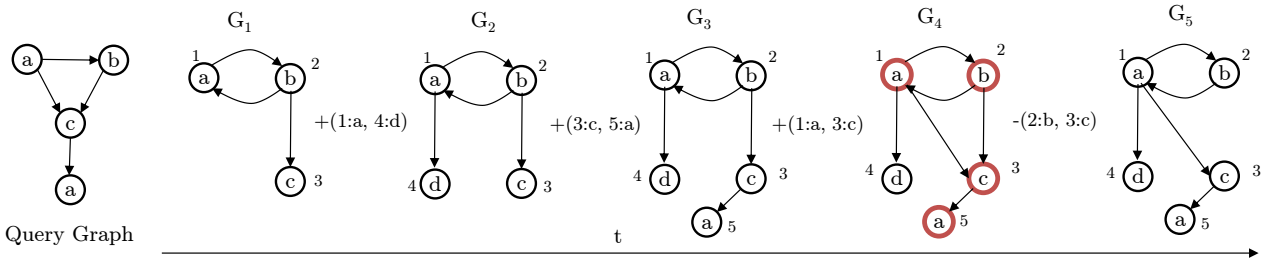


Fig. 2: Five temporal snapshots G_t of a dynamic graph where $t \in \{1, 2, 3, 4, 5\}$. Each snapshot G_t is obtained from previous one G_{t-1} by applying an edge update. Vertices $\{1, 2, 3, 5\}$ in G_4 matches the given query graph via subgraph isomorphism.

theoretical bounds to the proposed algorithms where all the algorithms presented are sequential. In this paper, we show that the dual simulation matching method introduced in this work [20] can be extended and adopted as a graph pruning technique for exact SIM in dynamic graphs.

Recent growth in graph-structured data generated from social networks, communications networks, road networks, etc. and advances distributed computing have paved the way for research on large-scale graph processing on distributed memory environments with commodity hardware [2], [3]. However, most of these work focus on large static graphs. Recently, an algorithm for pattern matching in large scale static graphs on distributed shared memory environment has been proposed [15]. This work assumes a distributed shared memory abstraction and uses Trinity framework [24]. Similarly, distributed algorithms for subgraph matching has been proposed [6], [25]. But these algorithms focus on detecting subgraph simulation matches [26] on static graphs.

In [16], [27] Gao et al. proposed a system and set of optimizations for SIM matching on very large distributed graphs on a vertex-centric [2] graph processing system. But approximate SIM was their focus.

In 2009 Stotz et al. presented an incremental SIM algorithm for matching subgraph patterns in dynamic graphs [28], [23]. This is a sequential algorithm and the evaluations were conducted only for very small graphs. Also in [10] Choudhury et al. proposed a query decomposition based method to perform incremental subgraph isomorphism matching in dynamic graphs. But this work also presents a sequential algorithm for dynamic graphs.

Fan et al in [29] presented a query preserving graph pruning algorithm using bi-simulation. This work presents a theoretical study of the application of bi-simulation based graph pruning for reachability queries and bounded simulation matching. Our work focuses on exact SIM by using dual simulation as a graph pruning technique.

IV. Incremental Subgraph Isomorphism Matching In Dynamic Graphs

The computation time of SIM in a dynamic graph is known to be unbounded relative to the size of edge updates Δe_u or the incremental match set ΔM [23]. But as we observe below, the set of subgraphs that can potentially be in ΔM as the result of an edge update is within a neighborhood of edge update, bounded by the query diameter.

Observation 4.1: Let $e_u = (v_i, v_j)$ be an edge update to the data graph G_t which results in G_{t+1}. Then $\forall\ v_k \in V(\Delta M)$, $s_{(v_k,v_i)} \leq d_Q$ and $s_{(v_k,v_j)} \leq d_Q$.

Observation 4.1 can be exploited to reduce the computation space for incremental subgraph matching over a dynamic graph so as to decrease computation latency. Using Observation 4.1, we develop a distributed incremental algorithm (Algorithm 1) for SIM in dynamic graphs (D-ISI) via a simple framework that re-uses legacy SIM libraries developed for small static graphs. The proposed algorithm is designed specifically to deliver low-latency analytics on dynamic graphs with a larger diameter as compared to the diameter of the query graph ($d_{G_t} >> d_Q$). We develop a graph pruning scheme described in the next section for small diameter dynamic graphs where this is not the case.

Road networks and social networks are example graphs that demonstrate these behaviors. Road networks are examples of large diameter graphs, where in the social networks the effective diameter of the graph is very low (< 10) [30]. As a result, in social networks, subgraph matches in a significant portion of the graph can potentially be affected by an edge update compared to road networks for a given query graph Q.

Algorithm 1 takes an edge stream as input and processes edge updates in batches. Each batch of edge updates collected at time t starts a three stage process. First, edge updates are assigned to different partitions of the dynamic graph using an edge partitioning algorithm running on a single processing node (master), where each partition (G_t^w) is processed by a single worker (w). We use a simple hash based partitioning strategy which assigns the edge updates to different partitions by hashing the source vertex of each edge update. While there has been some work on graph partitioning strategies [31], evaluating those strategies on this algorithm is beyond the scope of this paper. Second, edge updates are applied to the respective graph partitions which result in G_{t+1}. Third, each worker starts processing its graph partition to find the portion of ΔM caused by the edge updates assigned to the worker.

Algorithm 2 presents the core of the distributed algorithm executed at each worker for computing ΔM. Each worker loops through the edge updates from Δe_u assigned to it and finds the subgraphs affected by these updates in its partition using a distributed breadth first search limited to depth d_Q.

Utilizing Observation 4.1, we note that ΔM for this worker can be found by restricting the search for matches to only these subgraphs. Existing subgraph isomorphism libraries can be used to find these matches. Algorithm exploits parallelism at multiple levels. Edge updates are first processed in parallel by distributing it among the workers followed by distributed subgraph construction for each edge update. Conflicting edge updates (additions/removals of the same edge) in a batch are removed before the partitioning step and duplicate matches due to overlaps in the constructed subgraphs are removed in line 7 of Algorithm 2.

Note that in the $UpdateGraph$ subroutine of our Algorithm 1 (line 8), G^w_{t+1} refers to the new data graph created after worker w applies *all* edge updates from edge stream Δe^w_u. Although temporary matches may exist at internal points of the edge stream, the algorithm only finds and outputs final matches that exist in G^w_{t+1}. If desired, the algorithm can output matches at a more fine-grained scale by reducing the size of Δe_u.

Each worker starts a distributed depth limited breadth first search (DDLBFS) (Algorithm 3) independently to construct the subgraph for each of its edge updates. Execution follows the bulk synchronous parallel (BSP) execution model [32]. Each worker maintains a map that tracks the vertices visited by each independent search. This map is used at the end of DDLBFS to send the portions of subgraphs to respective workers (line 20-21 in Algorithm 3).

Algorithm 1 Distributed Incremental Subgraph Isomorphism Matching In A Dynamic Graph (D-ISI)

```
1: procedure D-ISI(Δe_u)
2:     if master then PARTITION(Δe_u)
3:         for each w ∈ WORKERS do
4:             SENDUPDATES(Δe_u^w)
5:     else
6:         Δe_u^w ← RECEIVEUPDATES(.)
7:                 ▷ Apply edge updates to G_t^w
8:         G_{t+1}^w ← UPDATEGRAPH(G_t^w, Δe_u^w)
9:         ΔM ← PROCESS(G_{t+1}^w, Δe_u^w)
10:        Barrier(WORKERS)
11:        M_{t+1} ← M_t ⊕ ΔM
```

Algorithm 2 Computing ΔM for Δe^w_u

```
1: procedure PROCESS(G_{t+1}^w, Δe_u^w)
2:     F ← matches found for each subgraph
3:     for each e ∈ Δe_u^w do s ← e.source t ← e.target
4:         SG_e ← DDLBFS(s, t, G_{t+1}^w, Q, d_Q)
5:         F[SG_e] ← MATCHISO(SG_e, Q)
6:         Barrier(WORKERS)
7:     return FINDDIFF(F,M_t)
```

V. Distributed Graph Pruning

As described previously, Algorithm 1 is likely to work well only for large diameter data graphs with comparatively small query graphs. Our experimental results show that the performance of the algorithm degrades rapidly with increasing query graph diameters, especially in small diameter graphs. This is due to the fact that the size of the subgraph constructed

Algorithm 3 Distributed Depth Limited Breadth First Search

```
1: procedure DDLBFS(source, target, G_{t+1}^w, Q, d_Q)
2:     M ← vertices visited by each processor
3:     for ss ← 1 to d_Q do
4:         if ss = 1 then
5:             M[w].add(source)
6:             M[w].add(target)
7:             N ← NEIGHBORS(source, target)
8:             root ← w
9:             SENDMESSAGES(N, root)
10:        else
11:            MSGS ← Messages sent to this worker
12:            for each m ∈ MSGS do
13:                v ← m.vertex
14:                root ← m.root
15:                if NOTVISTED(v, root) then
16:                    M[root].add(v)
17:                    N ← NEIGHBORS(v)
18:                    SENDMESSAGES(N, root)
19:        Barrier(WORKERS)
20:    for each root worker w ∈ M do
21:        SENDSUBGRAPH(M[w], G_{t+1}^w)
22:    SG ← RECEIVESUBGRAPH(.)
23:    return SG
```

in Algorithm 2 grows rapidly with increasing d_Q on small diameter graphs. In this section, we present a distributed graph pruning algorithm that can significantly improve the performance of exact SIM on small diameter graphs.

Our proposed graph pruning algorithm maintains a pruned graph of the underlying dynamic graph G_T based on dual simulation, where we maintain the maximum dual simulation match of G_t for Q as the pruned graph at each time point t. The following proposition 5.1 can be easily verified from the definitions of dual simulation and SIM [20].

Proposition 5.1: In a data graph, the maximum dual simulation match for a query graph Q contains all subgraphs that match Q via SIM.

We refer readers to Fig. 1 which illustrates this proposition.

We develop a distributed algorithm to maintain the maximum dual simulation match in a dynamic graph. Our algorithm follows the BSP model and can take $O(|E_{SCC}|)$ super-steps to complete in the worst case, where $|E_{SCC}|$ denotes the number of edges in the largest strongly connected component of the data graph. However, we observed that the number of super-steps required in practice to be significantly less.

Algorithm 4 shows the integration of D-IDS (Distributed incremental dual simulation) with the D-ISI (Distributed incremental subgraph isomorphism). Symbols used in the algorithms presented in this section with their definitions are summarized in Table I. As shown in line 9 of Algorithm 4 we perform distributed graph pruning on the updated dynamic graph before processing the edge updates to find ΔM. As a result, the rest of the processing occurs on the pruned graph which can be very small, compared to the original graph. Given the NP-Hard nature of the problem, this reduction in size can significantly improve the performance of SIM in dynamic graphs.

While the proposed pruning technique does not bring down the computational complexity of SIM, it helps to decrease computation time by reducing the size of the data graph to be searched for matches along with the communication overhead of subgraph construction (Line 4 in Algorithm 2).

Algorithm 4 D-ISI with Distributed Graph Pruning

```
1:  procedure D-ISI(Δe_u)
2:      if master then
3:          PARTITION(Δe_u)
4:          for each w ∈ WORKERS do
5:              SENDUPDATES(e_u^w)
6:      else
7:          e_u^w ← RECEIVEUPDATES(.)
8:          G_{t+1}^w ← UPDATEGRAPH(G_t^w, e_u^w)
9:          G_{t+1,Dsim}^w ← MAXIMUMDUALSIM(G_{t+1}^w, e_u^w)
10:         ΔM ← PROCESS(G_{t+1,Dsim}^w, e_u^w)
11:         Barrier(WORKERS)
12:         M_{t+1} ← M_t ⊕ ΔM
```

Associated with each vertex v, D-IDS maintains a set of states: 1) *match* state (0/1) to indicate whether the vertex is in the maximum dual simulation match (1 if the vertex is in the match). 2) Set of query vertices that v matches (match set of v) via dual simulation ($MS[v]$). 3) Match sets of each parent and child of v ($P[v], C[v]$).

Given an initial graph G_0, D-IDS is initialized using a distributed dual simulation matching algorithm for static graphs for which Algorithm 5 shows the high-level steps associated with each vertex.

Algorithm 5 executes in iterations which we call super-steps (ss) following the BSP execution model. In the first super-step, match set of each vertex v, $MS[v]$ is initialized by adding all query vertices that can match v based on its label (line 6 of Algorithm 5). $match$ status in all the vertices whose match set contain some matches is set to 1. Labels of matching vertices are sent to their parents and children to be received in the next super-step. In the next super-step, each currently matched vertex v, create parent and child match sets ($P[v], C[v]$) from the labels received. They are then evaluated with their match set $MS[v]$ to see if they satisfy dual simulation rules (see Section II). Query vertices that violate these rules are removed from the match set. $match$ status is set to 0 for each vertex with an empty match set. Removals from match sets from each vertex are notified to its parents and children as messages to be received in next super-step. In subsequent super-steps, vertices that receive removal notifications update their parent and child matches based on these removals. Then their match set is evaluated with updated parent and child match to see if they satisfy dual simulation rules. Query vertices that violate these rules are again removed from the match set and removals are notified to parents and children of these vertices as before. The algorithm stops when there are no more removal notifications.

Algorithm 6 provides an overview of our distributed graph pruning algorithm for dynamic graphs. Given a batch of edge updates Δe_u, initially we perform a pruning process on them to prune out *safe* edges updates. Safe edges updates are the edges updates that do not have an impact on the maximum dual simulation match (i.e. Safe to add/remove), and it is possible to verify that using local information associated with source and sink vertex of the edge update.

We use the propositions 5.2 and 5.3 to determine safe edge updates.

Proposition 5.2: Edge removal $e = (v, u)$ is safe, 1) $match[v] = 0$ OR $match[u] = 0$ (Either source or sink vertex not in the match set.) 2) $\nexists(v^q, u^q) \in E^Q : l^q(v^q) = l(v)$ AND $l^q(u^q) = l(u)$ (No edge in Q matches e).

Proposition 5.3: Edge addition $e = (v, u)$ is safe if $\nexists v^q \in V^Q : l^q(v^q) = l(v)$ OR $l^q(v^q) = l(u)$.

Algorithm to handle edge removals closely follows the steps in Algorithm 5. In the first super-step ($ss = 0$), the algorithm finds the removed matches due to the edge removals on incident vertices of removed edges. These removals are notified to the parents and children of respective vertices as match removal messages. In next subsequent super-steps, the vertices who receive the messages apply those removals to their parents and children match sets and evaluate the matching conditions. Removals are notified to parents and children as before. The algorithm terminates when there are no more removal notifications.

Processing edge additions can also be done incrementally with small modifications to Algorithm 5. In the first super-step (ss = 0), the algorithm will be executed only on the incident vertices of edge additions. Which will in turn send their labels to the parents and children. Vertices that receive the labels will execute the super-step 1 of Algorithm 5 where in the start of the super-step match set of each of these vertices v are re-initialized by adding all the query vertices that match the label of the v. The rest of the steps are similar as in Algorithm 5.

The distributed graph pruning algorithm D-IDS closely follow the sequential algorithms presented in [33] for incremental graph simulation matching. The correctness of the algorithms can be proved following the same lemmas presented in [33] summarized below.

Edge Removal: 1) Edge removals only removes vertices from match set that are no longer valid matches 2) Algorithm terminates when all invalid matches are removed.

Edge Addition: 1) Edge addition, only adds vertices to the match set if they are candidates (vertices whose label matches a label of a query vertex) 2) Algorithm terminates when all the invalid matches are removed.

A. An Illustrative Example

Fig. 3 provides an illustrative example of a small query and the data graph. We provide a detailed walk through of D-IDS in following sections using Fig. 3.

1) Initialization: Fig. 4 illustrates the states associated with each vertex at the end of each super-step of Algorithm 5. After the end of the first super-step(ss=0) vertices $\{1, 2, 3, 5\}$ send their labels to parents and children. As a result, in next super-step (ss = 1), each vertex v receiving labels from parents and children can construct $P[v]$ and $C[v]$. Vertices 1 and 5 removes all matching query vertices from $MS[1]$ and $MS[5]$

Algorithm 5 BSP Algorithm to Initialize The Graph For D-IDS

```
1: procedure INITIALIZE(G_0^w)
2:     match ← initialized to 0 for each vertex in G_0^w in ss 0
3:     if ss = 0 then
4:         for each v ∈ V_0 do
5:             if ∀v^q ∈ V^Q : l^q(v^q) = l_0(v) then
6:                 MS_0^w[v] ← MS_0^w[v] ∪ v^q
7:             if MS_0^w[v].size > 0 then
8:                 match[v] ← 1
9:             SENDTOPARENTSANDCHILDREN(v, l_0(v))
10:    else
11:        if ss = 1 then
12:            for each v ∈ V_0 ⋀ match[v] = 1 do
13:                L[v] ← labels from parents and children of v
14:                {P_0^w[v], C_0^w[v]} ← CREATEMATCHSETS(L[v])
15:                    ▶ R[v]: match removals from vertex v
16:                    ▶ EvalDSim: evaluate matching rules
17:                R[v] ← EVALDSIM(MS_0^w[v], P_0^w[v], C_0^w[v])
18:                if MS_0^w[v].size = 0 then
19:                    match[v] ← 0
20:                SENDTOPARENTSANDCHILDREN(R[v], v)
21:        else
22:            R_p ← GETPARENTREMOVALS
23:            R_c ← GETCHILDREMOVALS
24:            for each r ∈ R_c do
25:                v ← r.vertex          ▶ target vertex
26:                REMOVEFROMCHILDMATCH(r,v)
27:                R[v] ← EVALDSIM(MS_0^w[v], P_0^w[v], C_0^w[v])
28:                SENDTOPARENTSANDCHILDREN(R[v], v)
29:            for each r ∈ R_p do
30:                v ← r.vertex
31:                REMOVEFROMPARENTMATCH(r,v)
32:                R[v] ← EVALDSIM(MS_0^w[v], P_0^w[v], C_0^w[v])
33:                if MS_0^w[v].size = 0 then match[v] ← 0
34:                SENDTOPARENTSANDCHILDREN(R[v], v)
```

Algorithm 6 Distributed Incremental Dual Simulation Matching in a Dynamic Graph (D-IDS)

```
1: procedure MAXIMUMDUALSIM(G_{t+1}^w, Δe_u^w)
2:     {Δe_u^w+, Δe_u^w−} ← PRUNESAFEEDGES(Δe_u^w)
3:     G_{t+1,Dsim}^w ← PROCESSREMOVALS(G_{t+1}^w, Δe_u^w−)
4:     G_{t+1,Dsim}^w ← PROCESSADDITIONS(G_{t+1,Dsim}^w, Δe_u^w+)
5:     return G_{t+1,Dsim}^w
```

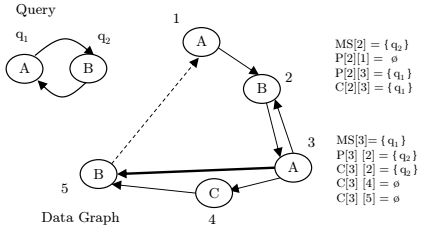


Fig. 3: Initial data graph (G_0) contains edges: $\{(1,2),(2,3),(3,2),(3,5),(3,4),(4,5)\}$. Edge $(5,1)$ added at time $t=1$ and edge $(3,5)$ removed at time $t=2$. Initial states (G_0) associated with vertex 2 and 3 are presented in the right.

SS	1	2	3	4	5
0	MS[1] = {q1}	MS[2] = {q2}	MS[3] = {q1}	MS[4] = ø	MS[5] = {q2}
1	MS[1] = ø	MS[2] = {q2} P[2][1] = {q1} P[2][3] = {q1} C[2][3] = {q1}	MS[3] = {q2} P[3][2] = {q2} C[3][2] = {q2} C[3][5] = {q2}	MS[4]= ø	MS[5] = ø
2	MS[1]= ø	MS[2] = {q2} P[2][3] = {q1} C[2][3] = {q1}	MS[3]= {q2} P[3][2] = {q2} C[3][2] = {q2}	MS[4]= ø	MS[5] = ø

Fig. 4: Vertex states at end of each super-step (ss) in the initialization stage of D-IDS

SS	1	2	3	4	5
0	MS[1] = {q1} C1[2] = {q2}	X	X	X	MS[5] = {q2} P[5][3] = {q1}
1	MS[1] = {q1} P[1][5] = {q2} C[1][2] = {q2}	MS[2] = {q2} P[2][1] = {q1} P[2]3] = {q1} C[2][3] = {q1}	MS[3] = {q2} P[3][2] = {q2} C[3][2] = {q2} C[3][5] = {q2}	MS[4] = ø	MS[5] = {q2} P[5][3] = {q1} C[5][1] = {q1}

Fig. 5: Vertex states at the end of each super-step (SS) after adding edge $(5,1)$. X indicates that those vertices did not involve in any processing in that super-step.

after evaluating dual simulation matching conditions using its match set with the newly created parent and child match sets. These removals are notified to parents and children of vertices (1,5). After receiving the removal notifications in super-step 2 vertex 2 and 3 update their parent/child match sets. But this does not cause any removals in $MS[2]$ or $MS[3]$.

2) Edge Addition: Fig. 5 illustrates the states associated with each vertex at the end of each super-step after adding the edge $(5,1)$. In the first super-step (ss=0), vertices 1 and 5 construct match sets $MS[1]$ and $MS[2]$ using its labels and they are added to the dual simulation match ($match[5]=1$ and $match[1]=1$). Labels of the vertices are sent to parents and children of vertices 1 and 5. Upon receiving labels, vertices 1, 2, 3 and 4 re-set their match set and match set of parents and children based on the received labels. After evaluating dual simulation matching conditions, using its match set with the newly created parent and children match sets, the algorithm will terminate since there are no changes to the match sets.

3) Edge Removal: Fig. 6 illustrates the states of each vertex at the end of each super-step after removing the edge $(3,5)$. In the first super-step, vertices 3 and 5 update their parent/child match sets. Vertex 3 removes the match set of vertex 5 from $C[5]$ and vertex 5 removes the match set of vertex 3 from $P[5]$. As a result, vertex 5 will lose query vertex q_2 from its match set making its $match$ status 0. These removals are notified to vertices 1, 2 and 4 in the next super-step. In the next super-step, the removal of vertex 5 causes vertex 1 to lose query vertex q_1 from its match set. Notification of this removal in

SS	1	2	3	4	5
0	X	X	$MS[3] = \{q_1\}$ $P[3][2] = \{q_2\}$ $C[3][2] = \{q_2\}$	X	$MS[5] = \emptyset$ $C[5][1] = \{q_1\}$
1	$MS[1] = \emptyset$ $C[1][2] = \{q_2\}$	$MS[2] = \{q_2\}$ $P[2][1] = \{q_1\}$ $P[2][3] = \{q_1\}$ $C[2][3] = \{q_1\}$	$MS[3] = \{q_1\}$ $P[3][2] = \{q_2\}$ $C[3][2] = \{q_2\}$	$MS[4] = \emptyset$	$MS[5] = \emptyset$ $C[5][1] = \{q_1\}$
2	$MS[1] = \emptyset$ $C[1][2] = \{q_2\}$	$MS[2] = \{q_2\}$ $P[2][3] = \{q_1\}$ $C[2][3] = \{q_1\}$	$MS[3] = \{q_1\}$ $P[3][2] = \{q_2\}$ $C[3][2] = \{q_2\}$	$MS[4] = \emptyset$	$MS[5] = \emptyset$ $C[5][1] = \{q_1\}$

Fig. 6: Vertex states at end of each super-step after removing edge $(3, 5)$. X indicates that those vertices did not involve in any processing in that super-step.

the next super-step to vertex 2 does not cause any removals.

VI. Experimental Evaluation

In this section, we evaluate the performance of our approach on six diverse real world graph data sets.

A. Implementation

We implemented D-ISI and D-IDS in C++ using MPI (MPICH2 [34]). An open source implementation of VF2 [14] algorithm was used for subgraph matching in D-ISI (line 5 in Algorithm 2) [35]. Lemon graph library [36] was used to store the graph partitions locally within each worker.

As explained in Section V, during the execution of D-IDS, each vertex maintains the labels and the match sets of its parents and children in memory. We did some optimizations to reduce the memory footprint by using memory references (pointers). Instead of each vertex maintaining a copy of the labels for each of its parents and children, at each worker, we maintained a pool of labels in the memory. This pool contained all the unique vertex labels. Also, adjacent vertices with bidirectional edges, that are in the same worker, shared the parent/child match sets instead of keeping local copies.

B. Experimental Setup

All the experiments were conducted in Amazon EC2 on a cluster consisting of 5 c3.2xlarge [37] instances (data center limit). This represents a commodity cluster available for the general computer science community rather than HPC users. Each c3.2xlarge instance consists of eight virtual CPUs and 15 GB RAM.

C. Datasets

We performed experiments on six large-scale datasets (see Table II) that are publicly available from [30], [38], [39], This dataset includes three large diameter graphs of increasing size (road networks) and three small diameter graphs. Road networks (CA, CTR, USA) are sparse planar graphs with relatively uniform degree distribution and large diameter. Small diameter (DBLP, YT, LJ) graphs are comparatively dense graphs with power law degree distribution and a smaller diameter.

Dataset	$\mid V \mid$	$\mid E \mid$	Type
California R/N (CA)	1,965,206	2,766,607	Large diameter
Central USA R/N (CTR)	14,081,816	34,292,496	Large diameter
Full USA R/N (USA)	23,947,347	58,333,344	Large diameter
DBLP network (DBLP)	317,080	1,049,866	Small-diameter
YouTube (YT)	4,945,382	49,445,382	Small-diameter
Live Journal (LJ)	5,284,457	77,402,652	Small-diameter

TABLE II: Datasets

Edge updates are generated from randomly extracted edges of these static graphs. Edge additions always added new edges randomly connecting existing vertices where edge removals removed existing edges. Vertex labels were generated randomly from a fixed size dictionary. Subgraphs with various diameters were extracted/injected from/to these graph datasets and used as query graphs. All the query graphs had cycles and evaluations were done on query graphs with diameter (undirected) 1 ($|V| = 5$), 2 ($|V| = 12$) and 3 ($|V| = 17$).

D. Performance Metrics

We used latency as the main metrics for evaluating the performance. Latency is defined as the time difference between the time that the first edge update of a Δe_u entered the system and the time the algorithm finished finding matches for all the edges in Δe_u.

We used the sequential exact SIM algorithm presented in [33] as a baseline for comparison. This is a widely cited representative sequential baseline for D-ISI which performs a neighborhood based search for exact SIM.

In order to report statistically significant results we ran our experiments several times on Amazon EC2. During our initial evaluations, we found that there is a variation in performance compared to the results in an isolated cluster. But we observed that this variation in performance did not change the overall performance behavior (How algorithm scales with increasing number of resources and graph size). To reduce the effect of variation in performance, final experiments for each plot were conducted together. The latency values reported below are mean values averaged over large number of edge updates.

E. Summary of the Results

We compared the latency of D-ISI with and without the proposed graph pruning algorithm (D-IDS) on the data sets listed in Table II. As shown in the Fig. 7 combining with D-IDS significantly improves the latency of D-ISI on small diameter graphs. But we observed that using D-IDS has no impact on the performance on road network datasets. Further analysis showed that, even though the graph pruning algorithm significantly reduces the size of the graphs (See Fig. 8) in all the datasets, the impact of graph pruning is minimal due to small uniform degree distribution and large diameter of road network datasets. But in small diameter networks, since edge updates affect a large portion of the graph, the impact

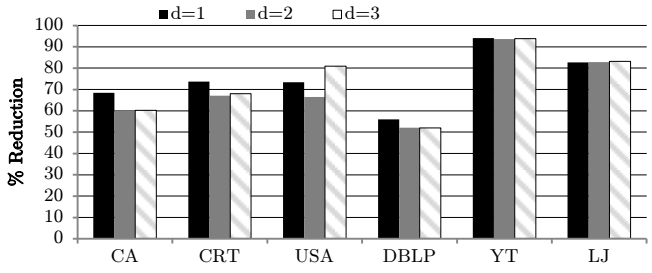
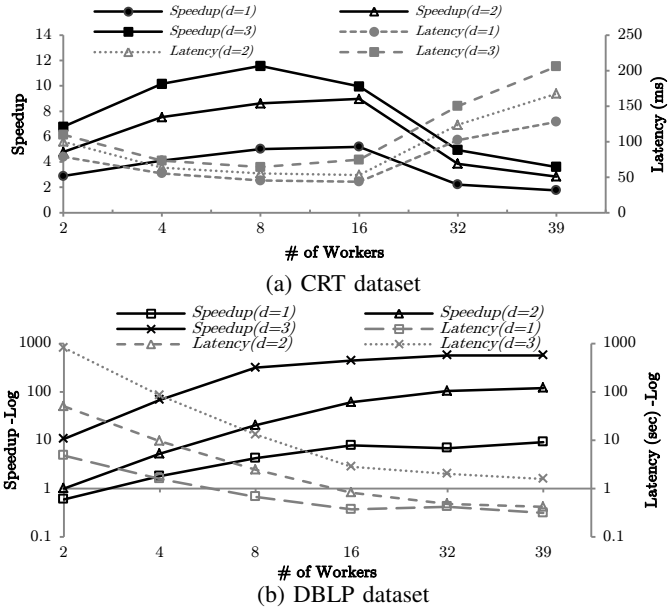


Fig. 8: Average percentage reduction of graph size (number of vertices) D-IDS. *d* denotes the diameter of query graph.



(a) CRT dataset

(b) DBLP dataset

Fig. 9: Speedup and latency of D-ISI + D-IDS for query graphs with various diameters. *d* denotes the diameter of query graph.

of graph pruning was comparatively high. Note that D-ISI failed to execute without graph pruning on the YT dataset due to the large size of the subgraphs created in the subgraph construction step (line 4 of Algorithm 2).

Fig. 8 shows the average percentage reduction (relative to the original graph) of the graph size (number of vertices) when D-IDS is used. As shown in Fig. 8 using D-IDS, we were able to reduce the graph size by over 60% for all the graphs on the average.

Fig. 9 presents the speedup and latency of D-ISI with D-IDS on a road network and a small diameter graph with increasing number of workers. As expected, on the road network dataset D-ISI algorithm with graph pruning was able to produce very low latency results compared to small diameter graphs. But the improvement in speedup tapers off when scaling up the number of workers on road network datasets. We were able to achieve significant improvements in speedup and reduction of latency when scaling up the number of workers on small diameter graph datasets. This is as a result of graph pruning

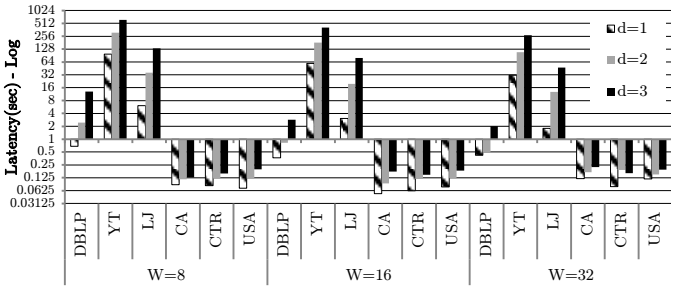


Fig. 10: Comparison of latency of D-ISI + D-IDS for query graphs with various diameters. *d* denotes the diameter of query graph and *W* denotes the number of workers.

where our algorithm (D-ISI + D-IDS) works on a much smaller subgraph compared to the baseline (Intractability of the exact SIM should also be taken into account).

We observed that the communication network in our cluster was saturated when scaling up the number of workers on small diameter datasets, increasing the overall communication latency of the system. We believe that further improvements can be achieved on a cluster with a high-performance communication network. Due scalability limitations in baseline algorithm, we were unable to compute speedups on YT and LJ datasets. A similar behavior of latency was observed in these datasets. Comparison of latencies on all datasets are summarized in Fig. 10. We also observed a higher improvement in speedup with increasing d_Q. The reason for this speedup is twofold: With the Increasing diameter of the query graph, edge updates can affect a large portion of the graph. As a result, a larger subgraph is constructed in the subgraph construction step. But since the baseline algorithm does not perform any graph pruning, reduction in the constructed subgraph due to graph pruning compared to the baseline becomes larger with the increasing diameter. Also increasing diameter in the query graph enables the DDLBFS to explore deeper, increasing the parallelism.

Evaluations with increasing $|\Delta e_u|$ showed that (See Fig. 11) D-ISI with D-IDS achieves higher throughput with increasing $|\Delta e_u|$ due to the higher parallelism achieved with increasing $|\Delta e_u|$. But this resulted in increase in latency.

VII. Conclusions

We presented a query preserving distributed graph pruning technique (D-IDS) to enable exact subgraph matching in small diameter dynamic graphs. Our graph pruning technique reduced the graph size significantly on real world graphs where it achieved over 60% reduction in graph size.

A simple distributed incremental exact subgraph matching algorithm (D-ISI) which can utilize the above mentioned graph pruning technique was presented. Our results show a significant improvement in performance on small diameter graphs when D-ISI was used with D-IDS. But no improvement in performance was observed on large diameter graphs.

In this work, we did not evaluate the impact of utilizing different graph partitioning schemes on the performance. We

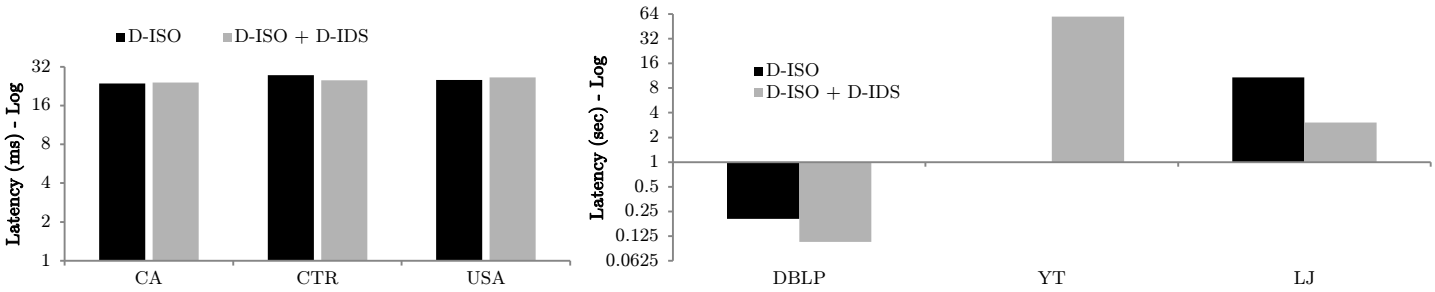
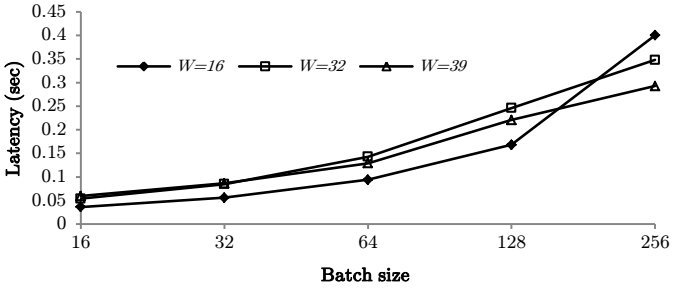
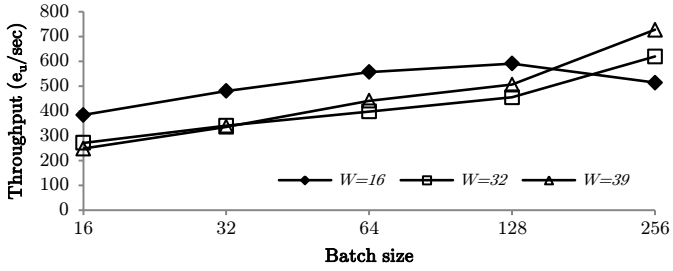


Fig. 7: Comparison of latency for various datasets with (D-ISO + D-IDS) and without (D-ISO) graph pruning.



(a) Latency



(b) Throughput

Fig. 11: Latency and throughput of D-ISI + D-IDS for different batch size. W denotes the number of workers.

plan to evaluate the above mentioned algorithms using different dynamic graph partitioning strategies and understand its impact on the performance.

VIII. Acknowledgment

This work has been funded by US NSF under grant ACI-1339756.

References

[1] D. J. Watts, "Networks, dynamics, and the small-world phenomenon 1," *American Journal of sociology*, vol. 105, no. 2, pp. 493–527, 1999.

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*.

[3] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*.

[4] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*. ACM, 2010, pp. 78–85.

[5] S. Salihoglu and J. Widom, "Optimizing graph algorithms on pregel-like systems," *Proceedings of the VLDB Endowment*, vol. 7, no. 7.

[6] S. Ma, Y. Cao, J. Huai, and T. Wo, "Distributed graph pattern matching," in *Proceedings of the 21st international conference on World Wide Web*.

[7] C. C. Aggarwal, H. Wang *et al.*, *Managing and mining graph data*. Springer, 2010, vol. 40.

[8] B. Gallagher, "Matching structure and semantics: A survey on graph-based pattern matching," 2006.

[9] D. Shasha, J. T. Wang, and R. Giugno, "Algorithmics and applications of tree and graph searching," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2002, pp. 39–52.

[10] S. Choudhury, L. Holder, G. Chin, K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," *arXiv preprint arXiv:1503.00849*, 2015.

[11] J. W. Raymond and P. Willett, "Maximum common subgraph isomorphism algorithms for the matching of chemical structures," *Journal of computer-aided molecular design*, vol. 16, no. 7.

[12] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1.

[13] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "Performance evaluation of the vf graph matching algorithm," in *International Conference on Image Analysis Proceedings, 1999*.

[14] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 10, 2004.

[15] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proceedings of the VLDB Endowment*, vol. 5, no. 9.

[16] J. Gao, C. Zhou, J. Zhou, and J. X. Yu, "Continuous pattern detection over billion-edge graph using distributed framework," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*.

[17] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: taking the pulse of a fast-changing and connected world," in *ACM european conference on Computer Systems (EuroSys)*, 2012.

[18] D. Ediger, R. McColl, J. Riedy, D. Bader *et al.*, "Stinger: High performance data structure for streaming graphs," in *IEEE Conference on High Performance Extreme Computing (HPEC), 2012*.

[19] A. Zeeshan, A. M. Masood, Z. M. Faisal, A. Kalim, and N. Farzana, "Prism: Automatic detection and prevention from cyber attacks," in *Wireless Networks, Information Processing and Systems*.

[20] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," *Proceedings of the VLDB Endowment*, vol. 5, no. 4.

[21] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979.

[22] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," in *Proceedings of the VLDB Endowment*, vol. 6, no. 2. VLDB Endowment, 2012, pp. 133–144.

[23] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: From intractable to polynomial time," *Proc. VLDB Endow.*, vol. 3, no. 1-2, Sep. 2010.

[24] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, New York, USA*, 2013.
[25] A. Fard, M. U. Nisar, L. Ramaswamy, J. A. Miller, and M. Saltz, "A distributed vertex-centric approach for pattern matching in massive graphs," in *IEEE International Conference on Big Data, 2013*.
[26] M. R. Henzinger, T. Henzinger, P. W. Kopke *et al.*, "Computing simulations on finite and infinite graphs," in *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*.
[27] J. Gao, C. Zhou, and J. X. Yu, "Toward continuous pattern detection over evolving large graph with snapshot isolation," *The VLDB Journal*.
[28] A. Stotz, R. Nagi, and M. Sudit, "Incremental graph matching for situation awareness," in *12th International Conference on Information Fusion, 2009.*
[29] W. Fan, J. Li, X. Wang, and Y. Wu, "Query preserving graph compression," 2012.
[30] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.
[31] J. Huang and D. J. Abadi, "Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 9, no. 7.
[32] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.
[33] W. Fan, X. Wang, and Y. Wu, "Incremental graph pattern matching," *ACM Transactions on Database Systems (TODS)*, vol. 38, no. 3, p. 18, 2013.
[34] "Mpich, a high performance and widely portable implementation of the message passing interface (mpi) standard," https://www.mpich.org, accessed: 2016-4-1.
[35] "Vflib: graph matching library," http://www3.cs.stonybrook.edu/ algorith/implement/vflib/implement.shtml, accessed: 2016-04-01.
[36] "Lemon: Library for efficient modeling and optimization in networks." https://lemon.cs.elte.hu/trac/lemon, accessed: 2016-04-01.
[37] "Amazon ec2 instance types." https://aws.amazon.com/ec2/instance-types/, accessed: 2016-04-01.
[38] "9th dimacs implementation challenge," http://www.dis.uniroma1.it/challenge9/download.shtml, Jun. 2010.
[39] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and Analysis of Online Social Networks," in *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.