Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU

Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Keshav Pingali
The University of Texas at Austin
{cxh,roshan,gill,pingali}@cs.utexas.edu

ABSTRACT

There is growing interest in graph pattern mining (GPM) problems such as motif counting. GPM systems have been developed to provide unified interfaces for programming algorithms for these problems and for running them on parallel systems. However, existing systems may take hours to mine even simple patterns in moderate-sized graphs, which significantly limits their real-world usability.

We present *Pangolin*, an efficient and flexible in-memory GPM framework targeting shared-memory CPUs and GPUs. Pangolin is the first GPM system that provides high-level abstractions for GPU processing. It provides a simple programming interface based on the extend-reduce-filter model, which allows users to specify application specific knowledge for search space pruning and isomorphism test elimination. We describe novel optimizations that exploit locality, reduce memory consumption, and mitigate the overheads of dynamic memory allocation and synchronization.

Evaluation on a 28-core CPU demonstrates that Pangolin outperforms existing GPM frameworks Arabesque, RStream, and Fractal by 49×, 88×, and 80× on average, respectively. Acceleration on a V100 GPU further improves performance of Pangolin by 15× on average. Compared to state-of-the-art hand-optimized GPM applications, Pangolin provides competitive performance with less programming effort.

PVLDB Reference Format:
Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Keshav Pingali. Pangolin: An Efficient and Flexible Parallel Graph Mining System on CPU and GPU. *PVLDB*, 13(8): 1190-1205, 2020.
DOI: https://doi.org/10.14778/3389133.3389137

1. INTRODUCTION

Applications that use graph data are becoming increasingly important in many fields. Graph analytics algorithms such as PageRank and SSSP have been studied extensively and many frameworks have been proposed to provide both high performance and high productivity [65, 62, 70, 78]. Another important class of graph problems deals with *graph pattern mining* (GPM), which has plenty of applications in areas such as chemical engineering [29], bioinformatics [5, 25], and social sciences [35]. GPM discovers relevant patterns in a given graph. One example is *triangle counting*, which is used to mine graphs in security applications [87]. Another example is *motif counting* [68, 12], which counts the frequency of certain structural patterns; this is useful in evaluating network models or classifying vertex roles. Fig. 1 illustrates the 3-vertex and 4-vertex motifs.

Compared to graph analytics, GPM algorithms are more difficult to implement on parallel platforms; for example, unlike graph analytics algorithms, they usually generate enormous amounts of intermediate data. GPM systems such as Arabesque [84], RStream [88], and Fractal [30] have been developed to provide abstractions for programmability. Instead of the vertex-centric model used in graph analytics systems [65], Arabesque proposed an *embedding-centric* programming model. In Arabesque, computation is applied on individual embeddings (*i.e.*, subgraphs) concurrently. It provides a simple programming interface that substantially reduces the complexity of application development. However, existing systems suffer dramatic performance loss compared to hand-optimized implementations. For example, Arabesque and RStream take 98s and 39s respectively to count 3-cliques for a graph with 2.7M vertices and 28M edges, while a custom solver (KClist) [26] counts it in 0.16s. This huge performance gap significantly limits the usability of existing GPM frameworks in real-world applications.

The first reason for this poor performance is that existing GPM systems provide limited support for application-specific customization. The state-of-the-art systems focus on generality and provide high-level abstraction to the user for ease-of-programming. Therefore, they hide as many execution details as possible from the user, which substantially limits the flexibility for algorithmic customization. The complexity of GPM algorithms is primarily due to combinatorial enumeration of embeddings and isomorphism tests to find canonical patterns. Hand-optimizing implementations exploit application-specific knowledge to aggressively prune the enumeration search space or elide isomorphism tests or both. Mining frameworks need to support such optimizations to match performance of hand-optimized applications.

The second reason for poor performance is inefficient implementation of parallel operations and data structures. Programming parallel processors requires exploring trade-offs between synchronization overhead, memory management, load balancing, and data locality. However, the state-of-the-art GPM systems target either distributed or out-of-

Proceedings of the VLDB Endowment, Vol. 13, No. 8
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3389133.3389137

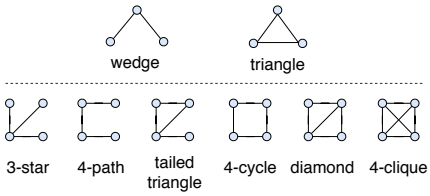


Figure 1: 3-vertex motifs (top) and 4-vertex motifs (bottom).

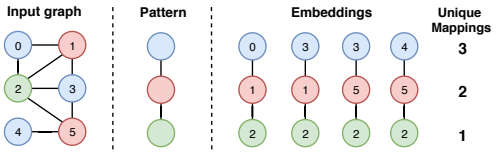


Figure 2: An example of the GPM problem.

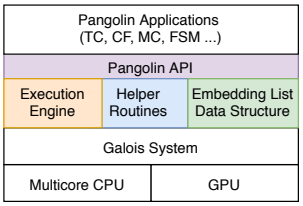


Figure 3: System overview of Pangolin (shaded parts).

core platforms, and thus are not well optimized for shared-memory multicore/manycore architectures.

In this paper, we present *Pangolin*, an efficient in-memory GPM framework that provides a flexible embedding-centric programming interface. Pangolin is based on the *extend-reduce-filter* model, which enables application-specific customization (Section 3). Application developers can implement aggressive pruning strategies to reduce the enumeration search space, and apply customized pattern classification methods to elide generic isomorphism tests (Section 4).

To make full use of parallel hardware, we optimize parallel operations and data structures, and provide helper routines to the users to compose higher level operations. Pangolin is built as a lightweight layer on top of the Galois [70] parallel library and LonestarGPU [18] infrastructure, targeting both shared-memory multicore CPUs and GPUs. Pangolin includes novel optimizations that exploit locality, reduce memory consumption, and mitigate overheads of dynamic memory allocation and synchronization (Section 5).

Experimental results (Section 6) on a 28-core CPU demonstrate that Pangolin outperforms existing GPM frameworks, Arabesque, RStream, and Fractal, by 49×, 88×, and 80× on average, respectively. Furthermore, Pangolin on V100 GPU outperforms Pangolin on 28-core CPU by 15× on average. Pangolin provides performance competitive to state-of-the-art hand-optimized GPM applications, but with much less programming effort. To mine 4-cliques in a real-world web-crawl graph (gsh) with 988 million vertices and 51 billion vertices, Pangolin takes ∼ 6.5 hours on a 48-core Intel Optane PMM machine [39] with 6 TB (byte-addressable) memory. To the best of our knowledge, this is the largest graph on which 4-cliques have been mined. In summary, Pangolin makes the following contributions:

- We investigate the performance gap between state-of-the-art GPM systems and hand-optimized approaches, and point out two key features absent in existing systems: *pruning enumeration space* and *eliding isomorphism tests*.
- We present a high-performance in-memory GPM system, Pangolin, which enables *application-specific optimizations* and provides transparent parallelism on CPU or GPU. To the best of our knowledge, it is the first GPM system that provides high-level abstractions for GPU processing.
- We propose novel techniques that enable the user to aggressively prune the enumeration search space and elide isomorphism tests.
- We propose novel optimizations that exploit locality, reduce memory usage, and mitigate overheads of dynamic memory allocation and synchronization on CPU and GPU.
- We evaluate Pangolin on a multicore CPU and a GPU to demonstrate that Pangolin is substantially faster than existing GPM frameworks. Compared to hand-optimized applications, it provides competitive performance while requiring less programming effort.

2. BACKGROUND AND MOTIVATION

We describe GPM concepts, applications, as well as algorithmic and architectural optimizations in state-of-the-art hand-optimized GPM solvers. Lastly, we point out performance limitations of existing GPM frameworks.

2.1 Graph Pattern Mining

In GPM problems, a pattern P is a graph defined by the user explicitly or implicitly. An explicit definition specifies the vertices and edges of the graph, whereas an implicit definition specifies the desired properties of the graph of interest. Given an *input graph* G and a set of patterns S_p, the goal of GPM is to find the *embeddings*, i.e., subgraphs in G that are isomorphic to any pattern $P \in S_p$. For explicit-pattern problems (e.g., triangle counting), the solver finds only the embeddings. For implicit-pattern problems (e.g., frequent subgraph mining), the solver needs to find the patterns as well as the embeddings. Note that *graph pattern matching* [36] finds embeddings only for a single explicit-pattern, whereas *graph pattern mining* (GPM) [3, 84] solves both explicit-pattern problems and implicit-pattern problems. In this work, we focus on connected patterns only.

In the input graph in Fig. 2, colors represent vertex labels, and numbers denote vertex IDs. The 3-vertex pattern is a blue-red-green chain, and there are four embeddings of this pattern in the input graph, shown on the right of the figure. In a specific GPM problem, the user may be interested in some pattern-specific statistical information (i.e., pattern frequency), instead of listing all the embeddings. The measure of the frequency of P in G, termed *support*, is also defined by the user. For example, in triangle counting, the support is defined as the total count of triangles.

There are two types of GPM problems targeting two types of embeddings. In a *vertex-induced* embedding, a set of vertices is given and the subgraph of interest is obtained from these vertices and the set of edges in the input graph connecting these vertices. Triangle counting uses vertex-induced embeddings. In an *edge-induced* embedding, a set of edges is given and the subgraph is formed by including all the endpoints of these edges in the input graph. Frequent subgraph mining (FSM) is an edge-induced GPM problem.

A GPM algorithm enumerates embeddings of the given pattern(s). If duplicate embeddings exist (*automorphism*), the algorithm chooses one of them as the *canonical* one (namely canonical test) and collects statistical information about these canonical embeddings such as the total count. The canonical test needs to be performed on each embedding, and can be complicated and expensive for complex problems such as FSM. Enumeration of embeddings in a graph grows exponentially with the embedding size (number of vertices or edges in the embedding), which is computationally expensive and consumes lots of memory. In addition, a graph isomorphism (GI) test is needed for each

embedding to determine whether it is *isomorphic* to a pattern. Unfortunately, the GI problem is not solvable in polynomial time [37]. It leads to compute and memory intensive algorithms [51] that are time-consuming to implement.

Graph analytics problems typically involve allocating and computing labels on vertices or edges of the input graph iteratively. On the other hand, GPM problems involve generating embeddings of the input graph and analyzing them. Consequently, GPM problems require much more memory and computation to solve. The memory consumption is not only proportional to the graph size, but also increases exponentially as the embedding size increases [84]. Furthermore, GPM problems require compute-intensive operations, such as isomorphism test and automorphism test on each embedding. Thus, GPM algorithms are more difficult to develop, and conventional graph analytics systems [34, 76, 60, 53, 45, 23, 41, 92, 27, 28] are not sufficient to provide a good trade-off between programmability and efficiency.

2.2 Hand-Optimized GPM Applications

We consider 4 applications: triangle counting (TC), clique finding (CF), motif counting (MC), and frequent subgraph mining (FSM). Given the input graph which is undirected, TC counts the number of triangles while CF enumerates all complete subgraphs [1] (i.e., cliques) contained in the graph. TC is a special case of CF as it counts 3-cliques. MC counts the number of occurrences (i.e., frequency) of each structural pattern (also known as *motif* or *graphlet*). As listed in Fig. 1, k-clique is one of the patterns in k-motifs. FSM finds frequent patterns in a *labeled* graph. A minimum support σ is provided by the user, and all patterns with support above σ are considered to be frequent and must be discovered. Note that a widely used support for FSM is *minimum image-based* (MNI) support (a.k.a. *domain support*), which has the anti-monotonic property [2]. It is calculated as the minimum number of distinct mappings for any vertex (i.e., domain) in the pattern over all embeddings of the pattern. In Fig. 2, the MNI support of the pattern is $min\{3, 2, 1\} = 1$.

Several hand-optimized implementations exist for each of these applications on multicore CPU [79, 4, 31, 17, 83], GPU [42, 59, 61, 52], distributed CPU [81, 38, 82], and multi-GPU [46, 44, 73]. They employ application-specific optimizations to reduce algorithm complexity. The complexity of GPM algorithms is primarily due to two aspects: combinatorial enumeration and isomorphism test. Therefore, hand-optimized implementations focus on either *pruning the enumeration search space* or *eliding isomorphism test* or both. We describe some of these techniques briefly below.

Pruning Enumeration Search Space: In general GPM applications, new embeddings are generated by extending existing embeddings and then they may be discarded because they are either not interesting or a duplicate (*automorphism*). However, in some applications like CF [26], duplicate embeddings can be detected eagerly before extending current embeddings, based on properties of the current embeddings. We term this optimization as *eager pruning*. Eager pruning can significantly reduce the search space. Furthermore, the input graphs are converted into directed acyclic graphs (DAGs) in state-of-the-art TC [46], CF [26], and MC [71] solvers, to significantly reduce the search space.

Eliding Isomorphism Test: In most hand-optimized TC, CF, and MC solvers, isomorphism test is completely avoided by taking advantage of the pattern characteristics. For example, a parallel MC solver, PGD [4], uses an ad-hoc method for a specific k. Since it only counts 3-vertex and 4-vertex motifs, all the patterns (two 3-motifs and six 4-motifs as shown in Fig. 1) are known in advance. Therefore, some special (and thus easy-to-count) patterns (*e.g.,* cliques[3]) are counted first, and the frequencies of other patterns are obtained in constant time using the relationship among patterns[4]. In this case, no isomorphism test is needed, which is typically an order-of-magnitude faster [4].

Summary: Most of the algorithmic optimizations exploit application-specific knowledge, which can only be enabled by application developers. A generic GPM framework should be flexible enough to allow users to compose as many of these optimization techniques as possible, and provide parallelization support for ease of programming. Pangolin is the first GPM framework to do so.

2.3 Existing GPM Frameworks

Existing GPM systems target either distributed-memory [84, 30, 47] or out-of-core [88, 91, 66] platforms, and they make tradeoffs specific for their targeted architectures. None of them target in-memory GPM on a multicore CPU or a GPU. Consequently, they do not pay much attention to reducing the synchronization overheads among threads within a CPU/GPU or reducing memory consumption overheads. Due to this, naively porting these GPM systems to run on a multicore CPU or GPU would lead to inefficient implementations. We first describe two of these GPM systems briefly and then discuss their major limitations.

Arabesque [84] is a distributed GPM system. It proposes "think like an embedding" (TLE) programming paradigm, where computation is performed in an embedding-centric manner. It defines a *filter-process* computation model which consists of two functions: (1) *filter*, which indicates whether an embedding should be processed and (2) *process*, which examines an embedding and may produce some output.

RStream [88] is an out-of-core single-machine GPM system. Its programming model is based on relational algebra. Users specify how to generate embeddings using relational operations such as `select`, `join`, and `aggregate`. It stores intermediate data (*i.e.*, embeddings) on disk while the input graph is kept in memory for reuse. It streams data (or table) from disk and uses relational operations that may produce more intermediate data, which is stored back on disk.

Limitations in API: *Most of the application-specific optimizations like pruning enumeration search space and avoiding isomorphism test are missing in existing GPM frameworks*, as they focus on providing high-level abstractions but lack support for application-specific customization. The absence of such key optimizations in existing systems results in a huge performance gap when compared to hand-optimized implementations. Moreover, some frameworks like RStream support only edge-induced embeddings but for applications

[1] A k-vertex complete subgraph is a connected subgraph in which each vertex has degree of $k-1$ (*i.e.*, any two vertices are connected).

[2] The support of a supergraph should not exceed the support of a subgraph; this allows the GPM algorithm to stop extending embeddings as soon as they are recognized as infrequent.

[3] Cliques can be identified by checking connectivity among vertices without generic isomorphism test.

[4] For example, the count of diamonds can be computed directly from the counts of triangles and 4-cliques [4].

like CF, the enumeration search space is much smaller using vertex-induced exploration than edge-induced one.

Data Structures for Embeddings: Data structures used to store embeddings in existing GPM systems are not efficient. Both Arabesque and RStream store embeddings in an array of structures (AoS), where the embedding structures consists of a vertex set and an edge set. Arabesque also proposes a space efficient data structure called the *Overapproximating Directed Acyclic Graph* (ODAG), but it requires extra canonical test for each embedding, which has been demonstrated to be very expensive for large graphs [84].

Materialization of Data Structures: The list or array of intermediate embeddings in both Arabesque and RStream is always materialized in memory and in disk, respectively. This has significant overheads as the size of such data grows exponentially. Such materialization may not be needed if the embeddings can be filtered or processed immediately.

Dynamic Memory Allocation: As the number of (intermediate) embeddings are not known before executing the algorithm, memory needs to be allocated dynamically for them. Moreover, during parallel execution, different threads might allocate memory for embeddings they create or enumerate. Existing systems use standard (`std`) maps and sets, which internally use a global lock to dynamically allocate memory. This limits the performance and scalability.

Summary: Existing GPM systems have limitations in their API, execution model, and implementation. Pangolin addresses these issues by permitting application-specific optimizations in its API, optimizing the execution model, and providing an efficient, scalable implementation on multicore CPU and GPU. These optimizations can be applied to existing embedding-centric systems like Arabesque.

3. DESIGN OF PANGOLIN FRAMEWORK

Fig. 3 illustrates an overview of the Pangolin system. Pangolin provide a simple API (purple box) to the user for writing GPM applications. The unified execution engine (orange box) follows the embedding-centric model. Important common operations are encapsulated and provided to the user in the helper routines (blue box), which are optimized for both CPU and GPU. The embedding list data structure (green box) is also optimized for different architectures to exploit hardware features. Thus, Pangolin hides most of the architecture oriented programming complexity and achieves high performance and high productivity simultaneously. In this section, we describe the execution model, programming interface (i.e., API), and example applications of Pangolin.

3.1 Execution Model

Algorithm 1 describes the execution engine in Pangolin which illustrates our *extend-reduce-filter* execution model. To begin with, a worklist of embeddings is initialized with all the single-edge embeddings (line 4). The engine then works in an iterative fashion (line 6). In each iteration, i.e., *level*, there are three phases: EXTEND (line 8), REDUCE (line 10) and FILTER (line 12). Pangolin exposes necessary details in each phase to enable a more flexible programming interface (Section 3.2) than existing systems; for example, Pangolin exposes the EXTEND phase which is implicit in Arabesque.

The EXTEND phase takes each embedding in the input worklist and extends it with a vertex (vertex-induced) or an edge (edge-induced). Newly generated embeddings then form the output worklist for the next level. The embedding

Algorithm 1 Execution Model for Mining

```
1: procedure MINEENGINE(G(V,E), MAX_SIZE)
2:     EmbeddingList in_wl, out_wl                ▷ double buffering
3:     PatternMap p_map
4:     INIT(in_wl)                    ▷ insert single-edge embeddings
5:     level ← 1
6:     while true do
7:         out_wl ← ∅                          ▷ clear the new worklist
8:         EXTEND(in_wl, out_wl)
9:         p_map ← ∅                           ▷ clear the pattern map
10:        REDUCE(out_wl, p_map)
11:        in_wl ← ∅                            ▷ clear the old worklist
12:        FILTER(out_wl, p_map, in_wl)
13:        level ← level + 1
14:        if level = MAX_SIZE - 1 then
15:            break                           ▷ termination condition
16:    return in_wl, p_map
```

size is increased with *level* until the user defined maximum size is reached (line 14). Fig. 4 shows an example of the first iteration of vertex-based extension. The input worklist consists of all the 2-vertex (i.e., single-edge) embeddings. For each embedding in the worklist, one vertex is added to yield a 3-vertex embedding. For example, the first 2-vertex embedding $\{0, 1\}$ is extended to two new 3-vertex embeddings $\{0, 1, 2\}$ and $\{0, 1, 3\}$.

After vertex/edge extension, a REDUCE phase is used to extract some pattern-based statistical information, i.e., pattern frequency or *support*, from the embedding worklist. The REDUCE phase first classifies all the embeddings in the worklist into different categories according to their patterns, and then computes the support for each pattern category, forming pattern-support pairs. All the pairs together constitute a pattern map (*p_map* in line 10). Fig. 5 shows an example of the reduction operation. The three embeddings (top) can be classified into two categories, i.e., triangle and wedge (bottom). Within each category, this example counts the number of embeddings as the support. As a result, we get the pattern-map as {[triangle, 2], [wedge, 1]}. After reduction, a FILTER phase may be needed to remove those embeddings which the user are no longer interested in; e.g., FSM removes infrequent embeddings in this phase.

Note that REDUCE and FILTER phases are not necessary for all applications, and they can be disabled by the user. If they are used, they are also executed after initializing single-edge embeddings (line 4) and before entering the main loop (line 6). Thus, infrequent single-edge embeddings are filtered out to collect only the frequent ones before the main loop starts. Note that this is omitted from Algorithm 1 due to lack of space. If REDUCE is enabled but FILTER is disabled, then reduction is only required and executed for the last iteration, as the pattern map produced by reduction is not used in prior iterations (dead code).

3.2 Programming Interface

Pangolin exposes flexible and simple interfaces to the user to express application-specific optimizations. Listing 1 lists user-defined functions (APIs) and Algorithm 2 describes how these functions (marked in blue) are invoked by the Pangolin execution engine. A specific application can be created by defining these APIs. Note that all the functions are not mandatory; each of them has a default return value.

In the EXTEND phase, we provide two functions, `toAdd` and `toExtend`, for the user to prune embedding candidates aggressively. When they return false, the execution engine avoids generating an embedding and thus the search space is reduced. More specifically, `toExtend` checks whether

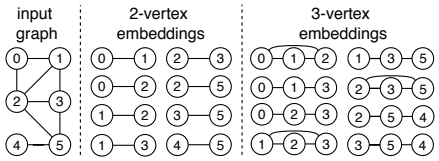


Figure 4: An example of vertex extension.

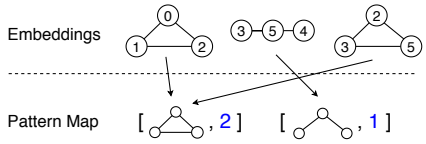


Figure 5: Reduction operation that calculates pattern frequency using a pattern map.

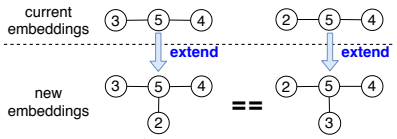


Figure 6: An example of automorphism.

Algorithm 2 Compute Phases in **Vertex-induced** Mining

1: **procedure** EXTEND(in_wl, out_wl)
2: **for** each embedding $emb \in in\_wl$ **in parallel do**
3: **for** each vertex v in emb **do**
4: **if** TOEXTEND(emb, v) $= true$ **then**
5: **for** each vertex u in $adj(v)$ **do**
6: **if** TOADD(emb, u) $= true$ **then**
7: insert $emb \cup u$ to out_wl

8: **procedure** REDUCE($queue$, p_map)
9: **for** each embedding $emb \in queue$ **in parallel do**
10: Pattern $pt \leftarrow$ GETPATTERN(emb)
11: Support $sp \leftarrow$ GETSUPPORT(emb)
12: $p\_map[pt] \leftarrow$ AGGREGATE($p\_map[pt]$, sp)

13: **procedure** FILTER(in_wl, p_map, out_wl)
14: **for** each embedding $emb \in in\_wl$ **in parallel do**
15: Pattern $pt \leftarrow$ GETPATTERN(emb)
16: **if** TODISCARD(pt, p_map) $= false$ **then**
17: insert emb to out_wl

```
1 bool toExtend(Embedding emb, Vertex v);
2 bool toAdd(Embedding emb, Vertex u)
3 bool toAdd(Embedding emb, Edge e)
4 Pattern getPattern(Embedding emb)
5 Support getSupport(Embedding emb)
6 Support Aggregate(Support s1, Support s2)
7 bool toDiscard(Pattern pt, PatternMap map);
```

Listing 1: User-defined functions in Pangolin.

a vertex in the current embedding needs to be extended. Extended embeddings can have duplicates due to *automorphism*. Fig. 6 illustrates *automorphism*: two different embeddings $(3, 5, 4)$ and $(2, 5, 4)$ can be extended into the same embedding $(2, 5, 3, 4)$. Therefore, only one of them (the canonical embedding) should be kept, and the other (the redundant one) should be removed. This is done by a *canonical test* in `toAdd`, which checks whether the newly generated embedding is a *qualified* candidate. An embedding is not qualified when it is a duplicate or it does not have certain user-defined characteristics. Only qualified embeddings are added into the next worklist. Application-specific knowledge can be used to specialize the two functions. If left undefined, `toExtend` returns true and `toAdd` does a default canonical test. Note that the user specifies whether the embedding exploration is vertex-induced or edge-induced. The only difference for edge-induced extension is in lines 5 to 7: instead of vertices adjacent to v, edges incident on v are used.

In the REDUCE phase, `getPattern` function specifies how to obtain the pattern of an embedding. Finding the canonical pattern of an embedding involves an expensive *isomorphism* test. This can be specialized using application-specific knowledge to avoid such tests. If left undefined, a canonical pattern is returned by `getPattern`. In this case, to reduce the overheads of invoking the *isomorphism* test, embeddings in the worklist are first reduced using their *quick patterns* [84], and then quick patterns are aggregated using their canonical patterns. In addition, `getSupport` and `Aggregate` functions specify the support of an embedding and the reduction operator for the support, respectively.

Lastly, in the FILTER stage, `toDiscard` is used to remove

```
// connectivity checking routines
bool isConnected(Vertex u, Vertex v)

// canonical test routines
bool isAutoCanonical(Embedding emb, Vertex v)
bool isAutoCanonical(Embedding emb, Edge e)
Pattern getIsoCanonicalBliss(Embedding emb)
Pattern getIsoCanonicalEigen(Embedding emb)

// to get domain (MNI) support
Support getDomainSupport(Embedding emb)
Support mergeDomainSupport(Support s1, Support s2)
```

Listing 2: Helper routines provided to the user by Pangolin.

uninteresting patterns. This usually depends on the support for the pattern (that is in the computed pattern map).

Complexity Analysis. Consider an input graph G with n vertices and maximum embedding size k. In the EXTEND phase of the last level (which dominates the execution time and complexity), there are up to $O(n^{k-1})$ embeddings in the input worklist. Each embedding has up to $k-1$ vertices to extend. Each vertex has up to d_{max} neighbors (candidates). In general, each candidate needs to check connectivity with $k-1$ vertices, with a complexity of $O(log(d_{max}))$ (binary search). An isomorphism test needs to be performed for each newly generated embedding (size of k) to find its pattern. The state-of-the-art algorithm to test isomorphism has a complexity of $O(e^{\sqrt{klogk}})$ [8]. Therefore, the overall worst-case complexity is $O(n^{k-1}k^2 d_{max} log(d_{max}) e^{\sqrt{klogk}})$.

Pangolin also provides APIs to process the embeddings or pattern maps at the end of each phase (e.g., this is used in clique-listing, which a variant of clique-finding that requires listing all the cliques). We omit this from Algorithm 2 and Listing 1 for the sake of brevity. To implement the application-specific functions, users are required to write C++ code for CPU and CUDA `__device__` functions for GPU (compiler support can provide a unified interface for both CPU and GPU in the future). Listing 2 lists the helper routines provided by Pangolin. These routines are commonly used in GPM applications; e.g., to check connectivity, to test canonicality, as well as an implementation of domain support. They are available on both CPU and GPU, with efficient implementation on each architecture.

Comparison With Other GPM APIs: Existing GPM frameworks do not expose `toExtend` and `getPattern` to the application developer (instead, they assume these functions always return true and a canonical pattern, respectively). Note that existing embedding-centric frameworks like Arabesque can be extended to expose the same API functions in Pangolin so as to enable application-specific optimizations (Section 4), but this is difficult for relational model based systems like RStream, as the table join operations are inflexible to allow this fine-grained control.

3.3 Applications in Pangolin

TC, CF, and MC use vertex-induced embeddings, while FSM uses edge-induced embeddings. Listings 3 to 5 show CF, MC, and FSM implemented in Pangolin (we omit TC

```
bool toExtend(Embedding emb, Vertex v) {
  return (emb.getLastVertex() == v);
}
bool toAdd(Embedding emb, Vertex u) {
  for v in emb.getVertices() except last:
    if (!isConnected(v, u)) return false;
  return true;
}
```

Listing 3: Clique finding (vertex induced) in Pangolin.

```
bool toAdd(Embedding emb, Vertex v) {
  return isAutoCanonical(emb, v);
}
Support getSupport(Embedding emb) { return 1; }
Pattern getPattern(Embedding emb) {
  return getIsoCanonicalBliss(emb);
}
Support Aggregate(Support s1, Support s2) {
  return s1 + s2;
}
```

Listing 4: Motif counting (vertex induced) in Pangolin.

due to lack of space). For TC, extension happens only once, i.e., for each edge (v_0, v_1), v_1 is extended to get a neighbor v_2. We only need to check whether v_2 is connected to v_0. If it is, this 3-vertex embedding (v_0, v_1, v_2) forms a triangle. For CF in Listing 3, the search space is reduced by extending only the last vertex in the embedding instead of extending every vertex. If the newly added vertex is connected to all the vertices in the embedding, the new embedding forms a clique. Since cliques can only grow from smaller cliques (e.g., 4-cliques can only be generated by extending 3-cliques), all the non-clique embeddings are implicitly pruned. Both TC and CF do not use REDUCE and FILTER phases.

Listing 4 shows MC. An extended embedding is added only if it is canonical according to automorphism test. In the REDUCE phase, the quick pattern of each embedding is first obtained and then the canonical pattern is obtained using an isomorphism test. In Section 4.2, we show a way to customize this pattern classification method for MC to improve performance. FILTER phase is not used by MC.

FSM is the most complicated GPM application. As shown in Listing 5, it uses the custom domain support routines provided by Pangolin. An extended embedding is added only if the new embedding is (automorphism) canonical. FSM uses the FILTER phase to remove embeddings whose patterns are not frequent from the worklist. Despite the complexity of FSM, the Pangolin implementation is still much simpler than hand-optimized FSM implementations [82, 1, 32], thanks to the Pangolin API and helper routines.

4. SUPPORTING APPLICATION-SPECIFIC OPTIMIZATIONS IN PANGOLIN

In this section, we describe how Pangolin's API and execution model supports application-specific optimizations that: (1) enable enumeration search space pruning and (2) enable the eliding of isomorphism tests.

4.1 Pruning Enumeration Search Space

Directed Acyclic Graph (DAG): In typical GPM applications, the input graph is undirected. In some vertex-induced GPM applications, a common optimization technique is *orientation* which converts the undirected input graph into a directed acyclic graph (DAG) [24, 6]. Instead of enumerating candidate subgraphs in an undirected graph, the direction significantly cuts down the combinatorial search space. Orientation has been adopted in triangle counting [74], clique finding [26], and motif counting [71].

```
bool toAdd(Embedding emb, Edge e) {
  return isAutoCanonical(emb,e)
}
Support getSupport(Embedding emb) {
  return getDomainSupport(emb);
}
Pattern getPattern(Embedding emb) {
  return getIsoCanonicalBliss(emb);
}
Support Aggregate(Support s1, Support s2) {
  return mergeDomainSupport(s1, s2);
}
bool toDiscard(Pattern pt, PatternMap map) {
  return map[pt] < MIN_SUPPORT;
}
```

Listing 5: Frequent subgraph mining (edge induced).

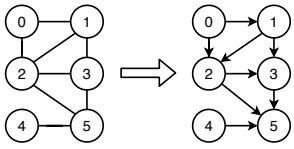


Figure 7: Convert an undirected graph into a DAG.

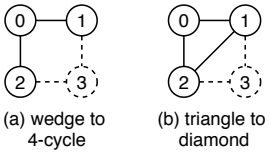


Figure 8: Examples of eliding isomorphism test for 4-MC.

Fig. 7 illustrates an example of the DAG construction process. In this example, vertices are ordered by vertex ID. Edges are directed from vertices with smaller IDs to vertices with larger IDs. Generally, vertices can be ordered in any total ordering, which guarantees the input graph is converted into a DAG. In our current implementation, we establish the order [44] among the vertices based on their degrees: each edge points to the vertex with higher degree. When there is a tie, the edge points to the vertex with larger vertex ID. Other orderings can be included in the future. In Pangolin, orientation is enabled by setting a flag at runtime.

Eager Pruning: In some applications like MC and FSM, all vertices in an embedding may need to be extended before determining whether the new embedding candidate is a (*automorphism*) canonical embedding or a duplicate. However, in some applications like TC and CF [26], duplicate embeddings can be detected eagerly before extending current embeddings. In both TC and CF, all embeddings obtained by extending vertices except (the last) one will lead to duplicate embeddings. Thus, as shown in Listing 3, only the last vertex of the current embedding needs to be extended. This aggressive pruning can significantly reduce the search space. The `toExtend` function in Pangolin enables the user to specify such *eager pruning*.

4.2 Eliding Isomorphism Test

Exploiting Memoization: Pangolin avoids redundant computation in each stage with memoization. Memoization is a tradeoff between computation and memory usage. Since GPM applications are usually memory hungry, we only do memoization when it requires small amount of memory and/or it dramatically reduce complexity. For example, in the FILTER phase of FSM, Pangolin avoids isomorphism test to get the pattern of each embedding, since it has been done in the REDUCE phase. This recomputation is avoided by maintaining a pattern ID (hash value) in each embedding after isomorphism test, and setting up a map between the pattern ID and pattern support. Compared to isomorphism test, which is extremely compute and memory intensive, storing the pattern ID and a small pattern support map is relatively lightweight. In MC, which is another application to find multiple patterns, the user can easily enable memo-

```
Pattern getPattern(Embedding emb) {
  if (emb.size() == 3) {
    if (emb.getNumEdges() == 3) return P1;
    else return P0;
  } else return getIsoCanonicalBliss(emb);
}
```

Listing 6: Customized pattern classification for 3-MC.

ization for the pattern id in each level. In this case, when it goes to the next level, the pattern of each embedding can be identified with its pattern id in the previous level with much less computation than a generic isomorphism test. As shown in Fig. 8, to identify a 4-cycle from a wedge or a diamond from a triangle, we only need to check if vertex 3 is connected to both vertex 1 and 2.

Customized Pattern Classification: In the REDUCE phase (Fig. 5), embeddings are classified into different categories based on their patterns. To get the pattern of an embedding, a generic way is to convert the embedding into a canonical graph that is isomorphic to it (done in two steps, as explained in Section 3.2). Like Arabesque and RStream, Pangolin uses the Bliss [51] library for getting the canonical graph or pattern for an embedding. This graph isomorphism approach is applicable to embeddings of any size, but it is very expensive as it requires frequent dynamic memory allocation and consumes a huge amount of memory. For small embeddings, such as 3-vertex and 4-vertex embeddings in vertex-induced applications and 2-edge and 3-edge embeddings in edge-induced applications, the canonical graph or pattern can be computed very efficiently. For example, we know that there are only 2 patterns in 3-MC (i.e., wedge and triangle in Fig. 1). The only computation needed to differentiate the two patterns is to count the number of edges (i.e., a wedge has 2 edges and a triangle has 3), as shown in Listing 6. This specialized method significantly reduces the computational complexity of pattern classification. The `getPattern` function in Pangolin enables the user to specify such *customized pattern classification*.

5. IMPLEMENTATION ON CPU AND GPU

The user implements application-specific optimizations using the Pangolin API and helper functions, and Pangolin transparently parallelizes the application. Pangolin provides an efficient and scalable parallel implementation on both shared-memory multicore CPU and GPU. Its CPU implementation is built using the Galois [70] library and its GPU implementation is built using the LonestarGPU [18] infrastructure. Pangolin includes several architectural optimizations. In this section, we briefly describe some of them: (1) exploiting locality and fully utilizing memory bandwidth [33, 10, 9]; (2) reducing the memory consumption; (3) mitigating the overhead of dynamic memory allocation; (4) minimizing synchronization and other overheads.

5.1 Data Structures for Embeddings

Since the number of possible k-embeddings in a graph increases exponentially with k, storage for embeddings grows rapidly and easily becomes the performance bottleneck. Most existing systems use array-of-structures (AoS) to organize the embeddings, which leads to poor locality, especially for GPU computing. In Pangolin, we use structure of arrays (SoA) to store embeddings in memory. The SoA layout is particularly beneficial for parallel processing on GPU as memory accesses to the embeddings are fully coalesced.

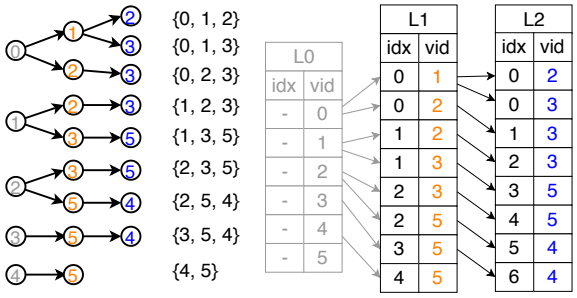


Figure 9: An example of the embedding list data structure.

Fig. 9 illustrates the embedding list data structure. On the left is the prefix-tree that illustrates the embedding extension process in Fig. 4. The numbers in the vertices are vertex IDs (VIDs). Orange VIDs are in the first level L_1, and blue VIDs belong to the second level L_2. The gray level L_0 is a dummy level which does not actually exist but is used to explain the key ideas. On the right, we show the corresponding storage of this prefix tree. For simplicity, we only show the vertex-induced case. Given the maximum size k, the embedding list contains $k-1$ levels. In each level, there are two arrays, index array (`idx`) and vertex ID array (`vid`). In the same position of the two arrays, an element of index and vertex ID consists of a pair (`idx`, `vid`). In level L_i, `idx` is the index pointing to the vertex of the same embedding in the previous level L_{i-1}, and `vid` is the i-th vertex ID of the embedding.

Each embedding can be reconstructed by backtracking from the last level lists. For example, to get the first embedding in level L_2, which is a vertex set of $\{0, 1, 2\}$, we use an empty vertex set at the beginning. We start from the first entry (0, 2) in L_2, which indicates the last vertex ID is '2' and the previous vertex is at the position of '0'. We put '2' into the vertex set $\{2\}$. Then we go back to the previous level L_1, and get the 0-th entry (0, 1). Now we put '1' into the vertex set $\{1, 2\}$. Since L_1 is the lowest level and its index is the same as the vertex ID in level L_0, we put '0' into the vertex set $\{0, 1, 2\}$.

For the edge-induced case, the strategy is similar but requires one more column `his` in each level to indicate the history information. Each entry is a triplet (`vid`, `his`, `idx`) that represents an edge instead of a vertex, where `his` indicates at which level the source vertex of this edge is, while `vid` is the ID of the destination vertex. In this way we can backtrack the source vertex with `his` and reconstruct the edge connectivity inside the embedding. Note that we use three distinct arrays for `vid`, `his` and `idx`, which is also an SoA layout. This data layout can improve temporal locality with more data reuse. For example, the first `vid` in L_1 (v_1) is connected to two vertices in L_2 (v_2 & v_3). Therefore v_1 will be reused. Considering high-degree vertices in power-law graphs, there are lots of reuse opportunities.

5.2 Avoiding Data Structure Materialization

Loop Fusion: Existing GPM systems first collect all the embedding candidates into a list and then call the user-defined function (like `toAdd`) to select embeddings from the list. This leads to materialization of the candidate embeddings list. In contrast, Pangolin preemptively discards embedding candidates using the `toAdd` function before adding it to the embedding list (as shown in Algorithm 2), thereby avoiding the materialization of the candidate embeddings

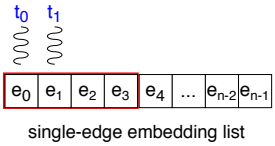


Figure 10: Edge blocking.

Figure 11: Inspection-execution.

(this is similar to *loop fusion* in array languages). This significantly reduces memory allocations, yielding lower memory usage and execution time.

Blocking Schedule: Since the memory consumption increases exponentially with the embedding size, existing systems utilize either distributed memory or disk to hold the data. However, Pangolin is a shared memory framework and could run out of memory for large graphs. In order to support processing large datasets, we introduce an *edge-blocking* technique in Pangolin. Since an application starts expansion with single-edge embeddings, Pangolin blocks the initial embedding list into smaller chunks, and processes all levels (main loop in Algorithm 1) for each chunk one after another. As shown in Fig. 10, there are n edges in the initial embedding list ($e_0 \sim e_{n-1}$). Each chunk contains 4 edges which are assigned to the 2 threads ($t_0 \sim t_1$) to process. After all levels of the current chunk are processed, the threads move to the next chunk and continue processing until all chunks are processed. The chunk size C_s is a parameter to tune; C_s is typically much larger than the number of threads. Blocking will not affect parallelism because there are a large number of edges in each chunk that can be processed concurrently. Note that the FILTER phase requires strict synchronization in each level, so edge-blocking cannot be applied for applications that use it. For example, we need to gather embeddings for each pattern in FSM in order to compute the domain support. Due to this, all embeddings needs to be processed before moving to the next level, so we disable blocking for FSM. Currently, edge-blocking is used specifically for bounding memory usage, but it is also potentially beneficial for data locality with an appropriate block size. We leave this for future work.

5.3 Dynamic Memory Allocation

Inspection-Execution: Compared to graph analytics applications, GPM applications need significantly more dynamic memory allocations and memory allocation could become a performance bottleneck. A major source of memory allocation is the embedding list. As the size of embedding list increases, we need to allocate memory for the embeddings in each round. When generating the embedding list, there are write conflicts as different threads write to the same shared embedding list. In order to avoid frequent `resize` and `insert` operation, we use *inspection-execution* technique to generate the embedding list.

The generation include 3 steps. In the first step, we only calculate the number of newly generated embeddings for each embedding in the current embedding list. We then use parallel prefix sum to calculate the `start` index for each current embedding, and allocate the exact amount of memory for all the new embeddings. Finally, we actually write the new embeddings to update the embedding list, according to the `start` indices. In this way, each thread can write to the shared embedding list simultaneously without conflicts. Fig. 11 illustrates the inspection process. At level i, there are 4 embeddings e_0, e_1, e_2, e_3 in the embedding list, which will generate 1, 2, 1, 3 new embeddings respectively. We get the `start` indices (0, 1, 3, 4) using prefix sum, and then allocate memory for the level $i+1$ embedding list. Next, each embedding writes generated embeddings from its `start` index in the level $i+1$ list (concurrently).

Although inspection-execution requires iterating over the embeddings twice, making this tradeoff for GPU is reasonable for two reasons. First, it is fine for the GPU to do the recomputation as it has a lot of computation power. Second, improving the memory access pattern to better utilize memory bandwidth is more important for GPU. This is also a more scalable design choice for the CPU as the number of cores on the CPU are increasing.

Scalable Allocators: Pattern reduction in FSM is another case where dynamic memory allocation is frequently invoked. To compute the domain support of each pattern, we need to gather all the embeddings associated with the same pattern (see Fig. 2). This gathering requires resizing the vertex set of each domain. The C++ standard `std` library employs a concurrent allocator implemented by using a global lock for each allocation, which could seriously limit performance and scalability. We leverage the Galois memory allocator to alleviate this overhead. Galois provides an in-built efficient and concurrent memory allocator that implements ideas from prior scalable allocators [13, 67, 75]. The allocator uses per-thread memory pools of huge pages. Each thread manages its own memory pool. If a thread has no more space in its memory pool, it uses a global lock to add another huge page to its pool. Most allocations thus avoid locks. Pangolin uses variants of `std` data structures provided by Galois that use the Galois memory allocator. For example, this is used for maintaining the pattern map. On the other hand, our GPU infrastructure currently lacks support for efficient dynamic memory allocation inside CUDA kernels. To avoid frequent `resize` operations inside kernels, we conservatively calculate the memory space required and pre-allocate bit vectors for kernel use. This pre-allocation requires much more memory than is actually required, and restricts our GPU implementation to smaller inputs for FSM.

Table 1: Input graphs (symmetric, no loops, no duplicate edges) and their properties ($\overline{d}$ is the average degree).

Graph	Source	# V	# E	$\overline{d}$	Labels
Mi	Mico [32]	100,000	2,160,312	22	29
Pa	Patents [43]	2,745,761	27,930,818	10	37
Yo	Youtube [22]	7,066,392	114,190,484	16	29
Pdb	ProteinDB [82]	48,748,701	387,730,070	8	25
Lj	LiveJournal [58]	4,847,571	85,702,474	18	0
Or	Orkut [58]	3,072,441	234,370,166	76	0
Tw	Twitter [56]	21,297,772	530,051,090	25	0
Gsh	Gsh-2015 [15]	988,490,691	51,381,410,236	52	0

5.4 Other Optimizations

GPM algorithms make extensive use of connectivity operations for determining how vertices are connected in the input graph. For example, in k-cliques, we need to check whether a new vertex is connected to all the vertices in the current embedding. Another common connectivity operation is to determine how many vertices are connected to given vertices v_0 and v_1, which is usually obtained by computing the intersection of the neighbor lists of the two vertices. A naive solution of connectivity checking is to search for one vertex v_0 in the other vertex v_1's neighbor list sequentially. If found, the two vertices are directly connected. To reduce complexity and improve parallel efficiency, we

Table 2: Execution time (sec) of applications in GPM frameworks on 28-core CPU (option: minimum support for 3-FSM; k for others). AR, RS, KA, FR, and PA: Arabesque, RStream, Kaleido, Fractal, and Pangolin respectively. '-': out of memory or disk, or timed out in 30 hours. FR for `Yo` is omitted due to failed execution. FR does not contain TC. †KA results are reported from their paper.

		Mi					Pa					Yo			
App	**Option**	**AR**	**RS**	**KA†**	**FR**	**PA**	**AR**	**RS**	**KA†**	**FR**	**PA**	**AR**	**RS**	**KA†**	**PA**
TC		30.8	2.6	0.2		**0.02**	100.8	7.8	0.5		**0.08**	601.3	39.8	2.2	**0.3**
CF	3	32.2	7.3	0.5	24.7	**0.04**	97.8	39.1	0.6	350.2	**0.2**	617.0	862.3	2.2	**0.7**
	4	41.7	637.8	3.9	30.6	**1.6**	108.1	62.1	1.1	410.1	**0.4**	1086.9	-	7.8	**3.1**
	5	311.9	-	183.6	488.9	**60.5**	108.8	76.9	1.5	463.5	**0.5**	1123.6	-	19.0	**7.3**
MC	3	36.1	7137.5	1.4	41.2	**0.2**	101.6	3886.9	4.7	236.3	**0.9**	538.4	89387.0	35.5	**5.5**
	4	353.0	-	198.2	243.2	**175.6**	779.8	-	**152.3**	561.1	209.1	5132.8	-	4989.0	**4405.3**
3-FSM	300	104.9	56.8	7.4	780.5	**3.9**	340.7	230.1	25.5	720.3	**14.7**	666.9	1415.1	132.6	**96.9**
	500	72.2	57.9	8.2	773.1	**3.6**	433.6	208.6	26.4	817.0	**15.8**	576.5	1083.9	133.3	**97.8**
	1000	48.5	52.9	7.8	697.2	**3.0**	347.3	194.0	28.7	819.9	**18.1**	693.2	1179.3	136.2	**98.0**
	5000	36.4	35.6	3.9	396.3	**2.4**	366.1	172.2	31.5	915.5	**27.0**	758.6	1248.1	155.0	**102.2**

generalize the binary search approach proposed for TC [46] to implement connectivity check in Pangolin. This is particularly efficient on GPU, as it improves GPU memory efficiency. We provide efficient CPU and GPU implementations of these connectivity operations as helper routines, such as `isConnected` (Listing 2), which allow the user to easily compose pruning strategies in applications.

In summary, when no algorithmic optimization is applied, programming in Pangolin should be as easy as previous GPM systems like Arabesque. In this case, performance gains over Arabesque is achieved due to the architectural optimizations (e.g., data structures) in Pangolin. To incorporate algorithmic optimizations, the user can leverage Pangolin API functions (e.g., `toExtend` and `toAdd`) to express application-specific knowledge. While this involves slightly more programming effort, the user can get an order of magnitude performance improvement by doing so.

6. EVALUATION

In this section, we compare Pangolin with state-of-art GPM frameworks and hand-optimized applications. We also analyze Pangolin performance in more detail.

6.1 Experimental Setup

We compare Pangolin with state-of-the-art GPM frameworks: Arabesque [84], RStream [88], G-Miner [19], Kaleido [91], Fractal [30], and AutoMine [66]. Arabesque, G-Miner, and Fractal support distributed execution, while the rest support out-of-core execution. None of them support GPU execution. Kaleido and AutoMine results are reported from their papers because they are not publicly available. We also compare Pangolin with the state-of-the-art hand-optimized GPM applications [11, 44, 26, 4, 73, 82, 83, 52].

We test the 4 GPM applications discussed in Section 3.3, i.e., TC, CF, MC, and FSM. k-MC and k-CF terminate when subgraphs reach a size of k vertices. For k-FSM, we mine the frequent subgraphs with $k-1$ edges. Table 1 lists the input graphs used in the experiments. We assume that input graphs are symmetric, have no self-loops, and have no duplicated edges. We represent the input graphs in memory in a compressed sparse row (CSR) format. The neighbor list of each vertex is sorted by ascending vertex ID.

The first 3 graphs — `Mi`, `Pa`, and `Yo` — have been previously used by Arabesque, RStream, and Kaleido. We use the same graphs to compare Pangolin with these existing frameworks. In addition, we include larger graphs from SNAP Collection [58] (`Lj`, `Or`), Koblenz Network Collection [56] (`Tw`), DistGraph [82](`Pdb`), and a very large web-crawl [15] (`Gsh`). Except `Pdb`, other larger graphs do not have vertex labels, therefore, we only use them to test TC, CF, and MC. `Pdb` is used only for FSM.

Unless specified otherwise, CPU experiments were conducted on a single machine with Intel Xeon Gold 5120 CPU 2.2GHz, 4 sockets (14 cores each), 190GB memory, and 3TB SSD. AutoMine was evaluated using 40 threads (with hyperthreading) on Intel Xeon E5-2630 v4 CPU 2.2GHz, 2 sockets (10 cores each), 64GB of memory, and 2TB of SSD. Kaleido was tested using 56 threads (with hyperthreading) on Intel Xeon Gold 5117 CPU 2.0GHz, 2 sockets (14 cores each), 128GB memory, and 480GB SSD. To make our comparison fair, we restrict our experiments to use only 2 sockets of our machine, but we only use 28 threads without hyperthreading. For the largest graph, `Gsh`, we used a 2 socket machine with Intel's second generation Xeon scalable processor with 2.2 Ghz and 48 cores, equipped with 6TB of Intel Optane PMM [39] (byte-addressable memory technology). Our GPU platforms are NVIDIA GTX 1080Ti (11GB memory) and Tesla V100 (32GB memory) GPUs with CUDA 9.0. Unless specified otherwise, GPU results reported are on V100.

RStream writes its intermediate data to the SSD, whereas other frameworks run all applications in memory. We exclude preprocessing time and only report the computation time (on the CPU or GPU) as an average of 3 runs. We also exclude the time to transfer data from CPU to GPU as it is trivial compared to the GPU compute time.

6.2 GPM Frameworks

Table 2 reports the execution time of Arabesque, RStream, Kaleido, Fractal, and Pangolin. The execution time of G-Miner and AutoMine is reported in Table 3a and Table 4 respectively (because it does not have other applications or datasets respectively). Note that Kaleido and AutoMine results on 28-core and 20-core CPU, respectively, are reported from their papers. We evaluate the rest on our 28-core CPU, except that we evaluate Pangolin for `gsh` on 48-core CPU. Fractal and AutoMine use DFS exploration [89, 50], whereas the rest use BFS. Pangolin is an order-of-magnitude faster than Arabesque, RStream, Fractal, and G-Miner. Pangolin outperforms Kaleido in all cases except 4-MC on `patent`. Pangolin on CPU is comparable or slower than AutoMine but outperforms it by exploiting the GPU.

For small inputs (e.g., TC and 3-CF with `Mi`), Arabesque suffers non-trivial overhead due to the startup cost of Giraph. For large graphs, however, due to lack of algorithmic (e.g., eager pruning and customized pattern classification) and data structure optimizations, it is also slower than Pangolin. On average, Pangolin is 49× faster than Arabesque.

For RStream, the number of partitions P is a key performance knob. For each configuration, we choose P to be the

Table 3: Execution time (sec) of Pangolin (PA) and hand-optimized solvers (σ: minimum support). PA-GPU and DistTC-GPU are on V100 GPU; PGD-GPU is on Titan Black GPU; rest are on 28-core CPU. †PGD-GPU results are reported from their paper.

(a) TC. GM: G-Miner.

Input	G-Miner	GAP	PA-CPU	DistTC-GPU	PA-GPU
Lj	5.2	0.5	0.6	0.07	**0.06**
Or	13.3	4.2	3.9	0.3	**0.2**
Tw	1067.7	40.1	38.8	4.3	**2.9**

(b) 4-CF.

Input	KClist	PA-CPU	PA-GPU
Lj	**1.9**	26.3	2.3
Or	**4.1**	82.3	4.3
Tw	**628**	28165	1509

(c) 3-MC.

Input	PGD	PA-CPU	PGD-GPU†	PA-GPU
Lj	12.7	19.5	∼**1.4**	1.7
Or	46.9	175	∼**7.7**	18.0
Tw	1883	9388	-	1163

(d) 3-FSM. DG: DistGraph.

	Mico			Patent			Youtube			PDB		
σ	DG	PA-CPU	PA-GPU	DG	PA-CPU	PA-GPU	DG	PA-CPU	PA-GPU	DG	PA-CPU	PA-GPU
300	52.2	3.9	**0.6**	19.9	14.7	**2.7**	-	**96.9**	-	281.4	**63.7**	-
500	52.9	3.6	**0.5**	18.7	15.8	**2.7**	-	**97.7**	-	279.5	**65.6**	-
1000	59.1	3.0	**0.4**	18.6	18.1	**2.7**	-	**98.0**	-	274.5	**73.4**	-
5000	58.1	2.4	**0.2**	18.4	27.0	**1.7**	-	**102.3**	-	322.9	**145.3**	-

(e) 4-FSM for Patent.

σ	DG	PA-CPU
15K	**129.0**	438.9
20K	**81.9**	224.7
30K	**26.2**	31.9

Table 4: Execution time (sec) of Pangolin (PA) and AutoMine (AM). Pangolin for Gsh is evaluated on Intel Optane-PMM machine. †AutoMine results are reported from its paper.

(a) Mi.

	AM†	PA-CPU	PA-GPU
TC	0.04	0.02	**0.001**
3-MC	0.12	0.20	**0.02**
4-MC	22.0	175.6	**5.3**
5-CF	11.4	60.5	**9.7**

(b) Gsh.

	AM†	PA
TC	4966	139.3
3-CF	-	659.3
4-CF	45399	23475

Table 5: Lines of code in Pangolin (PA) and hand-optimized (HO) applications (implementation name in parenthesis).

	TC	CF	MC	FSM
HO	(GAP) 89	(KClist) 394	(PGD) 2,538	(DistGraph) 17,459
PA	26	36	82	252

best performing one among 10, 20, 50, and 100. RStream only supports edge-induced exploration and does not support pattern-specific optimization. This results in extremely large search spaces for CF and MC because there are many more edges than vertices. In addition, RStream does not scale well because of the intensive use of mutex locks for updating shared data. Lastly, Pangolin avoids inefficient data structures and expensive redundant computation (isomorphism test) used by RStream. Pangolin is 88× faster than RStream on average (Kaleido [91] also observes that RStream is slower than Arabesque).

On average, Pangolin is 2.6× faster than Kaleido (7.4×, 3.3×, 2.4×, and 1.6× for TC, CF, MC, and FSM respectively). This is mainly due to DAG construction and customized pattern classification in Pangolin.

Pangolin is on average 80× faster than Fractal. Fractal is built on Spark and suffers from overheads due to it. More importantly, some optimizations in hand-optimized DFS-based applications like PGD [4] and KClist [26] are not supported in Fractal, which limits its performance.

AutoMine uses a key optimization [4, 26] to remove redundant computation that can only be enabled in DFS-based exploration. Due to this, when pattern size k is large like in 5-CF and 4-MC, AutoMine is faster than Pangolin. However, since Pangolin uses BFS-based exploration which easily enables GPU acceleration, Pangolin on GPU is on average 5.8× faster than AutoMine. It is not clear how to enable DFS mode for GPU efficiently, especially when k is large. Note that for all the applications, AutoMine can only do counting but not listing, because it has no automorphism test during extension (instead it uses post-processing to address the multiplicity issue). FSM in AutoMine uses frequency (which is not anti-monotonic) instead of domain support, and thus it is not comparable to FSM in Pangolin.

6.3 Hand-Optimized GPM Applications

We compare hand-optimized implementations with Pangolin on CPU and GPU. We report results for the largest datasets supported on our platform for each application. Note that all hand-optimized applications involve substantially more programming effort than Pangolin ones. As shown in Table 5, hand-optimized TC has 4× more lines of code (LoC) than Pangolin TC and the other hand-optimized applications have one or two orders of magnitude more LoC than Pangolin ones. The Pangolin code for MC is shown in Listings 4 and 6. The lines in the other Pangolin applications are as simple as that in MC. Hand-optimized solvers must handle parallelism, synchronization, memory allocation, etc, while Pangolin transparently handles all of that, making it easier for the user to write applications.

In Table 3a, we compare with GAP [11] and DistTC [44], the state-of-the-art TC implementations on CPU and GPU, respectively. It is clear from Table 2 and Table 3a that TC implementations in existing GPM frameworks are orders of magnitude slower than the hand-optimized implementation in GAP. In contrast, Pangolin performs similar to GAP on the same CPU. Pangolin is also faster than DistTC on the same GPU due to its embedding list data structure, which has better load balance and memory access behavior.

Table 3b compares our 4-clique with KClist [26], the state-of-the-art CF implementation. Pangolin is 10 to 20× slower than KClist on the CPU, although GPU acceleration of Pangolin significantly reduces the performance gap. This is because KClist constructs a shrinking local graph for each edge, which significantly reduces the search space. This optimization can only be enabled in the DFS exploration. In Table 3c, we observe the same trend for 3-MC compared with PGD, the state-of-the-art MC solver for multicore CPU [4] and GPU [73]. Note that PGD can only do counting, but not listing, as it only counts some of the patterns and the other patterns' counts are calculated directly using some formulas. In contrast, MC in Pangolin can do both counting and listing. Another limitation of PGD is that it can only handle 3-MC and 4-MC, while Pangolin handles arbitrary k. As PGD for GPU (PGD-GPU) [73] is not released, we estimate PGD-GPU performance using their reported speedup [73] on Titan Black GPU. Pangolin-GPU is 20% to 130% slower.

Table 3d and Table 3e compares our 3-FSM and 4-FSM, respectively, with DistGraph [82, 83]. DistGraph supports both shared-memory and distributed platforms. DistGraph supports a runtime parameter σ, which specifies the minimum support, but we had to modify it to add the maximum size k. On CPU, Pangolin outperforms DistGraph for 3-FSM in all cases, except for Pa with support 5K. For graphs that fit in the GPU memory (Mi, Pa), Pangolin on GPU is 6.9× to 290× faster than DistGraph. In comparison, the GPU implementation of DistGraph is only 4× to 9× faster than its CPU implementation [52] (we are not able able to run their GPU code and we cannot compare

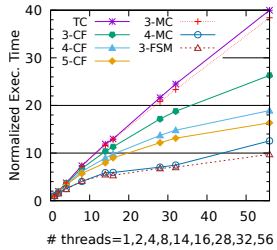


Figure 12: Strong scaling using Yo graph. σ=500 for FSM.

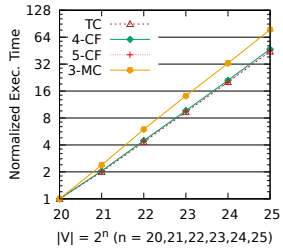


Figure 13: Execution time for RMAT graphs (log-log scale).

with their reported results as they do not evaluate the same datasets). For 4-FSM, Pangolin is 22% to 240% slower than DistGraph. The slowdown is mainly due to the algorithmic differences: DistGraph adopts DFS exploration and a recursive approach which reduces computation and memory consumption, while Pangolin does BFS exploration.

6.4 Scalability and GPU Performance

Although Pangolin is an in-memory processing system, Pangolin can scale to very large datasets by using large memory systems. To demonstrate this, we evaluate Pangolin on the Intel Optane PMM system and mine a very large real-world web crawl, Gsh. As shown in Table 4b, TC and 3-CF only take 2 and 11 minutes, respectively. 4-CF is much more compute and memory intensive, so it takes $\sim$ 6.5 hours.

Fig. 12 illustrates how the performance of Pangolin applications scales as the number of threads increases for different applications on Yo. Pangolin achieves good scalability by utilizing efficient, concurrent, scalable data structures and allocators. For TC, we observe near linear speedup over single-thread execution. In contrast, FSM's scalability suffers due to the overheads of computing domain support.

To test weak scaling, we use the RMAT graph generator [53] to generate graphs with vertices $|V|$ from 2^{20} to 2^{25} and average degree $\overline{d} = 20$. Fig. 13 reports the execution time normalized to that of rmat20 (log-log scale). The execution time grows exponentially as the graph size increases because the enumeration search space grows exponentially.

Fig. 14 illustrates speedup of Pangolin applications on GPU over 28-core CPU. Note that due to the limited memory size, GPUs fail to run some applications and inputs. On average, 1080Ti and V100 GPUs achieve a speedup of 6× and 15× respectively over the CPU. Specifically, we observe substantial speedup on CF and MC. For example, the V100 GPU achieves 50× speedup on 4-MC for Yo, demonstrating the suitability of GPUs for these applications.

6.5 Memory Consumption

The peak memory consumption for Arabesque, RStream, and Pangolin on the same 28-core CPU platform is illustrated in Fig. 15. We observe that Arabesque always requires the most memory because it is implemented in Java using Giraph [40] which allocates a huge amount of memory. In contrast, Pangolin avoids this overhead and reduces memory usage. Since Pangolin does in-memory computation, it is expected to consume much more memory than RStream which stores its embeddings in disk. However, we find that the difference in memory usage is trivial because aggressive search space pruning and customized pattern classification significantly reduce memory usage. Since this small memory cost brings substantial performance improvement, we believe Pangolin makes a reasonable trade-off.

6.6 Impact of Optimizations

We evaluate the performance improvement due to the optimizations described in Section 4 and Section 5. Due to lack of space, we present these comparisons only for the CPU implementations, but the results on the GPU are similar. Fig. 16a shows the impact of orientation (*DAG*) and user-defined eager pruning (*Prune*) on 4-CF. Both techniques significantly improve performance for TC (not shown) and CF. Fig. 16b demonstrates the advantage of using Galois memory allocators instead of std allocators. This is particularly important for FSM as it requires intensive memory allocation for counting support. Fig. 16c illustrates that customized pattern classification used in MC and FSM yields huge performance gains by eliding expensive generic isomorphism tests. Fig. 16d shows that materialization of temporary embeddings causes 11% to 37% slowdown for MC. This overhead exists in every application of Arabesque (and RStream), and is avoided in Pangolin. In Fig. 17a, we evaluate the performance of our proposed embedding list data structure with SoA layout and inspection-execution. Compared to the straight-forward embedding queue (mimic the AoS implementation used in Arabesque and RStream), the k-MC performance is 2.1× to 4.7× faster. Another optimization is employing binary search for connectivity check. Fig. 17b shows that binary search can achieve up to 6.6× speedup compared to linear search. Finally, Fig. 18 illustrates the last level cache (LLC) miss counts in the vertex extension phase of k-CF. We compare two data structure schemes for the embeddings, AoS and SoA. We observe a sharp reduction of LLC miss count by switching from AoS to SoA. This further confirms that SoA has better locality than AoS, due to the data reuse among embeddings.

7. RELATED WORK

GPM Applications: Hand-optimized GPM applications target various platforms. For triangle counting, Shun *et al.* [79] present a parallel, cache-oblivious TC solver on multi-core CPUs that achieves good cache performance without fine-tuning cache parameters. TriCore [46] is a multi-GPU TC solver that uses binary search to increase coalesced memory accesses, and it employs dynamic load balancing. There are several distributed TC solvers [81, 38, 44] too.

Chiba and Nishizeki (C&N) [24] proposed an efficient k-clique listing algorithm which computes the subgraph induced by neighbors of each vertex, and then recurses on the subgraph. Danisch *et al.* [26] refine the C&N algorithm for parallelism and construct DAG using a core value based ordering to further reduce the search space. PGD [4] counts 3 and 4-motifs by leveraging a number of proven combinatorial arguments for different patterns. Some patterns (*e.g.,* cliques) are counted first, and the frequencies of other patterns are obtained in constant time using these combinatorial arguments. Escape [71] extends this approach to 5-vertex subgraphs and leverages DAG to reduce search space.

gSpan [90] is an efficient sequential FSM solver which does depth-first search (DFS) based on a lexicographic order. GraMi [32] proposes an approach that finds only the minimal set of instances to satisfy the support threshold and avoids enumerating all instances. DistGraph [82] parallelizes gSpan for both shared-memory and distributed CPUs. Each

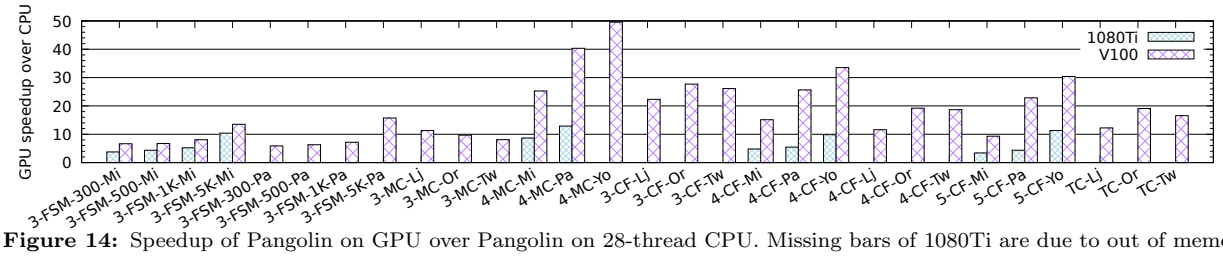


Figure 14: Speedup of Pangolin on GPU over Pangolin on 28-thread CPU. Missing bars of 1080Ti are due to out of memory.

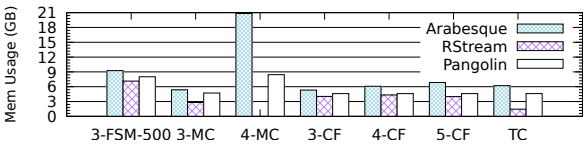
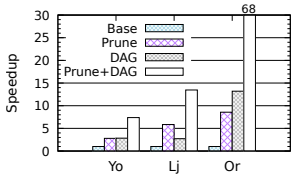
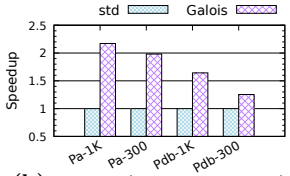


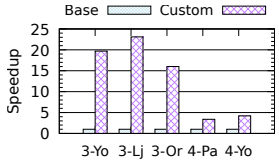
Figure 15: Peak memory usage in Arabesque, RStream, and Pangolin for `Pa` (4-MC in RStream runs out of memory).



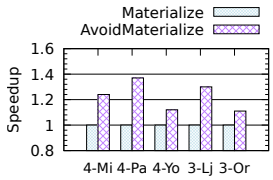
(a) 4-CF (pruning)



(b) 3-FSM (Galois allocator)



(c) k-MC (customized patterns)



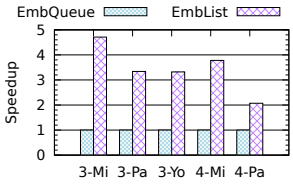
(d) k-MC (avoid materialization)

Figure 16: Speedup due to various optimizations: (a) eager pruning and DAG; (b) Galois scalable memory allocator; (c) customized pattern classification; (d) avoiding materialization.

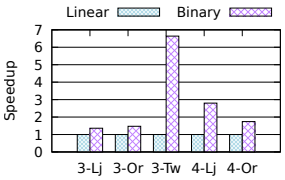
worker thread does the DFS walk concurrently. It introduces a customized dynamic load balancing strategy which splits tasks on the fly and recomputes the embedding list from scratch after the task is sent to a new worker. Scalemine [1] solves FSM with a two-phase approach, which approximates frequent subgraphs in phase-1, and uses collected information to compute the exact solution in phase-2. There are other GPM applications, e.g. maximal cliques [21], maximum clique [63, 2], and *subgraph listing* [77, 14, 54, 49, 55, 64, 57]. All the above hand-optimized solvers employ various optimizations to reduce computation and improve hardware efficiency. However, they achieve high performance at the cost of tremendous programming efforts, while Pangolin provides a unified model for ease of programming.

GPM Frameworks: For ease-of-programming, GPM systems such as Arabesque [84], RStream [88], G-Miner [19], and Kaleido [91] have been proposed. They provide a unified programming interface to the user which simplifies application development. However, their interface is not flexible enough to enable application specific optimizations. Instead of the BFS exploration used in these frameworks, Fractal [30] employs a DFS strategy to enumerate subgraphs, which reduces memory footprint. AutoMine [66] is a compiler based system using DFS exploration. In contrast, Pangolin uses the BFS approach that is inherently more load-balanced, and is better suited for GPU acceleration.

Approximate GPM: There are approximate solvers for TC [86, 72, 85], CF [69, 48], MC [80, 16], and FSM [7].



(a)



(b)

Figure 17: k-MC speedup of (a) using embedding list (SoA+inspection-execution) over using embedding queue (AoS) and (b) binary search over linear search.

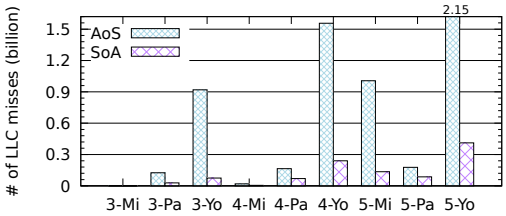


Figure 18: LLC miss counts in the vertex extension phase of k-CF using AoS and SoA for embeddings.

ASAP [47] is an approximate GPM framework that reduces computation at the cost of less than 5% error. Chen and Lui [20] propose another approximate GPM system based on random walk. Compared to approximate solutions, Pangolin focuses on exact GPM and achieves high performance without sacrificing accuracy.

8. CONCLUSION

We present Pangolin, a high-performance, flexible GPM system on shared-memory CPUs and GPUs. Pangolin provides a simple API that enables the user to specify eager enumeration search space pruning and customized pattern classifications. To exploit locality, Pangolin uses an efficient structure of arrays (SoA) for storing embeddings. It avoids materialization of temporary embeddings and blocks the schedule of embedding exploration to reduce the memory usage. It also uses inspection-execution and scalable memory allocators to mitigate the overheads of dynamic memory allocation. These application-specific and architectural optimizations enable Pangolin to outperform prior GPM frameworks, Arabesque, RStream, and Fractal, by 49×, 88×, and 80×, on average, respectively, on the same 28-core CPU. Moreover, Pangolin on V100 GPU is 15× faster than that on the CPU on average. Thus, Pangolin provides performance competitive with hand-optimized implementations but with much better programming experience.

Acknowledgments

The research was supported by NSF grants 1406355, 1618425, 1705092, and 1725322; NSFC grant 61802416; and DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563. We thank Intel for providing the Intel Optane DC PMM machine.

9. REFERENCES

[1] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 61:1–61:12, Piscataway, NJ, USA, 2016. IEEE Press.

[2] A. Aboulnaga, J. Xiang, and C. Guo. Scalable maximum clique computation using mapreduce. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 74–85, Washington, DC, USA, 2013. IEEE Computer Society.

[3] C. C. Aggarwal and H. Wang. *Managing and Mining Graph Data*. Springer US, 2010.

[4] N. K. Ahmed, J. Neville, R. A. Rossi, and N. Duffield. Efficient graphlet counting for large networks. In *ICDM*, pages 1–10, 2015.

[5] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):241–249, 2008.

[6] N. Alon, R. Yuster, and U. Zwick. Color-coding: A new method for finding simple paths, cycles and other small subgraphs within large graphs. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 326–335, New York, NY, USA, 1994. ACM.

[7] P. Anchuri, M. J. Zaki, O. Barkol, S. Golan, and M. Shamy. Approximate graph mining with label costs. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 518–526, New York, NY, USA, 2013. ACM.

[8] L. Babai, W. M. Kantor, and E. M. Luks. Computational complexity and the classification of finite simple groups. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 162–171, Nov 1983.

[9] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie. Analysis and optimization of the memory hierarchy for graph processing workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–386, Feb 2019.

[10] S. Beamer, K. Asanovic, and D. Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, IISWC '15, pages 56–65, Washington, DC, USA, 2015. IEEE Computer Society.

[11] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[12] A. R. Benson, D. F. Gleich, and J. Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.

[13] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[14] B. Bhattarai, H. Liu, and H. H. Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1447–1462, New York, NY, USA, 2019. ACM.

[15] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[16] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi. Counting graphlets: Space vs time. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, pages 557–566, New York, NY, USA, 2017. ACM.

[17] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi. Motif counting beyond five nodes. *ACM Trans. Knowl. Discov. Data*, 12(4):48:1–48:25, Apr. 2018.

[18] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, Nov 2012.

[19] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: An efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys 18, New York, NY, USA, 2018. Association for Computing Machinery.

[20] X. Chen and J. C. S. Lui. Mining graphlet counts in online social networks. *ACM Trans. Knowl. Discov. Data*, 12(4), Apr. 2018.

[21] J. Cheng, L. Zhu, Y. Ke, and S. Chu. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1240–1248, New York, NY, USA, 2012. ACM.

[22] X. Cheng, C. Dale, and J. Liu. Dataset for statistics and social network of youtube videos. http://netsg.cs.sfu.ca/youtubedata/.

[23] U. Cheramangalath, R. Nasre, and Y. N. Srikant. Falcon: A graph manipulation language for heterogeneous systems. *ACM Trans. Archit. Code Optim.*, 12(4), Dec. 2015.

[24] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, Feb. 1985.

[25] Y.-R. Cho and A. Zhang. Predicting protein function by frequent functional association pattern mining in protein interaction networks. *Trans. Info. Tech. Biomed.*, 14(1):30–36, Jan. 2010.

[26] M. Danisch, O. Balalau, and M. Sozio. Listing k-cliques in sparse real-world graphs*. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 589–598, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

[27] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali. Gluon:

A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 752–768, New York, NY, USA, 2018. ACM.

[28] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, V. Jatala, V. K. Nandivada, M. Snir, and K. Pingali. Gluon-Async: A Bulk-Asynchronous System for Distributed and Heterogeneous Graph Analytics. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques (PACT 2019)*, PACT '19. IEEE, 2019.

[29] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering*, 17(8):1036–1050, Aug 2005.

[30] V. Dias, C. H. C. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1357–1374, New York, NY, USA, 2019. ACM.

[31] E. R. Elenberg, K. Shanmugam, M. Borokhovich, and A. G. Dimakis. Beyond triangles: A distributed framework for estimating 3-profiles of large graphs. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 229–238, New York, NY, USA, 2015. ACM.

[32] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *PVLDB*, 7(7):517–528, 2014.

[33] S. Eyerman, W. Heirman, K. D. Bois, J. B. Fryman, and I. Hur. Many-core graph workload analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 22:1–22:11, Piscataway, NJ, USA, 2018. IEEE Press.

[34] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian. Parallelizing sequential graph computations. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD 17, page 495510, New York, NY, USA, 2017. Association for Computing Machinery.

[35] K. Faust. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Social Networks*, 32(3):221 – 233, 2010.

[36] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*, 2006.

[37] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[38] I. Giechaskiel, G. Panagopoulos, and E. Yoneki. PDTL: Parallel and distributed triangle listing for massive graphs. In *2015 44th International Conference on Parallel Processing*, pages 370–379, Sep. 2015.

[39] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali. Single machine graph analytics on massive datasets using intel optane DC persistent memory. *CoRR*, abs/1904.07162, 2019.

[40] Apache Giraph. http://giraph.apache.org/, 2013.

[41] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[42] O. Green, P. Yalamanchili, and L.-M. Munguía. Fast triangle counting on the gpu. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '14, pages 1–8, Piscataway, NJ, USA, 2014. IEEE Press.

[43] B. H. Hall, J. A. B., and T. M. The NBER patent citation data file: Lessons, insights and methodological tools. http://www.nber.org/patents/, 2001.

[44] L. Hoang, V. Jatala, X. Chen, U. Agarwal, R. Dathathri, G. Gill, and K. Pingali. DistTC: High performance distributed triangle counting. In *HPEC 2019 23rd IEEE High Performance Extreme Computing, Graph Challenge*, September 2019.

[45] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan. Multigraph: Efficient graph processing on gpus. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 27–40, Sep. 2017.

[46] Y. Hu, H. Liu, and H. H. Huang. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182, Nov 2018.

[47] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica. Asap: Fast, approximate graph pattern mining at scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 745–761, Berkeley, CA, USA, 2018. USENIX Association.

[48] S. Jain and C. Seshadhri. A fast and provable method for estimating clique counts using turán's theorem. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 441–449, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[49] M. Jha, C. Seshadhri, and A. Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 495–505, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee.

[50] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 000:1–31, 01 2004.

[51] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 135–149, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[52] R. Kessl, N. Talukder, P. Anchuri, and M. J. Zaki. Parallel graph mining with gpus. In *Proceedings of the 3rd International Conference on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications - Volume 36*, BIGMINE'14, pages 1–16. JMLR.org, 2014.

[53] F. Khorasani, R. Gupta, and L. N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT 15, page 3950, USA, 2015. IEEE Computer Society.

[54] H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrah. DUALSIM: Parallel subgraph enumeration in a massive graph on a single machine. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1231–1245, New York, NY, USA, 2016. ACM.

[55] K. Kim, I. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 411–426, New York, NY, USA, 2018. ACM.

[56] J. Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[57] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 8(10):974–985, 2015.

[58] J. Leskovec. Snap: Stanford network analysis platform, 2013.

[59] W. Lin, X. Xiao, X. Xie, and X. Li. Network motif discovery: A gpu approach. In *2015 IEEE 31st International Conference on Data Engineering*, pages 831–842, April 2015.

[60] H. Liu and H. H. Huang. Simd-x: Programming and processing of graph algorithms on gpus. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC 19, page 411427, USA, 2019. USENIX Association.

[61] Y. Liu, B. Schmidt, W. Liu, and D. L. Maskell. Cuda-meme: Accelerating motif discovery in biological sequences using cuda-enabled graphics processing units. *Pattern Recognition Letters*, 31(14):2170 – 2177, 2010.

[62] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *Proceedings Conf. Uncertainty in Artificial Intelligence*, UAI '10, July 2010.

[63] C. Lu, J. X. Yu, H. Wei, and Y. Zhang. Finding the maximum clique in massive graphs. *PVLDB*, 10(11):1538–1549, 2017.

[64] S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed graph pattern matching. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 949–958, New York, NY, USA, 2012. ACM.

[65] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[66] D. Mawhirter and B. Wu. Automine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP 19, page 509523, New York, NY, USA, 2019. Association for Computing Machinery.

[67] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 35–46, New York, NY, USA, 2004. ACM.

[68] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[69] M. Mitzenmacher, J. Pachocki, R. Peng, C. Tsourakakis, and S. C. Xu. Scalable large near-clique detection in large-scale networks via sampling. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 815–824, New York, NY, USA, 2015. ACM.

[70] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[71] A. Pinar, C. Seshadhri, and V. Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 1431–1440, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[72] M. Rahman and M. A. Hasan. Approximate triangle counting algorithms on multi-cores. In *2013 IEEE International Conference on Big Data*, pages 127–133, Oct 2013.

[73] R. A. Rossi and R. Zhou. Leveraging multiple gpus and cpus for graphlet counting in large networks. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, pages 1783–1792, New York, NY, USA, 2016. ACM.

[74] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proceedings of the 4th International Conference on Experimental and Efficient Algorithms*, WEA'05, pages 606–609, Berlin, Heidelberg, 2005. Springer-Verlag.

[75] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 84–94, New York, NY, USA, 2006. ACM.

[76] D. Sengupta and S. L. Song. Evograph: On-the-fly efficient mining of evolving graphs on gpu. In J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, editors, *High Performance Computing*, pages 97–119, Cham,

2017. Springer International Publishing.

[77] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 625–636, New York, NY, USA, 2014. ACM.

[78] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[79] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 149–160, April 2015.

[80] G. M. Slota and K. Madduri. Complex network analysis using parallel approximate motif counting. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 405–414, May 2014.

[81] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 607–614, New York, NY, USA, 2011. ACM.

[82] N. Talukder and M. J. Zaki. A distributed approach for graph mining in massive networks. *Data Min. Knowl. Discov.*, 30(5):1024–1052, Sept. 2016.

[83] N. Talukder and M. J. Zaki. Parallel graph mining with dynamic load balancing. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 3352–3359, Dec 2016.

[84] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 425–440, New York, NY, USA, 2015. ACM.

[85] C. E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *2008 Eighth IEEE International Conference on Data Mining*, pages 608–617, Dec 2008.

[86] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 837–846, New York, NY, USA, 2009. ACM.

[87] C. Voegele, Y.-S. Lu, S. Pai, and K. Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In *IEEE/Amazon/DARPA GraphChallenge, IEEE HPEC,*, 2017.

[88] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 763–782, Berkeley, CA, USA, 2018. USENIX Association.

[89] T. Washio and H. Motoda. State of the art of graph-based data mining. *SIGKDD Explor. Newsl.*, 5(1):5968, July 2003.

[90] Xifeng Yan and Jiawei Han. gspan: graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, pages 721–724, Dec 2002.

[91] C. Zhao, Z. Zhang, P. Xu, T. Zheng, and X. Cheng. Kaleido: An efficient out-of-core graph mining system on A single machine. *CoRR*, abs/1905.09572, 2019.

[92] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 301–316, Berkeley, CA, USA, 2016. USENIX Association.