Efficient parallel edge-centric approach for relaxed graph pattern matching

Sarra Bouhenni, Saïd yahiaoui, Nadia Nouali-Taboudjemat, Hamamache Kheddouci

The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Efficient Parallel Edge-Centric Approach for Relaxed Graph Pattern Matching

Sarra Bouhenni[123*] · Saïd Yahiaoui[2†] · Nadia Nouali-Taboudjemat[2‡] · Hamamache Kheddouci[3§]

Received: DD Month YEAR / Accepted: DD Month YEAR

Abstract Prior algorithms on graph simulation for distributed graphs are not scalable enough as they exhibit heavy message passing. Moreover, they are dependent on the graph partitioning quality that can be a bottleneck due to the natural skew present in real-world data. As a result, their degree of parallelism becomes limited. In this paper, we propose an efficient parallel edge-centric approach for distributed graph pattern matching. We design a novel distributed data structure called *ST* that allows a fine-grain parallelism, and hence guarantees linear scalability. Based on *ST*, we develop a parallel graph simulation algorithm called *PGSim*. Furthermore, we propose *PDSim*, an edge-centric algorithm that efficiently evaluates dual simulation in parallel. *PDSim* combines *ST* and *PGSim* in a Split-and-Combine approach to accelerate the computation stages. We prove the effectiveness and efficiency of these propositions through theoretical guarantees and extensive experiments on massive graphs. The achieved results confirm that our approach outperforms existing algorithms by more than an order of magnitude.

Keywords Graph pattern matching · Subgraph matching · Graph simulation · Dual simulation · Massive graph · Parallel algorithm

1 Introduction

Graph Pattern Matching (GPM) has been used in many application domains such as software plagiarism detection, in-database analytics and search engines [29,34]. It allows finding answers to an input query graph in a relatively larger data graph. GPM can be answered through different models that define the matching constraints based on the query graph. Subgraph isomorphism, which is an NP-Complete problem [17], is the most studied model in this domain. It defines a bijective mapping between the query graph vertices and the data graph vertices, such mapping must preserve the topology of the query graph. Among the most important studies on subgraph isomorphism, we find Ullman's algorithm [44], VF2 [5], QuickSI [41], GraphQL [20], SPath [51], Turbo$_{iso}$ [19], Boost$_{iso}$ [37, 45] and CFL-Match [2]. In the context of large graphs, various works addressed subgraph isomorphism such as [16,35,40,6,38,1,46] that use graph exploration to find the matching results or [43, 50,18,23,24,36,25] that are join-based. However, subgraph isomorphism may be too stringent for

[1]Ecole nationale Supérieure d'Informatique, BP M68, Oued Smar, 16309, Algerie
[*] cs_bouhenni@esi.dz
[2] CERIST, Centre de Recherche sur l'Information Scientifique et Technique, Ben Aknoun, 16030, Algérie
[‡]syahiaoui@cerist.dz · [†]nnouali@cerist.dz
[3]Université de Lyon, CNRS, Université Lyon 1, LIRIS, UMR5205, F-69622, France
[§] hamamache.kheddouci@univ-lyon1.fr

some applications where the subgraphs that should be considered as correct answers may vary from the ideal model defined by the input pattern. This limitation, i.e., the rigid constraints of subgraph isomorphism makes it impractical for the current applications of GPM in social networks. Moreover, subgraph isomorphism may return an exponential number of answers to the same query graph. Actually, when a query graph is an automorphism (isomorphic to itself), the same subgraph is considered several times by subgraph isomorphism. This particularity of subgraph isomorphism, if not handled by the proposed algorithm, not only makes the problem impractical, but it also becomes hard to analyze the returned subgraphs without an additional step.

In contrast, graph simulation [33] is a flexible model that answers GPM through the relaxation of the matching constraints imposed by subgraph isomorphism. It requires preserving the child relationships of input the query graph. Graph simulation returns a single match graph while it can be evaluated in quadratic time [9]. In addition to that, graph simulation is very flexible which makes it suitable for the current applications of social networks such as finding communities [8]. Furthermore, there are other relaxed models that extend graph simulation. For example, dual simulation [30] captures more topological structures of the query graph compared to graph simulation by imposing the matching constraints on both child and parent vertices, while being feasible in quadratic time. There is also strong simulation [30] that reinforces dual simulation by introducing the locality property. In addition, there are other GPM models that relax further the matching constraints of graph simulation, e.g. bounded simulation [9] that maps query edges to reachability paths in the data graph, surjective simulation [42] that extends bounded simulation for multi-labeled graphs, relaxation simulation [15] that allows child and parent constraints to be substituted with grand-children and grand-parents constraints. There is also taxonomy simulation [27] that allows mapping vertices having different labels based on a taxonomy hierarchy tree of the existing labels. Finally, double simulation [47] extends bounded simulation with the duality property. A thorough study of all the existing GPM models and their distributed algorithms can be found in [3].

Nevertheless, even though many works have addressed the problem of GPM for large graphs, the actual size of data graphs is still challenging. In fact, social networks are generating huge amounts of data continuously, e.g., Facebook was the largest network with 2.4 billion monthly active users in June 2019 [7]. These networks cannot fit on the memory of a single machine due to their large size; hence, they need distributed storage and processing. Among the challenges we encounter in such distributed environments, there is linear scalability, a key property for distributed graph algorithms. Yet, prior works addressing relaxed GPM in this context are limited due to the bounded level of parallelism that can be reached [31,14,12,39,22,28,13].

In this paper, we show that graph simulation and dual simulation can be both efficiently evaluated by parallel algorithms that achieve linear scalability. Actually, the different algorithms given by [14,39,22,28] adopt the vertex-centric paradigm of Pregel [32]. In Pregel, the data graph is seen as a distributed system where vertices are analogous to computing units that can execute their local programs in parallel and exchange messages with their neighboring vertices. The algorithm runs in a series of computations separated by a synchronization barrier and message exchanges. An initial message consisting of the query graph is sent to all the data vertices that will store their matching information locally. In each iteration the graph vertices share their matching information or removal messages with their neighbors that update their local matches accordingly. The distributed algorithm converges when no further messages are exchanged anymore. Nevertheless, the vertex-centric paradigm is limited because it generally exhibits a heavy message passing. Additionally, the natural skew present in real-world graphs does not allow for a higher degree of parallelism. Finally, the vertex-centric paradigm is more suitable for graph algorithms that require data locality. Indeed, data locality is important for graph problems where a large neighborhood of a vertex is required to evaluate the GPM model, e.g. subgraph isomorphism and strong simulation.

Furthermore, the algorithms proposed in [12,28,13] adopt Partial Evaluation in answering graph simulation queries. In Partial Evaluation, each worker of the distributed system evaluates graph simulation based only on the data stored locally, then, it communicates with a coordinator machine that propagates the newly computed matching information among the workers. At the reception of the matching information of other workers, a worker will update its local matching information accordingly. If there are any updates, another round of message exchange and computations are carried out until the algorithm convergence. Depending on the partitioning strategy adopted, some workers may remain inactive for most of the processing time. Moreover, these works do not provide mechanisms for parallelizing computations on the same machine, hence, limiting the degree of parallelism that can be achieved.

These shortcomings motivated our work to propose *PGSim* and *PDSim*, two parallel edge-based algorithms to answer GPM queries in parallel for the two models, respectively. Our main contributions are summarized as follows.

(1) We introduce *ST*, a novel distributed data structure for storing the data graph edges which ensures a high degree of parallelism. *ST* is composed of basic computing units called *STwigs*; a set of edges having the same source or the same destination vertex. *ST* ensures high scalability due to the independence between its elements that can be processed in parallel.
(2) We propose *PGSim*, an edge-centric algorithm for evaluating graph simulation in parallel on distributed data graphs. To the best of our knowledge, this is the first work on graph simulation that adopts an edge-centric programming model.
(3) Based on *ST* and *PGSim*, we propose *PDSim*, a fast parallel algorithm for evaluating dual simulation in the same context of massive graphs.
(4) In addition to that, we prove the effectiveness of our approach by giving theoretical guarantees on the correctness of the different algorithms proposed in this paper.
(5) Furthermore, we propose an implementation of the parallel algorithms on top of the in-memory distributed system Apache Spark [49].
(6) Finally, we prove the efficiency of *PGSim* and *PDSim* through extensive experiments and compare them to the state-of-the-art vertex-centric approach on different real-world and synthetic data graphs. The obtained results verified that *PGSim* and *PDSim* can be ten times faster than their vertex-centric counterparts.

The remainder of this paper is organized as follows. Section 2 iterates over related works while Section 3 is dedicated for defining the problem of relaxed graph pattern matching in addition to the preliminaries. Next, we introduce *PGSim*, the parallel edge-centric approach for graph simulation in Section 4. In Section 5, we propose the parallel edge-centric algorithm for dual simulation. Section 6 gives the implementation details in addition to the results of experimental evaluation of *PGSim* and *PDSim*, whereas Section 7 concludes the paper and gives future directions.

2 Related work

A first algorithm of graph simulation was given in [21]. Later in [9], Fan et al. proposed a quadratic-time algorithm for the context of labeled data graphs. Incremental graph simulation was addressed in [11]. Moreover, the problem of top-k queries based on graph simulation was addressed in [10]. However, these approaches were all centralized, hence cannot be directly employed for massive graphs.

Several approaches were proposed during the past decade to answer relaxed GPM on distributed graphs. Ma et al. proposed in [31] an algorithm that ships to the same worker the connected components of the data graph that are stored across different machines. Then, graph simulation is evaluated in a sequential way on each connected component. Finally, the union of the obtained results for each connected component is returned as an answer. Clearly, this approach is not scalable since we assume that a connected component of the data graph can reside on the memory of a single machine, which cannot always be the case. Fan et al. [12] adopted partial evaluation to find matches of graph simulation in a distributed graph. An iterative algorithm evaluates graph simulation on every machine in parallel for the available data vertices while the boundary nodes (vertices having children residing on a different machine) are unknown and hence kept non evaluated. In the next iteration of the distributed algorithm, each worker propagates the matching information of its boundary nodes through a central coordinator so that other machines complete their matching using the newly available data. In [13], another algorithm that uses partial evaluation was given based on GRAPE, a subgraph-mining system that adopts partial evaluation as its programming model. However, this approach is not fully distributed as it involves a coordinator machine that propagates updates made by each work to the other ones. The drawback of these approaches using partial evaluation is that their performance heavily depends on an efficient partitioning strategy of the data graph.

On the other hand, a vertex-centric approach, based on Pregel's programming model, was given in [14]. Each vertex in the data graph has its matching information saved locally as a match set. After that, vertices communicate their matching information with their neighbors and also send removal messages when a vertex does not satisfy graph simulation anymore. The removal messages allow other vertices to update their match sets accordingly and potentially generate new removal messages. The distributed algorithm converges when there are no new removals. Based on the same approach, the authors proposed a vertex-centric algorithm for dual simulation. Moreover, in [22], the authors adapted the same approach to the context of dynamic evolving graphs. Instead of computing the match set of graph simulation for every data vertex on each update event received, the algorithm updates the existing match set upon the addition, update or removal event propagated through the data graph edges. Another vertex-centric algorithm for evaluating dual simulation on distributed RDF graphs was given in [39]. Finally, in [28], the authors combined partial evaluation and the vertex-centric paradigm to evaluate graph simulation for acyclic and cyclic patterns separately. However, this work addressed only graph patterns having distinct labels, which limits the range of applications of such proposition.

PGSim uses the distributed data structure ST instead of sequential graph traversal to evaluate graph simulation. The fine-grain parallelism ensured by ST allows it to handle the problem of skewed degree distribution present in real-world graphs. The vertex-centric programming algorithms generally incur heavy message passing even for vertices residing on the same machine, which slows down the processing of a distributed graph and results into many rounds of computation. In contrast, our approach uses shared memory abstraction to propagate updates along the different rounds of the algorithm. Finally, our approach for evaluating dual simulation *PDSim* splits the computations to increase the degree of parallelism, resulting to an algorithm faster even than the vertex-centric graph simulation.

3 Background

This section introduces basic concepts related to the problem of GPM and particularly graph simulation and dual simulation. Throughout this paper, we consider simple directed and labeled graphs that are formally defined as follows.

Definition 1 (Directed labeled graph) A directed labeled graph $G = (V, E, f)$ is a simple graph where:

(1) V is a finite set of all the vertices in G,
(2) E is the set of edges $E \subseteq V \times V$ in which (v, v') denotes an edge from v to v',
(3) f is a function that maps each vertex $v \in V$ to a label value $f(v)$ in Σ, the set of all labels.

Graph simulation defines a mapping relation between two directed labeled graphs. We have a large graph called *data graph* G where $G = (V, E, f)$ and a relatively small graph called *pattern* (a.k.a. *query*) *graph* Q such that $Q = (V_q, E_q, f_q)$. Without loss of generality, we assume that Q is connected, otherwise, the answer of graph simulation is given as the union of the answers to the different connected components of Q. The formal definition of graph simulation is given below.

Definition 2 (Graph simulation) A data graph $G = (V, E, f)$ matches a pattern graph $Q = (V_q, E_q, f_q)$ via graph simulation, if there exists a binary match relation $R \subseteq V_q \times V$ such that:

(1) $\forall(u, v) \in R$, $f_q(u) = f(v)$, i.e., u and v have the same label,
(2) $\forall u \in V_q, \exists v \in V$ such that:
  (a) $(u, v) \in R$,
  (b) $\forall(u, u') \in E_q, \exists(v, v') \in E$ such that $(u', v') \in R$.

On the other hand, dual simulation imposes matching constraints on both parents and children of a data vertex. Its formal definition is given below.

Definition 3 (Dual simulation) A data graph $G = (V, E, f)$ matches a pattern graph $Q = (V_q, E_q, f_q)$ via dual simulation, if there exists a binary match relation $R \subseteq V_q \times V$ such that:

(1) $\forall(u, v) \in R$, $f_q(u) = f(v)$, i.e. u and v have the same label,
(2) $\forall u \in V_q, \exists v \in V$ such that:
  (a) $(u, v) \in R$,
  (b) $\forall(u, u') \in E_q, \exists(v, v') \in E$ such that $(u', v') \in R$ (child relationship),
  (c) $\forall(u'', u) \in E_q, \exists(v'', v) \in E$ such that $(u'', v'') \in R$ (parent relationship).

The answer to the query Q in G *w.r.t.* graph simulation or dual simulation is called the *match graph*, and it is simply the subgraph $G' = (V', E', f')$ of G that maps the vertices of Q to the vertices of G through the match relation $R \subseteq V_q \times V'$. Next, we define formally a subgraph and a maximum match graph.

Definition 4 (Subgraph) A graph $G' = (V', E', f')$ is a subgraph of $G = (V, E, f)$, iff:

(1) $V' \subseteq V$,
(2) $E' \subseteq E$,
(3) For each vertex v in V', we have $f'(v) = f(v)$.

Definition 5 (Maximum match graph) Let $G = (V, E, f)$ be a data graph, $Q = (V_q, E_q, f_q)$ a pattern graph and $R \subseteq V_q \times V'$ the maximum match relation of Q in G *w.r.t.* graph simulation (or dual simulation). The maximum match graph is $G' = (V', E', f')$, a subgraph of G verifying:

(1) R maps every vertex v in G' to a vertex u in Q, i.e., $\forall v \in V', \exists u \in V_q$ such that $(u, v) \in R$.
(2) $\forall (v, v') \in E', \exists (u, u') \in E_q$ such $(u, v) \in R$ and $(u', v') \in R$.

In what follows, we present the two parallel algorithms *PGSim* and *PDSim* for evaluating GPM queries on distributed data graphs based on graph simulation and dual simulation, respectively.

4 Parallel Edge-centric Graph Simulation

A particularity of graph simulation is that it requires availability of only the matching information from the children of a data vertex to decide whether it is a correct match or not. To illustrate this, a query graph Q_1, a data graph G_1 and their match graph G_s *w.r.t.* graph simulation are given in Figures 1, 2 and 3, respectively. We use query labels as identifiers only for simplicity purpose. For example, the query vertex B is mapped to data vertex 6 in the resulting match graph. Moreover, vertex B has a child vertex C, hence, at least one of the children of vertex 6 should be mapped to C. Notice that B has also a parent relationship with vertex C, however, this relation is not reflected in G_s. Such scenario is possible because we only care about the child relationships when answering queries via graph simulation.

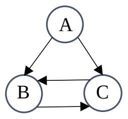


Fig. 1: Pattern Q_1

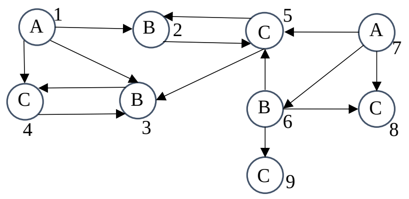


Fig. 2: Data graph G_1

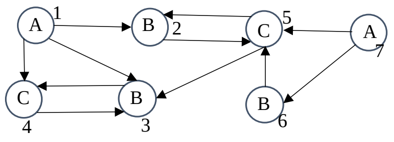


Fig. 3: Match graph G_s

To design an efficient and scalable algorithm, we consider the fact that only children of a data vertex are required in graph simulation, which allows us to avoid a sequential traversal of the data graph. Exploiting this property will decouple the different parts of the data graph and the computations performed on them, hence, increasing the degree of parallelism. Moreover, the data locality preserved by the vertex-centric programming model is more important for graph problems where a large neighborhood of a vertex is required to evaluate the GPM model, e.g. subgraph isomorphism and strong simulation. Therefore, this paradigm is more suitable for such problems, but is less accurate for the case of graph simulation because the message passing communication adopted causes a very low speed of computations. Moreover, the vertex-centric paradigm requires message passing even for vertices residing on the same physical machine even though the exchanged information between any pair of vertices is directly available if we adopt an edge view. These observations motivated our proposition of *PGSim* a parallel, edge-based algorithm that adopts a shared memory abstraction instead of message passing to evaluate graph simulation on distributed graphs.

In what follows, we introduce the different terminologies and data structures used by our approach. Then, we present the steps of the parallel algorithm *PGSim* and give its formal validation.

4.1 Terminologies and data structures

Graph simulation defines a set of matching constraints that must be respected by each data vertex in order to be considered as a correct match for a given query vertex. We define a matching constraint as follows.

Definition 6 (Matching constraint of a vertex) Given a query graph $Q = (V_q, E_q, f_q)$, the matching constraints of a query vertex $u \in V_q$ are given as the set of its children in Q. Let $\mathscr{C} : V_q \to 2^{V_q}$ be the function that maps a query vertex $u \in V_q$ to its constraints $\mathscr{C}(u) \subseteq V_q$. Each edge $(u, u') \in E_q$ defines a matching constraint u' for u. Consequently, $\mathscr{C}$ maps a vertex with zero children to an empty set of constraints.

As an example, the set of constraints of query graph Q_1 are given as $\{\mathscr{C}(A) = \{B, C\}, \mathscr{C}(B) = \{C\}, \mathscr{C}(C) = \{B\}\}$.

Now, let $G = (V, E, f)$ be a data graph, a data vertex $v \in V$ is matched to a query vertex $u \in V_q$, if and only if, $f(v) = f_q(v)$ and v respects the matching constraints $\mathscr{C}(u)$ of u. We say that constraints of u are also constraints of its candidates, such that a candidate is a data vertex having the same label as u. Therefore, a data vertex can have multiple constraints that should be met by its children, such that each constraint must be met by one or more of its outgoing edges. These edges sharing the same source vertex are grouped together to form a data structure that we call *STwig*, which will be defined next. Moreover, we call *ST* the set of all *STwigs* extracted from a data graph.

Definition 7 (STwig) Given a data graph, each vertex in this graph generates an *STwig* structure. Every *STwig* is composed of the set of outgoing edges from this vertex, that we refer to as root. The destination vertex of each edge is considered a child of this *STwig*.

An *STwig* has exactly one root but it can have zero or more children. An empty *STwig* is an *STwig* that has a root but no children, it represents an isolated data vertex or a data vertex with no children.

The set of *STwigs* related to our data graph G_1 is illustrated in Figure 4. *PGSim* processes these *STwigs* in parallel to evaluate graph simulation.

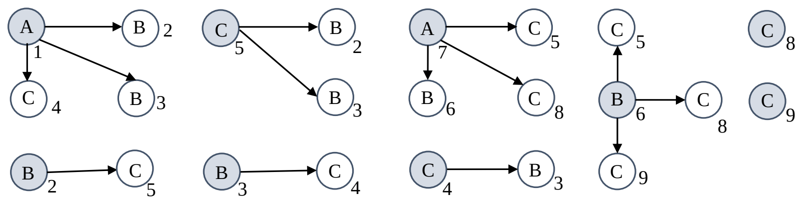


Fig. 4: The set of *STwigs* extracted from the data graph G_1 of Figure 2. The different *STwig* roots are colored in gray

Each *STwig* in the initial *ST*, which is extracted from the data graph, has its own local context. A local context is composed of two types of match sets; a *local match set* and *child match set*.

Definition 8 (Local match set) Given the *STwig* t, we define a local match set $M(t)$ as a set of matches (u, v, b) of the root vertex such that each pair $(u, v) \in V_q \times V$ is mapped to a Boolean value b indicating whether the matching is correct or not.

Definition 9 (Child match set) Given the *STwig* t, the child match set, noted $M_c(t)$, is simply the union of the local matches of the children of t. However, M_c does not include a match flag b, because we only keep the correct matches.

An *STwig* with a non empty match set has its root mapped to at least one query vertex. Consequently, an *STwig* that has an empty match set is not a correct match and therefore can be eliminated from *ST*. Intuitively, an empty *STwig* can only be mapped to query vertices having an empty set of constraints. Hence, its local match set is initialized but never updated afterward.

Definition 10 (ST) The set of all *STwigs* of a given data graph are stored in the distributed data structure *ST*. Each *STwig* in *ST* is mapped to a local context composed of a match set M and a child match set M_c.

ST is constructed off-line from the initial data graph and then processed in parallel. Furthermore, the global match set for graph simulation is defined as follows.

Definition 11 (Global match set) Given a query graph Q, and *ST*, the distributed set of *STwigs* extracted from data graph G. We compute the union of all the local match sets in *ST* *w.r.t.* $\mathscr{C}$, the set of constraints of Q. If there is at least one constraint in $\mathscr{C}$ that is not present in this union, the global match set M_g is $\emptyset$. Otherwise, it is given as this union.

Since we defined the basic concepts and data structures used by *ST*, we are now ready to introduce *PGSim* in the following section.

4.2 Parallel graph simulation via PGSim

Our approach for evaluating graph simulation in parallel consists of three points. First, a preliminary phase builds *ST* off-line by simply grouping the data graph edges that share the same source vertex together. Moreover, vertices without outgoing edges result into empty *STwigs*. Next, at the reception of an input query graph Q, we extract the set of constraints $\mathscr{C}$ in a straightforward way, as they are directly retrieved from the edges of Q. Finally, we compute the global match set *w.r.t.* graph simulation based on *ST* and $\mathscr{C}$.

We give Algorithm `PGSim` that takes as input *ST* and $\mathscr{C}$. It is an iterative algorithm that applies a set of transformations on the initial *ST*, resulting each time to a new refined data structure that is closer to the final match graph. Each iteration of *PGSim* transforms a given *STwig* by updating its local match set M in addition to its child match set M_c in parallel. The parallel algorithm executes two categories of steps (iterations); the initialization step and the computation steps. After its convergence, *PGSim* extracts the global match set M_g from the refined *ST*, also in parallel.

4.2.1 Initialization step

During the initialization (Algorithm `InitializeSTwig`), we evaluate for each *STwig* t its local match set based on the root label and the different children labels only. Each *STwig* runs a local program in parallel where it initializes the children match set M_c (Call of Procedure `InitMatch` in Lines 3–5) and its local match set M (Call of Procedure `InitMatch` in Line 6). Initially, M is composed of matches based only on the label constraint. Then, t verifies for each match in M (a pair (u, v)) whether the child constraints are respected or not by assigning a true or false value (Call of Procedure `GraphSim` in Line 7). The Boolean flag b attached to each pair (u, v) indicates whether this pair passes the child

Algorithm: PGSim

```
1  Input: $ST$, $\mathscr{C}$;
2  Output: $M_g$;
3  $I \leftarrow \emptyset$ ;
4  foreach $STwig\ t\ in\ ST$ do
5      $I_t \leftarrow$ InitializeSTwig($t, \mathscr{C}$);    // Initialize the local match set and child match set
6      $I \leftarrow I \cup I_t$;
7  $ST \leftarrow$ Refine($ST$) ;                  // Filter out STwigs with empty match sets
8  while $I \neq \emptyset$ do
9      $I_{tmp} \leftarrow \emptyset$;
10     foreach $STwig\ t\ in\ ST$ do
11         $I_t \leftarrow$ ComputeSTwig($t, I$) ;         // Evaluate graph simulation locally
12         $I_{tmp} \leftarrow I_{tmp} \cup I_t$;
13     $I \leftarrow I_{tmp}$;
14     $ST \leftarrow$ Refine($ST$) ;              // Filter out STwigs with empty match sets
15 $M_g \leftarrow$ ExtractM$_g$($ST, \mathscr{C}$) ;        // Extract the global match set
16 return $M_g$;
```

Procedure: ExtractM$_g$

```
1  Input: $ST, \mathscr{C}$;
2  Output: $M_g$;
3  $M_g \leftarrow \emptyset$;
4  foreach $STwig\ t\ in\ ST$ do
5      $M_g \leftarrow M_g \cup M(t)$ ;    // $M_g$ is initialized by the union of STwigs match sets
6  foreach $u\ in\ \mathscr{C}$ do
7      if $u \notin M_g$ then
8          return $\emptyset$ ;   // Every single query vertex must appear at least once in $M_g$
```

constraints or not. The matches that do not pass child constraints are collected in I and returned by the parallel algorithm (Lines 6–12). For now, an *STwig* root does not know whether its children satisfy the child constraint themselves or not, so here, I allows us to update the local child match set M_c. After that, the local match set M must be recomputed based on the new M_c. After the initialization, Algorithm *PGSim* reduces the size of ST by invoking Procedure `Refine` (Line 7) that removes from ST all the *STwigs* having an empty local match set.

Procedure `InitMatch` takes the sets of constraints $\mathscr{C}$ and a vertex v and computes the match set of v based only on the label similarity constraint. The initialization program of an *STwig* t invokes this procedure for each local edge to initialize $M(t)$ and $M_c(t)$.

Procedure `GraphSim` takes both $M(t)$ and $M_c(t)$ in addition to the set of constraints $\mathscr{C}$ as input parameters and verifies for each pair (u, v) in $M(t)$ whether there exist children of t that are matched to $\mathscr{C}(u)$. If at least one constraint in $\mathscr{C}(u)$ remains unmatched, (u, v) is mapped to $b = false$, otherwise the match flag b is set to *true*.

In the example of query graph Q_1 and data graph G_1, *PGSim* starts by building the initial ST data structure from G_1 where each vertex in G_1 results into an *STwig*. The initial ST contains the two types of *STwig* described earlier, i.e. seven non empty *STwig* and two empty *STwig*. In the initialization step, each *STwig* computes its local match set M and children match set M_c. Then, it updates M based on M_c which gives the local matches shown in Figure 5. The two empty *STwigs*

Algorithm: InitializeSTwig

```
1  Input: $t, \mathscr{C}$ ;
2  Output: $I_t$ ;
3  $M_c \leftarrow \emptyset$;
4  foreach $v \in t.children$ do
5      $M_c \leftarrow M_c \cup$ InitMatch($\mathscr{C}$, $v$) ;          // Initialize the child match set
6  $M_r \leftarrow$ InitMatch($\mathscr{C}$, t.root);                     // Initialize the STwig match set
7  $M \leftarrow$ GraphSim($\mathscr{C}$, $M_r$, $M_c$) ;         // Evaluate graph simulation for this STwig
8  $I_t \leftarrow \emptyset$;
9  foreach $(u, v, b)$ in $M$ do
10     if $b = false$ then
11         $M \leftarrow M \setminus (u, v, b)$ ;  // Filter out invalid matches ((u,v) that satisfy only
            label similarity)
12         $I_t \leftarrow I_t \cup (u, v)$ ;             // Collect the set of removed matches in $I_t$
13 return $I_t$;
```

Procedure: InitMatch

```
1  Input: $\mathscr{C}, v$;
2  Output: $M$;
3  $M \leftarrow \emptyset$ ;                                     // Initialize M with an empty set
4  foreach $u \in \mathscr{C}$ do
       /* Iterate over the set of vertices in the query graph, available also in
          $\mathscr{C}$                                                             */
5      if $f_q(u) = f(v)$ then
6          $M \leftarrow M \cup (u, v)$ ;        // Add (u, v) to M only if it respects the label
            similarity constraint
7  return $M$;
```

Procedure: GraphSim

```
1  Input: $\mathscr{C}, M, M_c$;
2  Output: $M'$;
3  $M' \leftarrow \emptyset$ ;                                      // Initialize the new match set
4  foreach $(u, v) \in M$ do
5      $b \leftarrow true$ ;                               // Initialize the match flag to true
6      foreach $c \in \mathscr{C}(u)$ do
7          if $\neg(M_c$ contains $c)$ then
8              $b \leftarrow false$ ;     // Change the match flag to false if v does not satisfy
                $\mathscr{C}(u)$ anymore
9      $M' \leftarrow M' \cup (u, v, b)$ ;                    // Add the updated matching info to M'
10 return $M'$ ;
```

rooted at 8 and 9 are initially mapped to query vertex C during the initialization phase. However, C has a child constraint that requires a vertex to have at least one of its children matched to query vertex B, but both *STwigs* are empty (they do not have any children), hence their initial matches become invalid (they are colored in red) and therefore will be broadcast and added to I, the global set of invalid matches for this first iteration of the parallel algorithm.

```
Procedure: Refine
1  Input: $ST$;
2  Output: $ST$;
3  foreach $STwig\ t\ in\ ST$ do
4      if $M(t) = \emptyset$ then
5          $ST \leftarrow ST \setminus t$ ; // Filter out every member of $ST$ having an empty match set
6  return $ST$;
```

4.2.2 Computation steps

A computation step starts if the set of invalid matches I is not empty. I is composed of the pairs (u, v) in every local match set M that did not pass the child constraints (Lines 8–13 in Algorithm `InitializeSTwig`). Consequently, the *STwigs* having a non empty match set run Algorithm `ComputeSTwig` in parallel to update their child match sets according to I (Line 3) and reevaluate their local constraints for graph simulation based on the new value of M_c (Call of Procedure `GraphSim` in Line 4). We also return the set of new invalid matches that should be propagated to other *STwigs* in ST (Lines 5–10). If the set of invalid matches I is not empty, another computation step (Call of Algorithm `ComputeSTwig`) is triggered to propagate these removals and update the local match sets accordingly. Afterward, the new invalid matches I are reevaluated (Lines 9–13 in Algorithm `PGSim`) and the *STwigs* having only invalid matches are filtered out (Call of Procedure Refine in Line 14). If there are new invalid matches, we repeat the computations again until yielding to an empty I. Then, Algorithm `PGSim` converges, and the global match set M_g is computed based on the final match set of each remaining *STwig* in ST (Lines 15–16).

```
Algorithm: ComputeSTwig
1  Input: $t$, $I$ ;
2  Output: $I_t$ ;
3  $M_c \leftarrow M_c \setminus I$ ;                        // Filter out invalid matches from $M_c$
4  $M \leftarrow$ GraphSim($\mathscr{C}$, $M$, $M_c$) ;   // Reevaluate graph simulation based on the new $M_c$
5  $I_t \leftarrow \emptyset$;
6  foreach $(u, v, b)$ in $M$ do
7      if $b = false$ then
           /* GraphSim sets match flag b to false when (u, v) is not a correct
              match anymore                                                */
8          $M \leftarrow M \setminus (u, v, b)$ ;               // Filter out the invalid match
9          $I_t \leftarrow I_t \cup (u, v)$ ;          // Keep track of the invalidated matches in $I_t$
10 return $I_t$;
```

Back to the example of query graph Q_1 and data graph G_1. At the end of the first iteration, the two *STwigs* rooted at 8 and 9 have both empty match sets, consequently, they are eliminated during the refinement phase of the ST before the next iteration. Since I is not yet empty, a second iteration of computation starts where each *STwig* in the refined ST updates its child match set based on the invalid matches of I. For example, both *STwigs* rooted at data vertices 6 and 7 are affected by the previous iteration. Actually, the child constraint for the *STwig* rooted at 6 is $\{C\}$, but since the two children $\{8, 9\}$ were filtered out, it eliminates them from its local M_c. Nevertheless, the third child 5 has a valid match, therefore, the local match set remains the same. The same thing applies for the

STwig rooted at vertex 7 that updates only its local M_c without changing its local match set (see the updated *ST* in Figure 5). At the end of this iteration, I remains empty which announces the end of the parallel algorithm in only two iterations. The global match set is the union of the local match sets for the remaining *STwigs* in *ST*. Moreover, the resulting match graph composed of the remaining *STwigs* in *ST*, is exactly the same as G_s given in Figure 3.

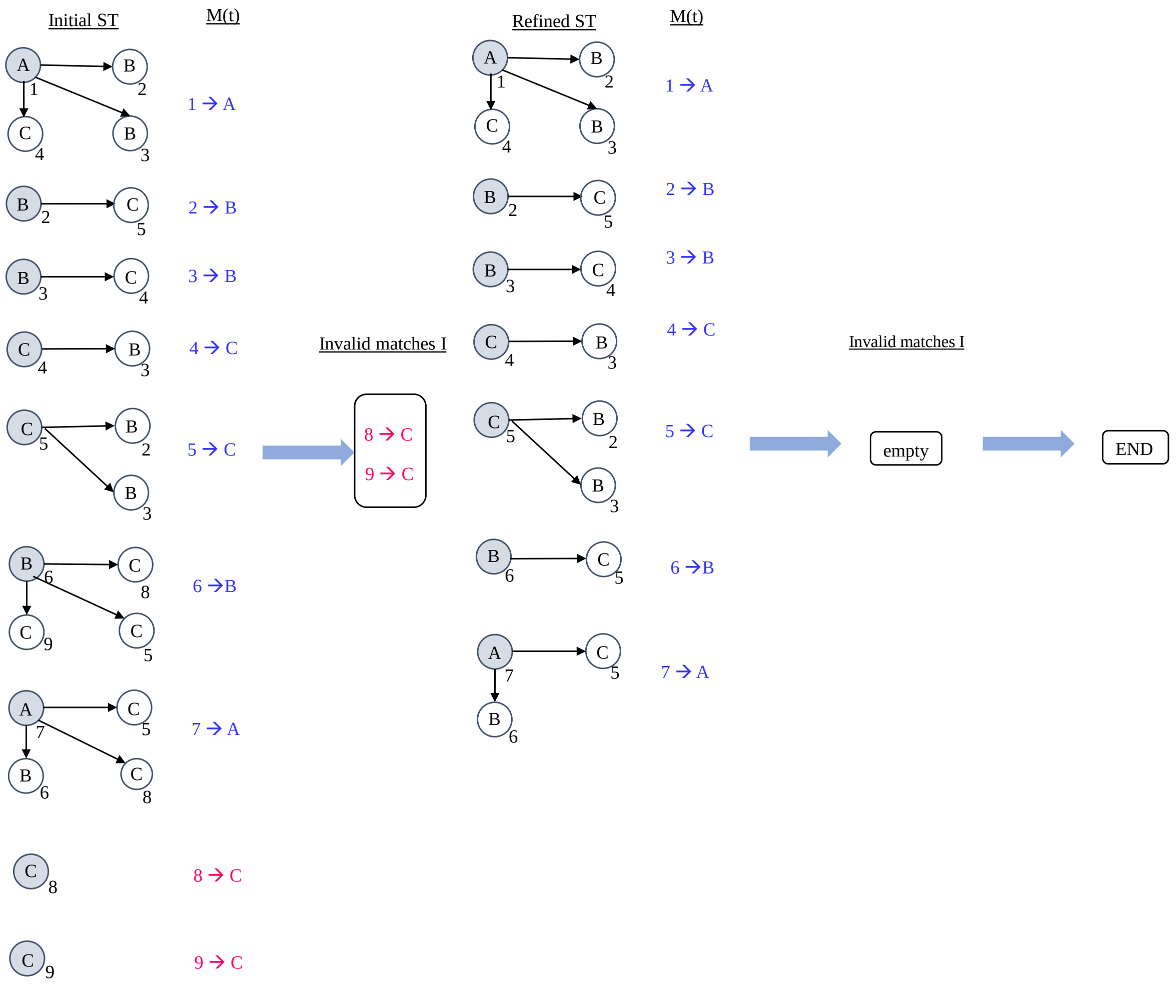


Fig. 5: A running example of *PGSim* on pattern graph Q_1 and data graph G_1. We illustrate the initial *ST* and the different transformations applied by *PGSim* to get the final match graph *w.r.t.* graph simulation. Values of the flag b of each match in $M(t)$ are illustrated by two colors; blue for true and red for false

4.3 Convergence and Correctness of *PGSim*

First, we prove the convergence of the proposed algorithm through the following Lemmas.

Lemma 1 *The maximum number of super-steps to run* PGSim *is* $|E|$.

Proof In the worst case, we have a successive elimination of matches one by one, such that only one match is filtered out in each iteration. A removed match in a given *STwig* is only propagated to its children's *STwigs* and such removal information can propagate over a path from the first removed *STwig* to cross all the edges of G. Therefore, the maximum number of iterations is equal to $|E|$. □

Let Δ_O be the maximum out degree of G and Γ be the maximum size of a local match set. We know that $\Gamma \leq l_q$ where $l_q \leq |V_q|$ is the highest label frequency in Q.

Lemma 2 *The time complexity of Algorithm* `InitializeSTwig` *is equal to* $\mathrm{O}(\Delta_O \times |V_q|)$.

Proof The *Initialize* program of a given *STwig* t initializes the match set of every child in t in $\mathrm{O}(\Delta_O \times |V_q|)$, then it initializes the match set of its root in $\mathrm{O}(|V_q|)$. After that, t evaluates graph simulation in $\mathrm{O}(|V_q| \times \Gamma)$. Finally, it takes at most $\mathrm{O}(\Gamma)$ to update M and populate I_t. Therefore, the time complexity of Algorithm `InitializeSTwig` is $\mathrm{O}(\Delta_O \times |V_q|)$. □

Lemma 3 *The time complexity of Algorithm* `ComputeSTwig` *is equal to* $\mathrm{O}(\Gamma \times |V_q|)$.

Proof The *Update* program of an *STwig* t updates M_c in at most $\mathrm{O}(\Gamma)$, then it reevaluates graph simulation in $\mathrm{O}(\Gamma \times |V_q|)$. Finally, it updates M and I_t in $\mathrm{O}(\Gamma)$. Hence, the time complexity of Algorithm `ComputeSTwig` is $\mathrm{O}(\Gamma \times |V_q|)$. □

Theorem 1 *Algorithm PGSim for evaluating graph simulation in parallel will terminate with time complexity* $\mathrm{O}(|V|/P \times |V_q|^2 \times |E|)$.

Proof The data structure ST has a maximum length equal to $|V|$. After its construction, it contains exactly $|V|$ *STwigs*. After the first iteration it gets smaller in size because the refine procedure can only remove elements from ST. Algorithm `PGSim` starts by iterating over ST, the set of *STwigs* and updating each *STwig* in parallel. Given P, the number of processors running in parallel, it will take $\mathrm{O}(|V|/P \times |V_q| \times \Delta_O)$ to finish the loop. After that, the second loop runs for at most $|E|$ iterations (from Lemma 1) such that in each iteration, it takes $\mathrm{O}(|V|/P)$ to refine ST in parallel and $\mathrm{O}(|V|/P \times |V_q| \times \Gamma)$ to update the *STwigs* in parallel. The global match set is extracted in $\mathrm{O}(|V|/P)$. Therefore, the time complexity of Algorithm `PGSim` in an environment with P processors is $\mathrm{O}(|V|/P \times |E| \times |V_q|^2)$. □

Theorem 2 *Algorithm PGSim computes the correct and complete match set w.r.t. graph simulation.*

Proof The correctness of *PGSim* can be verified by the following. (1) the parallel algorithm terminates (Theorem 1), (2) the parallel algorithm computes the correct matches *w.r.t.* graph simulation and (3) the parallel algorithm returns the complete set of matches *w.r.t.* graph simulation.

To prove the second property, we suppose that at the end of the parallel algorithm, there exists a match $(u, v) \in M_g$ not satisfying the constraints of graph simulation for $u \in V_q$. If $(u, v) \in M_g$, this means that at the end of *PGSim*, the *STwig* rooted at v has u in its local match set $M(t)$. If (u, v) does not satisfy graph simulation, then, either u and v do not have the same label or v does not satisfy the set of constraints $\mathscr{C}(u)$. However, the first case is impossible because $M(t)$ is initialized in Procedure `InitMatch` with only query vertices that share the same label. Moreover, if v does not satisfy $\mathscr{C}(u)$, it should have been already filtered out in the initialization program of t where we evaluate graph simulation through Procedure `GraphSim`, or during the following iterations where the matching constraints are reevaluated at every child's update. Furthermore, the only operation performed on the initial match set is removal, thus, $M(t)$ contains correct matches at the end of *PGSim*.

One can verify that M_g contains the complete matches *w.r.t.* graph simulation as follows. Actually, we are sure that any correct match necessarily belongs to the initial match set of a given t that was generated by Procedure `InitMatch` for every single data vertex v in the data graph. Next, we suppose that *PGSim* filters out incorrectly a correct match (u, v) from local $M(t)$. For any *STwig* $t \in ST$,

$M(t)$ is only updated inside Procedure `GraphSim` and the only removal of (u, v) made happens when $\mathscr{C}(u)$ are not satisfied anymore. Therefore, *PGSim* returns the complete match set M_g for graph simulation. □

5 Parallel Edge-centric Dual Simulation

In this section, we introduce our Split-and-Combine approach to evaluate dual simulation. We also introduce the parallel algorithm used by this approach, dubbed *PDSim*, and give theoretical guarantees on its correctness. Dual simulation requires the availability of both the child and parent information to decide on the matching of any data vertex. In addition, the existing algorithms for dual simulation (whether centralized or vertex-centric) all process the matching information for incoming and outgoing edges of a data vertex sequentially. However, in the case of high degree vertices, this strategy can lead to a load imbalance. Therefore, to address this problem, we run the computations related to the parent constraints and those related to the child constraints in parallel.

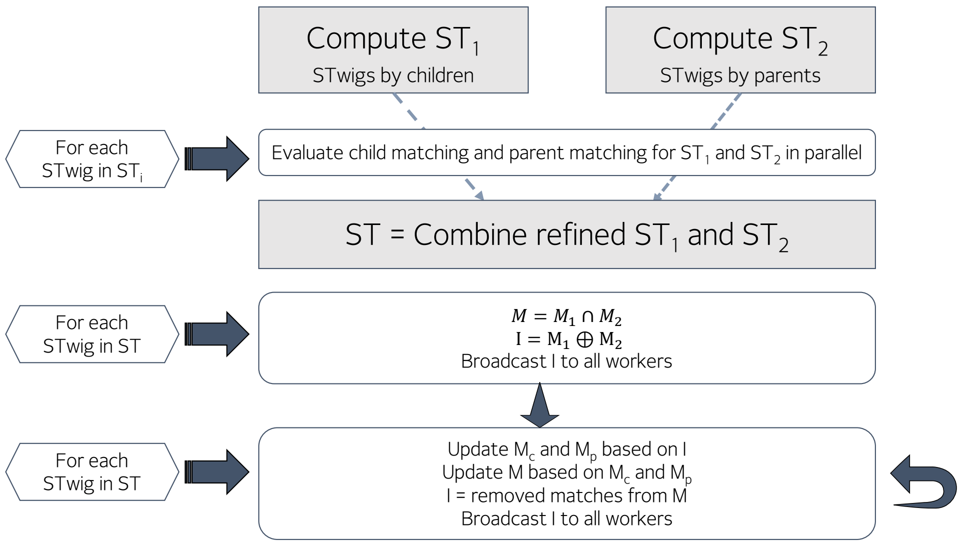

Compute ST_1
STwigs by children
Compute ST_2
STwigs by parents
For each STwig in ST_i
Evaluate child matching and parent matching for ST_1 and ST_2 in parallel
ST = Combine refined ST_1 and ST_2
For each STwig in ST
$M = M_1 \cap M_2$
$I = M_1 \oplus M_2$
Broadcast I to all workers
For each STwig in ST
Update M_c and M_p based on I
Update M based on M_c and M_p
I = removed matches from M
Broadcast I to all workers

Fig. 6: The split-and-combine approach for parallel dual simulation

5.1 A split-and-combine approach for parallel dual simulation

The split-and-Combine approach considers dual simulation as a set of constraints of two types: parent-based and child-based constraints. This separation allows generating two types of *STwigs*, the first type is the same as the *STwig* we have seen in *PGSim*, while the second type is based on the parents of a data vertex v, i.e. each data vertex results into an *STwig* that groups the edges having v as their destination. In the same way, the empty parent based *STwigs* have only a root and represent vertices with zero parents in the data graph. This phase is referred to as the Split phase. Figure 6 illustrates the different steps of computation followed by Algorithm `PDSim` for parallel dual simulation.

Algorithm `PDSim` takes as input the two distributed data structures ST_1 and ST_2, that were constructed before the Split phase (performed only once for each data graph off-line). The first *ST* contains the *STwigs* based on child relationship and the second one is built based on the parent relationship.

During the Split phase, we execute Algorithm `PGSim` on ST_1 using $\mathscr{C}_1$, the child constraints of the input query (Line 3), while a modified version of *PGSim* that checks the parent constraints $\mathscr{C}_2$ is executed on ST_2 (Line 4). We note that, here *PGSim* returns the refined ST without computing the global match set. At the end of this phase, the two refined data structures will have only edges of the data graph respecting the child constraints and parent constraints separately. However, dual simulation requires that the two types of constraints are met by these edges at the same time, hence, we call Algorithm `Combine` that takes as input ST_1 and ST_2 to find the match set M of each *STwig* *w.r.t.* dual simulation (Line 5).

Algorithm `Combine` groups the matching information of children M_c and parents M_p of a given *STwig*. It also computes a next local match set M by eliminating matches that do not satisfy dual simulation (Line 3). The match set M of a given *STwig* t is computed as the intersection of the match sets M_1 and M_2 for t in ST_1 and ST_2, respectively (Lines 4–8). Indeed, a data vertex is said to be a match of a given query vertex u w.r.t. dual simulation if and only if it satisfies the label constraint and both child constraints $\mathscr{C}_1(u)$ and parent constraints $\mathscr{C}_2(u)$. In addition to that, the filtered out matches are stored in I and returned by the parallel algorithm.

Nevertheless, the evaluation of dual simulation does not stop here. Actually, the same removal process that we have seen in *PGSim* must be triggered after each update made to a local match set M in the combined ST to propagate the removed matches across the other *STwigs*. Therefore, the same routine will be executed such that the set of removed matches I is broadcast at the end of every iteration and dual simulation is reevaluated again (based on both M_c and M_p) for each *STwig* affected by this removal until I becomes empty (Lines 6–13 of Algorithm `PDSim`). Consequently, Algorithm `PDSim` converges with the correct final match graph. The global match set M_g is then computed in the same way as the match set of graph simulation (Lines 14–15).

Algorithm: PDSim

```
1  Input: $ST_1$, $ST_2$, $\mathscr{C}_1$, $\mathscr{C}_2$;
2  Output: $M_g$;
3  $ST_1 \leftarrow PGSim(ST_1, \mathscr{C}_1)$ ;                 // Evaluate child-based constraints $\mathscr{C}_1$
4  $ST_2 \leftarrow PGSim(ST_2, \mathscr{C}_2)$ ;                 // Evaluate parent-based constraints $\mathscr{C}_2$
5  $(ST, I) \leftarrow$ Combine($ST_1, ST_2, \mathscr{C}_1, \mathscr{C}_2$ ) ;   // For each STwig, combine its two match sets
6  $ST \leftarrow$ Refine($ST$) ;          // Filter out STwigs in ST having empty match sets
7  while $I \neq \emptyset$ do
8      $I_{tmp} \leftarrow \emptyset$;
9      foreach STwig $t$ in $ST$ do
10         $I_t \leftarrow$ ComputeSTwig($t, I$) ; // Dual simulation is evaluated instead of graph simulation
11         $I_{tmp} \leftarrow I_{tmp} \cup I_t$;
12     $I \leftarrow I_{tmp}$;
13     $ST \leftarrow$ Refine($ST$) ;      // Filter out STwigs in ST having empty match sets
14 $M_g \leftarrow$ ExtractM$_g$($ST, \mathscr{C}_1 \cup \mathscr{C}_2$) ;     // Get the global match set $M_g$ w.r.t. dual simulation
15 return $M_g$;
```

The Split-and-Combine approach increases the degree of parallelism because first, we process the two distributed data structures ST_1 and ST_2 in parallel, which prunes out invalid matches as early as possible. After that, the remaining computations are also performed in parallel on a much smaller ST.

Algorithm: Combine

```
1 Input: $ST_1, ST_2, \mathscr{C}_1, \mathscr{C}_2$ ;
2 Output: $I, ST$ ;
3 $ST \leftarrow$ fullOuterJoin($ST_1, ST_2$) ;   // Combine the two STs based on the STwig root vertex
4 $I \leftarrow \emptyset$ ;   // Initialize the set of invalid matches
5 foreach $t$ in $ST$ do
6     $M(t) \leftarrow M_1(t) \cap M_2(t)$ ;   // Combine the two match sets using intersection
7     $I \leftarrow I \cup (M_1(t) \oplus M_2(t))$ ;   // Invalid matches are the ones appearing in only one match set
8 return $(ST, I)$
```

To illustrate the execution steps of *PDSim*, we use the same data graph G_1 and query graph Q_1. The two graphs G_{s1} of Figure 7 and G_{s2} of Figure 8 represent ST_1 and ST_2, respectively. Let us take the *STwig* rooted at 6, it has a non-empty local match set $M_1 = \{(B, 6)\}$ in ST_1 but that match set M_2 is empty in ST_2. Intuitively, the local match set of this *STwig*, named M, should be empty after calling Algorithm `Combine` because it does not respect the constraints of dual simulation. Indeed, M is given as the intersection between M_1 and M_2. We should inform the other *STwigs* of the combined ST about the removal of the match $(B, 6)$. The set of removed matches I contains elements that do not appear in both M_1 and M_2, i.e. $M_1 \oplus M_2$. Indeed, the local match set of *STwig* rooted at 6 becomes empty. The remaining iterations of Algorithm `PDSim` eliminates the *STwig* rooted at vertex 7. Next, the *STwigs* rooted at data vertices 5 and 2 will be filtered out one after another. The remaining *STwigs* form the final match graph given in Figure 9.

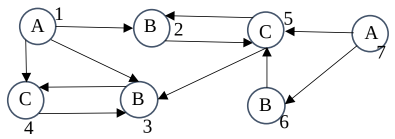


Fig. 7: G_{s1} resulting from ST_1

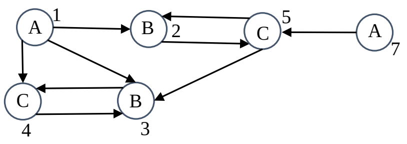


Fig. 8: G_{s2} resulting from ST_2

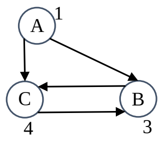


Fig. 9: Match graph G_d

5.2 Convergence and Correctness of *PDSim*

We give Theorem 3 and Theorem 4 that prove the convergence and correctness of Algorithm `PDSim`, respectively.

Theorem 3 *Algorithm `PDSim` will terminate with a time complexity $O(|V|/P \times |V_q|^2 \times |E|)$.*

Proof *PDSim* takes the same time complexity to process ST_1 and ST_2, which results into $O(|V|/P \times |E| \times |V_q|^2)$. After that the combine method takes $O(|V|/P)$ to combine ST_1 and ST_2, the remaining steps for dual simulation take at most $O(|V|/P \times |E| \times |V_q|^2)$ to converge (similar to the evaluation of graph simulation). Moreover, the global match set is extracted in at most $O(V/P)$. Consequently, the time complexity of Algorithm `PDSim` is $O(|V|/P \times |E| \times |V_q|^2)$. □

Theorem 4 *Algorithm `PDSim` returns the correct and complete match set w.r.t. dual simulation.*

Proof The correctness of PDSim is ensured by the following properties. (1) The parallel algorithm will terminate (Theorem 3), (2) the parallel algorithm returns the correct match set of dual simulation, (3) the algorithm returns the complete set of matches of dual simulation.

First, we suppose that at the end of the algorithm, there exists an incorrect match (u, v) in the returned match set M_g. If (u, v) is an incorrect match, then either the two vertices have different labels or v does not satisfy some of the constraints in $\mathscr{C}_1(u)$ or $\mathscr{C}_2(u)$. The first case cannot occur because every local match set is initialized with query vertices having similar labels and M_g is formed by the union of these local matches. Moreover, the local match set M of the *STwig* rooted at v in ST is initialized by the intersection of M_1 (only matches satisfying $\mathscr{C}_1$) and M_2 (only matches satisfying $\mathscr{C}_2$). Moreover, the next iterations of the parallel algorithm always remove the members of M not satisfying dual simulation. Furthermore, the only operations performed on M are removal operations, which ensures that invalid matches cannot be added to M during this stage, hence, the initial supposition is not valid. Therefore, Algorithm `PGSim` returns the correct matches *w.r.t.* dual simulation.

Next, we suppose that there exists a correct match (u, v) such that (u, v) is not part of the returned match set M_g. A match (u, v) is considered correct if and only if $f_q(u) = f(u)$, and v satisfies the child constraints $\mathscr{C}_1(u)$ and parent constraints $\mathscr{C}_2(u)$. Since we proved the correctness of Algorithm `PGSim` in Theorem 2, then a correct match is necessarily part of both $M_1(v)$ and $M_2(v)$. Therefore, if such match exists, then it has been filtered out during the combine phase or removal iterations following it in *PDSim*. However, the only removals made from the local match set $M(v)$ happen after evaluating dual simulation based on the updated child and parent match sets. Hence, Algorithm `PGSim` returns the complete matches *w.r.t.* dual simulation. □

6 Experimental evaluation

In this section, we evaluate the performance of the proposed parallel edge-centric algorithms and compare them to the vertex-centric one. First, we give details on the distributed implementation of *ST*, *PGSim* and *PDSim*. Then, we present the data sets used during the different experiments and give their characteristics. Next, we give the cluster configuration used and the environment in which these experiments were carried out. Finally, we present the different sets of experiments and discuss their results.

6.1 Distributed implementation of *PGSim* and *PDSim*

We implemented the distributed data structure ST on top of Apache Spark [49]. Apache Spark is an in-memory data processing framework that offers a distributed computation model based on Resilient Distributed Datasets (RDDs). An RDD is a distributed data structure that can be processed in parallel by applying a transformation on each element of the RDD. The immutability of Spark RDDs guarantees their resilience. If an RDD is lost while the distributed algorithm is running, it will be directly recomputed based on the set of operations that generated it first, hence allowing Spark to be fault-tolerant while avoiding costly I/O operations except for loading the initial RDD. ST inherits the properties of Spark RDDs, which makes it a distributed data structure for in-memory processing on a computing cluster. *PGSim* is implemented as a series of successive RDD transformations applied to the initial *ST*. Moreover, we implemented the Split-and-Combine approach of *PDSim* on top of Spark, using RDDs.

6.2 Experimental data sets

In addition to the synthetic graphs generated by the R-MAT model [4], we use four real-world datasets from SNAP Library [26]. Characteristics of these graphs are given in Table 1. Unless expressly stated otherwise, the number of distinct labels is fixed to $|\Sigma| = 500$.

Table 1: Characteristics of the data graphs used in the different experiments

Dataset (G)	$\|V\|$	$\|E\|$
Epinions	75,879	508,837
Amazon0601	403,394	3,387,388
WebGoogle	875,713	5,105,039
LiveJournal	4,847,571	68,993,773
Synthetic	up to 3,504,383	up to 83,886,080

6.3 Experimental setup

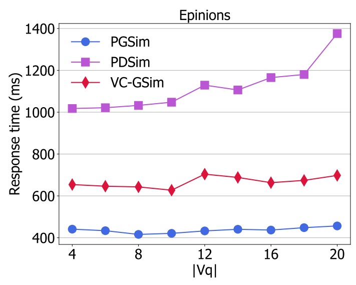
We executed our experiments on a Spark cluster composed of 10 nodes with 32GB memory and 16 cores for each; one node plays the role of the master while the remaining nodes are considered as workers.

To extract patterns of small size from the data graph, we implemented an algorithm that takes two input parameters that are $|V_q|$; the size of the subgraph, and its density α such that $|E_q| = |V_q|^\alpha$. The default value of α is set to $\alpha = 1.2$ in all our experiments. This algorithm executes a BFS traversal starting from a random vertex in the data graph, collects vertices randomly based on a fixed value of average out degree d_O computed such that $\alpha = 1.2$. Then, it extracts the subgraph connecting these vertices and adds up edges to match the input parameters. We used this algorithm to extract 50 different patterns randomly for each value of $|V_q|$.

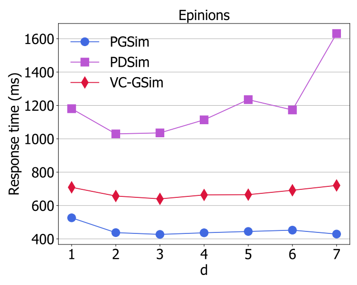
Furthermore, the R-Mat model [4] is used to generate synthetic data graphs. The generator takes as an input $|V|$, $|E|$ and $|\Sigma|$ such that $|\Sigma|$ is the number of distinct labels. We fix $|E| = 20 \times |V|$ for all the synthetic graphs generated in these experiments.

6.4 Experimental results

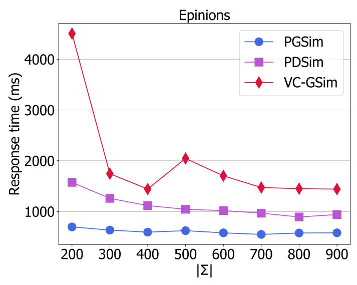
In this section, we present and discuss the results of performance evaluation for *PGSim* and *PDSim* in comparison to the state-of-the-art vertex-centric algorithm (*VC-GSim*) proposed in [14]. We have implemented *VC-GSim* on top of GraphX [48], a graph processing system that offers an implementation of Pregel [32] on top of Apache Spark.



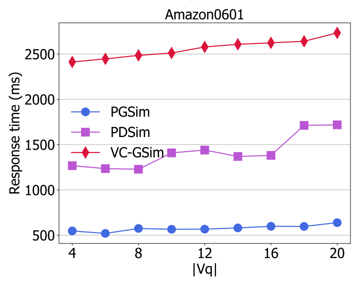
(a) Varying $|V_q|$ (Epinions)



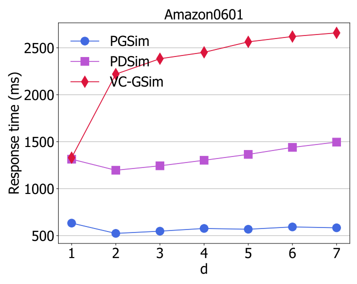
(b) Varying d (Epinions)



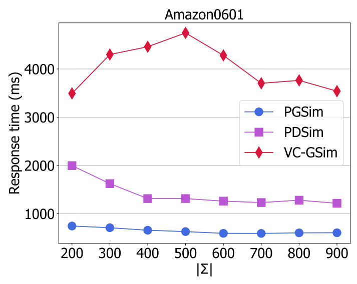
(c) Varying $|\Sigma|$ (Epinions)



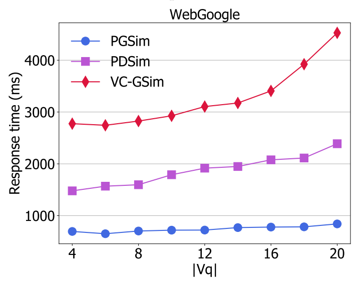
(d) Varying $|V_q|$ (Amazon0601)



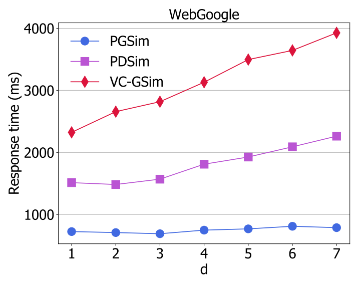
(e) Varying d (Amazon0601)



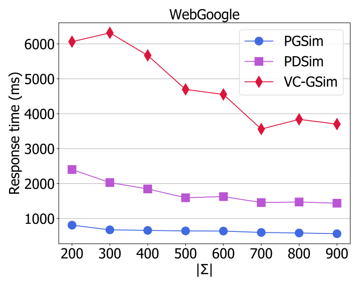
(f) Varying $|\Sigma|$ (Amazon0601)



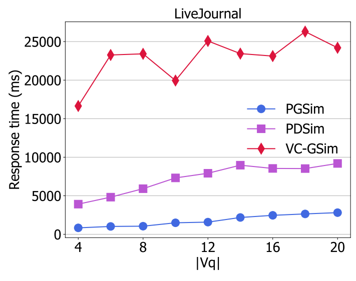
(g) Varying $|V_q|$ (WebGoogle)



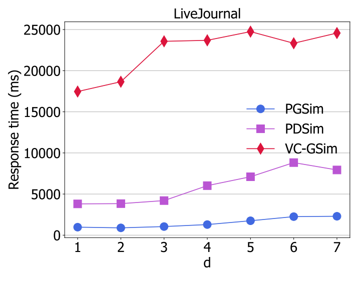
(h) Varying d (WebGoogle)



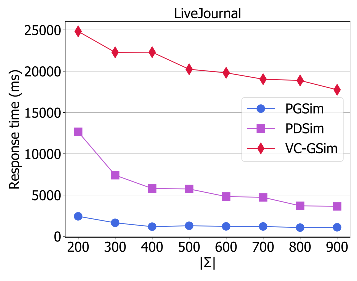
(i) Varying $|\Sigma|$ (WebGoogle)



(j) Varying $|V_q|$ (LiveJournal)

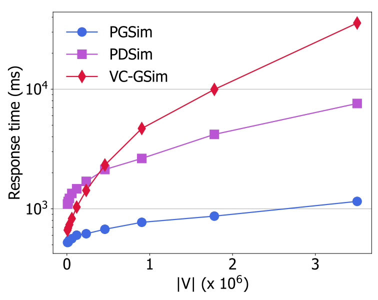


(k) Varying d (LiveJournal)

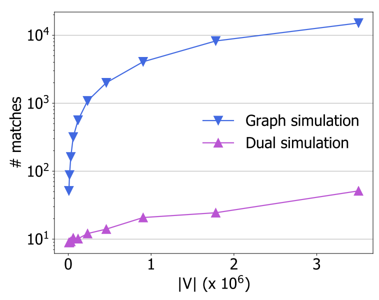


(l) Varying $|\Sigma|$ (LiveJournal)

Fig. 10: Performance evaluation of the parallel algorithms *PGSim* and *PDSim* in comparison to the vertex-centric one *VC-GSim* when varying the graph parameters: query graph size $|V_q|$, query graph diameter d and the number of distinct labels $|\Sigma|$

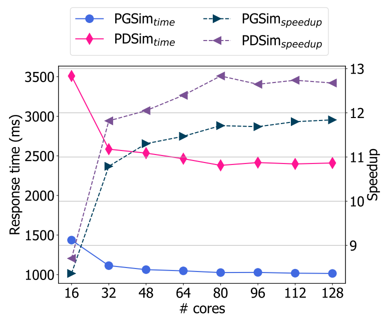


(a) Response time (ms) *w.r.t.* $|V|$

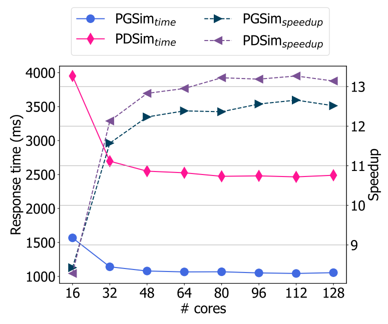


(b) Number of matches *w.r.t.* $|V|$

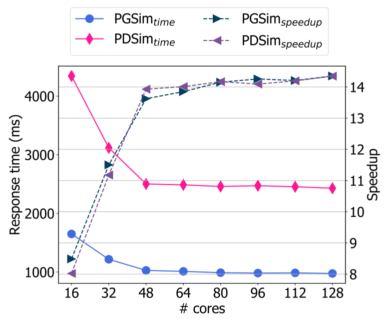
Fig. 11: Weak scaling of the parallel algorithms. Note: the Y axis is on log scale



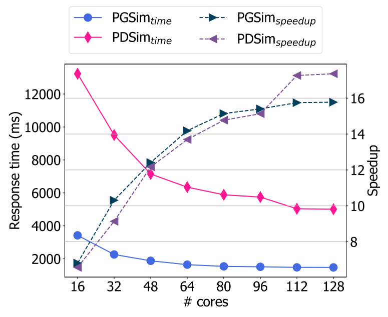
(a) Epinions



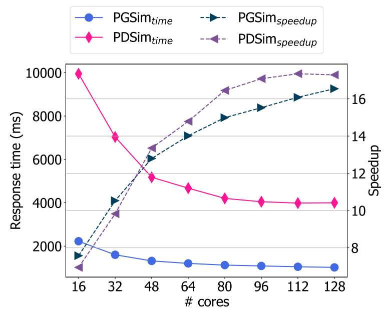
(b) Amazon0601



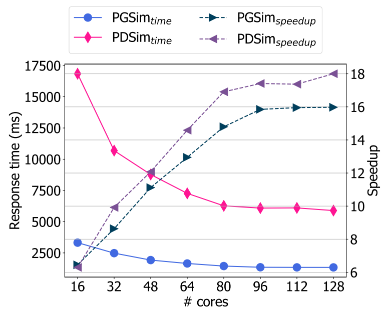
(c) WebGoogle



(d) LiveJournal



(e) Synthetic graph, scale = 21



(f) Synthetic graph, scale = 22

Fig. 12: Strong scaling of the parallel algorithms *PGSim* and *PDSim*

6.4.1 Varying the graph parameters

In this set of experiments, we evaluate the impact of three different parameters, $|V_q|$; the size of query graph, d; the diameter of the query graph and $|\Sigma|$; the number of distinct labels in the data graph, on the performance of *PGSim* and *PDSim*. The results are given in Figure 10.

First, we discuss the average response time when varying $|V_q|$. As we can see, the obtained results prove the superiority of our approach. *PGSim* is faster than *VC-GSim* for the four real datasets *Epinions*, *Amazon0601*, *WebGoogle* and *LiveJournal*. The improvement in response time achieved becomes more significant when the size of the data graph increases. Indeed, *PGSim* is ten times faster than the vertex-centric version for *LiveJournal*. Moreover, *PGSim* is sensitive to the size of the query graph as it increases in a linear curve with respect to $|V_q|$. On the other hand, the parallel algorithm *PDSim* for evaluating dual simulation behaves slower than *VC-GSim* for *Epinions*, which can be explained by the fact that dual simulation requires more computations to prune out invalid matches compared to graph simulation. However, it takes a shorter time to process queries from the other data sets. Actually, *PDSim* is very scalable when varying the size of query graphs or when varying the size of datasets, for a massive graph like *LiveJournal*, *PDSim*'s response time is closer to that of *PGSim*.

Next, we evaluate the behavior of *PGSim* and *PDSim* compared to *VC-GSim* with respect to the diameter of pattern graphs d, using the same data sets. For small to average graphs, we notice that the response time is not affected by d. However, for *LiveJournal*, the response time increases slightly *w.r.t.* this parameter, since the larger d gets, the more iterations are required to filter out all the invalid matches. *PGSim* is the least sensitive to the variations in d.

Finally, the number of distinct labels of the data graph $|\Sigma|$ is an important parameter that affects generally the size of the initial candidates set. We vary $|\Sigma|$ from 200 to 900 for the four real-world data sets. Here, the same query graphs are used with $|V_q| = 9$. Moreover, we use the same generated Σ on both data graphs and query graphs for each experiment. We evaluate graph simulation with *PGSim* and *VC-GSim*, dual simulation with *PDSim* and note the average response time. The response time decreases when increasing $|\Sigma|$, this is a normally expected behavior for *PGSim* and *PDSim* as we use a refinement right after the first iteration of the algorithm to prune out all the data vertices not having a label that exists in the query graph. The number of invalid matches increases when the data graph has a large set of distinct labels and inversely. On the other hand, the change in response time for *VC-GSim* is slow and the algorithm takes longer time to evaluate graph simulation in the different data sets.

6.4.2 Weak scaling experiment

In this set of experiments, we evaluate the weak scalability of *PGSim* and *PDSim*. This type of experiments consists of fixing the number of workers/cores of the cluster and increasing the size of the problem, which is the data graph G in our case. We generate synthetic data graphs of different sizes by varying the scale of the R-Mat model from 13 to 22 (resulting to $|V|$ that varies from 2^{13} to 2^{22}). We report the average response time for running graph simulation and dual simulation over 50 instances of queries having the same size ($|V_q| = 9$). The results of comparing *PGSim* and *PDSim* to *VC-GSim* are given in Figure 11a.

PGSim outperforms *VC-GSim* by one order of magnitude. We can see how the difference in response time between the two algorithms gets larger when the size of the data graph increases. For the first data graph having only $7.7k$ vertices, the two algorithms are very close with 600 *ms* while *PDSim* evaluates dual simulation for the same data graph in one second. Nevertheless, when increasing the size of data graphs, *PGSim* takes only 1.4 seconds to process the largest synthetic graph containing $3.5M$ vertices and $83.9M$ edges, while *VC-GSim* takes 42.6 seconds to get the same result.

Moreover, we can see in Figure 11b, the difference between the number of matches returned by graph simulation and dual simulation. *PDSim* prunes out a big part of the matches returned by graph simulation, which happens during the combine phase along with the remaining iterations that allow the parallel algorithm to converge to the correct match set. Even with these additional computations performed by *PDSim*, our approach for evaluating dual simulation outperforms significantly *VC-GSim* when increasing $|V|$. Indeed, it only takes 7.5 seconds to process the largest synthetic graph. Consequently, this set of experiments proves the scalability of our parallel algorithms.

6.4.3 Strong scaling experiment

We test the strong scalability of *PGSim* and *PDSim* by fixing the problem size (fixing the data graph) and varying the number of cores in the cluster. We report the average response time and speedup when evaluating graph simulation and dual simulation, respectively, on 50 instances of query graphs having the same size ($|V_q| = 9$). We run this set of experiments on the four real-world data sets in addition to synthetic graphs of different size. Figure 12 presents the obtained results.

PGSim scales very well with the number of cores. Indeed, the algorithm speedup increases linearly when increasing the number of cores for all the data graphs. We notice that the larger the data graph size, the higher speedups can be achieved. Indeed, the highest speedups of 16 and 18 are achieved by *PGSim* and *PDSim*, respectively, on the synthetic graph having the largest size ($|V| = 3.5M$ and $|E| = 83.9M$). Nevertheless, for remaining data graphs, the speedup curve starts flattening after some point due to reaching a maximum level of parallelism. For example, for the three data sets

Epinions (Figure 12a), Amazon0601 (Figure 12a) and WebGoogle (Figure 12a), the speedup of the two algorithms increases very fast when adding up 16 to 32 cores. After that, the speedup increases very slowly even when doubling the number of cores used.

The different experiments presented in this section prove the strong scalability of the distributed data structure *ST* and the two parallel algorithms *PGSim* and *PDSim*.

7 Conclusions

In this paper, we proposed *PGSim*, an efficient parallel edge-centric approach for evaluating graph simulation on distributed graphs. *PGSim* relies on the distributed data structure *ST* that groups the edges of the data graph and allows reaching higher degrees of parallelism while avoiding locality issues generally present in the vertex-centric graph algorithms. Moreover, we proposed *PDSim*, a Split-and-Combine approach for evaluating dual simulation based on *ST* and *PGSim*. We have provided theoretical guarantees on the correctness and the convergence of *PGSim* and *PDSim*. Moreover, the experimental results proved that the two propositions outperform the vertex-centric graph simulation by more than an order of magnitude. For future research, an interesting direction is proposing parallel algorithms for relaxed GPM in multi-labeled graphs. Another important challenge is addressing highly dynamic graphs; an important question would be whether or not the distributed data structure *ST* will maintain the same performance when update events arrive at high frequency.

Acknowledgements This work was supported by the Franco-Algerian program PHC Tassili BiGreen n°18 MDU 111 and by the DGRSDT grant FNRSDT N°253. The experiments presented in this work were carried out using the High Performance Computing Platform IBNBADIS provided by the Research Center on Scientific and Technical Information – CERIST (Algeria).

References

1. Bhattarai, B., Liu, H., Huang, H.H.: Ceci: Compact embedding cluster index for scalable subgraph matching. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1447–1462. ACM, Amsterdam, Netherlands (2019)
2. Bi, F., Chang, L., Lin, X., Qin, L., Zhang, W.: Efficient subgraph matching by postponing cartesian products. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1199–1214. ACM, San Francisco, California, USA (2016)
3. Bouhenni, S., Yahiaoui, S., Nouali-Taboudjemat, N., Kheddouci, H.: A survey on distributed graph pattern matching in massive graphs. ACM Computing Surveys **54**(2) (2021). DOI 10.1145/3439724
4. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-mat: A recursive model for graph mining. In: Proceedings of the 2004 SIAM International Conference on Data Mining, pp. 442–446. SIAM (2004)
5. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: An improved algorithm for matching large graphs. Proceedings of the 3rd IAPR Workshop on Graph-Based Representations in Pattern Recognition **219**(2), 149–159 (2001). DOI 10.1.1.101.5342
6. Csun, S., Luo, Q.: Parallelizing recursive backtracking based subgraph matching on a single machine. In: 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS), pp. 1–9. IEEE, Singapore, Singapore (2018)
7. Dustin, W.S.: Social media statistics 2020: Top networks by the numbers. https://dustinstout.com/social-media-statistics/ (2019). Accessed: 2021-03-01
8. Fan, W.: Graph pattern matching revised for social network analysis. In: Proceedings of the 15th International Conference on Database Theory, ICDT '12, p. 8–21. Association for Computing Machinery, New York, NY, USA (2012). DOI 10.1145/2274576.2274578. URL https://doi.org/10.1145/2274576.2274578
9. Fan, W., Li, J., Ma, S., Tang, N., Wu, Y., Wu, Y.: Graph Pattern Matching: From Intractable to Polynomial Time. Proceedings of the VLDB Endowment **3**(1-2), 264–275 (2010). DOI 10.14778/1920841.1920878
10. Fan, W., Wang, X., Wu, Y.: Diversified top-k graph pattern matching. Proceedings of the VLDB Endowment **6**(13), 1510–1521 (2013)

11. Fan, W., Wang, X., Wu, Y.: Incremental graph pattern matching. ACM Trans. Database Syst. **38**(3) (2013). DOI 10.1145/2489791. URL https://doi.org/10.1145/2489791
12. Fan, W., Wang, X., Wu, Y., Deng, D.: Distributed graph simulation: Impossibility and possibility. Proceedings of the VLDB Endowment **7**(12), 1083–1094 (2014). DOI 10.14778/2732977.2732983
13. Fan, W., Yu, W., Xu, J., Zhou, J., Luo, X., Yin, Q., Lu, P., Cao, Y., Xu, R.: Parallelizing sequential graph computations. ACM Transactions on Database Systems (TODS) **43**(4), 1–39 (2018)
14. Fard, A., Nisar, M.U., Ramaswamy, L., Miller, J.A., Saltz, M.: A distributed vertex-centric approach for pattern matching in massive graphs. In: 2013 IEEE International Conference on Big Data, pp. 403–411. IEEE, Santa Clara, CA, USA (2013). DOI 10.1109/BigData.2013.6691601
15. Gao, J., Liu, P., Kang, X., Zhang, L., Wang, J.: Prs: parallel relaxation simulation for massive graphs. The Computer Journal **59**(6), 848–860 (2016)
16. Gao, J., Zhou, C., Zhou, J., Yu, J.X.: Continuous pattern detection over billion-edge graph using distributed framework. In: 2014 IEEE 30th International Conference on Data Engineering, pp. 556–567. IEEE, Chicago, IL, USA (2014)
17. Garey, M.R., Johnson, D.S.: Computers and intractability: a guide to np-completeness (1979)
18. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pp. 289–300. ACM, Utah USA (2014)
19. Han, W.S., Lee, J., Lee, J.H.: Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, pp. 337–348. Association for Computing Machinery, New York, New York, USA (2013). DOI 10.1145/2463676.2465300. URL https://doi.org/10.1145/2463676.2465300
20. He, H., Singh, A.K.: Graphs-at-a-time: Query language and access methods for graph databases. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, pp. 405–418. Association for Computing Machinery, Vancouver, Canada (2008). DOI 10.1145/1376616.1376660
21. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proceedings of IEEE 36th Annual Foundations of Computer Science, pp. 453–462. IEEE, USA (1995)
22. Kao, J.S., Chou, J.: Distributed incremental pattern matching on streaming graphs. In: Proceedings of the ACM Workshop on High Performance Graph Processing, HPGP '16, p. 43–50. Association for Computing Machinery, Kyoto, Japan (2016). DOI 10.1145/2915516.2915519. URL https://doi.org/10.1145/2915516.2915519
23. Lai, L., Qin, L., Lin, X., Chang, L.: Scalable subgraph enumeration in mapreduce. Proceedings of the VLDB Endowment **8**(10), 974–985 (2015)
24. Lai, L., Qin, L., Lin, X., Zhang, Y., Chang, L., Yang, S.: Scalable distributed subgraph enumeration. Proceedings of the VLDB Endowment **10**(3), 217–228 (2016)
25. Lai, L., Qing, Z., Yang, Z., Jin, X., Lai, Z., Wang, R., Hao, K., Lin, X., Qin, L., Zhang, W., et al.: Distributed subgraph matching on timely dataflow. Proceedings of the VLDB Endowment **12**(10), 1099–1112 (2019)
26. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection (2014). URL http://snap.stanford.edu/data
27. Li, J., Cao, Y., Ma, S.: Relaxing graph pattern matching with explanations. In: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, pp. 1677–1686. ACM, Singapore Singapore (2017)
28. Li, J., Li, J., Wang, X.: A vertex-centric graph simulation algorithm for large graphs. In: Z. Xu, X. Gao, Q. Miao, Y. Zhang, J. Bu (eds.) Big Data, pp. 238–254. Springer, Singapore (2018)
29. Liu, C., Chen, C., Han, J., Yu, P.S.: Gplag: Detection of software plagiarism by program dependence graph analysis. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '06, pp. 872–881. Association for Computing Machinery, New York, NY, USA (2006). DOI 10.1145/1150402.1150522. URL https://doi.org/10.1145/1150402.1150522
30. Ma, S., Cao, Y., Fan, W., Huai, J., Wo, T.: Capturing topology in graph pattern matching. Proceedings of the VLDB Endowment **5**(4), 310–321 (2011)
31. Ma, S., Cao, Y., Huai, J., Wo, T.: Distributed graph pattern matching. In: Proceedings of the 21st International Conference on World Wide Web, WWW '12, pp. 949–958. Association for Computing Machinery, Lyon, France (2012). DOI 10.1145/2187836.2187963. URL https://doi.org/10.1145/2187836.2187963
32. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135–146 (2010)
33. Milner, R.: Communication and concurrency, vol. 84. Prentice hall Englewood Cliffs (1989)
34. Ogaard, K., Roy, H., Kase, S., Nagi, R., Sambhoos, K., Sudit, M.: Discovering patterns in social networks with graph matching algorithms. In: A.M. Greenberg, W.G. Kennedy, N.D. Bos (eds.) Social Computing, Behavioral-Cultural Modeling and Prediction, pp. 341–349. Springer, Berlin, Heidelberg (2013)

35. Peng, P., Zou, L., Özsu, M.T., Chen, L., Zhao, D.: Processing sparql queries over distributed rdf graphs. The VLDB Journal **25**(2), 243–268 (2016)
36. Qiao, M., Zhang, H., Cheng, H.: Subgraph matching: on compression and computation. Proceedings of the VLDB Endowment **11**(2), 176–188 (2017)
37. Ren, X., Wang, J.: Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. Proceedings of the VLDB Endowment **8**(5), 617–628 (2015)
38. Reza, T., Ripeanu, M., Tripoul, N., Sanders, G., Pearce, R.: Prunejuice: Pruning trillion-edge graphs to a precise pattern-matching solution. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 265–281. IEEE, Dallas, Texas, USA (2018). DOI 10.1109/SC.2018.00024
39. Schätzle, A., Przyjaciel-Zablocki, M., Berberich, T., Lausen, G.: S2x: Graph-parallel querying of rdf with graphx. In: F. Wang, G. Luo, C. Weng, A. Khan, P. Mitra, C. Yu (eds.) Biomedical Data Management and Graph Online Querying, pp. 155–168. Springer International Publishing, Cham (2016)
40. Serafini, M., De Francisci Morales, G., Siganos, G.: Qfrag: Distributed graph search via subgraph isomorphism. In: Proceedings of the 2017 Symposium on Cloud Computing, pp. 214–228. ACM, Santa Clara, CA (2017)
41. Shang, H., Zhang, Y., Lin, X., Yu, J.X.: Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. Proceedings of the VLDB Endowment **1**(1), 364–375 (2008)
42. Shemshadi, A., Sheng, Q.Z., Qin, Y.: Efficient pattern matching for graphs with multi-labeled nodes. Knowledge-Based Systems **109**, 256–265 (2016)
43. Sun, Z., Wang, H., Wang, H., Shao, B., Li, J.: Efficient subgraph matching on billion node graphs. Proceedings of the VLDB Endowment **5**(9), 788–799 (2012)
44. Ullmann, J.R.: An Algorithm for Subgraph Isomorphism. Journal of the ACM **23**(1), 31–42 (1976). DOI 10.1145/321921.321925
45. Wang, J., Ren, X., Anirban, S., Wu, X.W.: Correct filtering for subgraph isomorphism search in compressed vertex-labeled graphs. Information Sciences **482**, 363–373 (2019)
46. Wang, Z., Gu, R., Hu, W., Yuan, C., Huang, Y.: Benu: Distributed subgraph enumeration with backtracking-based framework. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 136–147. IEEE, Macao, Macao (2019)
47. Wu, X., Theodoratos, D., Skoutas, D., Lan, M.: Leveraging double simulation to efficiently evaluate hybrid patterns on data graphs. In: Z. Huang, W. Beek, H. Wang, R. Zhou, Y. Zhang (eds.) Web Information Systems Engineering – WISE 2020, pp. 255–269. Springer International Publishing, Cham (2020)
48. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: Graphx: A resilient distributed graph system on spark. In: First international workshop on graph data management experiences and systems, pp. 1–6. ACM, New York, USA (2013)
49. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I., et al.: Spark: Cluster computing with working sets. HotCloud **10**(10-10), 95 (2010)
50. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A distributed graph engine for web scale rdf data. Proceedings of the VLDB Endowment **6**(4), 265–276 (2013)
51. Zhao, P., Han, J.: On graph query optimization in large networks. Proceedings of the VLDB Endowment **3**(1-2), 340–351 (2010)