Towards Practical and Robust Labeled Pattern Matching in Trillion-Edge Graphs

Tahsin Reza[1,2], Matei Ripeanu[2], Christine Klymko[1], 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}@ece.ubc.ca {klymko1, sanders29, rpearce}@llnl.gov

***Abstract*—Subgraph pattern matching is fundamental to graph analytics and has wide applications. Unfortunately, high computational complexity limits the robustness guarantees of existing algorithms: they do not scale for modern large graph datasets and/or they have limitations in terms of accuracy or in terms of the intricacy of the patterns supported. We present algorithms, theory, and empirical evidence that iteratively eliminating vertices that do not meet local constraints dramatically reduces the search space for pattern matching in real-world graphs, and demonstrate a scalable implementation of our algorithms. We additionally identify the characteristics of patterns for which every non-eliminated vertex participates in a match. These techniques are an essential step to enable scalable, practical solutions for robust pattern matching in large-scale labeled graphs. We demonstrate the advantages of the proposed approach through strong and weak scaling experiments on massive-scale real-world (up to 257 billion edges) and synthetic (up to 2.2 trillion edges) graphs and at scales (256 compute nodes with 6,144 processors) orders of magnitude larger than those used in the past for similar problems.**

I. Introduction

Graph pattern matching, that is, finding subgraphs that match a small *template graph* within a massive *background graph*, is fundamental to graph analysis and has applications in multiple areas such as social networks [1], bioinformatics [2], and information mining [3]. A 'match' can be defined in multiple ways and variants of this problem include *exact* and *approximate matching* [4]. Exact matching is related to *subgraph isomorphism*, a problem not known to have a polynomial time solution [5].

As a refinement of exact matching problem, Berry et al. [6] introduced *type-isomorphism* in *semantic graphs*, that is, graphs where vertices and edges are labeled and a match identifies nodes or edges with the same label and adjacency structure in the template and the background graph. For real-world use cases: small template graphs (i.e., up to hundreds of vertices) yet, large real-world background graphs (i.e., graphs with billions of vertices, sparse, power-law degree distribution, and relatively uniform label distribution), algorithms based on heuristics are often able to determine whether a small template exists. Recently, several authors [1], [4], [7] have explored solutions in this space and demonstrated that label-based matching is often practical and sufficient for real-world applications such as social network analysis.

Background. Today, most methods for exact or approximate matching follow foundational work based on the *search-and-join* approach introduced by Ullman [5] (see §II for related work). A fundamental limitation of this class of methods is that the number of possible join operations is combinatorially large, making its application to generic patterns and massive graphs, with billions or trillions of edges, impractical. Moreover, these methods are difficult to parallelize and are not practical to implement on top of *vertex-centric* frameworks (e.g., Giraph [8], GraphLab [9], HavoqGT [10]), commonly adopted by today's large-scale, distributed memory machines.

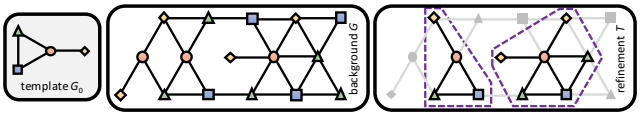

Fig. 1. A template graph $\mathcal{G}_0$ (left), a background graph $\mathcal{G}$ (center), and the output of vertex elimination $\mathcal{T}$ (right), a refined set of vertices that meet local constraints imposed by $\mathcal{G}_0$. The figures present vertex type as colored shapes. Connected components (purple outlines) in the active subgraph may contain multiple exact matches of $\mathcal{G}_0$ (one match and four matches, for this example).

Design Objectives. We aim to enable pattern matching on *semantic graphs*, whose vertices and edges are associated with a predefined *type* and/or *label* [6]. We aim for a solution that is highly-parallel, scalable, and caters to contemporary networked-data driven applications [4], [11]. A key design constraint for our solution is to leverage existing general-purpose graph processing frameworks targeting large distributed memory machines as they provide primitives with demonstrated flexibility to support a wide range of graph processing algorithms, have demonstrated good scalability when operating over a large number of nodes, and can accommodate massive graphs. As most of these frameworks are vertex-centric, our focus is on algorithmic solutions that have a natural vertex-centric description.

Approach. Figure 1 presents the algorithmic pipeline we have in mind. Our intuition is that, compared to discovering each instance of the pattern by exploring the entire graph using a tree-based *search-and-join* technique, it is typically cheaper to iteratively eliminate vertices and edges that do not meet the local constraints of the query pattern. Thus, we focus on identifying the set of vertices, $\mathcal{T}$, that participate in matches *by aggressively eliminating the copious numbers of vertices that cannot participate in a match*. Should a user be interested in identifying each individual match in the background graph, or in computing the total number of matches, the existing high-complexity algorithms can be used to operate over $\mathcal{T}$ at a

lower cost, as this is generally much smaller than the set of vertices in the background graph.

This paper focuses on evaluating the feasibility and effectiveness of this mechanism over two directions. On the one side we are interested in understanding how far vertex pruning can go. To this end, we focus on an exact matching scenario targeting *non-induced* subgraphs (although we believe our technique can be generalized to various forms of approximate matching), design algorithms for aggressive pruning, and prove that for some template subclasses (acyclic or edge-monocyclic and with unique vertex labels), the resulting pruned list is a complete enumeration of all the vertices that participate in a match (see §VII). For more general templates, the pruned list is a superset. On the other side, we focus on the scalability of these algorithms on distributed memory machines and demonstrate ability to process massive-scale graphs of over 2.2 trillion edges on a large number of nodes in under two minutes, a set of properties that make this approach an appealing candidate for including in a human-driven pipeline for graph mining.

The algorithms we propose iterate over two phases. The first phase uses local information propagated from neighbors at a one-hop distance to eliminate a vertex when it does not meet the constraints specified by the pattern (see §IV-A). For example, for the pattern in Fig. 1, in order to avoid elimination, each vertex with circle shape must have at least three neighbors of appropriate types who also have survived elimination. This iterative elimination process has roots in the well-known *label propagation* family of algorithms [12], has per-vertex communication requirements similar to the PageRank [13] algorithm and gracefully fits within the vertex-centric model. In the second phase, the algorithms use a version of *token passing* [14] to probe for the existence of a required cycle (see §IV-B).

Contributions. This paper:

(i) Presents a highly parallelizable, iterative vertex elimination technique that supports graph pattern matching (§IV). We provide a theoretical correctness proof (§VII) and define the scenarios in which our algorithm generates exact solution: for acyclic or edge-monocyclic template graphs (and independently from background graph size and topology).

(ii) Implements the proposed technique on top of HavoqGT [10], a high-performance vertex-centric graph processing framework, thus enabling asynchronous processing and balanced scale-free graph partitioning, as detailed in §V. The implementation operates on the framework's native graph data structure and does not require any expensive preprocessing (e.g., substructure indexing as [15]).

(iii) Demonstrates the applicability of this solution using real-world and synthetic datasets orders of magnitude larger than prior work (§VI). We evaluate scalability through two experiments: first, a strong scaling experiment using real datasets, including the largest known webgraph whose undirected version has over 257 billion edges; secondly, a weak scaling experiment using synthetic, R-MAT generated graphs of up to 2.2 trillion edges, on up to 256 compute nodes (6,144 processors). Finally, we demonstrate support for query patterns representative of practical queries and consider queries containing high frequency vertex labels, from ∼150 thousands up to ∼9.5 billions.

The advantage of the graph-pruning approach over the common tree-search method is highlighted when comparing with past work: Plantenga et al. [16], the largest scale experiment to date, report search time in order of hours in a billion-edge graph on 64 compute nodes. Our implementation can identify all instances of comparable patterns on a graph with twice the edge count, on the same number of compute modes, in under a minute.

TABLE I
SYMBOLIC NOTATION USED.

Object(s)	Notation
label set	$\mathcal{L} = \{0, 1, ..., n_\ell - 1\}$
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	$(q_i, q_j) \in \mathcal{E}_0$
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	$(v_i, v_j) \in \mathcal{E}$
label of a vertex	$\ell(v_i) \in \mathcal{L}$
degree of a vertex	$d(v_i)$
set of vertices adjacent to v_i	$adj(v_i)$
walk in the template, background	$\mathcal{W}_0$, $\mathcal{W}$
cycle in the template, background	$\mathcal{C}_0$, $\mathcal{C}$
vertex match function	$f_k(v_i) \in \mathcal{V}_0$
max number of iterations in local constraint checking	k_{max}

II. PRELIMINARIES

We aim to identify all structures within a large *background* graph, $\mathcal{G}$, that are identical to a small *template* graph, $\mathcal{G}_0$. We describe general graph properties in relation to the background graph $\mathcal{G}$; the same notation (also summarized in Table I for convenience) will be applied to $\mathcal{G}_0$ throughout the paper.

A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a collection of n vertices $\mathcal{V} = \{0, 1, ..., n-1\}$ and m pair-wise relationships between vertices, or edges $(i, j) \in \mathcal{E}$, where $i, j \in \mathcal{V}$. The first entry i is called the edge's *source* and the second j is the *target*. Although we develop techniques that are applicable more generally, here we only discus simple undirected graphs. A simple graph has no self edges ($\forall i \in \mathcal{V}, (i, i) \notin \mathcal{E}$) and no multiple edges ($\mathcal{E}$ is a set of edges). 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* $\mathcal{W}$ of length r in $\mathcal{G}$ is an ordered list of elements from $\mathcal{E}$ where source of each non-initial edge is the target of the previous edge, i.e. $\mathcal{W} = \{(i_0, i_1), (i_1, i_2), ...(i_{r-1}, i_r)\}$. The set of all walks from i to j is written $\{i \rightsquigarrow j\}$. The *hop*

distance of i to j in $\mathcal{G}$ is $\text{dist}(i,j) = \min_{\mathcal{W} \in \{i \rightsquigarrow j\}} |\mathcal{W}|$ The *hop diameter* of a graph is $\text{diam}(\mathcal{G}) := \max_{i,j \in \mathcal{V}} \text{dist}(i,j)$. A *connected* graph has finite diameter. A walk that has no repeated vertices is a *path* and one with no repeated edges is a *trail*. An r-length path with $i_0 = i_r$ is a *cycle*, $\mathcal{C}$. An *acyclic* graph has no cycles.

We further characterize graphs with with cycles. Two *disjoint* cycles have no edge in common. Two *distinct* cycles have at least one edge not in common. We define the *cycle degree* of edge $(i,j) \in \mathcal{E}$ as the number of distinct cycles (i,j) is in, written $\delta_{(i,j)}$. The maximum cycle degree is $\delta_{max} := \max_{\mathcal{E}} \delta_{(i,j)}$. A graph is *edge-monocyclic* if $\delta_{max} = 1$.

Throughout this paper, we will discuss two graph objects simultaneously, the *template graph* $\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$ that is relatively small ($n_0 := |\mathcal{V}_0| \leq 100$) and a large *background graph* $\mathcal{G}(\mathcal{V}, \mathcal{E})$ ($m := |\mathcal{E}|$ is generally larger than 10^7 and sometimes as large as 10^{12}). 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 $(q_i, q_j) \in \mathcal{E}_0$. We use subscript-0 for objects in this template graph (e.g. $\mathcal{W}_0$ is a walk in $\mathcal{G}_0$). 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, $\mathcal{G}$. When it is clear from context, we slightly abuse notation to avoid double subscripts, using q_0 or v_5 in place of q_{i_0} or v_{i_5}.

We assume $\mathcal{G}_0$ is connected, because if $\mathcal{G}_0$ has multiple components the matching problem can be easily reduced to solving it for each component individually.

Our goal is to find subsets of vertices $\mathcal{S} \subset \mathcal{V}$ that *exactly match* the template, $\mathcal{G}_0$.

Definition 1. *A vertex set $\mathcal{S} \subset \mathcal{V}$ is an* exact match *of template graph $\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$ (in notation, $\mathcal{S} \sim \mathcal{G}_0$) if there exists a bijective function $\phi : \mathcal{V}_0 \longleftrightarrow \mathcal{S}$ with the properties:*

(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* $(\phi(q_1), \phi(q_2)) \in \mathcal{E}$.

This paper proves that, for templates with unique labels the algorithms we propose have additional properties.

Assumption 1. (UNIQUE LABELS IN TEMPLATE) *For any $q_i, q_j \in \mathcal{V}_0$ such that $q_i \neq q_j$, we assume $\ell(q_i) \neq \ell(q_j)$.*

III. RELATED WORK

Models and Primary Techniques. Early work on graph pattern matching mainly focused on solving the problem of graph isomorphism [5]. The well-known Ullmann's algorithm and its extensions (in terms of join order and pruning strategies), e.g., VF2 [21], belong to the family of tree-search based algorithms. These solutions focus on preserving strict structural isomorphic properties, which have been identified as being unnecessary for many practical applications [4]. For large graphs, a tree search that fails mid-way and has to backtrack and restart 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.

SPARQL queries have been used for subgraph matching in RDF data [22]. A SPARQL query is disassembled into a set of edges and final results are constructed through multi-way joins. SPARQL has less expressive power than general subgraph matching and the space for possible join operations can be huge [19]. Cypher, a declarative graph query language for the open-source graph database Neo4j, borrows expression approaches from SPARQL [23].

Indexing frequent graph structures is an approach adopted by some in order to reduce the number of join operations and lower query response time, e.g., SpiderMine [15]. Unfortunately, for a billion-edge graph, this approach is infeasible. First, finding a large number of frequent substructures in a large graph is costly. Secondly, depending on template graph topology, the growth of the index size may be super-linear relative to the size of the graph [19].

G-Ray is an approximate matching algorithm for finding subgraphs in time linear to the size of the background graph. It leverages random walk with restart [24] to measure the probability of an graph-edge being a match for an edge in the template [7]. In [1], an approximate algorithm for top-k matching with early termination capability and its application in the context of social network analysis was presented.

Recently, based on graph simulation [25], a new family of matching algorithms has been proposed [1], [11] and [26]. As opposed to graph isomorphism, graph simulation algorithms relax matching constraints, e.g., matching based on vertex attributes and their connectivity constraints in the query [4]. Simulation-based algorithms have quadratic time-complexity and have been identified as a possible solution for emerging matching problems when large-scale graphs are involved [25].

Distributed Graph Pattern Matching. A number of recent projects consider the challenging task of pattern matching on large-scale graphs on distributed systems. Table II summarizes the key differentiating aspects and the scale achieved.

Plantenga [16], presents a MapReduce implementation of the *walk-based* algorithm for matching subgraphs, originally proposed in [6]. The implementation accommodates a rich set of constraints and can find both exact and inexact isomorphisms for a variety of small query graphs. This solution, however, requires $O(m_0)$ bulk synchronous phases, which makes its applicability to large templates questionable. SAHAD [17], is a MapReduce implementation of the *color coding* algorithm originally developed for counting non-induced tree substructures in protein-protein interaction networks. SAHAD follows a hierarchical sub-template explore-join approach which only works for tree-like patterns. Its application was presented only on graphs with up to ∼300M edges. Chakaravarthy et. al. [27] extended the color coding algorithm for subgraph counting to support patterns with cycles and presented a distributed implementation. However, they do not demonstrate the performance of their technique at scales that we are interested in. Sun et al. [19], present an inexact subgraph matching algorithm based on an explore-join approach and demonstrate it on larger query graphs than that of [16], yet not on real-world graphs and the inexactness of the algorithm

TABLE II
COMPARISON OF PAST WORK ON DISTRIBUTED PATTERN MATCHING.

Contribution	Model	Framework/ Language	Match Type	Max-Query Size	Metadata	Cluster Size Used	Max-Real Graph	Max-Synthetic Graph
Plantenga [16]	Graph Walk	Hadoop	(In)exact	4-cliques	Real	64	107B edges	R-MAT Scale 20
Zhao et al. [17]	Explore-Join	Hadoop	Approx.	12 vertices	Synthetic	40	N/A	269M edges
Gao et al. [18]	Explore-Join	Giraph	Approx.	50 vertices	Synthetic	28	3.7B edges	N/A
Sun et al. [19]	Explore-Join	C#.Net4	Inexact*	15 vertices	Synthetic	12	16.5M edges	4B nodes
Ma et al. [20]	Graph Simulation	Python	Inexact*	15 vertices	Type only	16	5.1M edges	100M nodes
Fard et al. [11]	Graph Simulation	GPS	Inexact*	N/A	N/A	8	300M edges	N/A

*Here, 'Inexact' indicates the authors did not guarantee their solution yields an exact match.

can lead to large numbers of false positive matches. Gao et al. introduce another approximate matching algorithm based on explore-join [18] and evaluate it on even larger queries than in [19]. Two inexact matching algorithms based on graph simulation are introduced in [11], [20], although results from both are only presented on relatively small real-world graphs.

IV. OUR SOLUTION: ITERATIVE VERTEX ELIMINATION

Our goal is an algorithm that eliminates the vertices that cannot be included in any match $\mathcal{S} \sim \mathcal{G}_0$. This approach is motivated by viewing the template $\mathcal{G}_0$ as specifying a list of implicit constraints that remaining vertices need to meet. Some classes of constraints are extremely easy to check due to their locality, while others are non-local and harder to check. For example, any vertex v whose label is not present in the template cannot be present in an exact match. A vertex in an exact match also needs to have edges to non-eliminated vertices with each label prescribed in the adjacency structure of its corresponding template vertex. Such local constraints that involve a vertex and its neighborhood can be checked iteratively. We call this process *Local Constraint Checking (LCC)* and describe a serial version in §IV-A. For an acyclic template that meets Assumption 1, we guarantee LCC produces no false positives (see §VII), i.e., all selected vertices are part of at least one match.

For more general templates, however, LCC is not guaranteed to eliminate all non-matching vertices (see Fig. 3), and requires complementary routines that check constraints that are non-local. To this end, we consider matching cycles in $\mathcal{G}_0$, a process we dub *Cycle Checking (CC)* and describe in §IV-B. Figure 2 illustrates the complete workflow using an example.

These constraint checking routines iteratively eliminate vertices, refining a set $\mathcal{T}$ that always contains all vertices that are included in an exact match, $\mathcal{T} \supset \bigcup_{\mathcal{S} \sim \mathcal{G}_0} \mathcal{S}$. The goal is to shrink $\mathcal{T}$ as aggressively as possible without throwing out any vertices that can participate in a match. Alg. 1 presents the overall iterative process. The rest of this section describes the serial algorithms for Local Constraint Checking (Alg. 2) and Cycle Checking (Alg. 3). The following sections describe the parallel versions of these two algorithms, their implementation and evaluation.

Algorithm 1 Vertex Elimination

Input: Template graph $\mathcal{G}_0$, background graph $\mathcal{G}$
Output: A vertex subset $\mathcal{T} \subset \mathcal{V}$ containing exact matches
1: **while** Vertices are being eliminated from $\mathcal{T}$ **do**
2: Refine $\mathcal{T}$ with Local Constraint Checking
3: Refine $\mathcal{T}$ with Cycle Checking

Algorithm 2 Local Constraint Checking (Serial Version)

Input: $\mathcal{G}_0$, $\mathcal{G}$, a vertex subset $\mathcal{T} \subset \mathcal{V}$, max iterations k_{max}
Output: A refined vertex subset $\mathcal{T} \subset \mathcal{V}$
1: `// initialize matching functions`
2: $\forall q \in \mathcal{V}_0, v \in \mathcal{T}$ set $f_0(v) = q$ when $\ell(v) = \ell(q)$
3: `// eliminate vertices iteratively`
4: **for** $k = 1, ..., k_{max}$ **do**
5: $\Delta\mathcal{T} \leftarrow 0$
6: **for** $v : f_{k-1}(v) \neq \boxtimes$ **do**
7: $f_k(v) \leftarrow f_{k-1}(v)$
8: `//check local constraints`
9: **for** $q \in adj(f_{k-1}(v))$ **do**
10: **if** no $v' \in adj(v)$ with $f_{k-1}(v') = q$ **then**
11: $f_k(v) \leftarrow \boxtimes$; $\mathcal{T} \leftarrow \mathcal{T} \setminus \{v\}$
12: $\Delta\mathcal{T} \leftarrow \Delta\mathcal{T} - 1$; **break**
13: **if** $\Delta\mathcal{T} = 0$ **then break**

A. Local Constraint Checking

Local constraint checking iteratively generates a sequence of *vertex match functions*, $f_k(v)$ for each iteration k, that map $\mathcal{V}$ onto $\mathcal{V}_0 \cup \{\boxtimes\}$, for $k = 0, 1, ..., k_{max}$, where $\boxtimes$ is a null value, which represents v not being part of any matching subset. Essentially, $f_k(v) = q$ means that, given the computed knowledge up to iteration k of our algorithm, vertex $v \in \mathcal{V}$ is still a possible match for vertex $q \in \mathcal{V}_0$. The $f_k(v)$ are related to $\phi^{-1}(v)$ with requirements similar to Definition 1, in the following sense. For $k > 0$, *(i')* The labels match: $\ell(f_{k-1}(v_1)) = \ell(v_1)$; and *(ii')* matching edges exist: $\forall (f_{k-1}(v_1), q_2) \in \mathcal{E}_0$ there exists v_2 with $f_{k-1}(v_2) = q_2$ and $(v_1, v_2) \in \mathcal{E}$.

The algorithm excludes the vertices that do not have a corresponding label in the template, then, iteratively, excludes the vertices that do not have similarly labeled neighbours as in the template. More formally, the initialization of Alg. 2 defines f_0 for every $v \in \mathcal{V}$. Every vertex $v_* \in \mathcal{V}$ with a label not represented in $\mathcal{G}_0$ is immediately eliminated, $f_0(v_*) = \boxtimes$. Every other $v \in \mathcal{V}$ is assigned the unique $q \in \mathcal{V}_0$ with matching label, $f_0(v) = q$ such that $\ell(v) = \ell(q)$. Then the

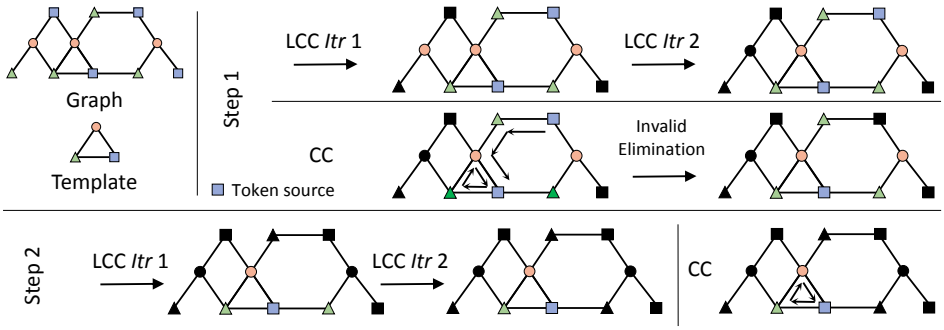

Fig. 2. Algorithm walk through for an example background graph and template depicting which vertices remain in $\mathcal{T}$ or are eliminated at the end of each step / iteration. Eliminated vertices are colored solid black. At the end of the cycle checking (CC) phase in Step 2, no vertex is eliminated and the algorithm terminates.

algorithm proceeds iteratively in step k, checking that the $\mathcal{G}$-neighborhood of each v does not violate the constraints specified in the $\mathcal{G}_0$-neighborhood of $f_{k-1}(v)$, and eliminates v if a single constraint is violated. For an acyclic template, this process is guaranteed to stop eliminating vertices after $k_{max} = \text{diam}\,(\mathcal{G}_0) + 1$ iterations (see Cor. 3).

Complexity. In each iteration, all active vertices visit all their respective neighbors. The worst case is when in each iteration only a few active vertices are eliminated and a large number of iterations is needed. In practice, for real-world scale-free graphs, the first few steps of LCC could reduce $|\mathcal{V}|$ by several orders of magnitude, yielding costs nowhere near the worst case bounds (e.g. see Fig. 8 for an example).

Discussion. Although in practice Alg. 2 aggressively removes a large part of $\mathcal{T}$, this is largely data dependent. However, as we prove in §VII), for template graphs that respect Assumption 1 and are acyclic this phase produces the set of all vertices participating in an exact match and the whole process can stop here. For the general case, complementary processes that check higher-order constraints (edge/distance constraints) are needed to further ensure a minimal number of false positives remain in $\mathcal{T}$.

For example, higher-order structure within $\mathcal{G}$ that would survive this phase, but does not contain the sought template, can be generated if the template contains a cycle. This happens if the background graph contains multiple *unrolled cycles* as in Fig. 3(a) and (b). There are several possible approaches to eliminate these structures; the next section describes a solution to directly check for cycles of the correct length.

B. Cycle Checking

We leverage a *token passing* approach to detect cycles of appropriate length (see Alg. 3). Let $\mathcal{K}_0$ be a set of cycle constraints to be checked; these can be user-specified or generated automatically from the template. Each member $\mathcal{C}_0 \in \mathcal{K}_0$ is a length $r = |\mathcal{C}_0|$ cycle in $\mathcal{G}_0$ beginning w.l.o.g. at q_0, $\mathcal{C}_0 = \{(q_0, q_1), (q_1, q_2), ..., (q_{r-1}, q_0)\}$. For each $v_0 \in \mathcal{T}$ with $\ell(v_0) = \ell(q_0)$, we initiate *tokens* that are passed through edges in $\mathcal{G}_0$ whose ends match the vertex labels in $\mathcal{C}_0$. After r steps we check to see if a self-issued token was received by each initial sender, thus completing an r-length cycle. Once all the cycles in $\mathcal{K}_0$ have been verified, we remove all initiating vertices that do not receive their own tokens in the number of expected steps.

We note that $\mathcal{K}_0$ should contain all orderings of each cycle (i.e., each cycle is presented r times starting with each participating vertex) to guarantee that each remaining vertex in $\mathcal{T}$ participates in each cycle in the template. In other words, each vertex participating in a cycle must issue tokens for that cycle. In practice, however, we use only a few members of $\mathcal{K}_0$ per CC phase, and harness the more efficient aggressive elimination from the complementary LCC phases.

Algorithm 3 Cycle Checking (Serial Version)

Input: $\mathcal{G}_0$, $\mathcal{G}$, a vertex subset $\mathcal{T} \subset \mathcal{V}$, set of cycles, $\mathcal{K}_0$.
Output: A refined vertex subset $\mathcal{T} \subset \mathcal{V}$

1: **for** $\mathcal{C}_0 \in \mathcal{K}_0$ **do**
2: Let (q_0, q_1) be the first edge in $\mathcal{C}_0$.
3: `// initialize cycles`
4: $\mathcal{A}_0 \leftarrow \emptyset$; $\mathcal{A} \leftarrow \emptyset$
5: **for** all $v_0 \in \mathcal{T}$ with $\ell(v_0) = \ell(q_i)$ **do**
6: $\mathcal{A}_0 \leftarrow \mathcal{A}_0 \cup \{v_0\}$
7: $\mathcal{A} \leftarrow \mathcal{A} \cup \{(v_0, v_0, 0)\}$
8: `// loop to process cycles`
9: **for** $s = 1, 2, ..., |\mathcal{C}_0|$ **do**
10: Let (q_i, q_j) be the s-th edge in $\mathcal{C}_0$.
11: $\mathcal{B} \leftarrow \emptyset$
12: **for** every $(v, v_0, s-1) \in \mathcal{A}$ **do**
13: **for** $v' \in adj(v) \cap \mathcal{T}$ **do**
14: **if** $\ell(q_j) = \ell(v')$ **then**
15: $\mathcal{B} \leftarrow \mathcal{B} \cup \{(v', v_0, s)\}$
16: $\mathcal{A} \leftarrow \mathcal{B}$
17: `// remove vertices without the cycle`
18: **for** every $v_0 \in \mathcal{A}_0$ **do**
19: **if** $(v_0, v_0, |\mathcal{C}_0|) \notin \mathcal{A}$ **then** $\mathcal{T} \leftarrow \mathcal{T} \setminus \{v\}$

In Alg. 3, several sets maintain the token initiators and instances of various tokens. The set $\mathcal{A}_0 \subset \mathcal{T}$ is the set of all vertices that initiate cycle attempts in $\mathcal{G}$. After the s-th stage of the algorithm, $\mathcal{A}$ is a collection of three-tuples (v, v_0, s) that store the vertex $v \in \mathcal{V}$ reached by the s-th step of a partial cycle in $\mathcal{G}$ beginning at token-issuing vertex v_0.

Complexity. The presence of a single cycle is verified by passing around tokens in $|\mathcal{C}_0|$ phases in $\mathcal{G}$, where, $|\mathcal{C}_0| \leq |\mathcal{V}_0|$. In Alg. 3, a token passing happens in a *breadth-first*

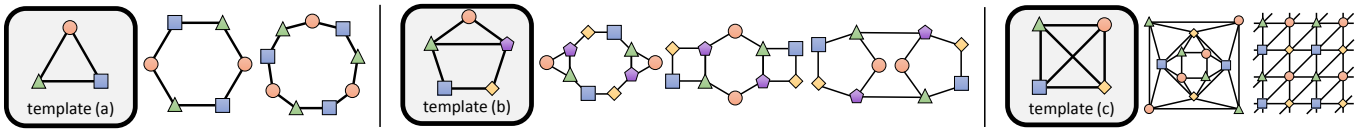

Fig. 3. Three example templates with structures not containing the searched template that survive various vertex elimination phases. Template (a) is a 3-cycle; cycles of length $3k$ with repeated labels meet neighborhood constraints, surviving Alg. 2. Template (b) contains 3-, 4-, 5-cycles; to its right are several examples that also would survive Alg. 2. Template (c) contains several 3- and 4- cycles; to its right are examples that meet both neighborhood constraints and vertex cycle constraints, surviving Alg. 1. The structure on the far right is doubly periodic (a 4×4 torus).

search manner, yielding per-token worst-case time-complexity $O(|\mathcal{V}| + |\mathcal{E}|)$. Let n_t be an upper bound the number of token-issuing vertices in $\mathcal{V}$ for all cycles in $\mathcal{K}_0$. Note n_t can be crudely bound by one-third of the number of the most abundant label in $\mathcal{T}$, as the shortest cycle is a triangle, or $n_t \leq |\mathcal{V}|/3$. Therefore, the time-complexity for checking all $|\mathcal{K}_0|$ cycles is at worst $O(|\mathcal{K}_0| n_t(|\mathcal{V}| + |\mathcal{E}|))$. For the majority of the cyclic query patterns, $|\mathcal{K}_0|$ is expected to be orders of magnitude smaller than both n_t and $|\mathcal{V}|$, e.g., for a triangle pattern, $|\mathcal{K}_0| = 3$; hence, compared to the other terms, $|\mathcal{K}_0|$ has much lesser influence on the overall complexity.

We observe that, in practice, the generated load is often low as the number of active vertices and edges drops drastically after the local constraint checking phase. These bounds highlight the importance of high selectivity in the local constraint checking phase.

Discussion. When used in conjunction the above two algorithms rapidly eliminate nearly all of the non-matching vertices in many cases. In §VII we demonstrate that these algorithms can guarantee that *all* remaining vertices in $\mathcal{T}$ participate in exact matches for template graphs that have unique labels (i.e., meet Assumption 1) and are either acyclic or edge-monocyclic. More generally, one needs additional complementary processes that check higher-order constraints (edge/distance constraints) to further ensure a minimal number of false positives remain in $\mathcal{T}$. Figure 3(c) presents an example where every vertex in the proposed match in the middle of Fig. 3(c) participates in the correct cycles, yet no edge does. The proposed match on the right of Fig. 3(c) meets all vertex and edge cycle constraints, yet still does not contain a match, demonstrating that there are cases where checking higher-order, non-local constraints are necessary to guarantee no false positives.

V. System Design: Parallel Algorithms and Distributed System Implementation

The serial algorithms described so far do not consider important implementation details required to harness parallelism and enable efficient memory access and communication. The following section describes the distributed, vertex-centric implementations of the two algorithms on top of HavoqGT (Highly Asynchronous Visitor Queue Graph Toolkit) [10], a MPI-based framework that supports implementing efficient graph algorithms in distributed environments. Pearce et al. [28] demonstrated that HavoqGT's *asynchronous visitor* abstraction and its implementation provide excellent scalability [29] [30] for several popular graph algorithms. HavoqGT's *delegate partitioned graph* evenly distributes the edges of each high-degree vertex, across the compute nodes to achieve load balancing; crucial to achieving scalability.

HavoqGT allows the implementation of graph algorithms as vertex-programs using an asynchronous visitor abstraction. The framework executes a user-defined vertex-program, called a *visitor*, on the traversed vertices and offers the ability to pass a visitor's state to other vertices [29]. The table below lists the visitor callback procedures and the state that need to be defined for algorithm implementation.

HavoqGT Visitor Interface

pre_visit	Performs a preliminary evaluation and returns true if the visitation should proceed
$visit$	Main visitor procedure
$vertex$	Store the state of the vertex to be visited

The visitor queue has two key primitives that may be used by a visitor or initiating algorithm: $push(visitor)$ pushes a new visitor into the distributed queue, and $do\_traversal()$ initializes and runs the asynchronous traversal to completion. When an algorithm needs to dynamically create new visitors, they are pushed onto the visitor queue using the $push()$ procedure. When an algorithm begins, an initial set of visitors are pushed onto the queue, then the $do\_traversal()$ procedure is invoked. The asynchronous traversal completes when all visitors have been processed, which is determined by a distributed quiescence detection algorithm.

A. Metadata Store

In this paper, we only consider vertex metadata (i.e., labels). Metadata is stored independent of the graph topology. 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. It ensures the metadata store is consistent with the topology of the HavoqGT's delegate partitioned graph [29]. On 256 nodes, for the Web Data Commons graph [31], the metadata store can be built in under two minutes. Our implementation stores attributes as unsigned integers. For string attributes, we store their 64-bit hash representation.

B. The Query Template

The system accepts a template query in the form of an adjacency list. It also requires a map of each vertex ID in the template and its associated label. For the cycle checking

phase, from the original template, we pre-compute the list of cycles for verification. In the current implementation, a walk is initiated from each vertex on each cycle, hence the inputs are all orderings of each cycle.

C. Distributed Algorithms

As described in the previous section, the proposed technique iterates over a sequence of two synchronous steps (Alg. 4): the local constraint checking (LCC) phase followed by cycle checking (CC). Invalidating a single vertex in either of these steps can trigger a cascading effect that ripples down the graph and may reveal additional invalid vertices and cycles, hence, the need to iterate. If a vertex is eliminated, at the end of a step, all the MPI processes are notified to perform another iteration. Alg. 5 lists the state maintained at each vertex and some initialization routines.

Algorithm 4 Distributed Pattern Matching

1: **do**
2: LOCAL_CONSTRAINT_CHECKING
3: **barrier** ▷ *implicit, as LCC phase is synchronous*
4: **if** vertices have been eliminated **then** ▷ *global detection*
5: CYCLE_CHECKING ▷ *Note: for cyclic templates, this routine is invoked regardlessly at least once*
6: **barrier** ▷ *implicit, as CC phase is synchronous*
7: **while** vertices are being eliminated ▷ *global detection*

Algorithm 5 Vertex State and Initialization

1: status: $\alpha(v_j) \leftarrow$ initialized to $true$ if $\ell(v_j) \in \mathcal{L}$ otherwise $false$
2: template vertex: $q_j \leftarrow q_j \in \mathcal{V}_0$, $\ell(q_j) = \ell(v_j)$ and $\ell(v_j) \in \mathcal{L}$
3: $\eta_j(v_i) \leftarrow$ *local constraint verification function* for v_j with internal state, returns $true$ if constraints are satisfied, $false$ otherwise. v_i is the neighbor which v_j received a message from.

1) Local Constraint Checking: This phase is implemented as an iterative process (Alg. 6). Each iteration executes an asynchronous traversal by invoking the $do\_traversal()$ method. Alg. 7 is the visitor implementation of the LCC vertex-program. When the execution begins, an initial set of visitors is created with $msg_{type} = init$. If the metadata of a vertex in the graph is a match for the metadata of any vertex in the template, it creates visitors for all its neighbors with $msg_{type} = alive$. When a vertex is visited with $msg_{type} = alive$, it verifies whether the originating vertex v_i satisfies one of its template constraints by invoking the function $\eta_j(v_i)$. In each iteration, if all the template constraints for a vertex are satisfied, it stays active ($\alpha(v_j) \leftarrow true$) for the next iteration. Otherwise, the vertex is marked inactive ($\alpha(v_j) \leftarrow false$) and never creates visitors again. Iterations continue until, at the end of a round, no vertex is marked inactive.

Message-complexity. In each iteration, a active vertex creates only one visitor per-neighbor, resulting in one message per directed edge of an active vertex.

Algorithm 6 Local Constraint Checking

1: **procedure** LOCAL_CONSTRAINT_CHECKING
2: $vq \leftarrow create\_visitor\_queue(\mathcal{G})$
3: **do**
4: $vq.do\_traversal()$
5: **barrier**
6: **for all** $v_k \in \mathcal{V}$ **do** ▷ $\mathcal{V} \leftarrow$ set of vertices on a MPI rank
7: **if** $\alpha(v_k) = true$ and $\eta_k = false$ **then**
8: $\alpha(v_k) \leftarrow false$ ▷ vertex eliminated
9: **else** reset η_k's internal state for next iteration
10: **barrier** ▷ iterations are synchronous, hence the barrier
11: MPI_AllReduce($\alpha(v_0), ..., \alpha(v_{n-1})$) ▷ required to handle delegates, the distributed edges of a high-degree vertex
12: **while** vertices are being eliminated ▷ global detection

Algorithm 7 Local Constraint Checking Visitor

1: visitor state: $v_j \leftarrow$ vertex to be visited
2: visitor state: $v_i \leftarrow$ vertex originated the $alive$ message
3: visitor state: $msg_{type} \leftarrow$ options are $init$ or $alive$
4: **procedure** PRE-VISIT($\mathcal{G}, vq$)
5: **if** $\alpha(v_j) = false$ **then return** $false$
6: **else if** $is\_delegate(v_j) = true$ **then return** true ▷ if v_j is a delegate vertex, it must forward the visitor to the controller
7: **else** invoke $\eta_j(v_i)$ **return** false ▷ if constraints are met, η_j's state is updated
8: **procedure** VISIT($\mathcal{G}, vq$)
9: **if** $\alpha(v_j) = false$ **then return** $false$
10: **if** $msg_{type} = init$ **then**
11: **for all** $v_k \in adj(v_j)$ **do**
12: $vis \leftarrow$ LCC_VISITOR(v_k, v_j, $alive$) ▷ constructor
13: vq.push(vis)
14: **return** $true$
15: **else if** $msg_{type} = alive$ **then return** $true$

2) Cycle Checking: This process iterates over $\mathcal{K}_0$, the set of cycle constraints to be checked and validates each cycle $\mathcal{C}_0 \in \mathcal{K}_0$ one at a time, as multiple instances of $\mathcal{C}_0$ may exist in $\mathcal{G}$. Cycle checking is implemented as single flow of computation (Alg. 8): token passing is carried out through an asynchronous traversal by invoking the $do\_traversal()$ method. An active vertex $v_j \in \mathcal{G}$, initially a match for the source vertex q_0 of $\mathcal{C}_0$, broadcasts a token to all its neighbors. A map γ is used to track all the vertices that initiate a token. A token contains the token source vertex in $\mathcal{G}$, a field storing the vertex forwarding the token and a *hop count* field initialized to one.

When an active vertex v_j receives a token, it verifies whether its metadata is a match for the next entry in the $\mathcal{C}_0$, 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 constraints are satisfied (i.e., η_j returns true), v_j sets itself as the forwarding vertex, increases the entry in the hop count field by one and broadcasts the token to all its neighbors. If any of the constraints are not met, a vertex immediately drops the token. If the current hop count is equal to $|\mathcal{C}_0|$ and v_j is the same as the source vertex in the token, a cycle has been found and v_j is marked as 'found' in γ. Once verification of a cycle $\mathcal{C}_0$ has been completed, the vertices that are not marked as found in γ are invalidated, i.e., $\alpha(v_j) \leftarrow false$ (Alg. 9).

Since the LCC phase removes the bulk of the invalid vertices, for efficiency reasons, we interleave cycle checking with local constraint checking (Alg. 9, lines 12 to 14): verification of each cycle constraints in $\mathcal{K}_0$ is followed by a LCC phase.

Message-complexity. A vertex keeps track of each token it has received, and only forwards a single copy of a token. This means, a token is passed on the same edge at most two times (the source and destination vertices each forward once). Hence, the total message-complexity of the distributed cycle checking is $O(|\mathcal{K}_0||\gamma||\mathcal{E}|)$.

3) Termination and Output.: For an acyclic pattern, the search terminates when no vertex is eliminated in an LCC iteration. For a cyclic pattern, the search terminates when all cycle constraints in $\mathcal{K}_0$ have been verified and no vertex is eliminated in the following LCC phase. The output is a set of vertices that survived the iterative elimination process, this set of vertices could still possibly participate in a match. If the pattern has unique labels and acyclic or edge-monocyclic, then this is the eact set of vertices that participate in a match. For more general templates, two enhancements are possible but not provided by the current implementation: first, we could keep track and output for each vertex in the output the set of vertices in the template it could possibly match; and second, we could also maintain a set of active edges and output it.

Algorithm 8 Cycle Checking Visitor

```
1: visitor state: v_j ← vertex to be visited
2: visitor state: v_i ← vertex originated the visitor
3: visitor state: v_γ ← vertex originated a token
4: visitor state: r ← hop-counter for the token
5: visitor state: msg_type ← options are init or token
6: procedure PRE-VISIT(G, vq)
7:     if α(v_j) = false then return false
8:     if η_j(v_i) = true then return true
9:             ▷ η_j ensures a token is not forward more than once
10: procedure VISIT(G, vq)
11:    if α(v_j) = false then return false
12:    if msg_type = init and q_j = q_0 ∈ C_0 then
13:        for all v_k ∈ adj(v_j) do
14:            v_γ ← v_j;   r ← 1          ▷ v_γ is the token source
15:            add v_j to γ
16:            vis ← CC_VISITOR(v_k, v_j, v_γ, r, token)
17:            vq.push(vis)        ▷ CC_VISITOR is the constructor
18:        return true
19:    else if msg_type = token then
20:        if η_j(v_i) = true and r = |C_0| and v_j = v_γ then
21:            γ(v_j) ← found return true
22:        else if η_j(v_i) = true and r < |C_0| then
23:            r ← r + 1                   ▷ forward the token
24:            for all v_k ∈ adj(v_j) do
25:                vis ← CC_VISITOR(v_k, v_j, v_γ, r, token)
26:                vq.push(vis)
27:            return true
```

Algorithm 9 Cycle Checking

```
1: procedure CYCLE_CHECKING
2:     vq ← create_visitor_queue(G)
3:     for all C_0 ∈ K_0 do
4:         γ ← map of token source vertices in G for C_0
5:         vq.do_traversal()
6:         barrier
7:         for all v_k ∈ γ do
8:             if γ(v_k) is not marked found then
9:                 α(v_k) ← false ▷ token source vertex eliminated
10:        barrier
11:        MPI_AllReduce(α(v_0), ..., α(v_{n-1}))
12:        if vertices have been eliminated then ▷ global detection
13:            LOCAL_CONSTRAINT_CHECKING
14:            barrier
```

VI. Evaluation

This section provides evaluation of the performance and scalability of our distributed pattern matching system. We present strong and weak scaling experiments on massive real-world and synthetic graphs. All runtime numbers provided are averages over 10 runs. We do not present scaling numbers for a single node as it does not involve network communication and benefits from data locality. We consider naturally occurring patterns and, unless otherwise specified, multiple instances of all the query patterns exist in the respective graphs.

Testbed. The testbed is the Catalyst cluster at at Lawrence Livermore National Laboratory, a 324 node experimental platform with Mellanox Infiniband interconnect. Each node has two 12-core Intel Xeon E5-2695v2 (2.4GHz) processors, 128GB of memory, and Intel 910 PCI-attached NAND Flash per node [32]. We run 24 MPI processes per node.

Datasets. We list the datasets used for evaluation below. All these graphs are undirected and two directed edges are used to represent each edge. The table presents the number of undirected edges.

Datasets Used for Evaluation

Graph	Type	$\|\mathcal{V}\|$	$\|\mathcal{E}\|$
Web Data Commons [31]	Real	3.5B	257B
R-MAT up to Scale 36 [33]	Synthetic	68B	2.2T

Web Data Commons (WDC) is a web-graph whose vertices are webpages and edges are hyperlinks between webpages. We create an undirected version of the graph. To create vertex labels, we extract the top-level domain names from the webpage URL, e.g., *org*, *gov* or *edu*. Total 2903 unique labels are distributed among 3.5B vertices.

The synthetic *R-MAT* graphs exhibit approximate power-law degree distribution similar to *scale-free* graphs. These graphs were created following the Graph 500 [34] standards: 2^{Scale} vertices with an edge factor of 16. For example, for a Scale 30 graph, $|\mathcal{V}| = 2^{30}$ and $|\mathcal{E}| = 16 \times 2^{30}$ (approximately). We leverage degree information to create vertex labels, computed using the formula, $\ell(v_i) = \lceil \log_2(d(v_i) + 1)) \rceil$.

A. Weak Scaling Experiments

Figure 5 shows runtimes for weak scaling experiments using synthetic R-MAT graphs, up to Scale 36 graphs and on up to 256 nodes (6,144 cores). Figure 4 shows the query patterns used for these experiments. (Note that, to stress the system, the patterns have the most frequent labels in the graph). We see consistent scaling all the way to the trillion-edge Scale 36 graph. Runtime is broken down to the individual iteration level

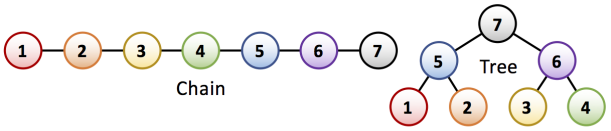

Fig. 4. Chain and Tree patterns with numeric labels. For the R-MAT graphs, these labels cover at least 44% of the vertices, with 1 being the most frequent label (9.5B instances in the Scale 36 graph). These patterns were constructed following the R-MAT characteristic, i.e., high-degree vertices have low degree neighbors.

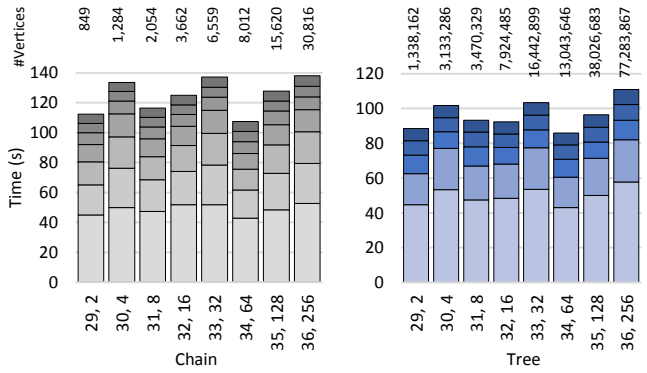

Fig. 5. Runtime for weak scaling experiments, broken down to individual LCC iterations, for the chain (left plots) and tree (right plots) patterns presented in Fig. 4. The X-axis labels present the R-MAT scale and the node count, respectively, used for the experiment. The number of vertices in each solution is shown on the top of respective bar plot. A flat line indicates perfect weak scaling.

to evaluate scaling and individual contribution of intermediate steps. As a graph gets pruned, the subsequent iterations require less time. The figure includes (at the top of each bar) the final number of active vertices that participate in the respective patterns. Except for Scale 31 and 34, at least 0.25% of vertices in the respective graphs participate in the tree pattern while the chain pattern is much rarer. The slight inconsistencies in scaling, both runtime and number of vertices that are match for the template, are due to the inherent non-deterministic nature of R-MAT graph generation.

B. Strong Scaling Experiments

Figure 7 shows the runtimes for strong scaling experiments using the WDC graph on up to 256 nodes (6,144 cores) and using the template queries presented in Fig. 6 and referred to as WDC-1, WDC-2, and WDC-3. Figure 7 shows the breakdown of time spent in each of the LCC and CC phases. We notice near perfect scaling for WDC-1 which is acyclic; hence, it does not invoke the CC phase. For the cyclic patterns WDC-2 and WDC-3, the earlier LCC phases scale almost linearly with increasing node count. However, the CC phases cannot always take advantage of more compute nodes. For WDC-2, on 64 nodes, 25% of the time is spent in the CC phases, while on 256 nodes it is 27%. This gets more expensive for WDC-3, on 64 nodes, 64% of the time is spent in the CC phases, while on 256 nodes it is 85%. Analysis shows that the presence of

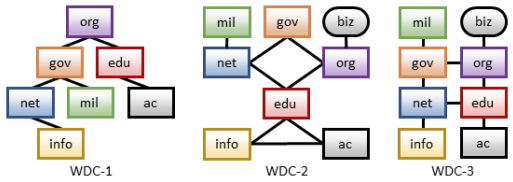

Fig. 6. WDC query patterns with top-level domain names as labels. These are the among the most frequent domains, covering ∼15% of the vertices in the WDC graph. *org* covers 220M vertices, the 2nd most frequent after *com*.

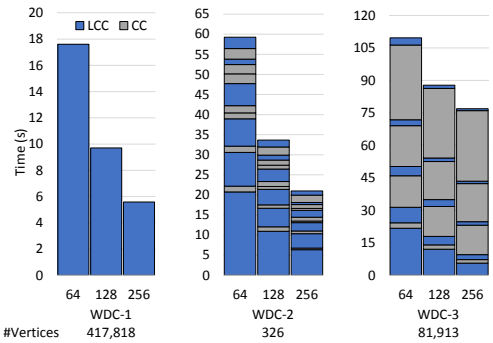

Fig. 7. Runtime for strong scaling experiments, broken down to individual phases (LCC and CC) for the three patterns presented in Fig. 6. The top row of X-axis labels represent the number of compute nodes. The bottom row of X-axis labels are number of vertices participating in the match. The two phases of the algorithm, LLC and CC are presented with different colours.

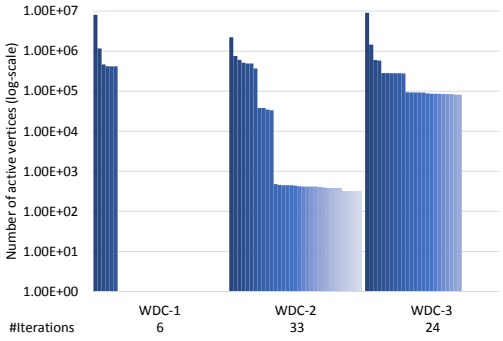

Fig. 8. Number of active vertices after each iteration for the same experiments as in Fig. 7. The bottom row of X-axis labels represent number of iterations required by the pattern. Note: the y-axis is on log-scale.

high-degree vertices in the token path prevents the CC phase from achieving good scaling (the WDC graph has vertices with degree over one million).

Explaining Performance. The runtime of the vertex-elimination based approach improves as more vertices are invalidated early. This reduces the number of visitors created, hence reducing communication since a vertex remains alive as long as its local constraints are satisfied. It is possible, however, to have many vertices that are eventually eliminated but are not identified as invalid early and live through multiple iterations; often an unavoidable artifact of the query pattern. Figure 8 shows an example of this scenario. Although, compared to WDC-2, 1282× more vertices belong to WDC-

1, i.e., more examples of WDC-1 are present in the graph, WDC-2 takes 3.8× longer to complete the search (on 256 nodes, Fig. 7). WCD-2 requires 33 iterations, compared to the six required by WDC-1. After rapid vertex elimination in the first 12 iterations, an additional 21 iterations (although short lived) are required to eliminate the remaining 152 invalid vertices. On 1,536+ processors, the processing rate in these later iterations becomes network-bound. This explains why we do not see perfect scaling for WDC-2. After the first LCC iteration, WDC-3 is mainly bottlenecked by the high cost of CC, as it has ∼82K vertices participating in a cyclic pattern.

To understand what properties of the graph and the query pattern influence performance, we have carried out additional experiments. Here, we briefly discuss our findings. We observe that performance of our graph pruning based pattern matching technique is influenced by the distribution of the metadata and concentration of the query subgraph in the graph. When coupled with a balanced graph partitioning technique, the influence of the graph topology is insignificant for the most part. When a pattern is abundant in the graph, a larger percentage of the vertices remain active and participate in communication throughout. The proposed technique, however, is particularly sensitive to the size and topology of the query subgraph. Patterns with larger diameters tend to require more iterations to complete the search (chain vs tree pattern in Fig. 5). Simpler patterns have fewer local constraints to satisfy; often leading to a large quantity of intermediate matches, thus requiring more iterations. Cyclic patterns rely on the CC phase to identify invalid cycles which has higher message complexity leading to worse scaling.

VII. Theoretical Guarantees

We present a few simple and useful results that characterize the output of Alg. 1. First, we demonstrate that Alg. 2 will not throw out any vertices that are in an exact match, we then provide an upper bound for the number of iterations Alg. 2 needs to be run when the template is acyclic and demonstrate that the remaining vertices are all part of at least one exact match. We finally prove similar properties for edge monocyclic templates.

Theorem 1. *Let $\mathcal{S} \subset \mathcal{V}$ be an exact matching of $\mathcal{G}_0$ (see Def. 1). No phase of Alg. 2 will remove any vertex from $\mathcal{S}$.*

Proof. Let v_1 be any vertex in $\mathcal{S}$. By Def. 1, there exists a bijective mapping, ϕ, from $\mathcal{V}_0$ onto $\mathcal{S}$. The matching function is initialized as $f_0(v_1) = \phi^{-1}(v_1) = q_1$, and no vertex in $\mathcal{S}$ is thrown out by initial checking of labels. Now, assume $f_k(v_1) = q_1$ and no vertex in $\mathcal{S}$ was eliminated in the k-th iteration. For any $q_2 \in adj(q_1)$, $(q_1, q_2) \in \mathcal{E}_0$, and we require a vertex with label $\ell(q_2)$ in $adj(v_1)$. By Def. 1 (ii), we have $(v_1, \phi(q_2)) \in \mathcal{E}$. Further by Def. 1(i) $\ell(\phi(q_2)) = \ell(q_2)$, and the constraint associated with (q_1, q_2) is met. All constraints are satisfied, so $f_{k+1}(v) = q_1$ and v_1 is not eliminated at the $(k+1)$-th iteration. This is similar for all other vertices in $\mathcal{S}$, so v_1 will never be eliminated. □

It is straightforward to see that Alg. 3 will not throw out any vertices that are in an exact match.

After r iterations, the vertices left seem like plausible matches in the sense that the same r-length walks (in terms of labels) can be taken from a surviving vertex as a correspondingly labeled vertex in the template.

Lemma 2. *Under Assumption 1, let r iterations of Alg. 2 be performed. If $f_r(v_0) = q_0$ then for every length r walk, $\mathcal{W}_0$, in $\mathcal{G}_0$ starting at q_0, there is a length r walk, $\mathcal{W}$, in $\mathcal{G}$ starting with v_0 with the same sequence of vertex labels as $\mathcal{W}_0$.*

Proof. Let $\mathcal{W}_0 = \{(q_0, q_1), (q_1, q_2), ..., (q_{r-1}, q_r)\}$ be an r-length walk starting at q_0 in the undirected version of the template, $\mathcal{G}_0$. If $f_r(v_0) = q_0$, then for each $i = 1, ..., r$ there exists a $v_i \in \mathcal{V}$ such that $(v_{i-1}, v_i) \in \mathcal{E}$, $f_{r-i}(v_i) = q_i$, with $\ell(v_i) = \ell(q_i)$. □

Lemma 2 is extremely useful. For acyclic graphs, it helps us derive the maximum number of iterations that can be taken before no more vertices are eliminated.

Corollary 3. *If $\mathcal{G}_0$ is acyclic, no more vertices are eliminated after $k_{max} := diam\,(\mathcal{G}_0) + 1$ iterations of Alg. 2.*

Proof. Because $\mathcal{G}_0$ is acyclic, this is a direct consequence of Lemma 2. □

Additionally, Lemma 2 shows that when $\mathcal{G}_0$ has unique labels and is acyclic, Alg. 2 successfully removes all vertices from $\mathcal{G}$ that do not participate in an exact match.

Corollary 4. *If Assumption 1 is met and $\mathcal{G}_0$ is acyclic, then after k_{max} iterations of Alg. 2 every remaining vertex in $v_0 \in \mathcal{T}$ is a member of at least one $\mathcal{S} \subset \mathcal{V}$ that matches $\mathcal{G}_0$.*

Proof. If $f_{k_{max}}(v_0) = q_0$, the we apply Lemma 2 to construct an $\mathcal{S} \subset \mathcal{V}$ containing v_0 and exactly matching $\mathcal{G}_0$. □

When $\mathcal{G}_0$ is not acyclic, Alg. 3 is also employed. However, checking cycle participation for vertices is not enough to guarantee that there are no false positive vertices $\mathcal{T}$ after completing Alg. 1. (see Fig. 3(c), the torus on the far right, for a simple pathological example). Additional constraints (e.g. distances and/or edge participation in cycles) are required to remove such structure. Under the additional assumption that $\mathcal{G}_0$ is edge-monocyclic (edges participate in at most one cycle; see §II), we have the following result that guarantees no false positive vertices. It is important to note that $\mathcal{K}_0$ must contain every cycle $\mathcal{C}_0$ in $\mathcal{G}_0$. To prove the result, we leverage the tree-like quality of edge-monocyclic graphs and the local constraints met by vertices in $\mathcal{T}$ to construct a matching $\mathcal{S} \subset \mathcal{T}$.

Theorem 5. *If Assumption 1 is met and $\mathcal{G}_0$ is edge-monocyclic, then when Alg. 1 terminates every remaining vertex in $v_0 \in \mathcal{T}$ is a member of at least one $\mathcal{S} \subset \mathcal{V}$ that matches $\mathcal{G}_0$.*

Proof. We prove that if v_0 survives local constraint checking and participates in every cycle listed in $\mathcal{K}_0$ involving q_0 then an $\mathcal{S}$ that matches $\mathcal{G}_0$ can be constructed.

Let $\mathcal{U}_0$ be any vertex spanning tree of $\mathcal{G}_0$, rooted at q_0, the unique vertex in the template for which $\ell(q_0) = \ell(v_0)$. Additionally, we have $\mathcal{R}_0$, the set of edges in $\mathcal{G}_0$ not in spanning tree $\mathcal{U}_0$. For each edge $(q_i, q_j) \in \mathcal{R}_0$, we have a single length-r cycle $\mathcal{C}_0$ involving (q_i, q_j) and $(r-1)$ edges in $\mathcal{U}_0$. These $|\mathcal{R}_0|$ cycles have no edge overlap by the edge-monocyclic property and $|\mathcal{K}_0| = |\mathcal{R}_0|$.

By Lemma 2 there exists at least one tree subgraph $\mathcal{U}$ in $\mathcal{T}$ that matches $\mathcal{U}_0$. Below, we show we are able to modify this tree within $\mathcal{T}$ until it yields an $\mathcal{S}$ that matches $\mathcal{G}_0$. If the vertex set of $\mathcal{U}$ is not a match, then there exists an edge (q_i, q_j) in $\mathcal{R}_0$ such that the corresponding vertices v_i, v_j in $\mathcal{U}$ are not connected in $\mathcal{G}$. We let (q_i, q_j) be the first such edge encountered in a deterministic breadth-first search tree ordering (where ties are broken by vertex number). In $\mathcal{G}_0$ edge (q_i, q_j) forms a cycle $\mathcal{C}_0$ emanating from the edge, up the spanning tree both directions until they meet at a mutual ancestor q_a. By the edge-monocylic property, the vertices in $\mathcal{C}_0$ contain no more crossing edges in $\mathcal{R}_0$, in particular q_i and q_j are not connected to any other vertex in $\mathcal{C}_0$ other than their respective parents. The vertices in $\mathcal{U}$ associated with those in $\mathcal{C}_0$ are not involved in a cycle with the same labels. For v_a to be in $\mathcal{T}$ there must be some other $|\mathcal{C}_0| - 1$ vertices in $\mathcal{T}$ that have a cycle matching $\mathcal{C}_0$, so we replace the part of $\mathcal{U}$ below v_a with the vertices that contain the cycle (including the other generations below, as guaranteed by Lemma 2) to get a new $\mathcal{U}'$ that still contains root v_0. Now either $\mathcal{U}'$ is a match or a new edge (q'_i, q'_j) exists in $\mathcal{R}_0$ such that the corresponding vertices v'_i, v'_j in $\mathcal{U}$ are not connected in $\mathcal{G}$. In this case, we repeat the process at most $|\mathcal{K}_0|$ times until we have constructed a match. □

For edge-monocyclic $\mathcal{G}_0$ we also have a guarantee on how many iterations of each algorithm are required for $\mathcal{T}$ to converge with no false positives.

Corollary 6. *If Assumption 1 is met and $\mathcal{G}_0$ is edge-monocyclic, after all cycles have been checked (from every vertex in the cycle) with Alg. 3, if $k_{max} := diam(\mathcal{G}_0) + 1$ iterations of Alg. 2 are run, then there are no false positives left in $\mathcal{T}$.*

The previous result shows we have no false positives for edge-monocyclic $\mathcal{G}_0$ if we run Alg. 3 (CC) followed by k_{max} iterations of Alg. 2 (LCC). In practice, to benefit from efficient aggressive elimination of LCC and to minimize cycle checking, we run LCC first, then check one cycle at a time followed by LCC, repeating until all vertices participating in all cycles have had their participation in the cycles checked. This strategy takes $n_c\overline{r}$ iterations of Alg. 3, where n_c is the number of template cycles and $\overline{r}$ is the average cycle length.

Although beyond the scope of this paper, the current algorithms could be adapted to design new algorithms that help with more general templates, e.g. those that violate Assumption 1 and/or have denser cyclic structure.

VIII. Conclusion

Graph analytics problems such as pattern matching, that are fundamentally complex yet serve a rich set of applications, have been relatively unexplored in the context of large-scale processing. This paper presents our work towards scalable labeled pattern matching on massive graphs. To this end, we have developed a vertex elimination-based algorithm that excludes vertices that do not satisfy the constrains in a given pattern. We designed the algorithm for implementation within a scalable vertex-centric distributed graph framework and implemented it on the top of HavoqGT [10]. Evaluation using a 257B edge real-world web-graph and a trillion-edge synthetic R-MAT graph confirms the scalability of our solution. Additionally, we demonstrate that for a subclass of patterns (that are either acyclic or edge-monocyclic and have unique labels) and regardless, of the background graph topology result of our vertex pruning algorithms includes all the vertices that participate in a match and only them.

The experimental results confirm the potency and effectiveness of our technique to scale subgraph pattern matching to a trillion edge graph. Today, the ability to scale graph analytics to such massive graphs is relevant; a recent work reports a Facebook 'users' graph consists of more than one trillion edges [35]. Our success stems from a number of key design ingredients: aggressive vertex elimination while harnessing massive parallelism, low message overhead and lightweight per-vertex state.

Future Work. We will continue our work on enriching the feature set and performance of our pattern matching system. Our goal is to realize a framework to support: (i) a variety of pattern matching scenarios - in addition to exact matching, we want to provide support for approximate, top-k matching [1], (ii) a richer set of subgraph patterns, e.g., patterns with repeating vertex labels and patterns with edge labels. At present, in the case of patterns with a high-concentration of cycles, the cycle checking phase becomes a bottleneck, primarily due to the high volume of tokens (messages) that have to be passed, an issue that is compounded when high-degree vertices participate in cycles.

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] 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

[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] 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

[8] "Giraph." [Online]. Available: http://giraph.apache.org/

[9] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: https://doi.org/10.14778/2212351.2212354

[10] "HavoqGT." [Online]. Available: http://software.llnl.gov/havoqgt/

[11] 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.

[12] J. Soman and A. Narang, "Fast community detection algorithm with gpus and multicore architectures," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 568–579.

[13] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[14] H. Kakugawa, S. Kamei, and T. Masuzawa, "A token-based distributed group mutual exclusion algorithm with quorums," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 9, pp. 1153–1166, Sept 2008.

[15] 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.

[16] 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

[17] 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.

[18] 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.

[19] 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

[20] 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

[21] 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

[22] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *The VLDB Journal*, vol. 19, no. 1, pp. 91–113, Feb. 2010. [Online]. Available: http://dx.doi.org/10.1007/s00778-009-0165-y

[23] "Chapter 3. Cypher - The Neo4j Developer Manual v3.1." [Online]. Available: https://neo4j.com/docs/developer-manual/current/cypher/

[24] H. Tong, C. Faloutsos, and J.-Y. Pan, "Fast random walk with restart and its applications," in *Proceedings of the Sixth International Conference on Data Mining*, ser. ICDM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 613–622. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2006.70

[25] 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

[26] 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.

[27] 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.

[28] 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

[29] R. Pearce, M. Gokhale, and N. 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

[30] S. Sallinen, K. Iwabuchi, S. Poudel, R. Pearce, and M. Ripeanu, "Graph colouring as a challenge problem for dynamic graph processing on distributed systems," in *Proceedings of the 2016 IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2016)*, ser. SC'16, 2016.

[31] O. L. Robert Meusel, Christian Bizer, "Web Data Commons - Hyperlink Graphs." [Online]. Available: http://webdatacommons.org/hyperlinkgraph/index.html

[32] "Catalyst." [Online]. Available: http://computation.llnl.gov/computers/catalyst

[33] 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.

[34] "Graph 500 benchmark." [Online]. Available: http://www.graph500.org/

[35] 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