PruneJuice: Pruning Trillion-edge Graphs to a Precise Pattern-Matching Solution

Tahsin Reza[1,2], Matei Ripeanu[2], Nicolas Tripoul[2], Geoffrey Sanders[1], Roger Pearce[1]
[1]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
[2]Department of Electrical and Computer Engineering, University of British Columbia
{treza, matei, mtripoul}@ece.ubc.ca, {sanders29, rpearce}@llnl.gov

***Abstract*—Pattern matching is a powerful graph analysis tool. Unfortunately, existing solutions have limited scalability, support only a limited set of search patterns, and/or focus on only a subset of the real-world problems associated with pattern matching. This paper presents a new algorithmic pipeline that: *(i)* enables highly scalable pattern matching on labeled graphs, *(ii)* supports arbitrary patterns, *(iii)* enables trade-offs between precision and time-to-solution (while always selecting all vertices and edges that participate in matches, thus offering 100% recall), and *(iv)* supports a set of popular data analytics scenarios. We implement our approach on top of HavoqGT and demonstrate its advantages through strong and weak scaling experiments on massive-scale real-world (up to 257 billion edges) and synthetic (up to 4.4 trillion edges) graphs, respectively, and at scales (1,024 nodes / 36,864 cores) orders of magnitude larger than used in the past for similar problems.**

I. Introduction

Pattern matching in graphs, that is, finding subgraphs that *match* a small *template graph* within a large *background graph*, is fundamental to graph analysis and has applications in multiple areas such as social network analysis [1], bioinformatics [2], and information mining [3]. A *match* can be broadly categorized as either *exact* - i.e., there is a bijective map between the vertices/edges in the template and those in the matching subgraph, or *approximate* - the template and the match are just similar by some defined similarity metric [4]. If the template size is not limited, exact matching is not known to have a polynomial time solution in the general case [5]. Berry et al. [6] introduced the problem of *type-isomorphism*: metadata graphs where vertices and edges are labeled and, in addition to topological constraints, a *match* identifies nodes and edges with the same labels in the template and the background graph. While the labeled version does not reduce the worst-case complexity of the original problem, past experience [1], [7], [8] has demonstrated that label-based matching can be a powerful tool with potential for practical, real-world applications such as social network analysis.

The Challenge. Applications that mine graphs consist of tens of billions of edges are common [9], [10]. However, existing pattern matching solutions (we survey related work in §C) have limited capabilities: most importantly, they do not scale to massive graphs and/or support only a restricted set of search templates. Additionally, the algorithms at the core of the existing techniques are not suitable for today's infrastructures relying on horizontal scalability and share-nothing clusters as most of these algorithms are inherently sequential and difficult to parallelize [5], [11], [12]. Finally, pattern matching is susceptible to combinatorial explosion of the intermediate or final state: for low selectivity queries, the number of subgraphs partially (or entirely) matching the template can grow exponentially with the number of nodes and edges in the already large background graph [13], [14], posing serious memory and communication challenges.

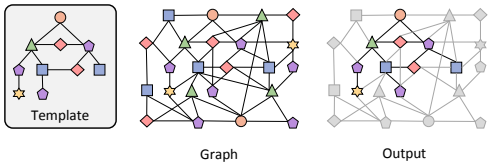


Fig. 1: An example of a background graph $\mathcal{G}$ (center), template $\mathcal{G}_0$ (left) and the output $\mathcal{G}^*$ after vertex and edge elimination (right). The output is a refined set of vertices and edges that contain every subgraph $\mathcal{H}$ that matches $\mathcal{G}_0$. The figures present vertex metadata as colored shapes. The eliminated vertices and edges are colored solid grey.

A New Approach for Scalable Pattern Matching. We propose a new algorithmic pipeline based on *graph pruning*. The idea of exploring graph pruning to support pattern matching stems from three key observations: First, the traditionally used *tree-search* techniques [5] generally attempt to enumerate all matches through explicit search. When a search path fails, such an unprofitable path is marked invalid and ignored in the subsequent steps. Similar to past works that use graph pruning [15], [16], [17] or, more generally, input reduction [18], we observe that it is much cheaper to first focus on eliminating the vertices and edges that do not meet the label and topological constraints introduced by the search template. Our experience shows that relatively simple pruning heuristics based on label and vertex neighborhood constraints can significantly prune away much of the background graph. *The key contributions of this paper is a pruning-based solution that limits the exponential growth of the state-space, scales to massive graphs and distributed memory machines with large number of processors, and supports arbitrary search templates*: the result of pruning is the *complete* set of all vertices and edges that participate in a match, with no false positives or false negatives.

Second, we observe that a *vertex-centric* formulation for such pruning algorithms exists, and this makes it possible to harness existing high-performance, vertex-centric frameworks (e.g., Giraph [19], GraphLab [20], HavoqGT [21]). In our vertex-centric formulation for pruning, a vertex must satisfy

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

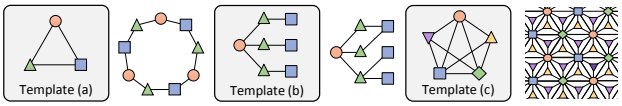


Fig. 2: Three examples of search templates and background graphs that justify the full set of pruning constraints. Template (a) is a 3-cycle; cycles of length $3k$ with repeated labels in the background graph meet neighborhood constraints, surviving LCC. Template (b) contains several vertices with non-unique labels; to its right there is a background graph that meets individual point-to-point path constraints, also surviving NLCC-path checking. Template (c) is characterized by two 4-cliques that overlap at a 3-cycle; the background graph structure to the right is doubly periodic (a 4×3 torus) and meets all edge and vertex cycle constraints, surviving NLCC-cycle checking. Templates (b) and (c) require NLCC-template-driven search to guarantee no false positives; template (a) only needs cycle checking in addition to LLC.

two types of constraints: *local* and *non-local*, to possibly be part of a match. *Local constraints* involve only the vertex and its neighborhood: a vertex in an exact match needs to *(i)* match the label of a corresponding vertex in the template, and *(ii)* have edges to vertices labeled as prescribed in the adjacency structure of this corresponding vertex in the template. *Non-local constraints* are topological requirements beyond the immediate neighborhood of a vertex (e.g., that the vertex must be part of a cycle). We describe how these constraints are generated, and our algorithmic solution to verify them in §III.

Third, we observe that, full match enumeration is not the most efficient avenue to support many high-level graph analysis scenarios. Depending on the final goal of the user, pattern matching problems fall into a number of categories which include: *(a)* determining if a match exists (or not) in the background graph (yes/no answer), *(b)* selecting all the vertices and edges that participate in matches, *(c)* ranking these vertices or edges based on their *centrality with respect to the search template*, i.e., the frequency of their participation matchs, *(d)* counting/estimating the total number of matches (comparable to the well-known triangle counting problem), or *(e)* enumerating all distinct matches in the background graph. The traditional approach is to perform *(e)* and to use the result of the enumeration to answer *(a) – (d)*. However, this approach is limited to small background graphs or is dependent on a low number of near and exact matches within the background graph (due to exponential growth of the search state-space). We argue that a pruning-based pipeline is not only a practical solution to *(a) – (d)* (and to other pattern-matching-related analytics), when full match enumeration is not the main interest, but also an efficient path towards full match enumeration. We demonstrate that a solution starting from the techniques we develop for pruning, efficiently supports match enumeration for two reasons: First, the pruned graph can be multiple orders of magnitude smaller than the background graph, and existing high-complexity enumeration routines (which otherwise would be intractable following the conventional approach) are now applicable. Second, our pruning techniques collect additional key information to accelerate match enumeration: for each vertex in the pruned graph, our algorithms build a list of its potential matches in the template (§V-C)).

Contributions. We capitalize on our preliminary study [15] that highlighted the effectiveness of pruning (yet in the context of a restricted set of templates) and design a pattern matching solution that is: *generic* - no restrictions on the set of patterns supported, *precise* - no false positives, *offers 100% recall* - retrieving all matches, *efficient* - low generated network traffic, and *scalable* - able to process graphs with up to trillions of edges on tens of thousands of cores (as demonstrated). In particular, we make the following contributions:

(i) Novel Asynchronous Algorithms. We have developed asynchronous vertex-centric algorithms able to prune the background graph to a precise, enumeration of all vertices and edges that participate in a match for *arbitrary* templates. The key gap we bridge is the ability support templates with repeated vertex labels, cycles, and arbitrary edge density. Importantly, the algorithms map well on a distributed asynchronous graph processing platform, thus enabling scalability and high-performance. For a given template, we propose heuristics to generate the constraints that are later used for pruning. We have also developed correctness proofs to show that these constraints eliminate all non-matching vertices and offer full recall (not included here due to space constraints).

(ii) Optimized Distributed Implementation. We offer an efficient implementation of these algorithms on the top of HavoqGT [21], an open-source asynchronous graph processing framework. The prototype includes two key optimizations that dramatically reduce the generated traffic: aggressive edge elimination, and what we call *work aggregation*, a technique that skips duplicate checks in non-local constraint checking, thus preventing possible combinatorial explosion. Additionally, our pruning implementation collects potential matching information: not only it prunes away all vertices and edges that do not participate in any match, but, for each of the vertices that remain, it collects their mappings to the search template. We use this information to accelerate match enumeration.

(iii) Proof of Feasibility. We demonstrate the applicability of this solution by experimenting on real-world and synthetic datasets orders of magnitude larger than prior work (§V). We evaluate scalability through two experiments: first, a strong scaling experiment using real-world datasets, including the largest openly available webgraph whose undirected version has over 257 billion edges; second, a weak scaling experiment using synthetic, R-MAT [22] graphs of up to 4.4 trillion edges, on up to 1,024 compute nodes (36,864 cores). We demonstrate support for search patterns representative of practical queries in both relatively low selectivity and *needle in the haystack* scenarios, and, to stress our system, consider patterns containing only the highest-frequency vertex labels (up to 14B instances). We show that our technique prunes the graph by orders of magnitude, which, combined with using the intermediary state generated by pruning, makes match enumeration feasible on graphs with trillions of edges.

(iv) Application Demonstration. We demonstrate the ability of our solution to support practical graph analytics queries. To this end, we use two real-world metadata graphs which we have curated from publicly available datasets, Reddit (3.9B vertices, 14B edges) and the smaller International Movie Database (IMDB), and demonstrate practical use cases of our technique to support rich pattern mining. (§V-D).

(v) Exploring Trade-offs, and the Impact of Strategic Design Choices and Optimizations. Our approach has the added flexibility that search can be stopped early, leading to the ability to trade faster time to an approximate solution (or even precise solution, yet without 100% precision guarantees) for precision (rate of false positives in the pruned graph) and precision guarantees. We also explore the impact of each optimization used (§A-A): the cumulative impact of these optimizations is a multiple orders of magnitude of runtime reduction, bringing pattern matching on massive metadata graphs in the realm of possible graph analytics.

II. Preliminaries

We aim to identify all structures within a large *background graph*, $\mathcal{G}$, identical to a small connected *template graph*, $\mathcal{G}_0$. We describe general graph properties for $\mathcal{G}$, and use the same notation (summarized in Table I) for other graph objects.

A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a collection of n vertices $\mathcal{V} = \{0, 1, ..., n-1\}$ and m edges $(i, j) \in \mathcal{E}$, where $i, j \in \mathcal{V}$ (i is the edge's *source* and j is the *target*). Here, we only discuss simple (i.e., no self-edges), undirected, vertex-labeled graphs, although the techniques are applicable to directed, non-simple graphs, with labels on both edges and vertices. An *undirected* $\mathcal{G}$ satisfies $(i, j) \in \mathcal{E}$ if and only if $(j, i) \in \mathcal{E}$. Vertex i's *adjacency list*, $adj(i)$, is the set of all j such that $(i, j) \in \mathcal{E}$. A *vertex-labeled graph* also has a set of n_ℓ labels $\mathcal{L}$ of which each vertex $i \in \mathcal{V}$ has an assignment $\ell(i) \in \mathcal{L}$.

A *walk* in $\mathcal{G}$ is an ordered subsequence of $\mathcal{V}$ where each consecutive pair is an edge in $\mathcal{E}$. A walk with no repeated vertices is a *path*. A path with equal first and last vertex is a *cycle*. An *acyclic* graph has no cycles.

We discuss several graph objects simultaneously: the *template graph* $\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$ the *background graph* $\mathcal{G}(\mathcal{V}, \mathcal{E})$, and the *current solution subgraph* $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$, with $\mathcal{V}^* \subset \mathcal{V}$ and $\mathcal{E}^* \subset \mathcal{E}$. Our techniques iteratively refine $\mathcal{V}^*$ and $\mathcal{E}^*$ until they converge to the union of all subgraphs of $\mathcal{G}$ that *exactly match* the template, $\mathcal{G}_0$.

For clarity, when referring to vertices and edges from the template graph, $\mathcal{G}_0$, we will use the notation $q_i \in \mathcal{V}_0$ and $(q_i, q_j) \in \mathcal{E}_0$. Conversely, we will use $v_i \in \mathcal{V}$ and $(v_i, v_j) \in \mathcal{E}$ for vertices and edges from the background graph $\mathcal{G}$ or the solution subgraph $\mathcal{G}^*$.

Definition 1. *A subgraph $\mathcal{H}(\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H}), \mathcal{V}_\mathcal{H} \subset \mathcal{V}, \mathcal{E}_\mathcal{H} \subset \mathcal{E}$ is an* exact match *of template graph $\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$ (in notation, $\mathcal{H} \sim \mathcal{G}_0$) if there exists a bijective function $\phi : \mathcal{V}_0 \longleftrightarrow \mathcal{V}_\mathcal{H}$ with the properties (Note that ϕ may not be unique for a given $\mathcal{H}$):*

(i) $\ell(\phi(q)) = \ell(q)$, *for all* $q \in \mathcal{V}_0$ *and*
(ii) $\forall (q_1, q_2) \in \mathcal{E}_0$, *we have* $(\phi(q_1), \phi(q_2)) \in \mathcal{E}_\mathcal{H}$
(iii) $\forall (v_1, v_2) \in \mathcal{E}_\mathcal{H}$, *we have* $(\phi^{-1}(v_1), \phi^{-1}(v_2)) \in \mathcal{E}_0$

Intuition for our Solution. The algorithms we develop here iteratively refines *vertex-match* functions $\omega(v) \subset \mathcal{V}_0$ such that, for every $v \in \mathcal{V}$, $\omega(v)$ stores a super-set of all template vertices v can possibly match. Set $\omega(v)$ converges to contain all possible values of $\phi^{-1}(v)$, were v involved in one or more matching subgraphs. When a single constraint involving $q \in \mathcal{V}_0$ is violated/unmet, q is no longer a possibility for v in a match and q is removed: $\omega(v) \leftarrow \omega(v) \setminus \{q\}$.

TABLE I: Symbolic notation used.

Object(s)	Notation
template graph, vertices, edges	$\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$
template graph sizes	$n_0 := \lvert\mathcal{V}_0\rvert$, $m_0 := \lvert\mathcal{E}_0\rvert$
template vertices	$\mathcal{V}_0 := \{q_0, q_1, ..., q_{n_0-1}\}$
template edges	$(q_i, q_j) \in \mathcal{E}_0$
set of vertices adjacent to q_i in $\mathcal{G}_0$	$adj(q_i)$
background graph, vertices, edges	$\mathcal{G}(\mathcal{V}, \mathcal{E})$
background graph sizes	$n := \lvert\mathcal{V}\rvert$, $m := \lvert\mathcal{E}\rvert$
background vertices	$\mathcal{V} := \{v_0, v_1, ..., v_{n-1}\}$
background edges	$(v_i, v_j) \in \mathcal{E}$
set of vertices adjacent to v_i in $\mathcal{G}$	$adj(v_i)$
maximum vertex degree in $\mathcal{G}$	d_{max}
label set	$\mathcal{L} = \{0, 1, ..., n_\ell - 1\}$
vertex label and label degree of q_i	$\ell(q_i) \in \mathcal{L}$,
matching subgraph, vertices, edges	$\mathcal{H}(\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$
solution subgraph, vertices, edges	$\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$
vertex match function	$\omega(v_i) \subset \mathcal{V}_0$
set of non-local constraints for $\mathcal{G}_0$	$\mathcal{K}_0$

Our previously developed pruning algorithms [15] required that all vertex labels in the template $\mathcal{G}_0$ were unique, and that $\mathcal{G}_0$ be *acyclic* or *edge-monocyclic* to ensure 100% precision. Our new techniques achieve 100% precision for arbitrary $\mathcal{G}_0$.

Remark 1. *Given an ordered sequence of all n_0 vertices $\{q_1, q_2, ..., q_{n_0}\} \subset \mathcal{V}_0$, a simple (although potentially expensive) search from $v_1 \in \mathcal{V}^*$ verifies if v_1 is in a match, with $\phi(q_1) = v_1$, or not. The search lists an ordered sequence $\{v_1, v_2, ..., v_{n_0}\} \subset \mathcal{V}^*$, with ϕ defined as $\phi(q_k) = v_k$. Search step k proposes a new v_k, checking Def.1 (i) and (ii). If all checks are passed, the search accepts v_k and moves on to step $(k+1)$, but terminates if no such v_k exists in $\mathcal{V}^*$. If the full list is generated with all label and edge checks passed then there exists a $\mathcal{H} \sim \mathcal{G}$ with $\mathcal{V}_\mathcal{H} = \{v_1, v_2, ..., v_{n_0}\}$.*

We call this *Template-Driven Search (TDS)* and develop an efficient distributed version in §IV, to apply to the solution $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$. If TDS has been applied successfully then there are no false positives remaining.

III. Graph Pruning for Scalable Matching

Our goal is to realize a technique which systematically eliminates all the vertices and edges that do not participate in any match $\mathcal{H} \sim \mathcal{G}_0$. This approach is motivated by viewing the template $\mathcal{G}_0$ as specifying a set of constraints the vertices and edges that participate in a match must meet. As a trivial example, any vertex v whose label $\ell(v)$ is not present in $\mathcal{G}_0$, cannot be present in an exact match. A vertex in an exact match also needs to have non-eliminated edges to non-eliminated vertices labeled as prescribed in the adjacency structure of the corresponding template vertex. Local constraints that involve a vertex and its neighborhood can be checked by having vertices communicate their (tentative) template match(s) with their one-hop neighbors in the *solution subgraph* $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$ (i.e., the currently pruned background graph). We call this process

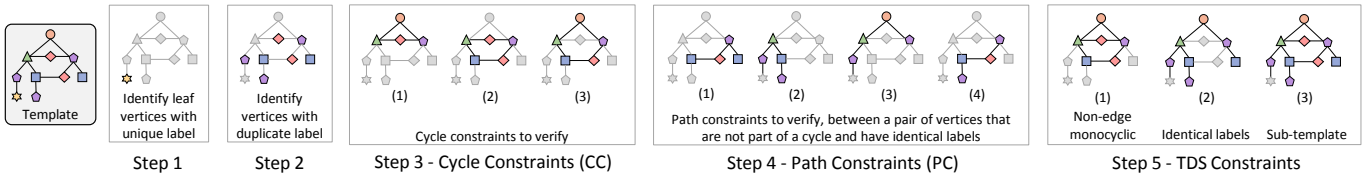


Fig. 3: A high-level depiction of non-local constraint generation for the template in Fig. 1. The figure shows the steps to generate required cycle constraints (CC), path constraints (PC) and higher-order constraints requiring template-drive search (TDS). (Due to limited space, the figure presents only a subset of the path and TDS constraints generated, however, sufficient to guarantee 100% precision.)

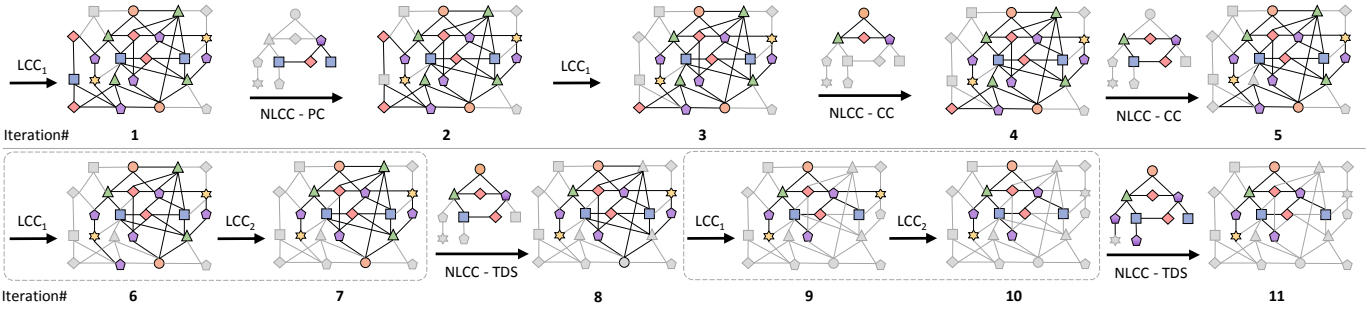


Fig. 4: Algorithm walk through for the example background graph and template in Fig. 1, depicting which vertices and edges in $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$ are eliminated (in solid grey) during each iteration. The NLCC constraints for $\mathcal{G}_0$ are listed in Fig. 3. Due to space limitations, the example does not show the application of some of the constraints in Fig. 3 that do not eliminate vertices or edges.

Local Constraint Checking (LCC). Our experiments show that LCC is responsible for removing the bulk of non-matching vertices and edges.

Some classes of templates (with cycles or repeated vertex labels), require additional routines to check non-local properties and to guarantee that all non-matching vertices are eliminated. (Fig. 2 highlights the need for these additional checks). To support arbitrary templates, we have developed a process which we dub *Non-local Constraint Checking (NLCC)*: first, based on the search template $\mathcal{G}_0$, we generate the set of constraints $\mathcal{K}_0$ that are to be verified, then prune the graph using each of them.

Alg. 1 presents an overview of our solution. This section provides high-level descriptions of the local and non-local constraint checking routines, while §IV provides the detailed distributed algorithms. For better understanding, Fig. 4 illustrates the complete workflow for the graph and pattern in Fig. 1 for which constraint generation is detailed in Fig. 3.

Algorithm 1 Main Pruning Loop

1: **Input:** background graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, template $\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$
2: **Output:** the solution subgraph $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$
3: generate non-local constraint set $\mathcal{K}_0$ from $\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$
4: $\mathcal{G}^* \leftarrow$ LOCAL_CONSTRAINT_CHECKING $(\mathcal{G}, \mathcal{G}_0)$
5: **while** $\mathcal{K}_0$ is not empty **do**
6: pick and remove next constraint $\mathcal{C}_0$ from $\mathcal{K}_0$
7: $\mathcal{G}^* \leftarrow$ NON_LOCAL_CONSTRAINT_CHECKING $(\mathcal{G}^*, \mathcal{G}_0, \mathcal{C}_0)$
8: **if** any vertex has been eliminated or
9: has one of its potential matches remove **then**
10: $\mathcal{G}^* \leftarrow$ LOCAL_CONSTRAINT_CHECKING $(\mathcal{G}^*, \mathcal{G}_0)$

Local Constraint Checking (LCC) involves a vertex and its neighborhood. The algorithm performs the following two operations. *(i) Vertex elimination*: the algorithm excludes the vertices that do not have a corresponding label in the template, then, iteratively, excludes the vertices that do not have neighbors as labeled in the template. For templates that have vertices with multiple neighbors with the same label, the algorithm verifies if a matching vertex in the background graph has a minimum number of distinct neighbors with the same label as prescribed in the template. *(ii) Edge elimination*: this excludes edges to eliminated neighbors and edges to neighbors whose labels do not match the labels prescribed in the adjacency structure of its corresponding template vertex (e.g., Fig. 4, Iteration #1). Edge elimination is crucial for scalability since, in a distributed setting, no messages are sent over eliminated edges, significantly improving the overall efficiency of the system (evaluated in §V, Fig. 8).

Non-local Constraint Checking (NLCC) aims to exclude vertices that fail to meet topological and label constraints beyond the one-hop neighborhood that is covered by LCC (Fig. 2). We have identified three types of non-local constraints which can be verified independently: *(i)* Cycle Constraints (CC), *(ii)* Path Constraints (PC), and *(iii)* constraints that require Template-Driven Search (TDS) (see Remark 1). For arbitrary templates, TDS constraints based on aggregating multiple paths/cycles enable further pruning and, when based on full template enumeration[1], insure that pruning yields no false positives. Checking TDS constraints, however, can be expensive. To reduce the overall cost, we first generate single cycle- and path-based constraints which are usally less costly to verify and prune the graph before using TDS (the effectiveness of this ordering is evaluated in Fig. 14(c)).

High-level Algorithmic Approach. Regardless of constraint type, NLCC leverages a *token passing* approach: tokens are issued by background graph vertices whose corresponding template vertices are identified to have non-local constraints. After a fixed number of steps, we check if a token has arrived

[1]In a number of corner cases not *all* TDS constraints need to be aggregated for full precision, we skip this discussion for lack of space.

where expected (e.g., back to the originating vertex for checking the existence of a cycle). If not, then the issuing vertex does not satisfy the required constraint and is eliminated. Along the token path, the system verifies that all expected labels are encountered and, where necessary, uses the path information accumulated with the token to verify that different/repeated node identity constraint expectations are met. Next, we discuss how each of the constraints is verified.

Cycle Constraints (CC). Higher-order structures within $\mathcal{G}$ that survive LCC, but do not contain $\mathcal{G}_0$, are possible if $\mathcal{G}_0$ contains a cycle (this happens if $\mathcal{G}$ contains one or more unrolled cycles as in Fig. 2, Template (a)). To address this, we directly check for cycles of the correct length.

Path Constraints (PC). If the template $\mathcal{G}_0$ has two or more vertices with the same label three or more hops away from each other, then structures in $\mathcal{G}$ that survive LCC, yet contain no match, are possible (Fig. 2, Template (b)). Thus, for every vertex pair with the same label in $\mathcal{G}_0$, we directly check the existence of a path of correct length and label sequence for prospective matching vertices in $\mathcal{G}^*$. Opposite to cycle checking, after a fixed number of steps, a token must be received by a vertex *different* from the initiating vertex but with an identical label.

TDS Constraints. These are partial or complete (i.e., including all edges of the template) walks on the template. The token *walks* the constraint in the background graph and verifies that each node visited meets its neighborhood constraints (Remark 1) - in our distributed memory setting, this is done by maintaining a history of the walk and checking that previously visited vertices are revisited as expected. TDS constraints are crucial to guarantee no-false positives for templates that are non-edge-monocyclic or have repeated labels (Fig. 2).

Token Generation. For CC and TDS constraints, a token must be initiated from each vertex that may participate in the substructure, whereas for PC, tokens are only initiated from terminal vertices.

Further Optimization - Work Aggregation. All NLCC constraints attempt to identify if a walk exists from a specific vertex and though vertices with specific labels. Since the goal is to identify the *existence* of any such path, and multiple intermediate paths in the background graph often exist, to prevent combinatorial explosion, our duplicate work detection mechanism prevents an intermediately vertex (in the token path) from forwarding a duplicate token (evaluated in §A-A).

Non-local Constraint Generation. For a relatively large class of templates (i.e., those with unique labels and mono-cyclic edges), LCC and CC are sufficient to generate a precise solution. For the general case, we generate non-local constraints using the following heuristic. (Fig. 3 shows, for a given template, the non-local constraints to be verified and Fig. 4 shows how pruning progresses.) First, all the leaf vertices with unique labels are identified and ignored from this process (as LCC guarantees pruning if there is no match). Next, if the template has cycles, then individual cycles are identified (e.g., Fig. 3, Step 3) and a cycle constraint is generated for each cycle. Next, vertices with identical label are identified and all path constraints are generated for all such pairs (e.g., Fig. 3, Step 4, pentagonal vertices).

Finally, we identify TDS constraints in three steps. First, for templates with multiple cycles sharing more than one edge (i.e., non-edge-monocyclic), a TDS cyclic constraint is generated through union of previously identified cycle constraints. This results in a higher-order cyclic structure with a maximal set of edges that cover all the non-edge-mono cycles (e.g., Fig. 3, Step 5(1)). Second, for templates with repeated labels, a new TDS constraint is generated through a union of all previously identified path constraints. This procedure generates higher order structure that covers all the template vertices with repeated labels (e.g., Fig. 3, Step 5(2)). The final step generates a TDS constraint as the union of the previously identified two constraints (e.g., Fig. 3, Step 5(3)). (Note that the above is a heuristic, more constraints could be generated by creating various possible combinations of cycles and paths. Only this third step is mandatory to eliminate all false positives.)

IV. Asynchronous Distributed Algorithms

This section presents the constraint checking algorithms on top of HavoqGT [23], a MPI-based framework that supports asynchronous graph algorithms in distributed environments. Our choice for HavoqGT's is driven by multiple considerations: first, unlike most graph processing frameworks that only support the Bulk Synchronous Parallel (BSP) model, HavoqGT has been designed to support asynchronous algorithms, essential to achieve high-performance; second, the framework has excellent scaling properties [24] [21]; and, finally, it enables load balancing: HavoqGT's *delegate partitioned graph* distributes the edges of each high-degree vertices across multiple compute nodes, which is crucial for achieving scalability for scale-free graphs with skewed degree distribution.

In HavoqGT, graph algorithms are implemented as vertex-callbacks: the user-defined *visit()* callback can only access and update the state of a vertex. The framework offers the ability to generate events (a.k.a. 'visitors' in HavoqGT lingo) that trigger this callback - either at the entire graph level using the $do\_traversal()$ method, or for a neighboring vertex using the $push(visitor)$ call (this enables asynchronous vertex-to-vertex communication). The asynchronous graph computation completes when all events have been processed, which is determined by a distributed quiescence detection algorithm [25].

Alg. 1 outlines the key steps of the graph pruning procedure. Below, we describe the distributed implementation of the local and non-local constraint checking routines. Alg. 2 lists the state maintained at each vertex and its initialization.

Local Constraint Checking is implemented as an iterative process (Alg. 3 and the corresponding callback, Alg. 4). Each iteration initiates an asynchronous traversal by invoking the $do\_traversal()$ method and, as a result, each active vertex receives a visitor with $msg_{type} = init$. In the triggered $visit()$ callback, if the label of a vertex v_j in the graph is a match for the label of any vertex in the template and the vertex is still

Algorithm 2 Vertex State and Initialization

1: status of vertex v_j: $\alpha(v_j) \leftarrow true$ (active) if $\exists q \in \mathcal{V}_0$ s.t. $\ell(v_j) = \ell(q)$, otherwise $false$ (i.e., v_j has been pruned)
2: set of possible matches in template for vertex v_j: $\omega(v_j) \leftarrow$ initially all $q_k \in \mathcal{V}_0$ s.t. $\ell(q_k) = \ell(v_j)$
3: map of active edges of vertex v_j: $\varepsilon(v_j) \leftarrow$ keys are initialized to $adj(v_j)$. The value field, which is initially $\emptyset$, is set to $\omega(v_i)$, for each $v_i \in \varepsilon(v_j)$ that has communicated its state to v_j.
4: set of already forwarded tokens by vertex v_j: $\tau(v_j) \leftarrow$ initially empty, used for work aggregation in NLCC

active, it creates visitors for all its active neighbors in $\varepsilon(v_j)$ with $msg_{type} = alive$ (Alg. 4, line #9). When a vertex v_j is visited with $msg_{type} = alive$, it verifies whether the sender vertex v_s satisfies one of its own (v_j's) template constraints by invoking the function $\eta(v_s, v_j)$. By the end of an iteration, if v_j satisfies all the template constraints: i.e, it has neighbors with required labels (and, if needed, a minimum of distinct neighbors with the same label as prescribed in the template), it stays active (i.e., $\alpha(v_j) = true$) for the next iteration. For templates that have multiple vertices with the same label, in any iteration, a vertex with that label in the background graph could match any of these vertices in the template, so each mach must be verified independently. If v_j fails to satisfy the required constraints for a template vertex $q_k \in \omega(v_j)$, q_k is removed from $\omega(v_j)$. At any stage, if $\omega(v_j)$ becomes empty, then v_j is marked inactive ($\alpha(v_j) \leftarrow false$) and never creates visitors again. Edge elimination excludes two categories of edges: first, the edges to neighbors: $v_i \in \varepsilon(v_j)$ from which v_j did not receive an $alive$ message, and, second, the edges to neighbors whose labels do not match the labels prescribed in the adjacency structure of the corresponding template vertex(s) in $\omega(v_j)$. A vertex v_j is also marked inactive if its active edge list $\varepsilon(v_j)$ becomes empty. Iterations continue until no vertex and/or edge is marked inactive.

Algorithm 3 Local Constraint Checking

1: $\eta(v_s, v_j)$ - tests if v_s satisfies a local constraint of v_j; returns $\omega(s)$
2: if constraints are satisfied, $\emptyset$ otherwise
3: **procedure** LOCAL_CONSTRAINT_CHECKING ($\mathcal{G}, \mathcal{G}_0$)
4: **do**
5: $do\_traversal(msg_{type} \leftarrow init)$
6: **barrier**
7: **for all** $v_j \in \mathcal{V}$ **do**
8: $\omega' \leftarrow \emptyset$ ▷ set of matches in template for neighbors of v_j
9: **for all** $v_i \in \varepsilon(v_j)$ **do**
10: **if** $\eta(v_i, v_j) = \emptyset$ **then**
11: $\varepsilon(v_j).remove(v_i)$ ▷ edge eliminated
12: **continue**
13: **else**
14: $\omega' \leftarrow \omega' \cup \eta(v_i, v_j)$ ▷ accum. matches of the nbrs.
15: reset the value field of $v_i \in \varepsilon(v_j)$ for the next iteration
16: **for all** $q_k \in \omega(v_j)$ **do** ▷ for each potential match
17: **if** $adj(q_k) \nsubseteq \omega'$ **then**
18: ▷ q_k does not meet neighbor requirements
19: $\omega(v_j).remove(q_k)$ ▷ remove from potential matches
20: **continue**
21: **if** $\varepsilon(v_j) = \emptyset$ **or** $\omega(v_j) = \emptyset$ **then**
22: $\alpha(v_j) \leftarrow false$ ▷ vertex eliminated
23: **while** vertices or edges are eliminated ▷ global detection

Algorithm 4 Local Constraint Checking Visitor

1: visitor state: v_j - vertex that is visited
2: visitor state: v_s - vertex that originated the visitor
3: visitor state: $\omega(v_s)$ - set of possible matches in template for vertex v_s
4: visitor state: msg_{type} - $init$ or $alive$
5: **procedure** VISIT($\mathcal{G}, vq$) ▷ vq - visitor queue
6: **if** $\alpha(v_j) = false$ **then return**
7: **if** $msg_{type} = init$ **then**
8: **for all** $v_i \in \varepsilon(v_j)$ **do**
9: $vis \leftarrow$ LCC_VISITOR(v_i, v_j, $\omega(v_j)$, $alive$)
10: $vq.push(vis)$
11: **else if** $msg_{type} = alive$ **then**
12: $\varepsilon(v_j).get(v_s) \leftarrow \omega(v_s)$

Non-local Constraint Checking routine iterates over $\mathcal{K}_0$, the set of non-local constraints to be checked, and validates each $\mathcal{C}_0 \in \mathcal{K}_0$ one at a time. Alg. 5 describes the solution to verify a single constraint: tokens are initiated through an asynchronous traversal by invoking the $do\_traversal()$ method. Each active vertex $v_j \in \mathcal{G}$ that is a potential match for the vertex q_0 at the head of a path $\mathcal{C}_0$, broadcasts a token to all its active neighbors in $\varepsilon(v_j)$. A map γ is used to track these token issuers. A $token$ is a tuple (t, r) where t is an ordered list of vertices that have forwarded the token and r is the hop-counter; $t_0 \in t$ is the token-issuing vertex in $\mathcal{G}$. The ordered list t is essential for TDS since it enables detection of distinct vertices with the same label in the token path.

Algorithm 5 Non-local Constraint Checking

1: **procedure** NON_LOCAL_CONSTRAINT_CHECKING($\mathcal{G}, \mathcal{G}_0, \mathcal{C}_0$)
2: $\gamma \leftarrow$ map of token source vertices (in $\mathcal{G}$) for $\mathcal{C}_0$; the value field
3: (initialized to *false*) is set to *true* if the token source vertex meets
4: the requirements of $\mathcal{C}_0$
5: $do\_traversal(msg_{type} \leftarrow init)$
6: **barrier**
7: **for all** $v_j \in \gamma$ **do**
8: **if** $\gamma.get(v_j) \neq true$ **then**
9: ▷ violates $\mathcal{C}_0$, eliminate potential match
10: $\omega(v_j).remove(q_0)$ **where** q_0 is the first vertex in $\mathcal{C}_0$
11: **if** $\omega(v_j) = \emptyset$ **then** ▷ no potential match left
12: $\alpha(v_j) \leftarrow false$ ▷ vertex eliminated
13: $\forall v_j \in \mathcal{V}$, reset $\tau(v_j)$

When an active vertex v_j receives a token with $msg_{type} = forward$, it verifies that if $\omega(v_j)$ is a match for the next entry in $\mathcal{C}_0$, if it has received the token from a valid neighbor (with respect to entries in $\mathcal{C}_0$), and that the current hop count is $< |\mathcal{C}_0|$. If these requirements are satisfied (i.e., μ returns $true$), v_j sets itself as the forwarding vertex (added to t), increments the hop count, and broadcasts the token to all its active neighbors in $\varepsilon(v_j)$. If any of the constraints are not met, v_j drops the token. If the hop count r is equal to $|\mathcal{C}_0|$ and v_j is the same as the source vertex in the token, for a cyclic template, a path has been found and v_j is marked $true$ in γ. For path constraints, an acknowledgement is sent to the token issuer to update its status in γ (Alg. 6, lines #28 – #31). Once verification of a constraint $\mathcal{C}_0$ has been completed, the vertices that are not marked $true$ in γ, are invalidated, i.e., $\alpha(v_j) \leftarrow false$ (Alg. 5, line #12). NLCC uses an unordered set $\tau(v_j)$ (Alg. 2, line #4) for *work aggregation* (see Alg. 6, line #14): at each vertex, this is used to detect if another copy of $token$ has already visited the vertex v_j taking a different path.

Algorithm 6 Non-local Constraint Checking Visitor

visitor state: v_j - vertex that is visited
visitor state: $token$ - the token is a tuple (t, r) where t is an ordered list of vertices that have forwarded the token and r is the hop-counter; $t_0 \in t$ is the vertex that originated the token
visitor state: msg_{type} - $init$, $forward$ or ack
$\mu(v_j, \mathcal{C}_0, token)$ - tests if v_j satisfies requirements of $\mathcal{C}_0$ for the current state of $token$; returns $true$ if constraints are met, $false$ otherwise
procedure VISIT($\mathcal{G}, vq$)
  if $\alpha(v_j) = false$ **then return**
  if $msg_{type} = init$ **and** $\exists q_k \in \omega(v_j)$ **where** $q_k = q_0 \in \mathcal{C}_0$ **then**
    ▷ initiate a token; v_j is the token source
    $t.add(v_j)$; $r \leftarrow 1$; $token \leftarrow (t, r)$; $\gamma.insert(v_j, false)$
    for all $v_i \in \varepsilon(v_j)$ **do**
      $vis \leftarrow$ NLCC_VISITOR($v_i, token, forward$)
      $vq.push(vis)$
  else if $msg_{type} = forward$ **then** ▷ v_j received a token
    if $token \notin \tau(v_j)$ **then** ▷ work aggregation optimization
      $\tau(v_j).insert(token)$
    else return ▷ ignore $token$ if it was previously forwarded by v_j
    if $\mu(v_j, \mathcal{C}_0, token) = true$ **and** $token.r < |\mathcal{C}_0|$ **then**
      ▷ the walk can be extended with v_j and it has not yet reached $|\mathcal{C}_0|$ length
      $token.t.add(v_j)$; $token.r \leftarrow token.r + 1$;
      for all $v_i \in \varepsilon(v_j)$ **do** ▷ forward the token
        $vis \leftarrow$ NLCC_VISITOR($v_i, token, forward$)
        $vq.push(vis)$
    else if $\mu(v_j, \mathcal{C}_0, token) = true$ **and** $token.r = |\mathcal{C}_0|$ **then**
      ▷ the walk can be extended with v_j and it has reached $|\mathcal{C}_0|$ length
      if $\mathcal{C}_0$ is cyclic **and** $t_0 = v_j$ **then**
        $\gamma.get(v_j) \leftarrow true$ **return** ▷ v_j meets requirements of $\mathcal{C}_0$
      else if $\mathcal{C}_0$ is acyclic **and** $t_0 \neq v_j$ **then**
        $vis \leftarrow$ NLCC_VISITOR($t_0, token, ack$)
        $vq.push(vis)$ ▷ send ack to the token originator $t_0 \in t$
  else if $msg_{type} = ack$ **then**
    $\gamma.get(v_j) \leftarrow true$ **return** ▷ v_j meets requirements of $\mathcal{C}_0$

Termination, Output, and Match Enumeration Queries. If NLCC is not required, the search terminates when no vertex is eliminated (or none of its potential matches is removed) in an LCC iteration. Otherwise, the search terminates when all constraints in $\mathcal{K}_0$ have been verified and no vertex is eliminated (or none of its potential matches is removed) in the following LCC phase. The output is: *(i)* the set of vertices and edges that survived the iterative elimination process and, *(ii)* for each vertex in this set, the mapping in the template where a match has been identified. A distributed enumeration or counting routine can operate on the pruned graph with this information: Alg. 6 can be slightly modified to obtain the enumeration of the matches in the background graph: the constraint used is the full template, work aggregation is turned off, and each possible match is verified.

Metadata Store. Metadata is stored independent of the graph topology itself (which uses CSR format [26]). At initialization, only the required attributes are read from the file(s) stored on a distributed file system. A light-weight distributed process builds the in-memory (or memory-mapped) metadata store. On 256 nodes, for the 257 billion edge Web Data Commons graph [27], the metadata store can be built in under two minutes. Although, in this paper, we consider vertex metadata (i.e., labels) only, edge metadata is also supported.

V. Evaluation

We present strong (§V-B) and weak (§V-A) scaling experiments of pruning on massive real-world and synthetic graphs; additionally we demonstrate full match enumeration starting from the pruned graph (§V-C); we evaluate the effectiveness of the optimizations our system incorporates (§A-A); we highlight the use of our system in the context of realistic data analytics scenarios (§V-D); we explore time-to-solution vs. precision/guarantees trade-offs (§V-E); and finally, we compare our solution with a recent work, QFrag [28] (§A-C).

Testbed. The testbed is the 2.6PFlop Quartz cluster at the Lawrence Livermore National Lab., comprised of 2,634 nodes and the Intel Omni-Path interconnect. Each node has two 18-core Intel Xeon E5-2695v4 @2.10GHz processors and 128GB of memory [29]. We run one MPI process per core.

Datasets. We summarize the main characteristics of the datasets used for evaluation and explain how we have generated vertex labels where necessary. For all graphs, we created undirected versions of the graphs; two directed edges are used to represent each undirected edge.

The datasets used for evaluation.

	Type	$\lvert\mathcal{V}\rvert$	$2\lvert\mathcal{E}\rvert$	d_{max}
Web Data Commons [27]	Real	3.5B	257B	95M
Reddit [30]	Real	3.9B	14B	19M
Internet Movie Database [31]	Real	5M	29M	552K
Patent [28]	Real	2.7M	28M	789
Youtube [28]	Real	4.6M	88M	2.5K
R-MAT up to Scale 37 [22]	Synthetic	137B	4.4T	612M

Web Data Commons (WDC) graph is a web-graph whose vertices are webpages and edges are hyperlinks. To create vertex labels, we extract the top-level domain names from the webpage URL, e.g., *.org* or *.edu*. If the URL contains a common second-level domain name, it is chosen over, the top-level domain name. For example, for *ox.ac.uk*, *.ac* is selected as the vertex label. A total of 2,903 unique labels are distributed among the 3.5B graph vertices. We curated the *Reddit* social-media graph from an open archive [30] of billions of public posts and comments from Reddit.com. Reddit allows its users to rate (upvote or downvote) others' posts and comments. The graph has four types of vertices: *Author, Post, Comment* and *Subreddit* (a category for posts). For the Post and Comment there are three possible labels: *Positive, Negative,* and *Neutral* (indicating the overall balance of positive and negative votes) or *No* rating. An edge is possible between an Author and a Post, an Author and a Comment, a Subreddit and a Post, a Post and a Comment (to that post), and between two Comments that have a parent-child relationship. The *International Movie Database (IMDB)* graph was curated from the publicly available data [31]. The graph has five types of vertices: *Movie, Genre, Actress, Actor* and *Director*. An edge is only possible between a Movie type vertex and a non-Movie type vertex. We use the smaller *Patent* and *YouTube* graphs to compare published results by Serafini et al. [28]. The synthetic *R-MAT* graphs exhibit approximate power-law degree distribution. These graphs were created following the Graph 500 [32] standards: 2^{Scale} vertices and an

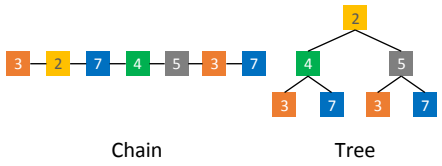


Fig. 5: Chain and Tree patterns used. Both patterns have two pairs of vertices with the same (numeric) label, hence, require non-local constraint checking (NLCC), more precisely, path checking. The labels used are the most frequent in the R-MAT graphs and cover ∼30% of all the vertices in the graphs.

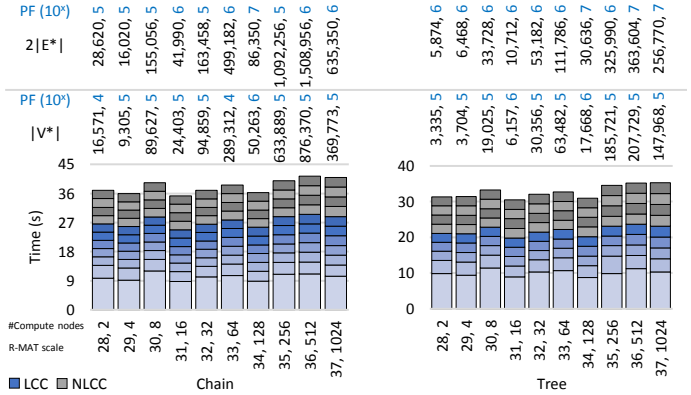


Fig. 6: Runtime and pattern selectivity for weak scaling experiments, broken down to individual iterations, for the chain (left) and tree (right) patterns presented in Fig. 5. The X-axis labels present the R-MAT scale and the node count used for the experiment. (Each node hosts two processors, each with 18 cores.) The number of vertices and edges in each pruned solution is shown on the top of respective bar plot. The pruning factors (PF), i.e., the *order of magnitude* reduction in number of vertices/edges compared to the original background graph are also shown. A flat line indicates perfect weak scaling. Time for LLC and NLCC phases is presented with different colors.

undirected edge factor of 16. For example, a Scale 30 graph has $|\mathcal{V}| = 2^{30}$ and $|\mathcal{E}| \approx 32 \times 2^{30}$ (as we create a directed version). We leverage degree information to create vertex labels, computed using the formula, $\ell(v_i) = \lceil \log_2(d(v_i)+1) \rceil$.

Search Templates. To stress our system, we use templates based on patterns naturally occurring, and relatively frequent, in the template graphs. The R-MAT (Fig. 5) and WDC (Fig. 7) patterns include vertex labels that are among the most frequent in the respective graphs. The Reddit and IMDB patterns (Fig. 11) include most of the vertex labels in these two graphs. We chose templates to exercise different constraint checking scenarios: the search templates have repeated vertex metadata and non-edge-monocyclic properties.

Experimental Methodology. All runtime numbers provided are averages over 10 runs. For weak scaling experiments, we do not present scaling numbers for a single node as this experiment does not involve network communication and benefits from data locality. For strong scaling experiments, the smallest experiment uses 64 nodes, as this is the lowest number of nodes that can load the graph in memory.

A. Weak Scaling Experiments

To evaluate the ability to process massive graphs, we use weak scaling experiments and the synthetic R-MAT graphs up to Scale 37 (∼4.4T edges) and up to 1024 nodes (36,864 cores). Fig. 5 shows the two search pattern used and Fig. 6 presents the runtimes. Since there are multiple vertices in the

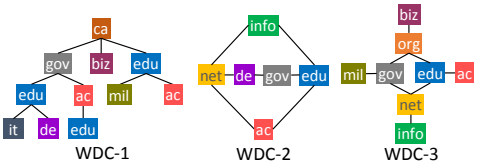


Fig. 7: WDC patterns using top/second-level domain names as labels. The labels selected are among the most frequent, covering ∼22% of the vertices in the WDC graph: *org* covers ∼220M vertices, the 2nd most frequent after *com* and *mil* is the least frequent among these labels, covering ∼153K vertices.

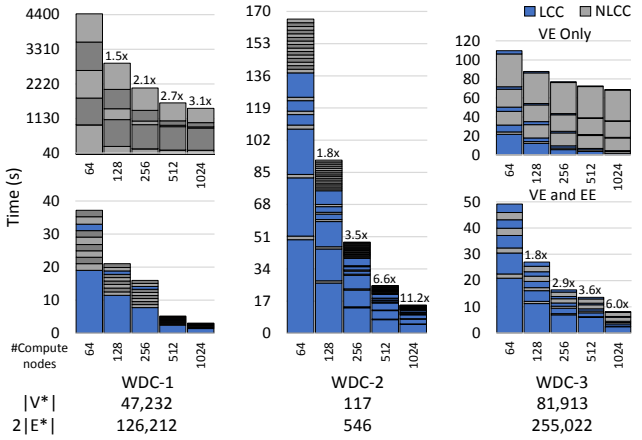


Fig. 8: Runtime for strong scaling experiments, broken down to individual phases (LCC and NLCC are in different colors) for the three patterns presented in Fig. 7. The top row of X-axis labels represent the number of compute nodes. (Each node hosts two processors, each with 18 cores.) The last two rows are number of vertices and edges in the pruned graph, respectively. For better visibility, for WDC-1 (left plots), runtime for different iterations are split into two scales on the Y-axis: LCC and NLCC-path constraints are at the bottom, and LCC and NLCC-TDS constraints are at the top. To highlight the impact of edge elimination, for WDC-3 (right plots), we present results with edge elimination disabled (top) and enabled (bottom). Speedup over the 64 node configuration is also shown on the top of each stacked bar plot.

pattern with identical labels (at more than two-hop distance), the patterns require NLCC - path checking - to ensure no false positives in the pruned graph. We see steady scaling all the way to the Scale 37 graph, which has ∼4.4 trillion edges, on 1024 nodes (36,864 cores). Runtime is broken down to the individual iteration to evaluate scaling and the individual contribution of each intermediate step. As a graph gets pruned, the subsequent iterations require less time. Fig. 6 includes (at the top of each bar) the final number of vertices and edges that participate in the respective patterns. Note that the NLCC phases (needed to guarantee a precise solution) do not delete any vertex or edge, hence, no further LCC phase is invoked.

B. Strong Scaling Experiments

Fig. 8 shows the runtimes for strong scaling experiments when using the real-world WDC graph on up to 1024 nodes (36,864 cores). Intuitively, pattern matching on the WDC graph is harder than on the R-MAT graphs, as the WDC graph is both denser and has a more skewed degree distribution. We use the patterns presented in Fig. 7. WDC-1 is acyclic, yet has multiple vertices with the same label and thus, requires non-local constraint checking (PC and TDS). For better visibility, the plot splits checking initial LCC and NLCC-path constraints (bottom left) from NLCC-TDS constraints (top left). We notice near perfect scaling for the LCC phases, however, some of

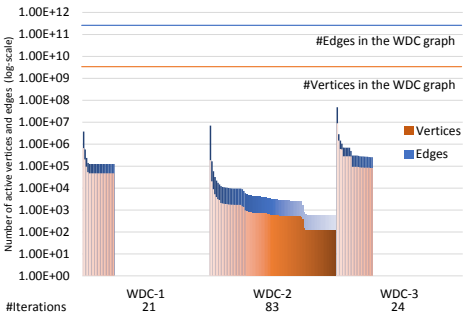


Fig. 9: Evolution of the number of active vertices and edges after each iteration for the same experiments as in Fig. 8. The bottom row of X-axis labels represent number of iterations required for a precise solution. Note that the Y-axis is on log scale.

the NLCC phases do not show linear scaling (explained in §A-B). WDC-2 is an example of a pattern with multiple cycles sharing edges, and relies on CC and TDS constraint checking to guarantee no false positive matches. WDC-2 shows near-linear scaling with ∼1/3 of the total time spent in the first LCC phase and little time spent in the NLCC phases. WDC-3 is a monocyclic template and, when edge elimination is used (bottom right) shows near linear scaling for both LCC and NLCC phases. The top right plot highlights the key performance impact of edge elimination: without it, the NLCC phases take almost one order of magnitude longer and the entire pruning 2–9× longer.

TABLE II: Match enumeration statistics: Number of matches for the chain and tree patterns (Fig. 5, top table), and WDC (Fig. 7), Reddit and IMDB (Fig. 11) patterns (bottom table) and the enumeration times, starting from the pruned graphs. Note that for WDC-1 and WDC-3, due to the extremely large number of matches (over half billion in each case) we stop enumeration early.

R-MAT Scale	#Compute Nodes	Chain		Tree	
		Count	Time (s)	Count	Time (s)
28	2	2,716	10.36	1,186	10.38
31	16	3,747	10.54	1,488	10.40
34	128	7,529	11.28	3,766	11.28
37	1024	55,710	10.10	32,532	5.53

Pattern	WDC-1	WDC-2	WDC-3	RDT-1	RDT-2	IMDB-1
Count	668M*	2,444	1.49B*	24K	518K	1.68M
Time	4min	1.84s	40h+	6.78s	4.85s	10h
#Compute Nodes	64	64	16	64	64	8

C. Match Enumeration

As our technique prunes the graph by orders of magnitude (see Fig. 9 and 12(b)), match counting and full match enumeration are now feasible. Table II (top) lists the number of distinct matches and the time to enumerate the chain and tree patterns on some of the R-MAT graphs we used. While these results prove that our match enumeration routines scale well, the match counts and enumeration time for WDC, Reddit and IMDB patterns listed on Table II (bottom) are more revealing.

There are three important takeaways: First, while our match enumeration technique is able to enumerate an immense number of matches (see, for example results for WDC-1 and WDC-3 with 500+ million matches, or even IMDB-1 with 1.5+ million matches), presenting results as pruned vertex/edge

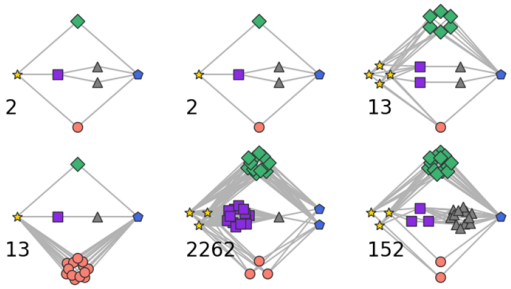


Fig. 10: The WDC-2 pattern matches in the background graph. The number of matches in each of the six connected components are also shown.

sets (with less than 100,000 vertices) avoids the combinatorial explosion and makes is feasible to imagine further analytics. Second, as Fig. 10 clearly shows, presenting the results as the union of *all* matches (rather than explicit match enumeration) is not only more space efficient, but also, in some cases even easy to directly understand by a human analyst. Finally, while we omit details due to lack of space, we note that key to supporting match enumeration, is edge pruning: this reduces the edge density in the pruned WDC graph by a factor of 10–15× (compared to using vertex pruning alone).

D. Example Use Case: Social Network Analysis

We demonstrate the ability of our scalable pattern matching technique to support complex data analytics scenarios in the context of social networks. Today's user experience on social media platforms is tainted by the existence of malicious actors such as bots, trolls, and spammers. This highlights the importance of detecting unusual activity patterns that may indicate potential malicious attacks. We present three use cases: one for the IMDB graph and two queries that attempt to uncover suspicious activity in the Reddit dataset.

Fig. 11 summarizes the scenarios we target and presents the corresponding search patterns. Fig. 12 shows runtime for these scenarios, broken down to individual LCC and NLCC iteration levels. Although RDT-1 is much less frequent than RDT-2, on the same 64 nodes, pruning for RDT-1 takes more than 3× longer to complete as it spends more time verifying the non-local constraints. Although both patterns have a 6-cycle, RDT-2 allows verification of the two smaller cycles in isolation. (For NLCC, a longer path typically results in larger generated message traffic.) IMDB-1, on 8 nodes, spends the majority of the time verifying non-local, specifically TDS, constraints.

E. Pruning Precision/Guarantees vs. Time-to-Solution

Our approach gradually refines $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$ down to the complete set of vertices and edges that participate in at least one match and guarantees no false positives. Given that this is an iterative process, it is natural to investigate at what rate $\mathcal{G}^*$ is refined, and whether there are opportunities to trade between the precision (or existence of precision guarantees) of an intermediary solution, and compute time.

Fig. 13 shows the evolution of the precision of the intermediate solution over time for the various patterns. (We define precision as the ratio between the number of distinct vertices that participate in at least one match, and the size of refined

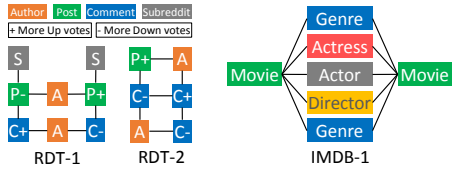
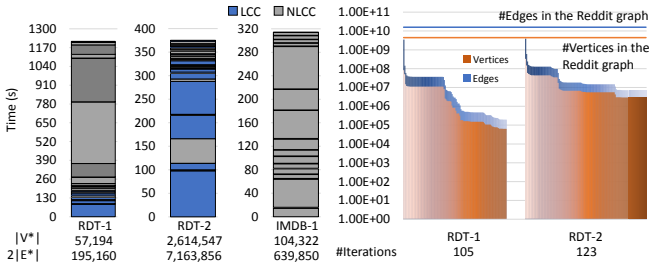
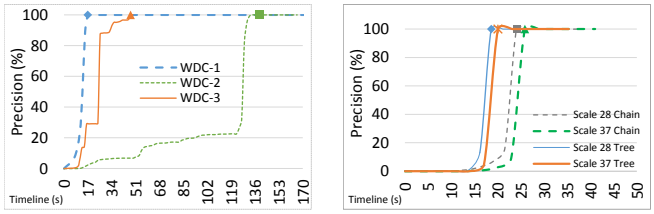


Fig. 11: The scenarios and their corresponding templates for the Reddit (RDT) and IMDB graphs: RDT-1 (left): identify users with adversarial poster/commenter relationship. Each author makes at least two posts or two comments, respectively. Comments to posts, that with more upvotes (P+), have a balance of negative votes (C-) and comments to posts, with more downvotes (P-), have a positive balance (C+). The posts must be under different Subreddits (category). RDT-2 (center): identify all poster/commenter pairs where the commenter makes at least two comments to the same post, one directly to the post and one in response to a comment. The poster also makes a comment in response to a comment. The commenter always receives negative rating (C-) to a popular post (P+), however, comments (to the same post) by the poster has a positive rating (C+). IMDB-1 (right): find all the actresses, actors, and directors that worked together at least on two different movies that fall under at least two similar genres.



(a) (b)

Fig. 12: (a) Runtime for the graph analytics patterns presented in Fig. 11. The labels on X-axis represent the number of vertices and edges in the pruned graph. Note that Y-axes have different scales. (b) Number of active vertices and edges after each iteration for the same experiments for Reddit as in (a). The labels on bottom row of X-axis represent the number of iterations required. Note that the Y-axis is on log scale.



(a) WDC (b) R-MAT

Fig. 13: Vertex set precision over the lifetime of an execution for (a) WDC (Fig. 7) and (b) R-MAT (Fig. 5) patterns. The X-axis presents the timeline while the Y-axis is the precision achieved by the end of an iteration. The markers indicate the moment in time when 100% precision has been achieved. The presented timeline for the WDC-1 is limited to 170th second for better visibility (WDC-1 achieves 100% precision in less than 20 seconds). For the R-MAT patterns, we only show numbers for Scale 28 and 37.

vertex set $\mathcal{V}^*$ at the end of an iteration.) We note that: *(i)* the rate at which precision improves is pattern dependent, and *(ii)* for some patterns, even after precision reaches 100% and no more vertices are pruned, the algorithm continues to operate to offer guarantees that no false positives are left. For example, in Fig. 13(a), WDC-1 quickly reaches 100% precision, however, 99% of the execution time is spent to verify five non-local constraints (top left WDC-1 plot in Fig. 8) to guarantee no false positive match. WDC-3, however, shows different behaviour: the complex structure does not reach 90% precision until the very end, and converges to 100% quickly afterwards. We believe that the rate at which precision is achieved, is partly influenced by the order in which constraints are verified and our heuristics for constraint verification ordering leave room for improvement.

Please continue to Appendix for the remaining of the paper: evaluation (§A), a brief discussion on the potential limitations of this work (§B), related work (§C), and conclusion (§D).

Acknowledgement

This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 (LLNL-CONF-735657). Experiments were performed at the Livermore Computing facility.

References

[1] W. Fan, X. Wang, and Y. Wu, "Diversified top-k graph pattern matching," *Proc. VLDB Endow.*, vol. 6, no. 13, pp. 1510–1521, Aug. 2013. [Online]. Available: http://dx.doi.org/10.14778/2536258.2536263

[2] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp, "Biomolecular network motif counting and discovery by color coding," *Bioinformatics*, vol. 24, no. 13, pp. i241–i249, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btn163

[3] Z. Zeng, J. Wang, L. Zhou, and G. Karypis, "Out-of-core coherent closed quasi-clique mining from large dense graph databases," *ACM Trans. Database Syst.*, vol. 32, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1242524.1242530

[4] D. CONTE, P. FOGGIA, C. SANSONE, and M. VENTO, "Thirty years of graph matching in pattern recognition," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 18, no. 03, pp. 265–298, 2004. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0218001404003228

[5] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1976. [Online]. Available: http://doi.acm.org/10.1145/321921.321925

[6] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and algorithms for graph queries on multithreaded architectures," in *2007 IEEE International Parallel and Distributed Processing Symposium*, March 2007, pp. 1–14.

[7] 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, pp. 264–275, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920878

[8] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad, "Fast best-effort pattern matching in large attributed graphs," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '07. New York, NY, USA: ACM, 2007, pp. 737–746. [Online]. Available: http://doi.acm.org/10.1145/1281192.1281271

[9] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1804–1815, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824077

[10] S. A. Myers, A. Sharma, P. Gupta, and J. Lin, "Information network or social network?: The structure of the twitter follow graph," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14 Companion. New York, NY, USA: ACM, 2014, pp. 493–498. [Online]. Available: http://doi.acm.org/10.1145/2567948.2576939

[11] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 10, pp. 1367–1372, Oct. 2004. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2004.75

[12] B. D. Mckay and A. Piperno, "Practical graph isomorphism, ii," *J. Symb. Comput.*, vol. 60, pp. 94–112, Jan. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jsc.2013.09.003

[13] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: A system for distributed graph mining," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 425–440. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815410

[14] N. P. Roth, V. Trigonakis, S. Hong, H. Chafi, A. Potter, B. Motik, and I. Horrocks, "Pgx.d/async: A scalable distributed graph pattern matching engine," in *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, ser. GRADES'17. New York, NY, USA: ACM, 2017, pp. 7:1–7:6. [Online]. Available: http://doi.acm.org/10.1145/3078447.3078454

[15] T. Reza, C. Klymko, M. Ripeanu, G. Sanders, and R. Pearce, "Towards practical and robust labeled pattern matching in trillion-edge graphs," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017, pp. 1–12.

[16] A. Lulli, E. Carlini, P. Dazzi, C. Lucchese, and L. Ricci, "Fast connected components computation in large graphs by vertex pruning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 760–773, March 2017.

[17] F. Zhou, S. Mahler, and H. Toivonen, *Simplification of Networks by Edge Pruning*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 179–198. [Online]. Available: https://doi.org/10.1007/978-3-642-31830-6_13

[18] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu, "Efficient processing of large graphs via input reduction," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 245–257. [Online]. Available: http://doi.acm.org/10.1145/2907294.2907312

[19] "Giraph," 2016. [Online]. Available: http://giraph.apache.org/

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 17–30. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387883

[21] R. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 549–559. [Online]. Available: https://doi.org/10.1109/SC.2014.50

[22] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the Fourth SIAM Int. Conf. on Data Mining*. Society for Industrial Mathematics, 2004, p. p. 442.

[23] "Havoqgt," 2016. [Online]. Available: http://software.llnl.gov/havoqgt/

[24] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling techniques for massive scale-free graphs in distributed (external) memory," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 825–836. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2013.72

[25] M. P. Wellman and W. E. Walsh, "Distributed quiescence detection in multiagent negotiation," in *Proceedings Fourth International Conference on MultiAgent Systems*, 2000, pp. 317–324.

[26] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 18:1–18:11. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654078

[27] O. L. Robert Meusel, Christian Bizer, "Web data commons - hyperlink graphs," 2016. [Online]. Available: http://webdatacommons.org/hyperlinkgraph/index.html

[28] M. Serafini, G. De Francisci Morales, and G. Siganos, "Qfrag: Distributed graph search via subgraph isomorphism," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 214–228. [Online]. Available: http://doi.acm.org/10.1145/3127479.3131625

[29] "Quartz," 2017. [Online]. Available: https://hpc.llnl.gov/hardware/platforms/Quartz

[30] "Reddit public data," 2016. [Online]. Available: https://github.com/dewarim/reddit-data-tools

[31] "Imdb public data," 2016. [Online]. Available: http://www.imdb.com/interfaces

[32] "Graph 500 benchmark," 2016. [Online]. Available: http://www.graph500.org/

[33] T. Plantenga, "Inexact subgraph isomorphism in mapreduce," *J. Parallel Distrib. Comput.*, vol. 73, no. 2, pp. 164–175, Feb. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2012.10.005

[34] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. S. A. Kumar, and M. V. Marathe, "Sahad: Subgraph analysis in massive networks using hadoop," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, May 2012, pp. 390–401.

[35] G. M. Slota and K. Madduri, "Complex network analysis using parallel approximate motif counting," in *Proc. 28th IEEE Int'l. Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2014, pp. 405–414.

[36] V. T. Chakaravarthy, M. Kapralov, P. Murali, F. Petrini, X. Que, Y. Sabharwal, and B. Schieber, "Subgraph counting: Color coding beyond trees," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 2–11.

[37] J. Gao, C. Zhou, J. Zhou, and J. X. Yu, "Continuous pattern detection over billion-edge graph using distributed framework," in *2014 IEEE 30th International Conference on Data Engineering*, March 2014, pp. 556–567.

[38] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 788–799, May 2012. [Online]. Available: http://dx.doi.org/10.14778/2311906.2311907

[39] S. Ma, Y. Cao, J. Huai, and T. Wo, "Distributed graph pattern matching," in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12. New York, NY, USA: ACM, 2012, pp. 949–958. [Online]. Available: http://doi.acm.org/10.1145/2187836.2187963

[40] 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 *2013 IEEE International Conference on Big Data*, Oct 2013, pp. 403–411.

[41] W.-S. Han, J. Lee, and J.-H. Lee, "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*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 337–348. [Online]. Available: http://doi.acm.org/10.1145/2463676.2465300

[42] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "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*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/2588555.2610511

[43] "Largetriplestores." [Online]. Available: https://www.w3.org/wiki/LargeTripleStores

[44] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia, "Graphframes: An integrated api for mixing graph and relational queries," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '16. New York, NY, USA: ACM, 2016, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2960414.2960416

[45] "Graphframes." [Online]. Available: http://graphframes.github.io/

[46] 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*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[47] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 599–613. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685096

[48] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, Jul. 2015. [Online]. Available: http://dx.doi.org/10.14778/2809974.2809983

[49] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi, "Pgx.d: A fast distributed graph processing engine," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New

York, NY, USA: ACM, 2015, pp. 58:1–58:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807620

[50] F. Zhu, Q. Qu, D. Lo, X. Yan, J. Han, and P. Yu, "Mining top-k large structural patterns in a massive network," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 807–818, 8 2011.

[51] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: An efficient algorithm for testing subgraph isomorphism," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 364–375, Aug. 2008. [Online]. Available: http://dx.doi.org/10.14778/1453856.1453899

[52] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang, "Fast graph pattern matching," in *2008 IEEE 24th International Conference on Data Engineering*, April 2008, pp. 913–922.

[53] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, "Computing simulations on finite and infinite graphs," in *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, ser. FOCS '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 453–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795662.796255

[54] G. Liu, K. Zheng, Y. Wang, M. A. Orgun, A. Liu, L. Zhao, and X. Zhou, "Multi-constrained graph pattern matching in large-scale contextual social graphs," in *2015 IEEE 31st International Conference on Data Engineering*, April 2015, pp. 351–362.

APPENDICES

A EVALUATION (CONTINUED)

A. Impact of Design Decisions and Strategic Optimizations

Edge Elimination. Fig. 8 (right plots) show scalability and performance gains as a result of enabling edge elimination. Without edge elimination, the WDC-3 pattern results in 3,180,678 edges selected (some are false positives). Edge elimination identifies the true positive matches and reduces the number of active edges to 255,022. In other words, the graph is 12.5× sparser which in turn improves overall message efficiency of the system.

Asynchronous Communication. Our system is designed to harness the advantages of an asynchronous graph processing framework, yet a synchronous one could easily support the same algorithms. Fig. 14(a) shows runtime for two patterns that benefit the most from asynchronicity: for WDC-1 and RDT-1 (2.65× and 3.5× gains, respectively) compared with a synchronous version that adds a barrier after each NLCC token propagation step. Asynchronous NLCC makes it possible for all walks to progress independently without synchronization overheads. Synchronous NLCC is implemented within HavoqGT as well.

Work Aggregation. Fig. 14(b) shows the performance gains enabled by the work aggregation strategy employed by NLCC (presented in §III and Alg. 6). The magnitude of the gain: 10-50%, is data dependent and more pronounced when the pattern is abundant, e.g., WDC-1 has 600M+ instances (Table II).

Constraint Selection. For patterns for which full TDS is required for precision guarantees, the path and cycle constraints are there for performance optimizations (their goal is to prune away the non-matching part of the graph early) and we are interested to evaluate the impact of these optimizations. To this end, we compare time-to-solution for a configuration that generates and uses all NLCC constraints, and one that uses only the TDS constraints required to guarantee 100% precision. Our experiments show that, although it increases the number of iterations, verifying simpler and smaller substructures first is extremely effective: for some patterns (e.g., RDT-1) the system is not able to complete in reasonable time without these constraints, for others (e.g., WDC-2) these constraints enable a 2.39× speedup (Fig. 14(c)).

B. Load Balancing

Load imbalance issues are inherent to problems involving irregular data structures, such as graphs, when these need to be partitioned for processing over multiple nodes. For our pattern matching solution, load imbalance can be further caused by two artifacts: first, over the course of execution, our solution causes the workload to mutate (i.e., we prune away vertices and edges), and, second, by nonuniform distribution of matches in the background graph: the vertices and edges that participate in the matches may reside on a small, potentially concentrated, portion of the graph. Load imbalance can indeed occur: for example, for the relatively rare WDC-2 pattern when using 128 nodes, for example, the vertices that participate in

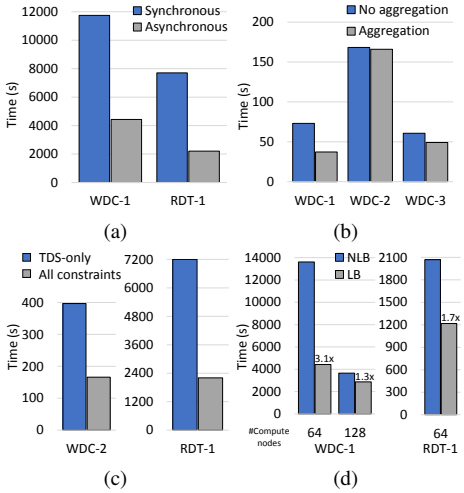


Fig. 14: (a) Comparing synchronous and asynchronous NLCC. (b) Impact of work aggregation on runtime for the WDC patterns (for the sake of readability, only a subset of non-local constraints are considered for WDC-1). (c) Runtime performance when only TDS constraints are used vs. all NLCC constraints used. (Here, RDT-1 does not finish after two hours). Note that in (a) and (c), we did not apply load balancing to RDT-1. All experiments in (a), (b) and (c) use 64 compute nodes. (d) Impact of load balancing on runtime for the the WDC-1 and RDT-1 patterns. We compare two cases: without load balancing (NLB) and with load balancing (LB). For WDC-1, we show results for two scales, on 64 and 128 nodes. Speedup achieved by LB over NLB is also shown on the top of each bar.

the final selection are distributed over as few as 114 partitions out of 4,608. The distribution is concentrated: 90% of the matching edges are on 85 partitions while more than half of of the matching edges are located on only 15 partitions. For the more frequent WDC-1 pattern, 99% of the matching vertices are part of a single connected component. 50% of the matching edges are on less than 5% of the total partitions on a 64 node deployment, which becomes less than 3% on a 128 node deployment.

We employ a pseudo-dynamic, load balancing strategy. First, we checkpoint the current state of execution: the pruned graph, i.e., the set of active vertices and edges and the per-vertex state indicating template matches, $\omega(v_j)$ in Alg. 2. Next, using HavoqGTs graph partitioning module, we reshuffle vertex-to-processor assignment to evenly distribute vertices and edges across processing cores. Processing is then resumed on the rebalanced workload. (Note that, depending on the size the the pruned graph, it is possible to resumed processing on a smaller deployment.) Over the course of the execution, checkpointing and rebalancing can be repeated as needed.

As a proof of feasibility, to examine the impact of this technique, we analyze the runs for WDC-1 and RDT-1 patterns. (We chose some of the real-world workloads as they are more likely to lead to imbalance than synthetically generated load.) Fig. 14(d) compares performance of the pruning algorithms with and without load balancing. For these examples, we perform workload rebalancing only once: for WDC-1, before verifying the TDS constraints, and for RDT-1, when the pruned graph is four orders of magnitude smaller. The extent of load imbalance is more severe for WDC-1 on the smaller 64 node deployment compared to using 128 nodes; workload

rebalncing improves runtime by 3.1× and 1.3× on 64 and 128 nodes, respectively. In the case of RDT-1, the gain in runtime as a results of load balancing is 1.7×. Given the pruned graphs are orders magnitude smaller then the original graph, check pointing, rebalancing, and relaunching the computation takes less than two minutes, which is negligible compared to the gain achieved in time-to-solution. In Table II, we run enumeration for WDC-1, WDC-2 and RDT-1 on the rebalanced graphs.

TABLE III: Performance comparison between QFrag and our pattern matching system. The table shows runtime in seconds for full enumeration for QFrag; pruning and full enumeration for our distributed system (labeled PruneJuice-distributed), and for a single node implementation of our graph pruning-based approach tailored for a shared memory system (labeled PruneJuice-shared). For PruneJuice, we split time-to-solution into pruning (top row) and enumeration (bottom row) times. We use the same graphs (Patent and YourTube) and some of the query templates (Q4 – Q7) used for evaluation of QFrag in [28].

	QFrag		PruneJuice-distributed		PruneJuice-shared	
	Patent	Youtube	Patent	Youtube	Patent	Youtube
Q4	4.19	8.08	0.238	0.704	0.100	0.400
			0.223	1.143	0.010	0.010
Q6	5.99	10.26	0.874	2.340	0.070	1.730
			0.065	0.301	0.005	0.010
Q7	6.36	11.89	0.596	1.613	0.130	0.820
			0.039	0.180	0.005	0.010
Q8	10.05	14.48	0.959	2.633	0.100	1.370
			0.049	0.738	0.001	0.010

C. Comparison with Existing Work: QFrag

We empirically compare our work with a recent work (2017) on pattern matching from the database community: QFrag [28].

Similar to our solution, QFrag targets exact pattern matching, yet there are two main differences: QFrag assumes that the entire graph fits in the memory of each compute node and uses data replication to enable search parallelism. More importantly, QFrag employs a sophisticated load balancing strategy between parallel instances of a search to achieve good scalability. QFrag is implemented on top of Apache Spark and Giraph [19]. In QFrag, each replica runs an instance of a pattern enumeration algorithms called $\text{Turbo}_{\text{ISO}}$ [41]. Through evaluation, the authors demonstrated QFrag's performance advantages over two other distributed pattern matching systems: *(i)* TriAD [42], an MPI-based distributed RDF [43] engine based on an asynchronous distributed join algorithm, and *(ii)* GraphFrames [44], [45], a graph processing library for Apache Spark, also based on distributed join operations.

Given that we have demonstrated the good scalability of our solution (Serafini et al. [28] demonstrate equally good scalability properties for QFrag yet on much smaller graphs), we are interested to establish a comparison baseline at single node scale. To this end, we run experiments on a modern shared memory machine with 60 CPU-cores, and use four template queries and two real-world graphs (Patent and Youtube) that were used for evaluation of QFrag [28]. We run QFrag with 60 threads and HavoqGT with 60 MPI processes. The results are summarized in Table III: QFrag runtimes for match enumeration (first pair of columns) are comparable with the results presented in [28], so we have reasonable confidence that we replicate their experiments well. With respect to combined pruning and enumeration time, our system (second pair of columns, presenting pruning and enumeration time separately) is consistently faster than QFrag on all the graphs, for all the queries. We note that our solution does not take advantage of shared memory of the machine at the algorithmic or implementation level (we use different processes, one MPI process per core), and has the system overhead of MPI-communication between processes. Additionally, unlike QFrag, our system is not handicapped by the memory limit of a single machine as it supports graph partitioning. To highlight the effectiveness of our technique and get some intuition on the magnitude of the MPI overheads in this context, we implemented our technique for shared memory and present runtimes (when using 60 threads) for the same set of experiments in Table III (right two columns). We notice up to an order of magnitude better performance, the main cost for our technique, compared to the distributed implementation running on a single node.

In summary, our system works about 4–10× faster than QFrag, and, if excluding distributed system memory overheads and considering the pruning time for the shared memory solution and conservatively reusing enumeration runtime for the distributed solution, it is about 6–100× faster than QFrag.

B Limitations / Discussion

We categorize the limitations of our proposed system based on their respective sources.

Limitations stemming from major design decisions. Our system inherits limitations of a system that performs exact matching (compared to a system that focuses on approximate matching, e.g., based on graph simulation [7]). Similarly, our system inherits all limitations of its communication and middleware infrastructure, MPI and HavoqGT. One example is the lack of flow control in these infrastructures which sometimes leads to message buildup and system collapse.

Limitations stemming from the targeted uses cases. In the same vein, we note that our system targets a graph analytics scenario (queries that need to cover the entire graph), rather than the traditional graph database queries that attempt to find a specific pattern around a vertex indicated by the user (where other systems may perform better).

Limitations stemming from attempting to design a generic system. Systems optimized for specific patterns may perform better (e.g., systems optimized to enumerate triangles or treelets [34] or systems relying on multi-join indices to support patterns with limited diameter).

Limitations stemming from incomplete understanding and work in progress. While we propose heuristics that appear to work well for our experiments, one of the key challenges is identifying an optimal set of constraints and their execution order. We believe graph statistics at different stages in execution can be used to dynamically select effective constraints.

C Related Work

The volume of related work on graph processing in general [46], [19], [20], [47], [48], [49] and on pattern match-

TABLE IV: Comparison of past work on distributed pattern matching. The table highlights the characteristics of the solution presented (exact vs. approximate matching), its implementation infrastructure, and summarizes the details of the largest-scale experiment performed. We highlight the fact that ***our solution is unique in terms of demonstrated scale, ability to perform exact matching, and ability to retrieve all matches***.

Contribution	Model	Framework/ Language	Match Type	Max-Query Size	Metadata	#Compute Nodes	Max-Real Graph	Max-Synthetic Graph
Plantenga [33]	Graph Walk	Hadoop	Approx.	4-cliques	Real	64	107B edges	R-MAT Scale 20
QFrag [28]	Tree-based	Spark	Exact	7 edges	Real	10	117M edges	N/A
SAHAD [34]	Explore-Join	Hadoop	Approx.	12 vertices	Synthetic	40	N/A	269M edges
FASICA [35]	Explore-Join	MPI	Approx.	12 vertices	N/A	15	117M edges	Erdős-Renyi 1M edges
Chakaravarthy et.al. [36]	Explore-Join	MPI	Approx.	10 vertices	N/A	BG/Q-512	2.7M edges	R-MAT
PGX.D/Async [14]	Async. DFS	Java/C++	Exact	4 edges	Synthetic	32	N/A	2B edges (Unif. rand.)
Gao et al. [37]	Explore-Join	Giraph	Approx.	50 vertices	Synthetic	28	3.7B edges	N/A
Sun et al. [38]	Explore-Join	C#.Net4	Exact	15 vertices	Synthetic	12	16.5M edges	4B vertices
Ma et al. [39]	Graph Simulation	Python	Approx.	15 vertices	Type only	16	5.1M edges	100M vertices
Fard et al. [40]	Graph Simulation	GPS	Approx.	N/A	N/A	8	300M edges	N/A

ing algorithms in particular [5], [11], [12], [50], [6], [1] is humbling. We summarize closely related work in Table IV.

Sequential Algorithms. Early work on graph pattern matching mainly focused on solving the problem of graph isomorphism [5]. The well-known Ullmann's algorithm [5] and its extensions (in terms of join order and pruning strategies), e.g., VF2 [11] and QuickSI [51], belong to the family of tree-search based algorithms. A recent effort, $\text{Turbo}_{\text{ISO}}$ [41] is considered to be the state-of-the-art of tree-search based sequential subgraph isomorphism algorithm. For large graphs, a tree search that fails mid-way and has to backtrack, can be expensive. Efficient distributed implementation of this approach is difficult due to the costs associated with maintaining large intermediate search state across multiple physical nodes that participate in the search. Perhaps the best known exact matching algorithm that does not belong to the family of tree-search based algorithms is Nauty due to McKay [12], which is based on *canonical labeling* of the background graph. This approach, however, has high preprocessing overhead.

Subgraph Indexing. In the same spirit as database indexing, *indexing of frequent subgraph structures* is an approach attempted by some in order to reduce the number of join operations and lower query response time, e.g., SpiderMine [50] and R-Join [52]. Unfortunately, for a billion-edge graph, this approach is infeasible. First, searching frequent subgraphs in a large graph is expensive. Second, depending on topology of the template(s) and the background graph, the size of the index is often superlinear relative to the size of the graph [38].

Distributed Solutions. Here, we focus on projects that provide distributed pattern matching and demonstrate it at some scale. Table IV summarizes the key differentiating aspects and the scale achieved. The best scale is offered by Plantenga's [33] MapReduce implementation of the *walk-based* algorithm for inexact matching, originally proposed in [6]. Unlike ours, Plantenga's system can find exact/approximate matches only for a restricted class of small patterns. Plantega demonstrated performance using a 107 billion edge graph, the largest-scale experiment to date (excluding ours). SAHAD [34], is a MapReduce implementation of the *color-coding* algorithm originally developed for finding tree-like patterns (treelet) in protein-protein interaction networks. SAHAD follows a hierarchical sub-template *explore-join* approach. Its application was presented only on small graphs with up to ∼300M edges. FASICA [35] is also a color-coding-based system for approximate treelet counting, whose MPI-based implementation offers superior performance to SAHAD. Chakaravarthy et al. [36] extended the color-coding algorithm to count patterns with cycles and presented a MPI-based distributed implementation. However, they demonstrated performance on graphs with only a few million edges. Although QFrag outperforms many of its competitors in terms of time-to-solution, it replicates the entire graph in the memory of each node which limits its applicability to small graphs only. PGX.D/Async [14] relies on asynchronous depth-first traversal and incorporates flow control with a deterministic guarantee of search completion under a finite amount of memory. Both QFrag and PGX.D/Async are demonstrated at a much smaller scale (in terms of graph sizes and number of compute nodes) than in this paper. Sun et al. [38] present an exact subgraph matching algorithm which follows the explore-join approach and demonstrate it on large synthetic graphs and larger query graphs than in [33].

Approximate Matching. Recently, a new family of approximate matching algorithms based on graph simulation [53] has been proposed [1], [40], [54]. As opposed to exact matching, graph simulation algorithms relax matching constraints, e.g., matching based on vertex attributes and their connectivity constraints in the query [7]. Simulation based algorithms have quadratic/cubic time-complexity and have been proposed as a possible solution for emerging matching problems when large-scale graphs are involved [53]. Two inexact matching algorithms based on graph simulation are introduced in [40], [39], although results from both are only presented on relatively small real-world graphs. Gao et al. introduce another approximate matching algorithm based on explore-join [37] and evaluate it on even larger query patterns than in [38]. Here, a query pattern is converted into a single-sink Directed Acyclic Graph (DAG) and message transition follows its topology.

D Conclusion

This paper presents a new algorithmic pipeline to support pattern matching on large-scale metadata graphs on large

distributed-memory machines. We capitalize on the idea of *graph pruning* and develop asynchronous algorithms that use both vertex and edge elimination to iteratively prune the original graph and reduce it to a subgraph which represents the union of all matches. We have developed pruning techniques that *guarantee a solution with* 100% *precision* (i.e., no false positives in the final pruned graph) and 100% *recall* (i.e., all vertices and edges participating in matches are included) for *arbitrary search patterns*, including patterns with repeated vertex metadata, and patterns that have arbitrary cycles. Our algorithms are *vertex-centric* and *asynchronous*, thus, they map well onto existing high-performance graph frameworks. Our evaluation using up to 257 billion edge real-world web-graphs and up to 4.4 trillion edge synthetic R-MAT graphs, on up to 1,024 nodes (36,864 cores), confirms the scalability of our solution. We demonstrate that, depending on the search template, our approach prunes the graph by orders of magnitude which enables full pattern enumeration and counting on graphs with trillions of edges. Our success stems from a number of key design ingredients: asynchronicity, aggressive vertex and edge elimination while harnessing massive parallelism, intelligent work aggregation to ensure low message overhead, effective pruning constraints, and lightweight per-vertex state.

E Artifact Description

Here, we present an example of searching a pattern in a R-MAT generated graph using our program. The code is developed on top of HavoqGT. Detailed instructions on how to use our tool are available at https://github.com/LLNL/havoqgt

Clone the code from https://github.com/LLNL/havoqgt.git

You will require the latest releases of OpenMPI or MAVPICH2 and the Boost library to run HavoqGT. The code has only been tested on latest generation of Linux distributions. Once you have checked out the code, make sure you are on the master branch.

Go to the directory, build/quartz/ Setup CMake environment by running the following script:

./scripts/quartz.llnl.gov/do_cmake.sh

(Make necessary adjustments to the script for CMake to work within your environment.)

The next step is to generate a graph in HavoqGT format: Go to the directory, build/quartz/ and build the R-MAT generator:

make generate_rmat

Create a directory, e.g., /usr/graph/, to store the generated graph.

Assuming you are in a Slurm environment, run the following command to generate a R-MAT graph:

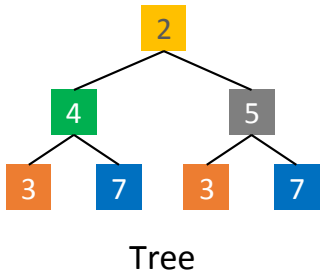
srun -N1 –ntasks-per-node=4 –distribution=block ./src/generate_rmat -s 18 -p 1 -f 1 -o /dev/shm/rmat -b /urs/graph/rmat

This will create a graph with four partitions, to be run using four MPI processes. Note that this is a Scale 18 graph. (Notice the parameter for the -s flag.) The mmap/binary graph file will be store in /usr/graph/

Next, we build the pattern matching executable:

make run_pattern_matching_beta

We will search the following Tree pattern on the graph we just created. The numeric values on each vertex is the label of the respective vertex.



Tree

We use degree information to create numeric vertex labels, computed using the formula. $\lceil \log_2(d(v_i)+1) \rceil$. Here, $d(v_i)$ is the degree of a vertex v_i.

The input pattern is available at https://github.com/LLNL/havoqgt (See the instructions on the readme page regarding how to download the sample input pattern.)

See the instructions on the readme page regarding how create the output directory before you run the program. (https://github.com/LLNL/havoqgt)

Once you have the input pattern, e.g., /usr/pattern/ and output, e.g., /usr/results directories setup, use the following command to search the pattern stored in /usr/pattern/.

Note that we do not need to provide vertex labels for the Tree pattern as we will use labels based on vertex degree and the program will generate labels when no input label is provided, i.e., the -l flag is not set. (The readme page has links to other datasets that require input labels.)

srun -N1 –ntasks-per-node=4 –distribution=block ./src/run_pattern_matching_beta -i /dev/shm/rmat -b /usr/graph/rmat -p /usr/pattern/ -o /usr/results

The program logs status information to the standard output so you know the current state of the execution.

See the instructions on the readme page regarding how to interpret the results and retrieve the pruned graph. You will find scripts (written in python) that will help you to parse the result files. (https://github.com/LLNL/havoqgt)

Also, instructions on how to enumerate a pattern on the pruned graph are available on the readme page. (https://github.com/LLNL/havoqgt)

We encourage you to try the Tree pattern on a Scale 28 or larger R-MAT graph and compare the results you obtain with the numbers we have reported in Section V-A.

The instructions page also explains how to partition a graph (from a given edge list) for distributed processing, provide required vertex labels and use the scripts to generate non-local constraints for a given pattern.