Fractal: A General-Purpose Graph Pattern Mining System

Vinicius Dias
Carlos H. C. Teixeira
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
viniciusvdias@dcc.ufmg.br
carlos@dcc.ufmg.br

Dorgival Guedes
Wagner Meira Jr.
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
dorgival@dcc.ufmg.br
meira@dcc.ufmg.br

Srinivasan Parthasarathy
The Ohio State University
Columbus, USA
srini@cse.ohio-state.edu

ABSTRACT

In this paper we propose Fractal, a high performance and high productivity system for supporting distributed graph pattern mining (GPM) applications. Fractal employs a dynamic (auto-tuned) load-balancing based on a hierarchical and locality-aware work stealing mechanism, allowing the system to adapt to different workload characteristics. Additionally, Fractal enumerates subgraphs by combining a depth-first strategy with a from scratch processing paradigm to avoid storing large amounts of intermediate state and, thus, improves memory efficiency. Regarding programmer productivity, Fractal presents an intuitive, expressive and modular API, allowing for rapid compositional expression of many GPM algorithms. Fractal-based implementations outperform both existing systemic solutions and specialized distributed solutions on many problems - from frequent graph mining to subgraph querying, over a range of datasets.

CCS CONCEPTS

• **Mathematics of computing → Graph enumeration**; *Graph algorithms*; • **Computer systems organization → Distributed architectures**; • **Software and its engineering** → *Distributed programming languages*;

KEYWORDS

graph pattern mining; distributed systems

ACM Reference Format:
Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira Jr., and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319875

1 INTRODUCTION

Graph pattern mining (GPM) plays an important and increasing role in a number of existing and emerging applications, from extracting motifs from gene networks [39, 59] to brain networks [9], from searching for patterns over semantic data (e.g., RDF) [16] to social media analysis [56], and from community discovery [3, 14] to link spam detection [34].

Graph algorithms tend to be complex and non-trivial to develop especially in distributed environments. The advent of Pregel [37] and related graph analytics systems [22, 23, 61] sought to address this issue by offering a simpler way to implement and design efficient distributed variants of algorithms. Unfortunately, such systems are focused on matrix-based algorithms, and may not be suitable for all GPM problems [53]. At its core, GPM methods perform subgraph enumeration, which may be computationally and storage intensive, where a tremendous amount of intermediate state can be generated even when running on small-scale networks (e.g., 5-10k nodes). Moreover, the irregular topology presented in scale-free graphs makes GPM quite challenging regarding load balancing in parallel and distributed settings.

This complexity, in turn, has led to the development of distributed algorithms for specialized (domain-specific) GPM problems, such as frequent subgraph mining [1], motif counting [47], and clique counting [19], that do not generalize to other GPM problems. Systems such as Arabesque [53] and NScale [45] have emerged as first generation, general-purpose solutions for GPM. While both potentially offer programming interfaces suitable for processing coarse-grained GPM applications, their computation models fail to support fine-grained interactive analysis. In fact, those systems adopt a BFS-style subgraph enumeration to balance the work at the end of each synchronization step, generating and shuffling a huge amount of intermediate state between workers, what leads to increased overhead at larger scales.

In this work we propose Fractal, a distributed system for general-purpose graph pattern mining built on Spark [64]. Our design goals for Fractal are simple: (i) a flexible yet simple API focused on analyst productivity; and (ii) an efficient systemic support for a range of GPM kernels on modern distributed architectures that handles their irregular memory and computational demands. In order to meet those twin goals, Fractal makes the following contributions.

- *Flexible, expressive and compositional API.* Fractal's API and programming model are designed from the ground up to be simple and to reflect fundamental GPM operations (denoted as primitives). In this work, we show how the API is flexible to compose a wide range of GPM applications using just a few lines of code. To the best of our knowledge no other system can tackle such a range of GPM applications.
- *Mitigating irregular memory demand.* The amount of intermediate state in GPM applications is hard to predict, being a significant source of overhead. The use of modern garbage collected languages – widely used in popular stacks for data analytics – compounds this issue leading to high unpredictability in performance [20, 36, 41, 42]. Fractal combines a depth-first strategy with a "from scratch processing" paradigm to keep the memory requirements bounded. Our results show that it can improve performance and reliability of GPM applications: executions up to three orders of magnitude faster than comparable systems or specialized baselines, while leaving more memory for the user application.
- *Adaptive load balancing.* As mentioned, GPM algorithms are irregular by nature. Balancing the load and minimizing communication overhead is central to performance efficiency. Fractal incorporates a novel hierarchical work stealing and communication mitigating strategy that is aware of task locality and reduces the communication overhead, achieving a nearly-ideal load balancing in many scenarios.
- *Novel graph reduction and filtering.* Fractal relies on a novel graph reduction optimization to speed up the enumeration phase of many GPM methods. A data analyst can specify a reduced (materialized) view of the input graph. This potentially benefits a range of GPM applications, reducing their memory footprint, when subgraphs (or patterns) of interest lie in localized (sub-)regions of the original input graph.
- *Extensive evaluation.* We perform an extensive evaluation of Fractal, and demonstrate that it outperforms both specialized distributed algorithms and general-purpose systems on a range of GPM kernels and input graphs.

2 BACKGROUND

We briefly review the model and key definitions needed to understand Fractal and the GPM algorithms executed in it.

2.1 Graph Model

Without loss of generality, we adopt in this work an input graph G with vertices and non-directed edges which may have multiple labels as described in Definition 1.

Definition 1. ***(Graph)*** *A graph G is represented by three sets, $V(G)$, $E(G)$ and $L(G)$ which are the sets of vertices, edges, labels (or keywords) of G and one map function f_L. Each edge $e = (v, u) \in E(G)$ connects a pair of vertices v and $u \in V(G)$. The edges are not directed and there are no self-loops in G. Formally, $(v_i, v_j) = (v_j, v_i)$ and $e = (v_i, v_i) \notin E(G)$. The labels (keywords) of a vertex or an edge are defined according the function $f_L : V(G) \cup E(G) \rightarrow \mathbb{P}(L(G))$ (power set).*

Definition 2. ***(Subgraph)*** *Let G and S be graphs. We say that S is an subgraph of G iff $V(S) \subseteq V(G)$ and $E(S) \subseteq E(G)$.*

According to Definition 2, a subgraph is represented by a set of vertices and edges embedded in the input network G. In particular, Fractal works with *connected subgraphs.*

Isomorphism and patterns. Graph isomorphism (Def. 3) is the problem of verifying whether two (sub)graphs have an identical structure (topology), being fundamental to a variety of GPM applications such as motif counting, frequent pattern mining and graph matching. Given a set of (sub)graphs $\mathcal{S} = \{S_i, S_2, \ldots, S_N\}$, the isomorphism relation divides $\mathcal{S}$ into equivalence classes, where each class contains graphs that are isomorphic among themselves.

Definition 3. ***(Isomorphism)*** *Two (sub)graphs G and H are isomorphic iff there is a bijective function $\pi: V(G) \Rightarrow V(H)$ such that $(v_i, v_j) \in E(G)$ iff $(\pi(v_i), \pi(v_j)) \in E(H)$.*

The concept of **pattern** is related to isomorphism since two (sub)graphs G and H in the same class have the same pattern. In practice, a pattern is a template for a subgraph and, thus, a subgraph is an instance of its pattern. In this work, we adopted the depth-first search (DFS) coding algorithm [62] to determine the canonical labeling of a labeled (sub)graph S, which is given by the function $\rho(S)$. Basically, the canonical labelling is an string that represents the pattern of a given (sub)graph through the ordering of its edges. This is is a popular and an efficient algorithm to perform isomorphic checks (i.e., comparison of strings) between (sub)graphs.

2.2 Graph Pattern Mining Problems

Let G be input graph and $\mathcal{S} = \{S_1, \ldots, S_N\}$ be the set all of distinct subgraphs and $\mathcal{P} = \{P_1, \ldots, P_M\}$ be the set of canonical patterns, both enumerated from G. In other words,

$\mathcal{P} = \bigcup_{S' \in \mathcal{S}}\{\rho(S')\}$). We briefly review popular GPM kernels studied in the literature and also evaluated in Fractal.

Motif extraction & counting. A *motif* P is defined as a connected and induced subgraph pattern in an input graph G. The goal is to count frequencies of all motifs (patterns) having k vertices, *i.e.*, we want to compute $|\{S' \in S \mid |V(S')| = k \text{ and } \rho(S') = P'\}|, \forall P' \in \mathcal{P}$. This kernel usually ignores the labels in G and it is widely used in bioinformatics [39, 44].

Cliques listing & counting. A k-node clique is a complete subgraph having k nodes in an input graph. In this case, only the topology of the subgraphs is considered. Thus, we may formally define the set of k-node cliques (or k-cliques) in G as follows: $\{S' \in \mathcal{S} \mid |V(S')| = k \text{ and } |E(S')| = \frac{k(k-1)}{2}\}$.

Frequent subgraph mining (FSM). An FSM task seeks to obtain all frequent subgraph patterns from a labeled input graph G. A pattern P is frequent if it has a support $s(P)$ above a threshold α, *i.e.*, if $s(P) \geq \alpha$. Generally, $s(P)$ is calculated based on the set of isomorphic subgraphs, defined as $\{S' \in \mathcal{S} \mid \rho(S') = P\}$. In this work, we adopt the *minimum image-based support* [7] as the support function $s(\cdot)$ to leverage the anti-monotonic property. Therefore, we may define the result set of FSM as $\{P' \in \mathcal{P} \mid s(P') \geq \alpha\}$. For a more detailed description of FSM, see [17, 53].

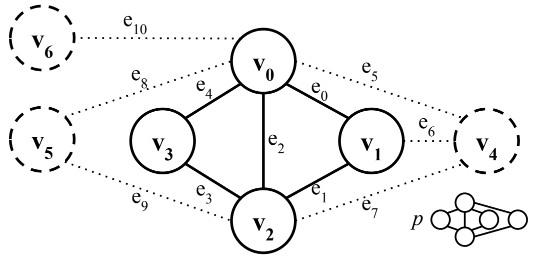
Subgraph querying or listing. Querying a subgraph pattern is maybe the naivest GPM application known. The task is to list all the subgraphs in an input graph G that are isomorphic to a user-defined pattern P. Formally, we seek to retrieve the following set of subgraphs: $\{S' \in \mathcal{S} \mid \rho(S') = P\}$.

Keyword-based subgraph search. Given an attributed graph G with keywords (or labels) on nodes and edges and a query represented by a set of keywords $\mathcal{K} = \{w_1, \ldots, w_C\}$, the task is to retrieve subgraphs in G as follow. A subgraph $S' \in \mathcal{S}$ is retrieved if its keywords cover $\mathcal{K}$ and each edge in $E(S')$ is responsible for, at least, one these covers. Formally, this problem seeks to find the set of subgraphs given by $\{S' \in \mathcal{S} \mid \mathcal{K} \subseteq L(S') \text{ and } \mathcal{K} \not\subset L(S) \setminus f_L(e), \forall e \in E(S')\}$. This problem is widely studied in the context of RDF data [16].

3 COMPUTATION MODEL

Fractal adopts a model with three types of computation **primitives**: *extension*, *aggregation*, and *filtering*. An analyst may design a sequence of these primitives to solve a particular GPM problem. Primitives are applied on subgraphs of the input graph G to enumerate more subgraphs, to prune the search space, or to summarize the results:

Extension **(E)**. This is a core primitive of the Fractal computation model, responsible for generating the solution space of any GPM problem - specifically, it represents the *subgraph enumeration* step of GPM problems. This primitive receives a set of subgraphs as input and extends them by using their own neighborhood in G, producing a set of larger subgraphs. Our model supports the following extension strategies, which are illustrated in Figure 1:



edge-induced	vertex-induced	pattern-induced (p)
$\{v_4, e_5\}$, $\{v_5, e_8\}$	$\{v_4, e_5, e_6, e_7\}$	$\{v_4, e_5, e_7\}$
$\{v_4, e_6\}$, $\{v_5, e_9\}$	$\{v_5, e_8, e_9\}$	$\{v_5, e_8, e_9\}$
$\{v_4, e_7\}$, $\{v_6, e_{10}\}$	$\{v_6, e_{10}\}$	

Figure 1: Subgraph extensions: the user can define three types of subgraph extensions. The subgraph above (composed of vertices and edges in solid lines) has 6, 3 and 2 possible extensions for the edge-, vertex- and pattern-induced extensions, respectively.

- *Edge-induced extension* expands a subgraph S edge-by-edge, considering its neighborhood, which is often used for frequent subgraph mining.
- *Vertex-induced extension* expands a subgraph S vertex-by-vertex, that is, whenever a vertex v is added, all edges that connect v to S are included. This is often used in motif extraction, clique listing kernels.
- *Pattern-induced extension* expands subgraphs vertex-by-vertex, but guided by a user-defined reference pattern P (*e.g.*, graph querying and matching).

The above extension strategies must avoid redundant (symmetric) enumerations. Fractal addresses this issue by combining the extension algorithms with canonical subgraph checking, for vertex(edge)-induced subgraphs [53], or symmetry breaking [24], for pattern-induced subgraphs.

Aggregation **(A)**. The aggregation primitive summarizes a set of subgraphs into patterns and metrics for downstream processing. In other words, it receives a set of subgraphs as input and maps them to key/value entries for subsequent reduction. For aggregation one needs to define three functions: *(1)* a mapping function to extract a key from a subgraph; *(2)* a second mapping function to extract a value from a subgraph; and *(3)* a reduction function to reduce the values sharing the same key. This computation primitive is fundamental for any GPM application that relies on frequency counts, or any application that works with aggregates (*e.g.*, sum).

Filtering **(F)**. The filtering primitive within the Fractal model is used to prune subgraphs that do not meet the application criteria. Currently, Fractal supports two options.

The *local filtering* alternative prunes a subgraph by using local information. For instance, in clique listing algorithms, a subgraph may be trivially classified as a non-clique by just examining its local topology. The *aggregation filtering* option prunes a subgraph by considering a source of information provided by an upstream aggregation primitive. This particular filter can be leveraged within algorithms like FSM, where subgraphs that do not belong to the current set of frequent subgraphs may be discarded. Note that both types of filtering are performance-critical for a range of applications as they limit the enumeration of irrelevant subgraphs and reduce the search space for a given task.

Fractal's design allows a developer to specify a computation workflow concisely and typically in fewer steps than contemporary GPM systems [45, 53]. For instance, counting 3-cliques can be implemented in a single step: three extensions followed by an aggregation primitive (`EEEA-`) and a single synchronization point (represented as "`-`"). Other BFS-style GPM systems for the same tasks [45, 53] require at least three steps separated by synchronization points. Additionally, their BSP [57] design result in expensive synchronization and communication overheads if either the input or the intermediate data being processed are large.

3.1 Fractal Programming Interface

Designing a flexible and expressive API for distributed GPM applications is challenging. An API is expressive when it is easily readable and interpretable and it is flexible when it is capable of representing a wide range of applications. Fractal's API is *subgraph-centric* [10, 45, 53] in that it exposes a small set of intuitive and modular operators to construct complicated GPM applications. All operators act on a state object, called a *fractoid*. We discuss these in turn next.

Fractoid. A fractoid holds the state of a Fractal application during the execution process. Such state includes an array of primitives representing the user workflow and any aggregation result required for computation. One can derive a fractoid from either another fractoid or from the input graph. Fractal supports three types of fractoids – edge-induced, vertex-induced and pattern-induced.

Initialization operators (Fig. 2). The entry point to an application is the `FractalContext`, responsible for configuring and initializing all the required resources to build and run Fractal routines. Since our current implementation runs on top of Spark [64], we instantiate a `FractalContext` (C_1) directly from a `SparkContext`. In order to obtain the first fractoid, the user must first create a fractal graph from the context by passing an input path (I_1) and then ask for a vertex-induced fractoid (B_1), an edge-induced fractoid (B_2) or a pattern-induced fractoid (B_3) (see Figure 3 for a vertex-induced example).

C_1	`new FractalContext(sc: SparkContext)`
I_1	`def adjacencyList(path: String): FractalGraph`
B_1	`def vfractoid(): Fractoid // by-vertex`
B_2	`def efractoid(): Fractoid // by-edge`
B_3	`def pfractoid(p: Pattern): Fractoid // by-pattern`

Figure 2: Initialization operators.

```
val sc = new SparkContext(..)
val fctx = new FractalContext(sc)
val graph = fctx.adjacencyList (graphPath)
val vfrac = graph.vfractoid()
```

Figure 3: Initialization of a Fractal application.

Workflow operators (Fig. 4) . These operators are used to describe the processing performed over fractoids in order to explore the space of solutions (subgraphs) or must be explored in the input graph.

```
W1    def expand(depth: Int): Fractoid
      def aggregate[K,V](aggName: String)(
          key: (Subgraph, Computation, K) => K,
          value: (Subgraph, Computation, K) => V,
W2        reduction: (V, V) => V,
          aggFilter: (K, V) => Boolean
      ): Fractoid
      def filter(
W3      f: (Subgraph, Computation) => Boolean
      ): Fractoid
      def filter(aggName: String)(
W4      f: (Subgraph, Aggregation[K,V]) => Boolean
      ): Fractoid
W5    def explore(n: Int): Fractoid
```

Figure 4: Workflow operators.

The `expand` function (W_1) represents the extension primitive (`E`) and enumerates subgraphs by extending the subgraphs given as input. Considering a GPM application that uses the edge-induced extension method, an m-expansion over k-edge subgraphs generates all (and unique) subgraphs of size $(k + m)$ edges. Fractal also supports user-defined aggregations (W_2), which represent the aggregation primitive (`A`). For instance, in motif counting, one may be interested in creating an aggregation where the key is the subgraph pattern (`key`), the value is the number one (`value`), the reduction function is a simple sum operation (`reduction`). The last parameter (`aggFilter`) is an optional filtering step applied to the final reduced mappings.

The filtering primitive (`F`) is mapped to the Fractal API via two options: *local filter* (W_3) and *aggregation filter* (W_4). The local filter prunes subgraphs based on local information. Alternatively, the aggregation filter discards a subgraph by considering the result of an aggregation operator: a key/-value mapping (`Aggregation[K,V]`). Our last workflow operator

(W_5) is used to keep applications clean and concise. This operator chains a workflow fragment n times, simplifying the implementation of iterative algorithms.

Output operators (Fig. 5). These operators represent the application output of subgraphs or aggregations for downstream analysis. A straightforward way to expose those outputs is through Spark's built-in support of RDDs (of subgraphs (O_1)), or alternatively through key/value pairs (O_2), obtained from a named aggregation. We highlight that because we use RDDs to export Fractal's output, the system inherits its resilience for these operators.

```
O1  def subgraphs(): RDD[Subgraph]
O2  def aggregation(aggName: String): RDD[(K,V)]
```

Figure 5: Output operators.

Implementing applications with Fractal. Fractal's API allows an intuitive and interactive experience since every partial result of a workflow (*fractoids*) can be easily executed and verified separately. Users can combine any sequence of primitive components and perform successive refinements in their analysis. Existing systems lack such support since they view applications as atomic jobs waiting to be executed in batch mode [28, 45]. We present the implementations of the various applications used in this paper in Appendix A.

4 FRACTAL SYSTEM DETAILS

Fractal considers a distributed and parallel environment, organized with a single application master and many workers.

System architecture. As illustrated in Fig. 6, the user interacts with Fractal by submitting commands to the application master (a). The master contains the execution engine, which manages the underlying cluster resources and coordinates the execution of *fractal steps* (b). Workers represent instances that perform the actual GPM processing. Specifically, each worker ($w_0, \ldots, w_{n-1}$) is a process running

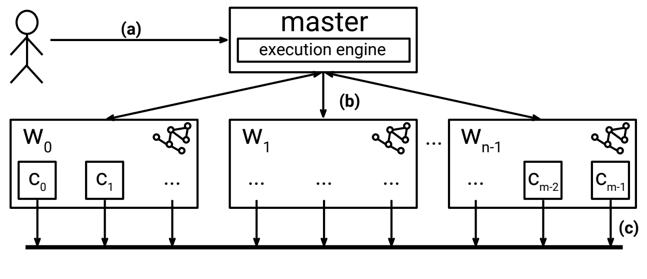


Figure 6: System architecture. (a) Programs are created with the Fractal API and submitted to the master. (b) The master initiates the setup of workers and schedules Fractal steps. (c) Workers execute the steps.

on multiple cores ($c_0, \ldots, c_{m-1}$). Communication occurs between master and workers, but also among workers via an actor-model paradigm. Such design eases the initialization and the asynchronous assignment of work amongst workers (c).

System initialization. The master is the first to start, followed by the workers. The master acts as a central point for hand-shaking among workers. Worker initialization includes the creation of internal structures – including a global identifier for the worker and each one of its cores – and the reading of the input graph in-memory. Input graphs may be stored on the local file system or on HDFS [50]. Then, each worker sends a registration message to the master. When the master acknowledges the registration of the workers, it broadcasts their addresses. Thus, every worker knows how to reach the others, in addition to the master.

Scheduling and execution. The application master prepares the application workflow for execution. Inspired by the concept of dependencies among distributed collections [64], a sequence of primitive components can be pipelined whenever there are no global synchronization points. A synchronization point is required when the user calls an output operator, triggering the evaluation of the workflow (O_1 and O_2, Fig. 5), or when the workflow contains a filter clause that reads from an aggregation not yet computed (W_4, Fig. 4). We denote such blocks of pipelined computation as *fractal steps*, and they are the scheduling units of our system. The master submits *fractal steps* to be executed by the workers. Within each submission, the application master piggybacks the fractoid for execution. Then, each core of every worker starts the execution over an empty subgraph and an initial partition of extensions from the input graph, determined on-the-fly using its unique core identifier. For both vertex- and pattern-induced fractoids, the initial extensions are single vertices, while edge-induced fractoids use single edges (Fig 1).

Proof of concept over Spark and Akka. Our current version of Fractal is built on top of Spark 2.0 [64], however its design is independent enough to fit other platforms. Specifically, we leverage Spark's batch computation model to represent and schedule Fractal steps for execution. Thus, each Fractal step corresponds to a Spark job, and the computation performed by each core corresponds to the processing of a Spark partition in the underlying system. This part covers all communication requirements between master and workers (Fig. 6b). Additionally, we extend Spark's execution model with an actor model provided by Akka 2.5.3[1], so we can support communication between workers (Fig. 6c). Fractal is open-source and its source code is publicly available[2].

4.1 Memory-efficient Subgraph Processing

Many graph pattern mining (GPM) algorithms suffer from a combinatorial explosion of the search space, often requiring the maintenance of large intermediate state (memory) and thereby overloading the underlying system. This is the case of current systems since they adopt a BFS-style subgraph enumeration to balance the work at the end of each synchronization step [45, 53].

Motivating example. To illustrate this issue, we estimate the amount of memory necessary to keep all vertex-induced subgraphs occurring in the medium-sized Mico network [17]. We consider that each subgraph can be represented solely by its vertices (with no memory overheads), *i.e.*, `NumberOfVertices` × `BytesPerVertex`. The memory requirements quickly become unbearable for subgraphs with four or five vertices, resulting in demands of 163.27*GB* and 46.37*TB*, respectively.

Solution. Fractal avoids the need to maintain intermediate state by (1) enumerating subgraphs with a depth-first search algorithm and (2) recomputing the subgraphs from scratch after a synchronization point. This is possible due to Fractal's computational model, designed to support multiple primitive components in a single Fractal step. Specifically, Fractal uses a data structure called `SubgraphEnumerator` (Fig. 7). Each enumerator is identified by an enumeration prefix, which represents the current subgraph under extension process. Extensions candidates of this subgraph are generated with `computeExtensions()`. If the prefix is empty, then this function generates the set of vertices or edges of the input graph (according to the fractoid's type and the respective core). In Fractal, an enumerator is consumed once the `extend()` procedure is processed, which updates the next enumerator with the current subgraph augmented by one pre-computed extension.

```
class SubgraphEnumerator {
  val prefix: Subgraph
  def computeExtensions(): Unit
  def extend(): SubgraphEnumerator
}
```

Figure 7: Subgraph enumerator: a data structure designed to support transparent subgraph enumeration.

[1]https://akka.io
[2]http://github.com/dccspeed/fractal

Algorithm 1 DFS-PROCESSING(*step*)

```
1: $senum \leftarrow$ CREATE-SUBGRAPH-ENUMERATOR()
2: PROCESS($senum$, $step$, 0)
3: function PROCESS($senum$, $step$, $idx$)
4:     $p \leftarrow step[idx]$; $sg \leftarrow senum.prefix$
5:     if IS-EXTENSION($p$) then                                ▷ E
6:         $senum.computeExtensions()$
7:         while not EMPTY($senum$) do
8:             PROCESS($senum.extend()$, $step$, $idx + 1$)
9:     else if IS-FILTER($p$) and FILTER($sg$) then             ▷ F
10:        PROCESS($senum$, $step$, $idx + 1$)
11:    else if IS-AGGREGATION($p$) then                         ▷ A
12:        AGGREGATE(KEY($sg$), VALUE($sg$))
```

Algorithm 1 presents a DFS-based method executed by each core to process subgraphs in Fractal. Its input is a *Fractal step*, *i.e.*, a sequence (array) of pipelined primitives to be executed. The algorithm initiates by creating an empty subgraph enumerator, which is given as parameter to the function *process* (lines 1-2). This function (lines 3-12) applies the primitives over the subgraph enumerators recursively, reusing the data structures on each enumeration level. For example, the first primitive is indexed by zero in the *step* array. In case of extension (lines 4-8), the algorithm invokes the method recursively for each possible extension of the current subgraph. In case of filtering (lines 9-10), we only call the process function pointing to the next primitive if subgraph passes the filter. Finally, the algorithm handles the aggregation according to the user's reduction function (lines 11-12), which marks the end of the recursive call.

Algorithm 2 FROM-SCRATCH-EXECUTION(*fractoid*)

```
1: $workflow \leftarrow$ PRIMITIVES($fractoid$)
2: $steps \leftarrow$ ARRAY(); $step \leftarrow$ ARRAY(); $idx \leftarrow 0$
3: while $idx <$ LEN($workflow$) do
4:     $p \leftarrow workflow[idx{+}{+}]$
5:     if $idx =$ LEN($workflow$) or IS-SYNC-POINT($p$) then
6:         ADD($steps$, COPY($step$))
7:     ADD($step, p$)
8: for $step$ in $steps$ do
9:     DFS-PROCESSING($step$)
10: function IS-SYNC-POINT($p$)
11:     return IS-GLOBAL-FILTER($p$) and $p.aggregation()$
```

Our subgraph enumeration strategy is memory-efficient due to its DFS-style and the reuse of the subgraph enumerator on each enumeration level. However, many GPM applications (*e.g.*, FSM) require multiple global aggregations, making impracticable the use of Algorithm 1 directly. In such scenarios, Fractal recomputes the primitives from scratch to reduce the intermediate state of the GPM applications. Algorithm 2 shows how Fractal splits the fractoid workflow into steps and submits them for execution. The input of FROM-SCRATCH-EXECUTION is a fractoid, containing the sequence of primitives used to build the steps (line 1). Primitives are assigned to the current step until a synchronization point, which is marked by a filter reading from an aggregation (W_4) that is not yet computed (lines 10-11) or the end of the workflow (line 3). Whenever a primitive matches a synchronization point, the algorithm adds a copy of the current step to the pool of steps and proceeds accumulating other primitives (line 6). Thus, steps always accumulate computation from their ancestors: primitives in steps $\{0, \ldots i-1\}$ also belong to step i. Finally, the algorithm calls the enumeration procedure (Alg. 1) for each step built previously (lines 8-9). We highlight that aggregation results from W_4 operators are not recomputed: the execution engine reuses their results on every subsequent step once they are computed.

An important advantage of the proposed solution is that Fractal will not crash due to *out-of-memory* errors and, hence, more memory is available for the user-defined aggregations. Also, we avoid the cost of accessing precomputed subgraphs, including potentially out-of-core accesses. However, two potential concerns regarding our solution are: (1) the cost of recomputing the subgraphs from scratch and (2) it can lead to imbalance among workers. In fact, trading off memory for redundant processing is beneficial since the cost of enumerating subgraphs (during the combinatorial explosion phase of the algorithm) will dominate the execution time of a GPM task. The load imbalance among workers is addressed next.

4.2 Near-optimal Load Balancing

Although pipelining is a powerful feature for big-data workloads [64], for graph mining it can be tricky due to the irregular nature of degree distributions in real graphs. In such scenarios, load-balancing becomes key to achieve both good performance and resource utilization since standard pipelining levers task independence to coordinate parallelism.

Motivating example. Figure 8 shows the resource utilization (CPU) of Fractal when we employ a simple pipeline on a single machine with 28 cores for an application that finds all 4-cliques. Each core initially takes a partition of the graph vertices and enumerates all 4-cliques rooted by vertices in their respective partition. A critical scalability issue is observed: the resource utilization drops very quickly as some cores finish their work early, while others keep running as stragglers for a long time (long tail).

Solution. In this work, we propose a **hierarchical work stealing** strategy for dynamic work balancing, improving the resource utilization of the underlying system. Our strategy is composed of two levels the first one focuses on communication within cores of the same worker (*internal work stealing*, or WS_{int}), and the second focuses on coordination across cores of different workers (*external work stealing*, or WS_{ext}). Naturally, thread communication within a single worker (internal) is more efficient, since access to the memory is (often) shared. On the other hand, inter-process communication is expensive because it involves serializing, sending, receiving and deserializing data structures, as we will see next. Thus, WS_{int} is always preferred to WS_{ext}.

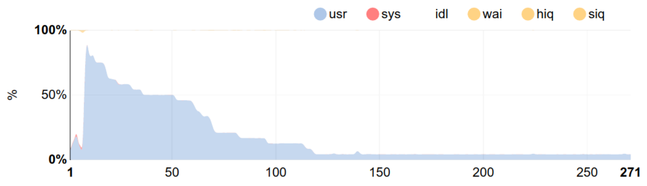


Figure 8: Subgraph enumeration without any work balancing: CPU is not well utilized due to skewness.

We implement work stealing directly over the subgraph enumerator abstraction (see Fig. 7). In particular, we make the extension function (`extend()`) thread-safe and efficient, to allow a fine-grained work sharing among execution cores. Upon an `extend()` call, Fractal copies the subgraph prefix, consumes an extension (thread-safe), and adds the new extension to the prefix of the new enumerator. Because we end up with a very short critical section (consumption of extensions), work stealing in Fractal comes with a small overhead and little contention among execution threads. Indeed, the depth-first enumeration maintains one enumerator per extension level, which can be locked and consumed independently. Subgraph enumerators also facilitate work sharing among distributed workers: a subgraph enumerator (prefix) represents a unique independent piece of work that can be shipped to any worker for processing.

For example, consider the execution state presented in Figure 9, where subgraphs are enumerated vertex-by-vertex from the graph of Figure 1 in parallel. The four available cores (c's) are organized in two workers (w's) and all cores finished their original assigned work, except for c_0. In case (a), c_1 can accommodate a WS_{int} by extending the second subgraph enumerator from c_0, since both belong to w_0. Hence, such operation generates a new subgraph enumerator with the prefix composed of edge (v_0, v_2) in c_1, ceasing its idleness and mitigating imbalance. In case (b), c_2 triggers a WS_{ext} because no core in w_1 has work to share. Thus, c_2 sends a work stealing request to w_0, which in turn forwards the request to c_0. A separate thread in w_0 is then responsible for extending the first non-empty c_0 enumerator and shipping this piece of work back to the requester c_2 at w_1. Such procedure fills

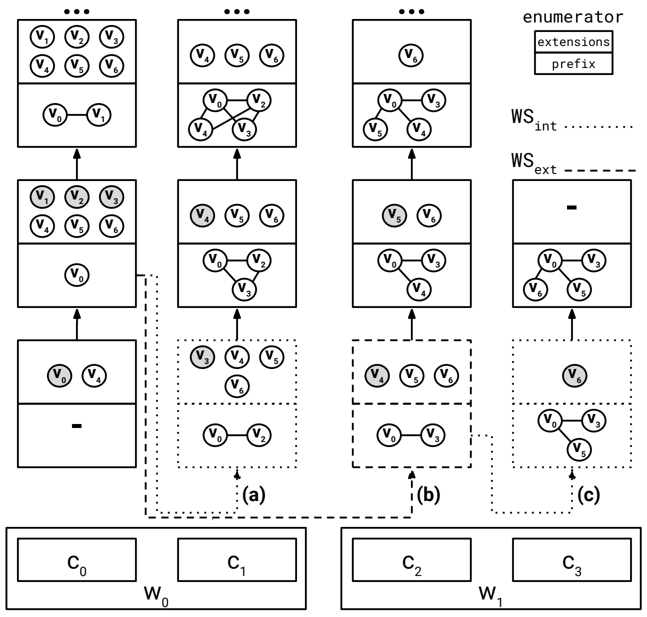


Figure 9: Work stealing happens (a)(c) internally among cores of the same worker or (b) externally among cores of different workers. To reduce network communication, we use remote work stealing only when no other local core has work to share.

c_2 with a new enumerator with a prefix composed by edge (v_0, v_3). Finally, in case (c), c_3 leverages the previous external request to perform a low-cost WS_{int} with c_2. The result is a new enumerator with prefix composed of path (v_0, v_3, v_5).

As we show in Sec. 5.2.2, our method is adaptive to different workload characteristics and does not assume anything about the input distribution. Also, we verify empirically the importance of the two levels of work stealing and show significant gains on CPU utilization and system's performance.

4.3 Fast Enumeration via Graph Reduction

Exploratory routines over graph data often exhibit locality during processing: the working set of visited vertices and edges is reduced or it shrinks as the algorithm progresses. Also, because the cost of graph processing engines is directly related to the input size, being able to work with just the essential regions of the graph can significantly reduce the overall cost of such computations. This is the case of subgraph querying, keyword search, and other search algorithms.

Motivating example. Consider the keyword search problem over a knowledge graph derived from Wikidata [18], and queries $Q_1 = \{paris, revolution, author\}$ and $Q_2 = \{tom, cruise, drama\}$. First, we execute these queries on the original graph G. Then, we execute the same queries on a reduced graph G', built by keeping only those vertices and edges that are associated to, at least, one query keyword. We compare the execution on G against the execution on G', in terms of the reduction of the input graph and the extension cost (EC) associated with the queries. Specifically, EC represents the *number of tests* performed to determine the set of candidate subgraph extensions throughout the execution of the query. The outcome is a significant reduction in the extension cost performed by Fractal. In particular, for Q_1 we may see reductions of 54.97% and 65.27% in the number of vertices and edges, respectively. More important, such pruning also reduced the extension cost (EC) by 92.54%. We observe similar results for Q_2, with an impressive cost reduction of 99.87%.

Solution. Fractal supports a novel **graph reduction** procedure that allows an user to specify a reduced graph for downstream processing. The graph reduction procedure is only applied between two fractal steps (at the synchronization point) to minimize overheads and design complexity. Formally, the goal of graph reduction is to specify a reduced graph G_i for each step i, based on the subgraphs enumerated during the previous step ($\mathcal{S}_{i-1}$). This can be represented as a recurrence relation (Equation 1), where functions f_i are algorithm-specific and provided by the user.

$$G_i = \begin{cases} f_i(\{\}) & i = 0 \\ f_i(\mathcal{S}_{i-1}) & i > 0 \end{cases} \tag{1}$$

Fractal exposes the graph reduction operators (see Fig. 10) to the user via the filter function (f_i).These special operators are called from a fractal graph in order to filter vertices (R_1) and/or edges (R_2). For example, before querying the input graph, the user can use these operators to filter vertices and edges that should not belong to any of the subgraph results. Note that many GPM applications can leverage this procedure **transparently**. For instance, Frequent Subgraph Mining (FSM) can use such a procedure to ensure that only the vertices and edges that actively have participated in at least one subgraph of the previous fractal step are kept in the next reduced graph. In such cases, at each step, Fractal keeps track of the subset of extensions necessary to accomplish the previous step re-computation, transferring the burden from the user to the system.

```
R1  def vfilter(f: (v: Vertex, g: Graph) => Boolean)
R2  def efilter(f: (e: Edge, g: Graph) => Boolean)
```

Figure 10: Graph reduction operators are used to filter the input graph to reduce subgraph enumeration cost.

5 EXPERIMENTAL RESULTS

All experiments, unless otherwise specified, were run on a cluster with 10 machines, each one having an Intel Xeon E52680 with hyperthreading (14 cores, 28 execution threads) and 25 MB cache, 500 GB RAM, running CentOS Linux 3.10. The machines were connected by Gigabit Ethernet.

Table 1: Graphs used for evaluation.

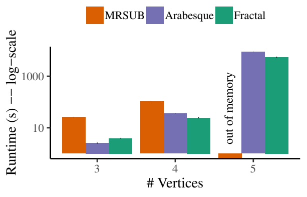
Graph (G)	|V(G)|	|E(G)|	|L(G)|	**Density**
Mico	100K	1.08M	29	2.1×10^{-4}
Patents	2.74M	13.96M	37	3.7×10^{-6}
Youtube	4.58M	43.96M	80	4.1×10^{-6}
Wikidata	15.51M	18.55M	2,569	1.5×10^{-7}

Datasets. In Table 1 we describe the graphs used in our evaluation. Note that such datasets were also used in previous works in order to evaluate graph mining algorithms and systems [1, 17, 53]. In Mico [17], vertices are authors (labeled with their research field) and edges represent co-authorship. Patents [25] has patents published in US as vertices and their citations as edges; the labels on vertices are given by the year in which the patents were released. Youtube [11] contains videos posted from February 2007 to July 2008. In this graph, there is an edge between two vertices if their videos are related. The label of a vertex is computed by combining the video's rating and length. Finally, the Wikidata graph (≈4M unique keywords) used in this work was derived from a knowledge base [58]. Such network models subjects and objects as vertices and it uses predicates as edges. Edge labels represent different types of predicates. Moreover, vertices and edges are associated with a set of keywords. Throughout this section we refer to these graphs by their name followed by a suffix indicating whether that specific version is single-labeled (-SL) or multi-labeled (-ML). Confidence intervals are presented for a confidence of 95%.

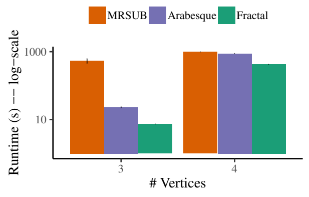
JVM-based Baselines. We compare Fractal (a JVM-based system) with several specialized JVM-based distributed algorithms including those for Motifs (MRSUB [47]), subgraph querying (SEED [33]) and Cliques (QKCount [19]). We also compare Fractal with general-purpose JVM-based systems, such as Arabesque [53], and GraphFrames [13] where possible. We were unfortunately unable to compare it to NScale [45] (the code is not public and the authors were unable to provide us with their code). The GPM kernels used are implementations of the problems described in Section 2.2.

5.1 Fractal: Comparative Performance

Motifs. Figure 11 compares the performance of Fractal with baselines (Arabesque and MRSUB) on the Motifs benchmark. Considering the single-labeled input graphs (Mico-SL and Youtube-SL), we observe that when the amount of work is small, Arabesque outperforms Fractal (see Mico-SL with 3-vertex motifs). Fractal pays a small *setup overhead* to support its work stealing environment and such overhead becomes significant when the amount of work is small. However, Fractal becomes more efficient as we target larger subgraphs (4- or 5-node motifs) or when a larger network is involved (Youtube-SL), obtaining a speedup of up to 1.6× for Mico-SL and 3.10× for Youtube-SL. MRSUB, a recent specialized approach performs worse than the other two methods across the board (running out of memory in one instance).



(a) Mico-SL



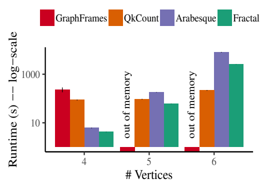
(b) Youtube-SL

Figure 11: Motifs runtime on Mico-SL and Youtube-SL.

Cliques. We evaluate the cliques application on Fractal and baselines (Arabesque, GraphFrames and QKCount) in Figure 12. Fractal outperforms Arabesque in almost every scenario. On Youtube-SL the performance gains are even more obvious (see Figure 12b). Fractal obtains speedups that range from 5.19× to 12.87× against Arabesque in all configurations considered. On this larger dataset, since Arabesque has to keep the subgraphs (compressed in ODAGs) from one step to another, this imposes extra memory and network costs to maintain that information consistent among workers. Arabesque, however, is able to mine 3-cliques faster than Fractal on Mico-SL (again due to the setup overhead).

Fractal competes well with the state-of-the-art, QKCount (a distributed algorithm for clique counting), outperforming it on many settings, while being slower on Mico-SL for cliques of size six. Fractal's mechanism to control memory pressure, efficient work stealing and its method to leverage pipelined computations and extension primitives (described in Section 3) allow it to compute cliques efficiently, without the need to keep any intermediate state.

FSM. We evaluate the performance of the FSM application implemented over Fractal, Arabesque, and ScaleMine [1], a high-performance specialized implementation. Scalemine relies on a two phase approach: in the first phase it estimates


(a) Mico-SL

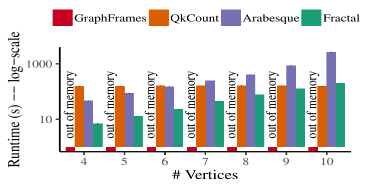
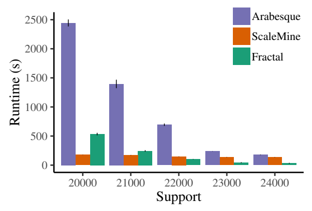

(b) Youtube-SL

Figure 12: Cliques runtime on Mico-SL and Youtube-SL. GraphFrames often ran out of memory.

search-space loads and uses that information for load balancing in the second phase. While Scalemine produces exactly the same set of frequent patterns, as Fractal and Arabesque, it does not retain the exact support counts for them (*i.e.*, the frequency counts are approximate). For this set of experiments, we consider two labeled graphs and we vary the minimum support of the algorithm (see Figure 13).


(a) Patents-ML

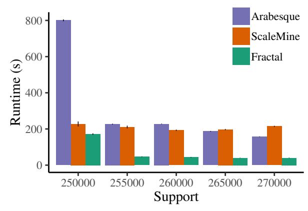

(b) Youtube-ML

Figure 13: FSM performance. Fractal's stateless characteristic allows competitive scalability with Scalemine.

While Fractal has the initial setup overhead (for work stealing), Scalemine's first phase (also used for load estimation) can be quite expensive especially when there is less overall work [1]. Fractal's stateless operation provides a better scalability against Arabesque, showing speedups of up to 4.57× (when the support is 20k). For higher values of support, it outperforms ScaleMine (in spite of being an exact algorithm) due its fast and balanced enumeration strategy, achieving a speedup of 4.12× when the support is 24k. For lower values of support, Scalemine outperforms Fractal. This is a surprisingly good result for Fractal, since Scalemine is implemented on C++ with MPI, demonstrating the effectiveness of work stealing, limiting memory pressure and graph reduction optimizations within Fractal.

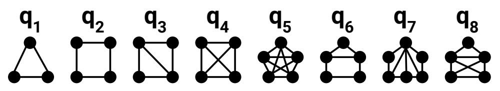
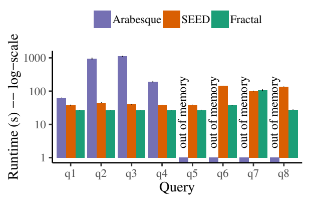


Figure 14: Queries for subgraph querying evaluation.


(a) Patents-SL

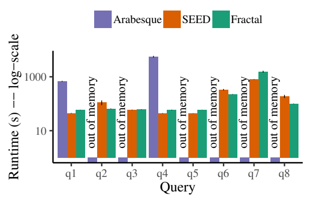

(b) Youtube-SL

Figure 15: Subgraph querying performance.

Subgraph Querying. In Figure 15, we evaluate the performance of the subgraph querying application on Fractal, SEED and Arabesque. SEED is the state-of-the-art subgraph enumeration system implemented over Hadoop, which computes larger subgraphs by joining smaller ones. We use the same queries supported by SEED [33] to evaluate our system (see Fig. 14). We implemented the same queries in Arabesque, for comparison with a general purpose approach for GPM.

Considering the Patents-SL graph (Figure 15a), SEED outperforms Fractal only for q_7, because the execution plan generated in this case is most effective. Specifically, SEED computes the matches of the pattern $q3$ and joins them to obtain q_7, reducing significantly the subgraph enumeration cost. Meanwhile, Arabesque executions finish successfully only for queries that are easier to enumerate (q_1 and q_4) or have fewer edges (q_2 and q_3). The other executions fail with out-of-memory errors since the number of subgraphs and their sizes start to pressure the memory, even for compressed representations like Arabesque's ODAGs. Fractal's pattern-induced extension and stateless enumeration allow a more efficient subgraph querying, specially compared to edge-induced approaches like Arabesque, across the board.

On Youtube-SL (Figure 15b), SEED also performs best when the query allows an execution joining plan with overlapping structures. Indeed, SEED outperforms Fractal for cliques (q_1, q_4, and q_5) and for q_7, because of that pattern's symmetry. In the remaining configurations, Fractal outperforms SEED (q_2, q_6, and q_8) or remains competitive (q_3). Overall, again Fractal demonstrates competitive performance with a state-of-the-art specialized baseline.

Keyword Search. For keyword search, due to space limitations, we report the runtime performance in Section 5.2.3.

5.2 Fractal Drilldown

Fractal's key systemic contributions (memory demand reduction, hierarchical work stealing and graph reduction) were found to be useful across a range of GPM kernels. We next drill down on some of those in turn, with specific kernels.

5.2.1 Memory footprint analysis. In this section we drill down on the memory costs of a couple of applications w.r.t the memory optimization facilitated by Fractal's computation model. Our metric is the *average memory usage* among all workers in the execution. A lower value of this metric indicates a better memory footprint, *i.e.*, less prone to *out of memory* errors or long garbage collection pauses (which may cause performance degradation and unpredictability).

We consider the following applications for this experiment: *(1) cliques*, representing applications in which the enumeration phase is the bottleneck; and *(2) motifs*, an application that not only enumerates all subgraphs up to a given depth but has to perform expensive isomorphic checks and to aggregate pattern counts. We use Arabesque as baseline for this analysis since it is the only distributed system for general-purpose GPM with source code available. Table 2 summarizes our results.

Table 2: Memory per worker.

	$\|V\|$	**Arab. (GB)**	**Frac. (GB)**	$\times$
Cliques Youtube-ML	3	22.9 ± 1.2	10.9 ± 0.1	2.1×
	4	57.5 ± 1.4	12.8 ± 0.1	4.5×
	5	117.4 ± 1.4	11.8 ± 0.1	10.0×
	6	204.3 ± 1.1	11.6 ± 0.0	17.6×
Motifs Mico-ML	3	0.2 ± 0.0	0.4 ± 0.0	0.6×
	4	1.8 ± 0.3	0.4 ± 0.0	4.9×
	5	46.9 ± 1.0	0.9 ± 0.3	49.9×

First, we consider the multi-labeled network, Youtube-ML. Fractal is able to keep the memory requirements relatively constant (range from 10.88 and 12.84 GB). Some variation is expected due to the non-deterministic behavior of the Java Virtual Machine (*e.g.*, garbage collection) in multi-threaded environments. Meanwhile, in Arabesque, another GPM system that levers JVM, we see a significant increase in the memory used by workers, which is a direct outcome of how the system keeps its intermediate state across enumeration depths. Specifically, subgraphs are kept in the memory of each worker in a compressed data structure (ODAG) per pattern [53]. As there are more patterns templates in a multi-labeled network, Arabesque must keep more ODAGs in memory, increasing the working memory of the workers. In particular, the workers of the baseline system require an average of 204.28 GB of memory in the enumeration depth of five, while Fractal needs only 11.58 GB. This represents a reduction factor of 17.64× regarding memory.

Second, we consider the Motifs application. In this example, we show that the intermediate state of workers significantly grows in the baseline system as we increase the exploration depth (even for moderately sized) graphs. Indeed, the amount of memory used by Arabesque increases 49.86×. On the other hand, Fractal's executions require no more than 0.94 GB of memory per worker (on average).

5.2.2 Hierarchical Work Stealing. In this section, we evaluate the *hierarchical work stealing* environment within Fractal. We focus on the FSM algorithm, which is a multi-step application and, consequently, has the potential to exhibit a richer per-level behavior. The input graph considered is Patents-ML and we set the support to $20k$ for this drilldown experiment.

Since our work stealing strategy is composed of two levels of balancing (*internal* and *external*), our evaluation consider four configurations: *1.Disabled*, where we disable both levels; *2.Internal*, where we enable only the internal work stealing (WS_{int}); *3.External*, where we enable only the external work stealing (WS_{ext}); and *4.Internal+External*, where we enable both levels ($WS_{int} + WS_{ext}$), representing our complete strategy. We seek to evaluate the effectiveness of each of them in mitigating imbalance. Figure 16 presents the execution times of the parallel tasks discriminated by step and scenario. The rows represent the five fractal steps and columns represent the four working stealing configurations. The y-axis is the individual runtime of each task (x-axis).

In the first configuration (*1.Disabled*), we can see the raw imbalance in load. As expected, the execution becomes more skewed for later steps, as we are enumerating bigger subgraphs (step 4 is a extreme case, for example). However, we observe a significant improvement in skew reduction across all steps when the internal work stealing is enabled (*2.Internal*). Note that, in this case, some imbalance across workers still exists since the original work is only allowed to be shared among threads in the same process. In the next configuration, we enabled only the external work stealing (*3.External*). We may see a better load balancing among the tasks, as each one has more options to steal work, but the communication overhead of work requests increases the execution time in comparison with the internal work stealing alone (*2.Internal*). Finally, the configuration which combines both strategies (*4.Internal+External*) results in near perfect load balancing as well as reduces the communication overhead in Fractal.

5.2.3 Graph Reduction. We evaluate the effectiveness of the graph reduction in Fractal. Our goal is to compare our implementation performance over the original graph (G) against the performance over the reduced graph (G_0), obtained by removing vertices and edges that do not contain any of the query keywords. We focus on the following evaluation queries [16]: $Q_1 = \{woody, allen, romance\}$, $Q_2 = \{mel, gibson, director\}$, $Q_3 = \{classic, fantasy, funny, author\}$, and $Q_4 = \{author, classic, award\}$. Figure 17 shows the runtime (log-scale) as we vary the number of cores in the executions. The results for queries Q_1 and Q_2 are presented in pairs (one

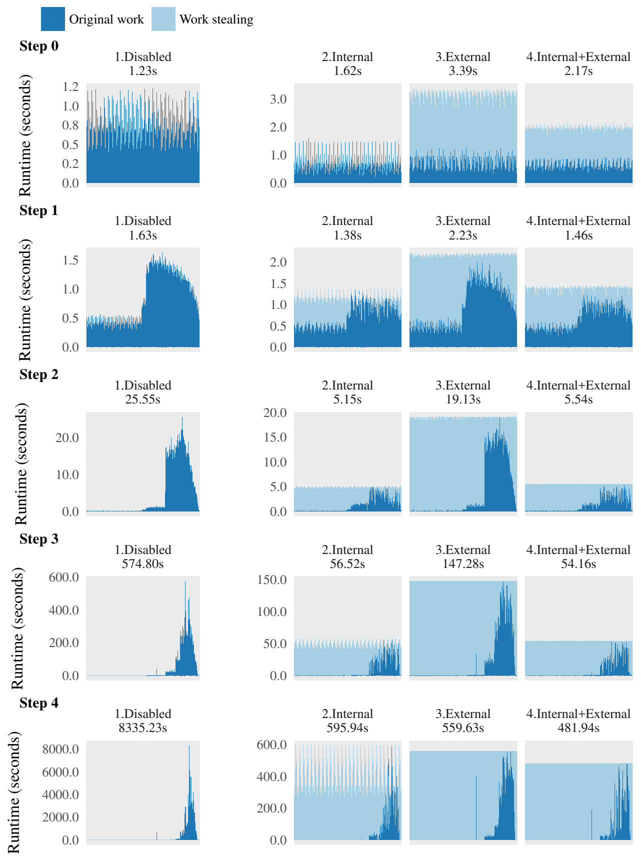


Figure 16: Work stealing evaluation. Imbalance becomes evident with load balancing strategies disabled *(1.Disabled)*. Internal work stealing allows a good intra-worker load balancing at low communication cost *(2.Internal)*. External work stealing allows an efficient load balancing at a higher communication overhead *(3.External)*. Applying both strategies gives the best trade-off between load balancing and communication overhead *(4.Internal+External)*. Times of each step are noted on top of each chart.

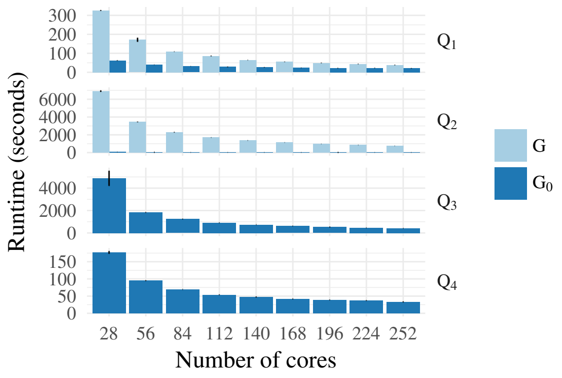


Figure 17: Graph reduction benefits (G_o), for keyword search. For Q_1 and Q_2, graph reduction is effective in reducing the input graph (particularly in Q_2 where runtime is reduced drastically), and scales well. For Q_3 and Q_4, we only show executions with graph reduction enabled since the standard alternative timed out.

for each query) and indicate whether the graph reduction optimization is used. The execution of the queries Q_3 and Q_4 without the optimization did not terminate within a time limit of four hours (14,400s) - only the results with the graph reduction optimization is included (most of which take at most a few 100 seconds or less). Overall we see anywhere from one to two orders or magnitude improvement in runtime for keyword search queries using the reduced graph.

We may see significant benefits from graph reduction for queries Q_1 and Q_2. However, the performance and effectiveness of the graph reduction optimization depends on the query being executed. For instance, the subgraph extension cost for Q_1 is reduced by 4.5× using the graph reduction optimization (see Section 4.3) which is substantial, but not as much as for Q_2. The performance improvement obtained with Q_2 means that Q_2's subgraphs lie in less-dense regions, which reduces the number of subgraphs enumerated by Fractal (extension cost was 77.96× lower). On the other hand, queries that match often in dense regions of the graph are prone to encounter several invalid extensions that are not relevant to the query and degrade performance.

Now, we evaluate the scalability of Fractal as we increase the number of cores for Q_3 and Q_4. Note that, these queries present a heavier workload compared to the previous two. Regarding Q_3, we observe a near perfect speedup, having a extension cost of approximately $1.5T$. In addition, the execution of query Q_4 presents an extension cost of approximately $46B$, while maintaining an efficiency of over 60%.

5.2.4 COST analysis. Motivated by McSherry et al. [38], we evaluate Fractal against state-of-the-art single-thread graph mining algorithms in terms of the COST metric. The COST is defined as the number of execution threads a system needs to outperform an efficient single-thread implementation. We lever recent single threaded JVM based implementations for this purpose. For the Motifs, Cliques and Graph Querying (q_2 and q_3) kernels we use Gtries [15, 46]. For FSM we use Grami [17]. Representative results are reported in Figure 18 and additional results for cliques and triangles are discussed, along with other baselines (e.g. Neo4j) in Appendix C.

One may observe that the COST typically range from 3-4 threads. For instance, Fractal beats Gtries for motifs (36.5k seconds) with 3 cores (≈ 30k seconds). Fractal outperforms both Gtries for cliques (2416s) and Grami (1154s) when using

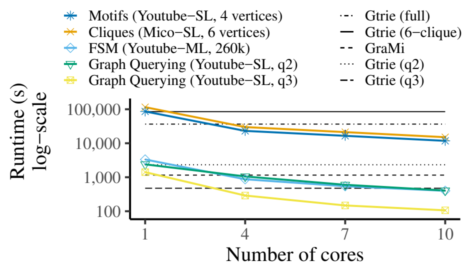


Figure 18: COST analysis: number of cores that Fractal needs to reach state-of-the-art single-thread methods.

4 cores, taking 844s and 872s respectively. Finally, Fractal outperforms the baseline for Graph Querying in both queries: the baseline evaluates q_2 in 2328s against 1055s for Fractal with 4 threads and q_3 in 474s against 289s for Fractal also with 4 threads. These numbers are representative for a large majority of our experimental settings. The positive exceptions to this (lower COST) arise in long-running tasks dominated by enumeration computations – here we see COST values as low as 2 threads (*e.g.* motifs on Mico). The negative exceptions (higher COST) arise when overheads dominate due to short duration tasks. For example, with the 3-cliques counting application on Youtube the COST value blows up to 16 threads (of Fractal). The overheads associated with initialization, actor set up and thread management cause this blowup. We discuss these issues in Section 6.

5.2.5 Scalability. We review the strong scalability of Fractal on four of our most time-consuming kernels (see Figure 19). We observe that if sufficient work exists the efficiency of Fractal is reasonable when compared to a single node (28-thread) implementation. For motifs kernel, Fractal achieves around 85% efficiency. For cliques kernel, Fractal achieves around 90% parallel efficiency on Mico-SL and Youtube-SL. The efficiency for motifs and cliques is higher because enumeration dominates the cost of these applications. For FSM, a challenging task for most graph systems (due to the number of aggregations and data transfers required), we observe around 75% parallel efficiency except when there is insufficient work (Youtube-ML,support:255K). For subgraph querying, the efficiency depends on the query: patterns that are harder to enumerate (q_6) achieves better performance in Fractal – around 80% efficiency – than more symmetric and dense ones (q_2, q_7) – around 65% efficiency. Aggregations in the latter two applications lead to increased data movement costs, limiting parallel efficiency.

6 OVERHEADS AND LIMITATIONS

In this section we briefly review where each optimization may not pay off, including overhead costs.

Constant memory demand. This optimization does not pay off when available memory on the machine exceeds the needs of the application. In such cases there is no need to recompute as one may process directly by maintaining relevant embedding lists. We observed this in several short-duration tasks. For example, small motifs (Figure 11), and triangles on smaller datasets (Appx. C). Note that the overhead cost associated with enabling this optimization is negligible but it can lead to significant load imbalance (see Sec. 4.2). We next discuss the overheads associated with the work stealing component of Fractal which seeks to alleviate this problem.

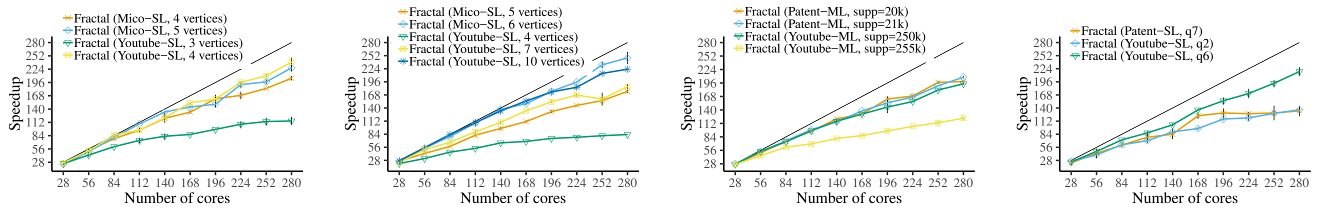
Work stealing. We use a low-overhead profiler[3] to monitor Fractal executions and measure the time spent on work stealing related code. Our experiments consider several algorithms and number of workers. We find that the overhead of work stealing is about 1.05%, with standard error of 0.44%. In terms of the initialization cost of the actor system, we observe this typically takes about one to two seconds. Such overheads impact especially the performance of executions with reduced amount of work. For instance, see Fig. 11 the 3-node motifs case.

Graph reduction. This optimization is effective in applications exploiting some local properties from the input graph, such as graph querying. In cases where the target subgraph instances occur in dense regions of the input graph, graph reduction can only reduce the input graph itself but not the magnitude of subgraph enumerations which often is the dominant cost. For instance, consider extracting k-cliques from Mico-SL and from its reduced version composed of only the vertices and edges occurring in at least one k-clique. While the reduction itself is substantial – at least 29.09% and 75.28% less vertices and edges, respectively –, the extension cost (EC) (which dominates computation time) remains unchanged. The net reduction in computation time is a negligible 0.5%, accounting for overhead costs of about 1%.

7 RELATED WORK

A popular distributed fault-tolerant system, Pregel [37], offers a “think like a vertex" (TLV) programming paradigm, which simplifies the design of graph analytics algorithms (e.g. Pagerank, HITS, belief propogation and shortest path) [6, 21, 31]. Over the past years many optimizations and variants of Pregel's TLV model have been proposed [22, 51, 54, 61]. Some of these systems [51, 54] present a subgraph-centric model, but it is not transparent to the users and subgraphs are used to reduce data communication among machines. Other matrix-inspired cloud-based graph processing systems such as System-ML [5], PEGASUS [30] and GBASE [29] has also been examined. However, in such systems every

[3]https://github.com/jvm-profiling-tools/async-profiler



(a) Motifs **(b)** Cliques **(c)** FSM **(d)** Subgraph Querying

Figure 19: Fractal scalability.

iteration typically requires a full matrix operation, which may be overkill for many applications (e.g. keyword search) where only a small part of the graph needs to be active (Section 5.2.3). Adapting such systems for graph pattern mining problems such as FSM and clique listing is non-trivial.

Single-machine systems such as Galois [40], GraphChi [32], Ligra [49] are tightly integrated to the underlying architecture. However, these efforts lack the ability to process iterative problems that accrue large intermediate state and **none** support the central primitives for GPM algorithms (*e.g.* frequent graph mining) or work in a heterogeneous, live environment, with machine downtimes (fault tolerance).

There have indeed been algorithmic advances or specialized frameworks for each of the applications considered, like graph pattern matching [2, 33, 48, 65], graph motif extraction [52], frequent subgraph mining [1, 4, 26, 35], and also RDF and keyword search related problems [16, 27]. While many of those frameworks are highly efficient for their individual application domain, none of them, to our knowledge, generalize and support different types of applications (see comparison with Scalemine [1] in Figure 13).

Arabesque [53] and NScale [45] represent the first generation of general purpose distributed systems that operate on a subgraph-centric programming model for graph processing. NScale (built on Hadoop) was not designed to handle FSM and related GPM problems, and it is unclear if it can scale to problems that generate large intermediate state (potentially overwhelming the Hadoop File System). On the other end of the spectrum, G-Miner [10] is a brand new MPI-based C++ framework for graph pattern mining and subgraph exploration. Unlike the above systems and Fractal, G-Miner does not focus on programmer productivity and cannot tolerate machine downtime. Indeed, the code for triangles counting in G-miner has 192 lines of C++, while it may be described in Fractal with 3 lines only (3-cliques).

8 CONCLUSIONS

We present a novel system (Fractal) to support various graph pattern mining and matching algorithms in a distributed setting. Fractal relies on a simple computational model and a flexible, expressive and composable API (expressing key primitives) to enhance programmer productivity. Fractal's system architecture employs memory optimizations, a novel graph reduction strategy coupled with an integrated hierarchical work stealing environment, supporting irregular graph computations efficiently on a modern data center. Results validate the effectiveness of Fractal over general purpose systems as well as over specialized algorithmic frameworks on a wide range of graph pattern mining kernels.

Fractal is under continuous improvement. On the system architecture front we are examining ways to improve its scalability via adaptive system-algorithm co-designs for such GPM kernels [8]. We plan to examine if data placement strategies for partitioning the input graph [27, 60] can help in this context. We also plan to support dynamic graphs [43].

9 ACKNOWLEDGMENTS

This work is partially funded by Fapemig, CNPq, CAPES, and by projects InWeb (MCT/CNPq 573871/2008-6), INCT-Cyber (MCT/CNPq 465714/2014-5), MASWeb (FAPEMIG-PRONEX APQ-01400-14), EUBra-BIGSEA (H2020- EU.2.1.1 690116, Brazil/MCTI/RNP GA-000650/04), and EUBra-Atmosphere (H2020-EU.2.1.1 777154) as well as grants from the US National Science Foundation: CNS-1513120; CCF-1629548; CCF-1747447; a grant from the Ohio Supercomputing Center - PAS0166; and gifts from Adobe Research and Google. The authors would like to thank G. Siganos, S. Blanas and Y. Wang, for their constructive feedback on this work. The first author would also like to thank the Ohio State University (OSU) for sponsoring a pre-doctoral visit to OSU, where some of this work was initiated. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the corresponding funding agencies or institutions.

REFERENCES

[1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. Scalemine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph. In *Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC ’16)*.

[2] Foto N Afrati, Dimitris Fotakis, and Jeffrey D Ullman. 2013. Enumerating subgraph instances using map-reduce. In *Data Engineering (ICDE), 2013 IEEE 29th Int. Conf. on*. IEEE, 62–73.

[3] Austin R. Benson, David F. Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* (2016).

[4] Mansurul A Bhuiyan and Mohammad Al Hasan. 2015. An iterative MapReduce based frequent subgraph mining algorithm. *Trans. on Knowl. and Data Engineering* (2015).

[5] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.* (2016).

[6] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer networks and ISDN systems* (1998).

[7] Björn Bringmann and Siegfried Nijssen. 2008. What is Frequent in a Single Graph?. In *Pacific-Asia Conf. on Advances in Knowl. Discovery and Data Mining (PAKDD’08)*.

[8] Gregory Buehrer, Srinivasan Parthasarathy, and Yen-Kuang Chen. 2006. Adaptive Parallel Graph Mining for CMP Architectures. In *Int. Conf. on Data Mining*. 97–106.

[9] E. Bullmore and O. Sporns. 2009. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience* (2009).

[10] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *EuroSys Conf.* ACM, 32.

[11] Xu Cheng, Cameron Dale, and Jiangchuan Liu. [n. d.]. Dataset for “Statistics and Social Network of YouTube Videos”. http://netsg.cs.sfu.ca/youtubedata/. ([n. d.]).

[12] Maximilien Danisch, Oana Denisa Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs. In *WWW*.

[13] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. 2016. GraphFrames: An Integrated API for Mixing Graph and Relational Queries. In *Int. Workshop on Graph Data Managem. Exp. and Sys. (GRADES ’16)*. ACM, New York, NY, USA.

[14] Imre Derényi, G Palla, and Tamás Vicsek. 2005. Clique Percolation in Random Networks. *Physical Review Letters* (2005).

[15] Ahmad Naser eddin and Pedro Ribeiro. 2017. Scalable Subgraph Counting Using MapReduce. In *Symp. on Applied Computing (SAC ’17)*.

[16] Shady Elbassuoni and Roi Blanco. 2011. Keyword Search over RDF Graphs. In *Int. Conf. on Information and Knowl. Managem. (CIKM ’11)*.

[17] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GRAMI: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.* (2014).

[18] Fredo Erxleben, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandečić. 2014. Introducing Wikidata to the Linked Data Web. In *Int. Semantic Web Conf. (LNCS)*. Springer.

[19] Irene Finocchi, Marco Finocchi, and Emanuele G Fusco. 2014. Clique counting in MapReduce: theory and experiments. *arXiv preprint arXiv:1403.0734* (2014).

[20] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping Out Garbage Collection from Big Data Systems. In *Workshop on Hot Topics in Operating Systems*.

[21] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory. In *Symp. on Discrete algorithms*.

[22] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *Symp. on Operating Systems Design and Implementation*.

[23] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Symp. on Operating Systems Design and Implementation*.

[24] Joshua A Grochow and Manolis Kellis. 2007. Network motif discovery using subgraph enumeration and symmetry-breaking. In *RECOMB*.

[25] Hall B. H., A. B. Jaffe, and M. Trajtenberg. 2001. The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools. http://www.nber.org/patents/. (2001).

[26] Steven Hill, Bismita Srichandan, and Rajshekhar Sunderraman. 2012. An iterative MapReduce approach to frequent subgraph mining in biological datasets. In *Conf. on Bioinformatics, Computational Biology and Biomedicine*.

[27] Jiewen Huang, Daniel J Abadi, and Kun Ren. 2011. Scalable SPARQL querying of large RDF graphs. *Proc. VLDB Endow.* (2011).

[28] Eslam Hussein, Abdurrahman Ghanem, Vinicius Vitor dos Santos Dias, Carlos H.C. Teixeira, Ghadeer AbuOda, Marco Serafini, Georgos Siganos, Gianmarco De Francisci Morales, Ashraf Aboulnaga, and Mohammed Zaki. 2017. Graph Data Mining with Arabesque. In *Int. Conf. on Managem. of Data (SIGMOD ’17)*.

[29] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. 2012. Gbase: An Efficient Analysis Platform for Large Graphs. *The VLDB Journal* (2012).

[30] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2009. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *Int. Conf. on Data Mining (ICDM)*.

[31] Jon M Kleinberg. 1999. Authoritative sources in a hyperlinked environment. *J. ACM* (1999).

[32] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *USENIX Conf. on Operating Systems Design and Implementation (OSDI’12)*.

[33] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable distributed subgraph enumeration. *Proc. VLDB Endow.* (2016).

[34] Yutaka I. Leon-Suematsu, Kentaro Inui, Sadao Kurohashi, and Yutaka Kidawara. 2011. Web Spam Detection by Exploring Densely Connected Subgraphs. In *Int. Conf. on Web Intelligence and Intelligent Agent Technology (WI-IAT ’11)*.

[35] Wenqing Lin, Xiaokui Xiao, and Gabriel Ghinita. 2014. Large-scale frequent subgraph mining in mapreduce. In *Int. Conf. on Data Engineering*. IEEE.

[36] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. 2015. Trash Day: Coordinating Garbage Collection in Distributed Systems. In *Workshop on Hot Topics in Operating Systems*.

[37] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Int. Conf. on Managem. of Data*.

[38] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! But at what COST. In *Workshop on Hot Topics in Operating Systems*.

[39] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* (2002).

[40] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Symp. on Operating Systems Principles (SOSP ’13)*.

[41] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *Symp. on Operating Systems Design and Implementation*.

[42] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*.

[43] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. 2017. Motifs in Temporal Networks. In *Int. Conf. on Web Search and Data Mining (WSDM '17)*.

[44] Nataša Pržulj. 2007. Biological network comparison using graphlet degree distribution. *Bioinformatics* (2007).

[45] Abdul Quamar, Amol Deshpande, and Jimmy Lin. 2016. NScale: Neighborhood-centric Large-scale Graph Analytics in the Cloud. *The VLDB Journal* (2016).

[46] Pedro Ribeiro and Fernando Silva. 2014. G-Tries: A data structure for storing and finding subgraphs. *Data Mining and Knowl. Discovery* 28, 2 (2014).

[47] Saeed Shahrivari and Saeed Jalili. 2015. Distributed Discovery of Frequent Subgraphs of a Network Using MapReduce. *Computing* (2015).

[48] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *Int. Conf. on Managem. of Data*. ACM, 625–636.

[49] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Symp. on Principles and Practice of Parallel Programming (PPoPP '13)*.

[50] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. https://doi.org/10.1109/MSST.2010.5496972

[51] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conf. on Parallel Processing*. Springer.

[52] Gomonoeorge M Slota and Kamesh Madduri. 2014. Complex network analysis using parallel approximate motif counting. In *Int. Parallel and Distributed Processing Symp.*

[53] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A System for Distributed Graph Mining. In *Symp. on Operating Systems Principles (SOSP '15)*.

[54] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.* 7, 3 (2013).

[55] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. 2009. Doulion: counting triangles in massive graphs with a coin. In *Int. Conf. on Knowl. Discovery and Data Mining*.

[56] Johan Ugander, Lars Backstrom, and Jon Kleinberg. 2013. Subgraph Frequencies: Mapping the Empirical and Extremal Geography of Large Graph Collections. In *Int. Conf. on World Wide Web (WWW '13)*.

[57] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990).

[58] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: A Free Collaborative Knowl.base. *Commun. ACM* (2014).

[59] A. VÃązquez, R. Dobrin, D. Sergi, Oltvai Eckmann, J.-P., Z. N., and A.-L. BarabÃąsi. 2004. The topological relationship between the large-scale attributes and local interaction patterns of complex networks. *Proc. of the National Academy of Sciences of the United States of America* (2004).

[60] Ye Wang, Srinivasan Parthasarathy, and P. Sadayappan. 2013. Stratification driven placement of complex data: A framework for distributed data analytics. In *Int. Conf. on Data Engineering*.

[61] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow.* 7, 14 (2014).

[62] Xifeng Yan and Jiawei Han. 2002. gSpan: Graph-based substructure pattern mining. In *Int. Conf. on Data Mining*.

[63] Jaewon Yang and Jure Leskovec. 2015. Defining and Evaluating Network Communities Based on Ground-truth. *Knowl. Inf. Syst.* (2015).

[64] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *USENIX Conf. on Networked Systems Design and Implementation (NSDI'12)*.

[65] Zhao Zhao, Guanying Wang, Ali R Butt, Maleq Khan, VS Anil Kumar, and Madhav V Marathe. 2012. Sahad: Subgraph analysis in massive networks using hadoop. In *Parallel & Distributed Processing Symp.* IEEE.

A APPLICATIONS

Here, we discuss the implementation details of the evaluated applications using Fractal's API (Sec. 3.1).

Motifs (Listing 1). The algorithm establishes that the subgraphs will be induced by vertices, by calling `vfractoid` (line 1). Next, we expand the initial empty fractoid by generating all unique subgraphs with k vertices using the call `expand(k)`. Then, we use the `aggregate` operator to configure an aggregation represented by pairs of *pattern::count*. More specifically, we must specify the aggregation key (line 3), the initial value one (line 4) and the reduction function (line 5). Lastly, the actual mapping containing the motifs and their counts is obtained by calling `aggregation` in line 6.

```
1  val motifs = graph.vfractoid.expand(k).
2   aggregate [Pattern,LongWritable] ("motifs",
3     (subg,comp,value) => { subg.pattern },
4     (subg,comp,value) => { value.set(1); value },
5     (value1,value2) => { value1.set(value1.get +
          value2.get); value1 }
6   ).aggregation [Pattern,LongWritable] ("motifs")
```

Listing 1: Motifs application.

Cliques (Listing 2). This is also a vertex-induced implementation, as indicated by the `vfractoid` operator at line 1. To find k-cliques, we generate the next set of candidates by growing the subgraph (`expand` at line 1) and verifying the clique satisfiability criteria, i.e., the number of edges added from the last expansion must be equal to the number of vertices in the subgraph minus one (line 2). The last condition ensures that every vertex must be adjacent to every other vertex in the subgraph. Finally, we explore this snippet k times and obtain the k-cliques (line 3).

Frequent Subgraph Mining (Listing 3). We implement FSM as an edge-induced application. The first step is to obtain the frequent single edges that will determine the rest of

```
val cliques = graph.vfractoid.expand(1).
 filter((subg,comp)=>subg.nEdgesAdded==subg.
     nVertices-1).
 explore(k).subgraphs()
```

Listing 2: Cliques application.

the processing. We refer to this process as bootstrap (lines 1-9). It starts by obtaining the first fractoid, expanding the subgraphs by one to generate single edges and aggregating those edges according to their pattern (key) and initial domain support (value), which implements the *minimum image-based support* [7]. Next, we gather the aggregation result and initialize two sets, the last set of frequent patterns (`newFreqPatts`, line 11) and the cumulative set of frequent patterns (`freqPatts`, line 12). Finally, we repeat a sequence of filtering not frequent patterns (lines 15-16), adding one edge to subgraphs that are instance of frequent patterns (line 17) and performing again a pattern support counting to get the new set of frequent patterns (lines 18-23). With that we redefine the new set of frequent patterns (line 24) and proceed to check whether this set is not empty and the processing must continue or empty, indicating the halting condition.

```
1   val bootstrap = graph.efractoid.
2    expand(1).
3    aggregate [Pattern,DomainSupport] ("support",
4     (subg,comp,value) => { subg.getPattern },
5     (subg,comp,value) => { value.setSuppport(
6      minSupp); value.set(subg); value },
7     (value1,value2) => { value1.aggregate(value2);
8      value1 },
9     (patt,supp) => supp.hasEnoughSupport())
10  var fsm = bootstrap
11  var _fpatts = bootstrap.aggregation("support")
12  var fpatts = _fpatts
13  while (!newFreqPatts.isEmpty) {
14    fsm = fsm.
15     filter [Pattern,DomainSupport] ("support") {
16      (subg,agg) => agg.contains(subg.pattern)}.
17     expand(1).
18     aggregate [Pattern,DomainSupport] ("support",
19      (subg,comp,value) => { subg.getPattern },
20      (subg,comp,value) => { value.setSupport(
21       minSupp); value.set(subg);value },
22      (value1,value2) => { value1.aggregate(value2)
          ; value1 },
23      (patt,supp) => supp.hasEnoughSupport())
24    _fpatts = fsm.aggregation("support")
25    fpatts = fpatts.union(_fpatts)
26  }
```

Listing 3: FSM application.

Keyword Search (Listing 4). We implemented the candidate retrieval presented in [16]. For such we assume as input to the algorithm *(1)* an array of keywords representing the query (`keywords`) and *(2)* an inverted index from keywords to the set of edge identifiers that contains that word (`invIdxs`). Then, we consider an subgraph valid if its last edge (most recently added) contributes with a keyword that none of the previous edges represent. For this reason, we only generate candidates with, at most, the same number of edges than the length of the keywords array. The filtering function verifies for each inverted index (line 7) whether there is any other edge (but the last) that already contemplates the current keyword (lines 11-12). In affirmative case, we conclude that the last word is not valid w.r.t. the current keyword, so we proceed to check the remaining indexes if any. Otherwise, we mark the last edge as valid, i.e., the only condition in which the filtering function will return true. Finally, we configure an edge-induced fractoid with the filtering function described above and we explore the workflow by the length of the keyword query set (lines 21-21).

```
1   def lastEdgeIsValid(e: Subgraph,
2    c: Computation[Subgraph]): Boolean = {
3     val edges = e.edges(); val nEdges = e.nEdges
4     val lastEdge = edges.getLast()
5     val invIdxs = invIdxsBc.value
6     var valid = false; var i = 0
7     while (!valid && i < invIdxs.length) {
8       val ii = invIdxs(i)
9       if (ii.containsDoc(lastEdge)) {
10        var j = 0
11        while (j < nEdges - 1 &&
12          !ii.containsDoc(edges.get(j))) j += 1
13        if (j == nEdges - 1) valid = true
14      }
15      i += 1
16    }
17    valid
18  }
19  val results = graph.efractoid.
20   filter(lastEdgeIsValid).
21   explore(keywords.length).subgraphs()
```

Listing 4: Keyword Search application.

Subgraph Querying (Listing 5). This application uses the pattern-induced extension algorithm, generating new subgraphs according to a query pattern. In line 1, the user defines the query. Next, the first pattern fractoid is initialized using the query as input (line 2). Finally, the algorithm extends the subgraphs to the number of vertices in the pattern query (line 3) and returns the instances as result (line 3).

```
1   val query = new Pattern(/* pattern edges */)
2   val results = graph.pfractoid(query).
3    expand(query.nvertices).subgraphs()
```

Listing 5: Subgraph querying application.

```scala
val cliquesopt = graph.
 vfractoid(new KClistEnum(graph.adjLists())).
 expand(1).explore(k).subgraphs()
```

Listing 7: Optimized cliques application.

B ADVANCED PROGRAMMING

Fractal also accommodates advanced features for experienced users. For example, the user may implement a custom subgraph enumerator (Fig. 7) and pass it as an additional parameter to the fractoid operators (B_{1-3}, Fig. 2) to support complex GPM implementations. This is particularly useful when the application requires a specific policy for generating extension candidates, such as sampling, or in case it needs to maintain state during subgraph enumeration.

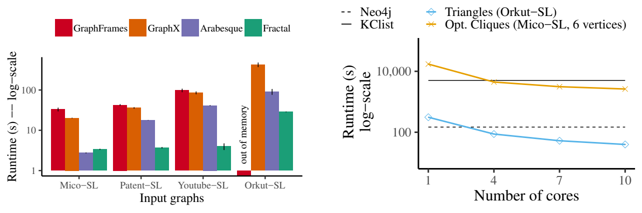
```scala
1  class KClistEnum extends SubgraphEnumerator {
2    val prefix: Subgraph // current subgraph
3    val dag: Map[Int,IntList] // DAG adj. lists
4    var cur: IntCursor = _ // extensions
5
6    override def computeExtensions(): Unit = {
7      extensions = dag.keys().cursor()
8    }
9    override def extend(): SubgraphEnumerator = {
10     val newAdjLists = new Map[Int,IntList]()
11     val u = cur.elem(); cur.moveNext()
12     val uneighborhood = dag.get(u)
13     for (v <- uneighborhood) {
14       val vneighborhood = dag.get(v)
15       val commonNeighborhood = uneighborhood.
16         intersection(vneighborhood)
17       newAdjLists.put(v, commonNeighborhood)
18     }
19     new KClistEnum(prefix.add(u), newAdjList)
20   }
21 }
```

Listing 6: KClist subgraph enumerator.

For example, KClist [12] is an optimized clique listing algorithm that reduces the clique search space by using special views of the input graph on each enumeration depth. The state to KClist enumeration is a DAG extracted from the induced-subgraph in the neighborhood of each extension candidate. We can accomplish such implementation in Fractal using a custom enumerator (Listing 6), responsible for maintaining the partial DAGs (line 3) and for extending existing cliques. In this case, the `computeExtensions` (lines 6-8) method reads from the current DAG to generate only extension candidates generating other cliques, while the `extend` (lines 9-20) method is responsible for setting up the next state (DAG) for the new enumerator. Furthermore, the usage of custom enumerators within Fractal is transparent: the user only needs to provide the enumerator parameter to the `vfractoid` initialization call. Listing 7 shows the implementation of this optimized version using Fractal's API. In Appendix C we evaluate the COST of this implementation in Fractal.

C ADDITIONAL RESULTS

Results on Triangle Counting: We now examine the performance of the triangle counting application on Fractal. We note that both fast approximate solutions [55] as well as distributed exact solutions [19] exist for this particular task but our goal here was to primarily compare Fractal's performance on this benchmark against other graph processing frameworks that directly support this common benchmark such as Arabesque [53], GraphFrames [13] and GraphX [23] in Figure 20a. We note that the triangles implementation in Fractal is the same as cliques (Listing 2) with $k = 3$. In this experiment we use datasets from Table 1 and an additional single-label graph Orkut [63] with 3.07M vertices representing users and 117.18M edges representing friendships among users. Fractal significantly outperforms the competing methods on three of the four datasets (up to an order of magnitude better), while being slightly slower than Arabesque on the smallest dataset due to setup overhead.



(a) Triangles performance **(b)** COST analysis

Figure 20: Additional results.

COST analysis: In Figure 20b we present the COST of the optimized version of Cliques (Listing 7), and Triangles (Listing 2 with $k = 3$). We consider Neo4j and the JVM based implementation of the KClist [12] algorithm, as single-thread baselines for triangles and cliques, respectively. Specifically for Neo4j, we use a built-in triangle counting implementation that serves as a strong baseline. Neo4j takes 147s to compute triangles on Orkut, while Fractal takes approximately 100s with 3 threads. For 6-cliques in Mico-SL, Fractal outperforms KClist (5032s) using 4 execution threads. Such COST remains consistent with previous results (Sec. 5.2.4) and shows that Fractal can also be used to implement highly optimized GPM algorithms efficiently and effectively.