ABSTRACT

There is growing interest in graph mining problems such as motif counting. Generic graph mining 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, a high-performance and flexible in-memory graph mining framework targeting shared-memory CPUs and GPUs. Pangolin is the first graph mining system that supports GPU processing. It provides a simple programming interface based on the extend-reduce-filter model, which enables user to specify application-specific knowledge for aggressive 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 Arabesque and RStream, two state-of-the-art graph mining frameworks, by 49× and 88× 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 mining applications, Pangolin provides competitive performance with much less programming effort.

1. INTRODUCTION

Applications that use graph data are becoming increasingly important in many fields such as world wide web, advertising, social media, and biology. 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. Another important class of graph problems deals with graph mining or graph pattern mining, which has applications in areas such as chemical engineering, bioinformatics, and social sciences. Graph mining discovers relevant patterns in a given graph. One example is triangle counting, which is used to mine graphs in security applications. Another example is motif counting, 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 algorithms, graph mining algorithms are more difficult to implement on parallel platforms; for example, unlike graph analytics algorithms, they usually generate enormous amounts of intermediate data. Systems such as Arabesque and RStream have been developed to provide abstractions for improving programmer productivity. Instead of the vertex-centric model used in graph analytics systems, 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 the Patent graph with 2.7M vertices and 28M edges, while a custom solver (KClist) counts it in only 0.16s. This huge performance gap significantly limits the usability of existing mining frameworks in real-world applications.

The first reason for this poor performance is that existing graph mining 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 mining 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 graph mining systems target either distributed or out-of-core platforms, and thus are not well optimized for shared-memory multicore/manycore architectures.

In this paper, we present Pangolin, a high-performance in-memory graph mining framework. Pangolin provides a simple yet flexible embedding-centric programming interface, based on the extend-reduce-filter model, which enables application-specific customization. Application developers can implement aggressive pruning strategies to reduce the enumeration search space, and apply customized pattern classification methods to elide generic isomorphism tests.

To make full use of parallel hardware, we optimize common 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 [58] parallel library and LonestarGPU [16] 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.

Experimental results on a 28-core CPU demonstrate that Pangolin outperforms existing state-of-the-art graph mining frameworks, Arabesque and RStream, by 49× and 88×, 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 graph mining 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 edges, Pangolin takes ~6.5 hours on a 48-core Intel Optane PMM machine [51] 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 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 consumption, and mitigate the overheads of dynamic memory allocation and synchronization on both CPUs and GPUs.
- We evaluate Pangolin on a multicore CPU and a GPU to demonstrate that Pangolin is not only substantially faster than existing mining frameworks but also competitive with hand-optimizing applications, while requiring less programming effort.

The rest of this paper is organized as follows. Section 2 provides background on existing mining frameworks and hand-optimized implementations. Section 3 introduces the Pangolin API and execution model. Section 4 describes how Pangolin enables application-specific optimizations using its API. Section 5 describes Pangolin’s implementation and architectural optimizations. Section 6 presents our evaluation of Pangolin and state-of-the-art mining applications and frameworks. Related work is discussed in Section 7 and conclusions are presented in Section 8.

## 2. BACKGROUND AND MOTIVATION

In this section, we start with describing graph mining concepts and applications. We discuss algorithmic and architectural optimizations in state-of-the-art hand-optimized mining applications. Lastly, we introduce the performance limitations of existing frameworks.

### 2.1 Graph Mining Concepts

Fig. 2 illustrates the basic concepts of graph mining. We are given an input graph that needs to be mined. A pattern is a template, while an embedding is an instance of that pattern in the graph (i.e., an appearance of the pattern). For the input graph and pattern in Fig. 2 there are four embeddings, shown on the right of the figure. In a graph mining problem, the user may provide a specific pattern such as triangle or clique, or require all “interesting” patterns to be discovered or counted. What makes a pattern interesting is defined by the user application; for example, in frequent subgraph mining, users are interested in patterns that appear frequently (above a given threshold) in the input graph. A measure of the pattern frequency in the input graph, termed support, is defined by the user.

There are two types of graph mining 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 is an edge-induced mining problem.

A graph mining algorithm enumerates the embeddings of the given pattern(s). If duplicate embeddings exist (automorphism), then the algorithm chooses one of them as the canonical one and collects statistical information about these canonical embeddings such as the total count. Enumeration of embeddings in a graph grows exponentially with
the embedding size (number of vertices or edges), which is computationally expensive and consumes lots of memory. In addition, a graph isomorphism test is needed for each embedding to determine whether it is isomorphic to a pattern. Unfortunately, graph isomorphism problem is not solvable in polynomial time\(^{29}\), and implementing an isomorphism test is compute and memory intensive\(^{42}\).

2.2 Hand-Optimized Mining 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, a k-clique is one of the patterns in k-motifs.

FSM finds frequent patterns in a labeled graph. A measure of frequency called support is provided by the application developer, and all patterns with support above a given threshold are considered to be frequent and must be discovered. A simple definition of support is the count of the embeddings associated with the pattern (used in by TC, CF, MC). A more widely used support definition is minimum image-based (MNI) support\(^{4}\) (a.k.a domain support). As shown in Fig. 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.

Several hand-optimized implementations exist for each of these applications on multicore CPU\(^{46,3,23,15,70}\), GPU\(^{34,49,50,43}\), distributed CPU\(^{68,30,69}\), and multi-GPU\(^{37,36,61}\). They employ application-specific optimizations to reduce algorithm complexity. The complexity of mining 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: The graphs are converted into directed acyclic graphs (DAGs) in state-of-the-art TC\(^{27}\), CF\(^{21}\) and MC\(^{59}\) solvers, to significantly reduce the search space. In general mining applications, new embeddings are generated by extending existing embeddings and then can be discarded because they are either not interesting or a duplicate (automorphism). However, in some applications like CF\(^{21}\), 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.

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\(^{3}\), 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\(^{4}\)) 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 order-of-magnitude faster\(^{3}\).

Summary: Most of the algorithmic optimizations exploit application-specific knowledge, which can only be enabled by application developers. A generic mining 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. Panguo is the first mining framework to do so.

2.3 Existing Mining Frameworks

Existing graph mining systems target either distributed-memory\(^{71,24,39}\) or out-of-core\(^{75}\) platforms, and they make tradeoffs specific for their targeted architectures. None of them target in-memory mining 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 mining systems to run on a multicore CPU or GPU would lead to inefficient implementations. We first describe two of these mining systems briefly and then discuss their major limitations.

Arabesque\(^{71}\) is a distributed graph mining 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\(^{75}\) is an out-of-core graph mining system on a single machine. Its programming model is based on relational algebra. Users specify how to generate embeddings using relational operations like select, join, and aggregate. It stores intermediate data (i.e., the embeddings) on disk while the input graph data is kept in memory for reuse. It

\(^{1}\)A k-vertex complete subgraph is a connected subgraph in which each vertex has degree of k – 1 (i.e., each vertex is connected to all other vertices in the subgraph).

\(^{4}\)MNI support has the anti-monotonic property, which states that the support of a supergraph should not exceed the support of a subgraph; this allows us 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\(^{3}\).
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 mining 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 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 mining systems are not efficient. Both Arabesque and RStream store embeddings in an array of structures (AoS), where the embedding structures consist 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 checking for each embedding, which has been demonstrated to be very expensive for large graphs.

**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:** The number of (intermediate) embeddings are not known before executing the mining algorithm, so memory for them needs to be allocated dynamically. 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 scaling significantly.

**Summary:** Existing graph mining 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**

This section introduces our graph mining framework, Pangolin, which uses the “think like an embedding” (TLE) programming paradigm. First, we describe the execution model used in Pangolin. We then introduce the programming interface (i.e. APIs) provided to the user. Finally, we present examples of applications expressed in Pangolin.

![Figure 3: An example of vertex extension.](image)

![Figure 4: The reduction operation that calculates pattern frequency using a pattern map.](image)

**Algorithm 1 Execution Model for Mining**

```plaintext
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) ↦ clear the pattern map
9:     p_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
```

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). The newly generated embeddings then form the output worklist for the next level. The embedding size is increased with level until the user-defined maximum size is reached (line 14). Fig. 3 shows an example of the first iteration of vertex-based extension. The input worklist consists 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. 4 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 to calculate the support. As a result, we get the pattern-map as {[(triangle, 2), [wedge, 1]]}. 


Algorithm 2 Compute Phases in Vertex-induced Mining

1: procedure EXTEND(in_wl, out_wl)
2:   for each embedding emb ∈ 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 ∪ u to out_wl

8: procedure REDUCE(queue, p_map)
9:   p_map ← ∅
10:  for each embedding emb ∈ queue in parallel do
11:     Pattern pt ← getPattern(emb)
12:     Support sp ← getSupport(emb)
13:     p_map[pt] ← AGGREGATE(p_map[pt], sp)

14: procedure FILTER(in_wl, p_map, out_wl)
15:  for each embedding emb ∈ in_wl in parallel do
16:    if toPrune(emb, p_map) = false then
17:      insert emb to out_wl

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 soon after initializing single-edge embeddings (line 4). However, we omit this 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 a vertex in the current embedding needs to be extended. Extended embeddings can have duplicates due to automorphism. Fig. 5 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. 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 [71], 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, toPrune is used to specify those embeddings the user is no longer interested in. This depends on the support for the embedding’s canonical pattern (that is in the computed pattern map).

Pangolin also provide 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 to the user. These routines are commonly used in mining applications; e.g., to check connectivity or canonicality. Pangolin also provides an implementation of domain (MIN) support. The helper functions are available on both CPU and GPU, with efficient implementation on each architecture.

Comparison With Other Graph Mining APIs: Existing graph mining frameworks do not expose toExtend and getPattern to the application developer (instead, they assume these functions always return true and a canonical pattern, respectively). However, existing embedding-centric frameworks like Arabesque can easily expose these functions to enable application-specific optimizations (Section 4) like in Pangolin.

3.3 Applications in Pangolin

TC, CF, and MC use vertex-induced embeddings, while FSM uses edge-induced embeddings. Listings 2 to 6 show how TC, MC, and FSM implemented in Pangolin (we omit TC due to lack of space). For TC, the extension happens only once: the second vertex is extended to get the third vertex. After that, all we need to check is whether the third vertex...
is connected to the first vertex. If it is, this 3-vertex embedding 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 3 shows MC. An extended embedding is added only if it is canonical according to an automorphism check. In the reduction phase, a quick pattern of each embedding is first obtained (by default) 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 mining application. As shown in Listing 5, it uses the custom domain (MNI) 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 this application, the Pangolin implementation is still much simpler than hand-optimized FSM implementations [69, 1, 26], 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. We first describe enabling enumeration search space pruning and then describe enabling the eliding of isomorphism tests.

4.1 Pruning Enumeration Search Space

Directed Acyclic Graph (DAG): In typical mining applications, the input graph is undirected. In some vertex-induced mining applications, a common optimization technique is orientation which converts the undirected input graph into a directed acyclic graph (DAG) [10, 5]. 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 [62], clique finding [21], and motif counting [39]. 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 among the vertices based on their degrees: edges will point towards the vertex with higher degree. When there is a tie, the edge points to the vertex with the larger vertex ID. Other orderings can be included in the future. In Pangolin, the user can enable orientation by simply setting a macro.

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 [21], 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 2, 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
5. IMPLEMENTATION OF PANGOLIN 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 shared-memory multicore CPU and GPU. Its CPU implementation is built using the Galois [58] library and its GPU implementation is built using the LonestarGPU [16] infrastructure. Pangolin includes several architectural optimizations. In this section, we briefly describe some of them: (1) exploiting locality and fully utilizing memory bandwidth [27, 8]; (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. Existing systems use array-of-structures (AoS) to organize the embeddings, which leads to poor locality, especially for GPU computing. In Pangolin, we use a structure of arrays (SoA) to store the embeddings in memory. The SoA layout is particularly beneficial for parallel processing on the GPU as memory accesses to the embeddings are fully coalesced.

Fig. 5 illustrates the embedding list data structure. On the left is the prefix-tree that illustrates the embedding expansion process in Fig. 3. 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 grey 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.

We can reconstruct each embedding 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

```c
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-motif.
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. The history information $his$ 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 Materialization of Data Structures

**Loop Fusion**: Existing mining systems first collect all the embedding candidates into a list and then call the user-defined function (like $toAdc$) to select embeddings from the list. This leads to materialization of the candidate embeddings list. In contrast, Pangolin preemptively discards embedding candidates using the $toAdc$ function before adding it to the embedding list (as shown in Algorithm 2), thereby avoiding the materialization of the candidate embeddings (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 as 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 process each chunk one after another. For large graphs, blocking will not affect parallelism, since there are still a large number of edges in each chunk that can be processed concurrently. Note that edge-blocking does not work for applications that require strict synchronization in each level. For example, we need to gather embeddings for each pattern in FSM in order to compute the domain support. In the way all embeddings needs to be processed before turning to the next level and thus we disable blocking for FSM. Currently edge-blocking is particularly used for bounding memory usage, however it is also potentially beneficial for data locality if we carefully determine the block size. We leave this for future work.

5.3 Dynamic Memory Allocation

**Inspection-Execution**: Compared to graph analytics applications, graph mining 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 the 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 three steps. In the first step, we only calculate the number of newly generated embeddings for each embedding in the current embedding list, and 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.

Although inspection-execution requires iterating over the embeddings twice, but 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**: The pattern reduction in FSM is another case where dynamic memory allocation is frequently invoked. To calculate the domain (MNI) support of each pattern, we need to gather all the embeddings associated with the same pattern. This gathering requires resizing the vertex set of each domain (as shown in Fig. 2). 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 [11, 55, 63]. 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, then 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 useful in maintaining the pattern map itself. 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.

5.4 Other Optimizations
Graph mining 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 $0$ and $1$ edges. Table 1 lists the first 3 graphs — Mi, Pa, and Yo — have no self-loops, and have no duplicate 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.

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 employ binary search for the connectivity check. Binary search is particularly efficient on GPU, as it improves GPU memory access efficiency [37]. We provide efficient CPU and GPU implementations of these connectivity operations as helper routines, e.g., isConnected (Listing 2), which allow the user to easily compose pruning strategies in applications.

### 6. Evaluation

In this section, we first present our experimental setup. Then, we compare Pangolin with state-of-art graph mining frameworks and hand-optimized applications. Finally, we analyze the performance of Pangolin in more detail.

#### 6.1 Experimental Setup

We compare Pangolin with state-of-the-art graph mining frameworks: Arabesque [71], RStream [75], and Kaleido [77]. Kaleido is concurrent work and is not publicly available, but we include numbers reported in their arxiv for comparison.

We test the 4 mining 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 shows 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 [48] ($Lj$, $Or$), Koblenz Network Collection ($Tw$), DistGraph [69] ($Pdb$), and a very large web-crawl [46] ($Gsh$). 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 4 socket machine with Intel Xeon Gold 5120 CPU 2.2GHz with 56 cores (14 cores per socket), 190GB memory, and 3TB 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, however, 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 [31] (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.

For RStream, the number of partitions $P$ is a key performance knob. For each configuration, we choose $P$ to be the best performing one among 10, 20, 50, and 100. Arabesque and Pangolin run all applications in memory, whereas RStream writes its intermediate data to the SSD. We exclude preprocessing time and only report the computation time (on the CPU or GPU) as an average of 3 runs.

#### 6.2 Graph Mining Frameworks

Table 1 reports the execution time of the four systems. On the same 28-core CPU, Pangolin is an order-of-magnitude faster than both Arabesque and RStream. We observe that for small inputs (e.g., TC and 3-CF with Mi), Arabesque suffers non-trivial overhead due to the startup cost of Giraph. Moreover, due to lack of eager pruning, lack of customized pattern classification, and ODAG data structure, it is also slower than Pangolin for larger datasets. On average, Pangolin is 49× faster than Arabesque.

RStream only supports edge-induced exploration and does not not 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 the average (Kaleido also observes that RStream is slower than Arabesque). Pangolin outperforms Kaleido in all cases except 4-MC on Pdb. On average, it 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.

#### 6.3 Hand-Optimized Mining Applications

We compare hand-optimized implementations with Pangolin on CPU and GPU. Depending on the memory usage of each application, we report results for the largest datasets supported on our platform.

For TC, we compare with the graph benchmark suite, GAP [9] in Table 1. TC implementations in existing mining frameworks are orders of magnitude slower than the hand-optimized implementation in GAP. In contrast, Pangolin achieves competitive performance compared to GAP, which makes it a practical framework for real-world use cases.
Table 3: Execution time (sec) of mining applications (option: minimum support for 3-FSM; k for other applications) in graph mining frameworks on 28 threads (‘-’ indicates out of memory or disk, or timed out in 30 hours).

| Applications      | Input | Archesse | RStream | Kaleido | Pangolin | Archesse | RStream | Kaleido | Pangolin | Archesse | RStream | Kaleido | Pangolin |
|-------------------|-------|----------|---------|---------|----------|----------|---------|---------|----------|----------|----------|---------|---------|----------|
| TC                | 3     | 30.78    | 2.56    | 0.17    | 0.02     | 100.75   | 7.81    | 0.72    | 0.08     | 901 29   | 38.82   | 2.24    | 0.34     |
| CF                | 4     | 32.23    | 7.29    | 0.46    | 0.04     | 97.82    | 35.10   | 0.56    | 0.15     | 617.04   | 862.25  | 2.16    | 0.69     |
| MC                | 3     | 31.19    | 311.89  | -       | 183.63   | 108.78   | 868.88  | 4.74    | 0.90     | 833 70   | 903.90  | 35.47   | 5.34     |
| 3-FSM             | 4     | 22.57    | 72.18   | 57.88   | 8.19     | 343.55   | 208.63  | 26.41   | 0.40     | 1086.88  | 7.83    | 3.08    | 7.32     |
| Youtube           | 3     | 36.96    | 1734.87 | 139.3   | 0.21     | 101.56   | 386.88  | 141.52  | 132.59   | 1123.63  | 18.99   | 7.32    | 0.56     |
| Arabesque        | 5     | 352.96   | 35.47   | 637.77  | 3.88     | 108.78   | 868.88  | 4.74    | 0.90     | 833 70   | 903.90  | 35.47   | 5.34     |
| RStream           | 5     | 352.96   | 35.47   | 637.77  | 3.88     | 108.78   | 868.88  | 4.74    | 0.90     | 833 70   | 903.90  | 35.47   | 5.34     |
| Kaleido           | 4     | 3 -      | -       | 22.57   | 72.18    | 343.55   | 208.63  | 26.41   | 0.40     | 1086.88  | 7.83    | 3.08    | 7.32     |
| Pangolin          | 5     | 3 -      | -       | 22.57   | 72.18    | 343.55   | 208.63  | 26.41   | 0.40     | 1086.88  | 7.83    | 3.08    | 7.32     |

Table 4: Execution time (sec) of Pangolin (PA) and hand-optimized mining applications (sup: minimum support).

| Applications | Mico | Paten | Youtube | PDB |
|--------------|------|-------|---------|-----|
| TC           | -    | 139.3 | 659.3   | 4-CF |
| 3-CF         | 580  | 152.28| 0.93    | 0.40 |
| 3-MC         | 4-CF | 152.28| 0.93    | 0.40 |
| 5-CF         | 4-CF | 152.28| 0.93    | 0.40 |
| 3-FSM-500    | 4-CF | 152.28| 0.93    | 0.40 |
| 3-MC         | 4-CF | 152.28| 0.93    | 0.40 |
| 5-CF         | 4-CF | 152.28| 0.93    | 0.40 |

Table 5: Execution time (sec) of Pangolin applications on Intel Optane-PMM machine.

Figure 9: Strong scaling of Pangolin (using Yo).

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 PPM system and mine a very large real-world web crawl, Gsh. As shown in Table 5, TC and 3-CF only take 2 and 11 minutes, respectively. 4-CF is much more compute and memory intensive, so it takes ~6.5 hours. To the best of our knowledge, this is the largest graph dataset for which 4-CF has been mined.

Figure 9 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 due to GPU’s larger computation power and the lack of DAG construction in PGD. In the future, we are considering to add support for DFS exploration to further improve the performance of Pangolin.
single-thread execution. In contrast, FSM's scalability suffers due to the overheads of computing domain support.

Fig. 10 illustrates speedup of Pangolin applications on GPU over 28 threads 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 $6 \times$ and $15 \times$ respectively over the CPU execution. Specifically, we observe substantial speedup on CF and MC. For example, the V100 GPU achieves $50 \times$ speedup on 4-MC for Pa, demonstrating the suitability of GPUs for these compute intensive applications.

6.5 Memory Consumption

The peak memory consumption for Arabesque, RStream, and Pangolin is illustrated in Fig. 11. We observe that Arabesque always requires the most memory because it is implemented in Java using Giraph framework that 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. For 4-MC, RStream runs out of memory due to its edge-induced exploration (Arabesque and Pangolin are using vertex-induced exploration).

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. 12a 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. 12b demonstrates the advantage of using Galois scalable memory allocator instead of std allocators. This is particularly important for FSM as it requires intensive memory allocation for counting support. Fig. 12c illustrates that customized pattern classification used in MC and FSM yields huge performance gains by eliding expensive generic isomorphism tests. Fig. 12d 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. 13, 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 $2.1 \times$ to $4.7 \times$ faster. Another optimization is employing binary search for connectivity check. Fig. 13b demonstrates that binary search can achieve up to $6.6 \times$ speedup compared to linear search. Finally, Fig. 14 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...
Figure 13: $k$-MC speedup of (a) using embedding list (SoA+inspection-execution) over using embedding queue (AoS) and (b) binary search over linear search.

Figure 14: LLC miss counts in the vertex extension phase of $k$-CF using AoS and SoA for embeddings.

to SoA. This further confirms that SoA has better locality than AoS, due to the data reuse among embeddings.

7. RELATED WORK

Mining Applications: Hand-optimized graph mining applications target various platforms. For triangle counting, Shun et al. [66] present a parallel, cache-oblivious TC solver on multicore CPUs that achieves good cache performance without fine-tuning cache parameters. Load balancing is applied in distributed TC solvers [68, 30] to evenly distribute workloads. TriCore [37] is a multi-GPU TC solver that uses binary search to increase coalesced memory accesses, and it employs dynamic load balancing.

Chiba and Nishizeki (C&N) [19] proposed an efficient $k$-clique listing algorithm which computes the subgraph induced by neighbors of each vertex, and then recurses on the subgraph. Searching over a local space of neighborhoods is more efficient than searching over the global space of the entire graph. Danisch et al. [21] refine the C&N algorithm for parallelism and construct DAG using a core value based ordering to further reduce the search space. PGD [5] counts 3 & 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 [59] extends this approach to 5-vertex subgraphs and leverages DAG to reduce search space.

Frequent subgraph mining (FSM) [58] is one of the most important mining applications. gSpan [72] is an efficient sequential FSM solver which implements a depth-first search (DFS) based on a lexicographic order called minimum DFS Code. GraMi [26] proposes an approach that finds only the minimal set of instances to satisfy the support threshold and avoids enumerating all instances. This idea has been adopted by most other frameworks. DistGraph [69] parallelizes gSpan for both shared-memory and distributed CPUs. Each worker thread does the DFS walk concurrently.

To balance workload, 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.

Other important mining applications includes maximal cliques [17], maximum clique [52, 2], and subgraph listing [64, 12, 44, 41, 45, 53, 47]. They employ various optimizations to reduce computation and improve hardware efficiency. They inspired our work to design a flexible interface for user-defined optimizations. However, they achieve high performance at the cost of tremendous programming efforts, while Pangolin provides a unified model for ease of programming.

Mining Frameworks: For high productivity and high performance, graph mining systems such as Arabesque [71], RStream [75], and Kaleido [77] 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 [21] employs a DFS strategy to enumerate subgraphs, which reduces memory footprint. Pangolin uses the BFS approach that is inherently more parallel than the DFS approach. In the future, we plan to also support DFS exploration.

Approximate Mining: There are approximate solutions for triangle counting [73, 60, 72], clique finding [57, 40], motif counting [67, 14], and frequent subgraph mining [6]. ASAP [59] is an approximate graph mining framework that supports various mining applications. It extends graph approximation theory to general graph patterns and incurs less than 5% error. Since approximation can reduce computation, it is much faster than exact frameworks such as Arabesque, and can scale to large graphs. In contrast, our work focuses on exact mining and shows that Pangolin can also achieve high performance without sacrificing accuracy.

8. CONCLUSION

We present Pangolin, a high-performance, flexible graph mining system on shared-memory CPUs and GPUs. Pangolin provides a simple programming interface 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 state-of-the-art mining frameworks, Arabesque and RStream, by 49× and 88×, 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. To mine 4-cliques in a web-crawl (gsh) with 988 million vertices and 51 billion edges, Pangolin takes ~ 6.5 hours on an 48-core Intel Optane machine with 6 TB memory.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] B. Bhattacharj, 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

[22] R. Datathri, 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.

[23] 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.

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

[25] 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 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’15, pages 229–238, New York, NY, USA, 2015. ACM.

M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. Proc. VLDB Endow., 7(7):517–528, Mar. 2014.

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.

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

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.

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.

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.

Apache Giraph. http://giraph.apache.org/, 2013.

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.

O. Green, P. Yalamanchili, and L.-M. Munguía. Fast triangle counting on the gpus. In Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms, IAA ’14, pages 1–8, Piscataway, NJ, USA, 2014. IEEE Press.

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.

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.

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.

J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In Third IEEE International Conference on Data Mining, pages 549–552, Nov 2003.

A. P. Iyer, Z. Liu, X. Jin, S. Venkataaraman, 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.

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.

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.

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

R. Kessel, 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.

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.

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.

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

L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. Proc. VLDB Endow., 8(10):974–985, June 2015.

J. Leskovec. Snap: Stanford network analysis platform, 2013.

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

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.

J. 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.

[52] C. Lu, J. X. Yu, H. Wei, and Y. Zhang. Finding the maximum clique in massive graphs. Proc. VLDB Endow., 10(11):1538–1549, Aug. 2017.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doilouion: 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.

[74] C. Voelgele, 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.