AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining

Daniel Mawhirter
Colorado School of Mines
dmawhirt@mymail.mines.edu

Bo Wu
Colorado School of Mines
bwu@mines.edu

Abstract

Graph mining algorithms that aim at identifying structural patterns of graphs are typically more complex than graph computation algorithms such as breadth first search. Researchers have implemented several systems with high-level and flexible interfaces customized for tackling graph mining problems. However, we find that for triangle counting, one of the simplest graph mining problems, such systems can be several times slower than a single-threaded implementation of a straightforward algorithm.

In this paper, we reveal the root causes of the severe inefficiencies of state-of-the-art graph mining systems and the challenges to address the performance problems. We build AutoMine, a single-machine system to provide both high-level interfaces and high performance for large-scale graph mining applications. The novelty of AutoMine comes from 1) a new representation of subgraph patterns and 2) compilation techniques that automatically generate efficient mining code with minimized memory consumption from a high-level abstraction. We have extensively evaluated AutoMine against 3 graph mining systems on 8 real-world graphs of different scales. Our experimental results show that AutoMine often produces several orders of magnitude better performance and can process very large graphs existing systems cannot handle.

CCS Concepts • **Computing methodologies → Shared memory algorithms**; • **Software and its engineering → Compilers**.

Keywords Graph mining, graph pattern matching, compiler

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359633

ACM Reference Format:
Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3341301.3359633

1 Introduction

Graph data, thanks to the flexibility of the structure, is ubiquitous in various domains, ranging from bioinformatics to social networks to web analytics. Efficiently processing large-scale graphs has attracted great attention leading to a number of highly optimized systems [25, 28, 29, 34, 40, 42, 64, 65]. Most of these systems provide a "think like a vertex" (TLV) or "think like an edge" (TLE) programming paradigm to implement *graph computation* algorithms. Example applications are breadth-first search (BFS) and PageRank, which can be modeled through iterative sparse matrix vector multiplication [24, 46]. In each iteration, the system traverses all active vertices or edges, but the processing of each vertex or edge only involves lightweight computation and generates limited intermediate data. For instance, in BFS an active vertex sends its label to all its unexplored neighbors. As such, the optimization efforts of *graph computation* problems mainly focus on communication reduction [18, 19], locality improvement [25, 65], and load balancing [11, 34].

Graph mining problems, however, are fundamentally differ from *graph computation* problems, because they involve much more complex algorithms and generate huge amounts of intermediate data. For example, the state-of-the-art algorithm to mine the frequency of size-4 cliques of a graph is $O(|E|\Delta T_{max})$, where E, Δ, and T_{max} respectively represent the edge set, the maximum degree, and the maximum number of triangles incident to an edge. The algorithm needs to enumerate all the triangles of the input graph, which are subgraphs of the size-4 cliques. The size of generated intermediate data can reach several TB for graphs with multiple million edges. The TLV or TLE based systems only maintain states on vertices or edges, which do not consider the subgraph pattern in *graph mining* problems. Therefore, neither do such systems provide a friendly interface to write graph mining algorithms, nor are they optimized to handle the large amount of intermediate data.

To address the mismatch, researchers have recently designed multiple systems that explicitly maintain states for subgraph patterns [10, 46, 51, 55]. Arabesque [46] is the first distributed system that proposes the "think like an embedding" paradigm, where an embedding is an instance of a subgraph pattern. By incrementally appending edges to embeddings, Arabesque can enumerate all the embeddings of any desired subgraph pattern, which are processed by user-defined *filter* and *process* functions. But distributed mining systems incur high overhead for small graphs and require enterprise clusters for large graphs. Wang et al. address this problem by proposing RStream [51], a single-machine graph mining system. RStream combines edge streaming for out-of-core processing and relational algebra operators for users to compose graph mining applications. Despite using less resource, it outperforms several state-of-the-art distributed mining systems on a variety of graph mining workloads.

Unfortunately, although Arabesque and RStream are specialized systems for graph mining problems, their performance is far from ideal. To perform triangle counting on a medium-sized graph (i.e., MiCo [14]) with 1.1 million edges, Arabesque needs 43 seconds on a 10-node cluster [51]. Our experiments on a 20-core machine show that RStream takes 2.5 seconds to process the same graph, but a single-threaded program based on a simple triangle counting algorithm finishes the execution in 0.97 seconds. To provide the high-level abstraction, both Arabesque and RStream implement generic yet low-efficiency graph mining algorithms that demand tremendous memory consumption. When facing two conflicting goals of providing a high-level abstraction and high performance, a classical problem in system design, they choose the former over the later.

In this paper, we present AutoMine, the first large-scale graph mining system to harmonize high-level abstraction and high performance on a single machine. A naive approach to provide the best of both worlds is to manually implement various graph mining algorithms and present easy-to-use interfaces to the user. However, this approach faces extreme difficulties because the topology of the subgraph pattern can take numerous forms and the user may be interested in mining different combinations of subgraph patterns. On the contrary, AutoMine does not explicitly implement any graph mining algorithm. It takes a high-level graph mining program as the input and automatically compiles it into efficient C++ code with low algorithm complexity and minimized memory consumption.

We face two challenges to implement AutoMine. The first challenge roots from the many possible algorithms to solve the same graph mining problem. AutoMine should automatically explore the algorithm space and properly rank the algorithms to select an optimized one for code generation. We point out that in the data mining community, researchers focus on one subgraph pattern (e.g., triangle counting) at a time and manually design algorithms [4, 22]. To the best of our knowledge, there exists no prior work on automatically generating efficient graph mining algorithms.

The second challenge is how to minimize memory consumption. Existing systems generate large amounts of intermediate data, because graph mining problems have nested dependencies: A subgraph pattern is built upon its own subgraphs. Those systems take an easy approach to meet the dependencies, which enumerates and stores *all* the embeddings of a simple sub-pattern before moving on to generate embeddings of a more complex one. AutoMine should also respect the dependencies but still find room to dramatically reduce memory consumption in the generated algorithm and code at compile time.

AutoMine addresses the challenges with three novel ideas. First, it represents an embedding by a vertex composition set (i.e., a set of sets of vertices), which 1) saves space compared to table or graph based representations and 2) provides the foundation for automatic algorithm and code generation. Second, AutoMine's schedule generator models a subgraph pattern mining problem as a graph tournament problem and generates algorithms to produce the composition set as well as encode its meaning. Third, when the user program is interested in multiple subgraph patterns, AutoMine's code generator automatically merges the generated algorithms for these patterns to minimize redundant work and maximize data sharing.

AutoMine is a flexible system that supports the sophisticated functionality of existing systems. AutoMine can process labeled graphs with a *support* parameter to filter out subgraph pattern whose frequency does not meet the threshold. Moreover, by leveraging memory mapped I/O, AutoMine can process out-of-core graphs that do not fit in the memory by taking advantage of the locality of the generated mining algorithms.

The proposed techniques allow AutoMine to be significantly faster than existing systems while still providing high-level interfaces. Our experimental results show that AutoMine often outperforms RStream and Arabesque by several orders of magnitude for 4 graph mining applications running on real-world graphs of different scales. Though AutoMine generates exact graph mining programs, it even outperforms ASAP [23], a state-of-the-art approximate graph mining system, by up to 68.8X for size-3 motif counting. We find that RStream's out-of-core processing cannot support triangle counting on a graph of 783 million edges given 2TB SSD space, while AutoMine successfully finishes triangle counting and size-4 clique counting on a much larger graph with 25.7 billion edges.

This paper makes the following contributions: 1) We present AutoMine, the first single-machine graph mining system to provide both high-level abstraction and high performance for graph mining applications. 2) We propose a space-efficient representation of embeddings that lays the foundation for automatic mining algorithm generation. 3) We propose a

set of modeling and optimization techniques to generate efficient graph mining programs in C++ with low complexity and minimized memory consumption. 4) We evaluate AutoMine by comparing it against 3 state-of-the-art graph mining systems on 8 real-world graphs. The results show that AutoMine substantially outperforms all these systems with minimal programming effort from the user.

2 Motivation

2.1 Single-threaded Triangle Counting vs. State-of-the-art Graph Mining Systems

Existing graph mining systems provide a high-level abstraction for users to easily write applications. To understand the performance of such systems, we follow the methodology used by McSherry et al. [32] and compare RStream, which was the fastest among 4 state-of-the-art mining systems (including Arabesque) [51], with a single-threaded program for triangle counting. The program implements a simple triangle counting algorithm used in many prior studies [4, 43, 47, 48] as shown in Algorithm 1. Although the performance of triangle counting can be dramatically improved by locality optimization (e.g., tiling-based data reorganization [62]), we stick with the unoptimized implementation for a fair comparison, because RStream may not apply similar optimizations.

Figure 1 shows the running times with 6 real-world graphs on a 20-core machine (details in Section 8). Observe that even though the single-threaded program only uses 1 core, it always outperforms RStream using 20 cores and produces up to 5.7X speedup. We point out that McSherry et al. [32] showed that their single-threaded benchmark outperforms the fastest *graph computation* systems by only up to 1.7X. Our results suggest that the high-level abstraction of the graph mining systems eats up even more performance.

Algorithm 1: Triangle counting.

```
   input  : G : the Graph.
   output : n : the number of triangles in G.
1  begin
2      n ← 0;
3      for v0 in V do
           // N(v) returns a set that contains
              all v's neighbors
4          for v1 in N(v0) do
5              s ← N(v0) ∩ N(v1);
6              n ← n + |s|;
7      n ← n/6;
```

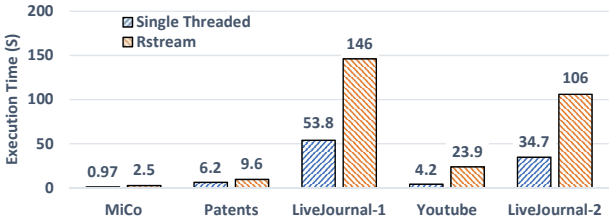

Figure 1. Performance comparisons on triangle counting.

2.2 Root causes and challenges

Reason 1: Existing graph mining systems implement generic but low-efficiency mining algorithms. The state-of-the-art systems provide a high-level abstraction to implement graph mining applications. They typically implement the bulk synchronous parallel (BSP) model and maintain a list (or lists) of embeddings. In each iteration, they try to append one more edge to each of the current embeddings to generate more complicated ones. The process continues until all the embeddings of the considered subgraph patterns have been enumerated. Figure 2 shows an example of the iterative process to perform triangle counting in RStream. The initial embeddings are a list of all the edges. To append new edges to generate wedge embeddings, RStream executes a join operation on the edge list. The complexity of the join operation is $O(|E|)$ if the edge list is sorted. In the next iteration, RStream joins the edge list with the list of wedge embeddings, whose worst case size is $|V|\Delta^2$. The simple triangle counting algorithm has complexity $O(|E|\Delta)$. Since $|V|\Delta$ is typically much larger than $|E|$, the triangle counting algorithm is much more efficient in practice, especially for power-law graphs. So the state-of-the-art systems have a serious shortcoming compared to specially designed graph mining algorithms.

Challenges for the remedy. A strawman approach is to implement the state-of-the-art graph mining algorithms as a library and provide high-level interfaces. Unfortunately, graph mining has a well-known combinatorial explosion problem as we increase the subgraph pattern size. For example, there exist only 6 different size-4 connected subgraph patterns but 21 size-5 connected patterns. Designing and implementing specialized algorithms for even the small subgraph patterns (e.g., size less than 7) is labor-intensive. Ideally, we should automatically generate the efficient mining algorithms, but we face an enormous challenge because they differ dramatically for different subgraph patterns. For instance, Ahmed et al. [4] propose for all the size-4 subgraph patterns 6 distinct algorithms with varying structures and complexities.

Reason 2: Existing graph mining systems have high memory consumption. To process the Patents graph with 16.5 million edges, RStream consumes more than 22GB memory, while the single-threaded program only needs 158MB

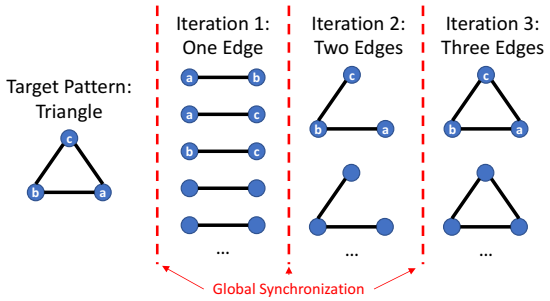

Figure 2. Embedding enumeration for triangle counting in RStream.

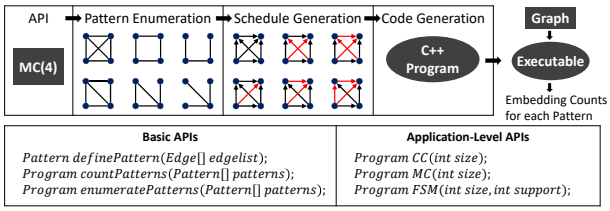

Figure 3. AutoMine architecture.

memory. RStream has such high memory consumption due to its conservative approach to deal with dependencies. As Figure 2 shows, RStream needs to generate all the wedge embeddings (indicated by the global synchronization) before moving to the next iteration to generate all the triangle embeddings. It hence has to allocate enormous memory space to store the generated intermediate data. If the required space does not fit in the main memory of one machine, RStream flushes the data into disk.

Challenges for the remedy. Figure 2 illustrates a simple idea to minimize memory consumption. To enumerate the triangle embedding (a, b, c), we only need to first generate the wedge embedding (a, b, c) it depends on, instead of all the wedge embeddings. However, it is difficult to generalize the idea to more complex subgraph patterns especially for automatic algorithm generation. Moreover, multiple subgraph patterns may share the same sub-pattern. For example, both the size-4 clique pattern and the chordal cycle pattern (i.e., clique minus a diagonal) have triangles in them. Thus, if the system does not store all the embeddings of the sub-pattern, it may need to re-generate or maintain duplicate embeddings, leading to extra space overhead.

3 Overview of AutoMine

In this work, we design the AutoMine system to bridge the gap between high-level abstraction and high performance for graph mining applications. AutoMine does not require the user to understand the mining algorithms or system optimization details, but presents a set of high-level APIs. AutoMine automatically generates highly efficient mining programs with low algorithm complexity and minimized memory consumption. We first present the overall architecture of the system with an end-user example and then the APIs.

The workflow of the AutoMine system has a compilation phase and an execution phase as shown in Figure 3. The compilation phase takes a high-level API (*MC(4)* in the illustrated example for size-4 motif counting) and generates an optimized graph mining program by invoking three components. The first component is the pattern enumerator. It understands the semantics of the high-level API and enumerates all the non-isomorphic subgraph patterns (6 in total for size-4 motif counting) that are involved in the mining task. The second component, the schedule generator, generates an optimized schedule (i.e., algorithm) to identify each of the subgraph patterns. Each schedule is represented by a colored graph with directions assigned to the edges. The last component is the code generator, which considers data reuse in the generated schedules and produces the final mining program in C++. In the execution phase, the mining program processes input graphs and returns the final results.

APIs. Figure 3 shows the major APIs to use AutoMine. The *definePattern* function defines a pattern with a list of 2-tuples, each representing an undirected edge. For example, to define a triangle pattern, user invokes the function as *Pattern p = definePattern([(a,b), (b, c), (c,a)])*. Since AutoMine only supports connected patterns, it warns the user if the provided list cannot form one. AutoMine supports two elementary APIs, *countPatterns* and *enumeratePatterns* to generate programs to respectively count and enumerate the embeddings of the given list of subgraph patterns. To make AutoMine easy to use, it implements APIs to support 3 popular graph mining applications: Clique Counting (*CC*), Motif Counting (*MC*), and Frequent Subgraph Mining (*FSM*) (details in Section 8). Each of these APIs invokes the pattern enumerator to generate the list of subgraph patterns, which is passed to *enumeratePatterns* or *countPatterns* to produce the final mining program.

We next describe the key techniques in the schedule generator and the code generator. Due to space limit, we omit the detailed description of the pattern enumerator, which is a necessary component but only involves engineering work.

4 Set Based Representation

Section 2 shows that the simple triangle counting algorithm is much more efficient than the generic algorithm implemented in the state-of-the-art graph mining systems. We make two observations about the algorithm. First, it exploits the local structure of the input graph. In the innermost loop, the algorithm discovers a set of vertices, each forming a triangle with the edge embedding (v_0, v_1). The advantage is that the algorithm can safely discard the edge embedding immediately since all the more complex embeddings (i.e. the

triangles) built on it are discovered in the same loop iteration. Second, each vertex of the discovered set corresponds to a distinct triangle incident on (v_0, v_1). The intersection operation performed on the neighbor sets of v_0 and v_1 generates the structure.

Inspired by this algorithm, we ask three questions: 1) Can we **generalize** the set based representation for any arbitrary pattern? 2) What **operations** should we use to compute a set? 3) How can we **compose** these set operations to discover the set? In this section, we explore the first two questions and consider the last question in the next section.

Consider a connected pattern $\mathcal{P}_k$ on k ($k > 2$) vertices, and a sub-pattern $\mathcal{P}_{k-1}$. An instance of $\mathcal{P}_k$ is an embedded subgraph denoted as $E_{\mathcal{P}_k}$ and composed of vertices $(v_0, ..., v_{k-1})$. We introduce a function $\mathcal{F}^k(E_{\mathcal{P}_{k-1}})$ which needs to meet two requirements. First, it should return a set $\mathbf{V}_k$ of all the vertices v_k that extend an embedding $E_{\mathcal{P}_{k-1}}$ into an $E_{\mathcal{P}_k}$. Second, it must only apply set operations on the neighbor sets of the $E_{\mathcal{P}_{k-1}}$'s vertices.

Intuitively, $\mathcal{F}^k$ exists because a graph is essentially a set of neighbor sets. The neighbor sets of $v_0, ..., v_{k-1}$ should have sufficient information for us to discover $\mathbf{V}_k$ precisely because $\mathcal{P}_{k-1}$ is connected. However, we only have four basic set operations with which to implement $\mathcal{F}^k$: union, complement, intersection, and subtraction. Only intersection and subtraction are anti-monotonic, meaning their output is no larger than the size of their largest input. So they are the preferred operations to use. Fortunately, the following lemma shows that these two are always sufficient to implement $\mathcal{F}^k$.

Lemma 1. *$\mathcal{F}^k$ can use only set intersection and subtraction to discover $\mathbf{V}_k$.*

Proof. In order to construct the function $\mathcal{F}^k$, suppose a vertex v_k which can form an embedding $E_{\mathcal{P}_k}$ with the vertices from $E_{\mathcal{P}_{k-1}}$. We partition $v_0, ..., v_{k-1}$ into two sets $\mathbf{V}_T$ and $\mathbf{V}_F$. $\mathbf{V}_T$ contains all the vertices that are neighbors of v_k in $\mathcal{P}_k$ and $\mathbf{V}_F$ includes the remaining vertices. Any vertex v_k in $\mathbf{V}_k$ must obey the following properties: $v_k \in \mathcal{N}(v)$ for each vertex v in $\mathbf{V}_T$, and $v_k \notin \mathcal{N}(v)$ for each vertex v in $\mathbf{V}_F$. We therefore construct $\mathcal{F}^k$ as follows:

$$\mathbf{V}_k = \mathcal{F}^k(E_{\mathcal{P}_{k-1}}) = \bigcap_{v \in \mathbf{V}_T} \mathcal{N}(v) - \bigcup_{v \in \mathbf{V}_F} \mathcal{N}(v)$$

Since $\mathcal{P}_k$ is connected, V_T is not empty. $\mathcal{F}^k$ first performs a reduction on V_T with intersection and then subtracts the neighbor sets of the vertices in V_F one by one from the result. $\mathbf{V}_k$ hence includes the vertices that neighbor all $v \in \mathbf{V}_T$ and none of $v \in \mathbf{V}_F$, completing $\mathcal{F}^k$ using only intersection and subtraction. □

The proof introduces an algorithm to discover and represent embeddings of the more complex pattern $\mathcal{P}_k$ based on any embedding of $\mathcal{P}_{k-1}$. The base pattern $\mathcal{P}_1$ is a vertex, with $\mathcal{P}_2$ being an edge. Hence, the vertex set $\mathbf{V}_k$ represents a set of embeddings of the non-trivial pattern $\mathcal{P}_k$ encoded by the sequence $\{\mathcal{F}^1, ..., \mathcal{F}^k\}$, where $\mathcal{F}^1$ returns the vertex set and $\mathcal{F}^2$ returns the neighbor set of a vertex. We name this sequence a **schedule** of set operations. In other words, once we have a schedule, we can iteratively apply it to all the edge embeddings to discover a set of sets with all the embeddings for an arbitrary pattern.

5 Schedule Generation

In the previous section, we show that once we have the schedule for a particular pattern, we have an algorithm to discover all its embeddings. This section presents the techniques to automatically generate an optimized schedule for any given pattern.

5.1 Modeling

The series of functions defined in Section 4 encodes the relationships among the vertices of a pattern. The functions $\{\mathcal{F}^1, ..., \mathcal{F}^k\}$ must be applied in order when computing patterns to respect their dependencies. As explored previously, this also implies an order in the discovery of the vertices $v_0, ..., v_{k-1}$. While there is a one-to-one mapping between a series of $\mathcal{F}^k$ and a vertex order, there can be many possible series' of functions for the same pattern. We next determine how to explore the space of possible schedules.

Given a pattern, we build a colored complete graph to encode all the neighborhood relationships of the vertices. Specifically, we color all of its present edges black and add red edges for the absent ones. Figure 4 shows an example of a colored complete graph for the chordal cycle pattern. With this complete graph, we need to make two decisions. First, we should assign an order to add vertices while discovering progressively more complex patterns. Second, we should assign directions to the edges. Direction encodes a critical property in this construction, denoting which vertex we should search for in the neighbor set of the other. A symmetric graph has the following important property:

$$v_a \in \mathcal{N}(v_b) \iff v_b \in \mathcal{N}(v_a)$$

Any pair of vertices which share an edge can therefore be discovered in any order. And the diversity of the space of possible orders gives rise to the diverse schedules for a given pattern.

The directional edges form a tournament of the complete graph, and each unique tournament identifies a distinct schedule. The tournament's edges define relationships encoded in the series of functions $\mathcal{F}^k$. Vertex v_k has incoming black edges from the vertices in $\mathbf{V}_T$ and incoming red edges from the vertices in $\mathbf{V}_F$, thus defining the schedule. Chordal cycle, as shown in Figure 4, has 5 unique acyclic tournaments, two of which are shown. The reason for the choice of acyclic tournaments is described in the following lemma:

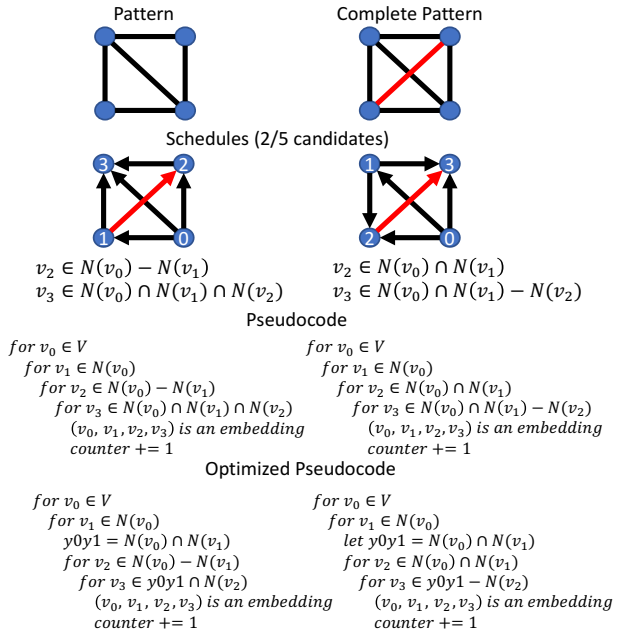

Figure 4. Scheduling for Chordal Cycle

Lemma 2. *A tournament must be acyclic for its corresponding schedule to exist.*

Proof. Suppose a vertex v is part of a cycle $[v_0, ..., v_{n-1}]$. Assume a valid series of functions $\mathcal{F}^k$ can be constructed according to this cyclic tournament, and recall that $\mathcal{F}^k$ is permitted to operate only on the vertices $[v_0, ..., v_{k-1}]$. Suppose v has an incoming edge from vertex v_{in} and an outgoing edge to v_{out}, both from its cycle, each of which has a defined value of k for its corresponding $\mathcal{F}^k$. The edges incident on v demand that $k_{v_{in}} < k_v < k_{v_{out}}$. But the cycle implies that there exists a path from v_{out} to v_{in} through the vertices $[v_0, ..., v_{n-1}]$, demanding that $k_{v_{out}} < k_{v_{in}}$. This forms a contradiction, and proves that a cyclic tournament cannot have a valid schedule. □

Corollary 3. *The proof of Lemma 2 demonstrates that an acyclic tournament gives the vertices a total order, which is the necessary condition for a schedule to exist.*

According to Lemma 2 and Corollary 3, acyclic tournaments and valid schedules have a one-to-one correspondence. We therefore choose to iterate over the possible tournaments in order to search the schedule space. In a k-vertex complete graph, there are $k!$ unique orderings of the vertices, and therefore $k!$ possible acyclic tournaments. The colored edges distinguish between some of these orders, making them non-isomorphic, and worth exploring. Note that cliques are a special case, in which all $k!$ permutations are in fact isomorphic, as there are no red edges to distinguish between them. Since Lemma 1 demonstrates the construction of a schedule from a vertex ordering, the space of all non-isomorphic colored acyclic tournaments defines the scheduling space for a particular pattern.

5.2 Multiplicity

We notice from the triangle counting algorithm that the corresponding schedule has a multiplicity problem. Given a triangle embedding, the same schedule (i.e., $\mathcal{N}(v_0) \cap \mathcal{N}(v_1)$) can actually observe it from any of its edges. Each undirected edge is represented symmetrically by two directed edges, so we over-count by a factor of 6. We need to automatically determine the multiplicity for a given schedule.

Symmetry in a pattern introduces this over-counting in all possible schedules for a pattern. The key point is that the order of vertex discovery for a schedule has a number of possibilities equal to the pattern's multiplicity. Consider the tailed triangle (i.e., a triangle with a dangling edge) with 4 vertices (a, b, c, and d). The three vertices, a, b, and c, form a triangle and d is a neighbor of c only. Now consider two schedules to observe the pattern. The first schedule is:

$$\mathcal{F}^2 = \mathcal{N}(v_0) \cap \mathcal{N}(v_1)$$
$$\mathcal{F}^3 = \mathcal{N}(v_2) - \mathcal{N}(v_0) - \mathcal{N}(v_1)$$

It first discovers a triangle and then the dangling edge. The schedule can observe the pattern in two orders: (a, b, c, d) and (b, a, c, d) because this pattern has a multiplicity of 2.

$$\mathcal{F}^2 = \mathcal{N}(v_0) - \mathcal{N}(v_1)$$
$$\mathcal{F}^3 = \mathcal{N}(v_0) \cap \mathcal{N}(v_2) - \mathcal{N}(v_1)$$

This schedule starts from the dangling edge to discover a wedge and then a triangle incident on the second edge of the wedge. The schedule can observe the pattern in two orders again: (c, d, a, b) and (c, d, b, a). Algorithm 2 generalizes this strategy to determine the multiplicity of any pattern by counting the automorphic vertex permutations. A schedule's result count divided by multiplicity yields the number of unique pattern instances in a graph.

Algorithm 2: Computing Multiplicty for a Pattern

```
input  : P_n : the Pattern.
output : M : the multiplicity of S counting P_n.
1 begin
2     M ← 0;
3     base_order ← range [0..n);
4     for order in permutations(base_order) do
5         Pattern P'_n ← empty;
6         for Edge (v_a, v_b) in P_n do
7             P'_n.add_edge(order[v_a], order[v_b]);
8             if equal(P_n, P'_n) then
9                 M ← M + 1;
```

5.3 Root Symmetry

Multiplicity introduces a computation redundancy problem, because a schedule may observe the same pattern several times. Root symmetry is a special case of multiplicity that only considers the first edge in the discovery order for a schedule, which we refer to as the root edge. If this edge is root symmetric, then we can halve the multiplicity by considering root edges in only one direction while processing the graph. The method for determining if a schedule is root symmetric is simple given the algorithm for multiplicity. If the two vertices incident on the root edge are interchangeable according to the isomorphism test in its inner loop, then the schedule is root symmetric.

One perspective to understand the efficiency of the root-symmetry property is that it prunes half of the directed edges from consideration as root edges in $\mathcal{F}^2$. Since the pattern is nested, applying the idea to each sub-pattern can further prune edges. We leave the generalization of this idea for arbitrary patterns to future work, but show that it can greatly improve the performance for clique patterns. Cliques have a special case of multiplicity and root-symmetry, in that every edge and vertex is indistinguishable, leading to a very high multiplicity of $k!$ for a k-clique. Even if we apply the basic root-symmetry optimization, the multiplicity is still $\frac{k!}{2}$. Observe that after applying the root symmetry optimization, the root edge becomes directional (i.e., $v_0 \rightarrow v_1$). If we remove v_0 and its edges from the pattern, the remaining sub-pattern is still a clique which is amenable to a second round of the application of the same technique. So a deep application of the root-symmetry idea at every level of $\mathcal{F}^{2..k}$ eliminates the multiplicity entirely for cliques.

6 Code Generation

This section crystallizes the scheduling idea into a useful system. We first describe how to generate code for a single pattern with data reuse optimizations, followed by the techniques to merge multiple schedules when having different patterns. We then present the infrastructure to support the generated code to process graphs.

6.1 Generating Code for a Single Pattern

Recall from previous section that given a pattern of size n, its schedule is represented by a series of functions $\mathcal{F}^k (0 \leq k \leq n)$, each depending on the vertices $[v_0, ..., v_{k-1}]$. Such a pattern naturally lends itself to a nested loop structure. At each loop level k, the loop body traverses the vertex set $\mathbf{V}_{k-1}$ and apply $\mathcal{F}^k$ to $[v_0, ..., v_{k-1}]$ to create a vertex set $\mathbf{V}_k$ for the next loop. When the execution reaches the innermost loop, it observes $[v_0, ..., v_n]$ as an embedding of the pattern. Figure 4 shows two schedules for the chordal cycle pattern as well as their corresponding loops to count the embeddings. The generated loop structure, despite its simplicity, is highly memory efficient. The only mandatory intermediate data for a pattern of size k is the series of vertices $[v_0, ..., v_{k-1}]$ and some indices to track positions in their containing sets. Once the corresponding loop is ready to move to another iteration, it is safe to discard all the vertex sets that store $[v_k, ..., v_{n-1}]$. Existing systems do not have this property, because their generic algorithms cannot keep track of the dependencies between embeddings.

While such an approach minimizes the memory footprint, it incurs redundant computation and data accesses. Because each function $\mathcal{F}^k$ depends on $k-1$ neighbor sets, it must access all of those sets each time it is computed. In the example shown in Figure 4, the generated code for both schedules accesses $\mathcal{N}(v_0)$, $\mathcal{N}(v_1)$, $\mathcal{N}(v_2)$ in the innermost loop. Observe that $\mathcal{N}(v_0) \cap \mathcal{N}(v_1)$ is loop-invariant, meaning that its result remains the same across iterations of the innermost loop. Ideally, we should store the result ahead of time, paying an up-front computation and data access cost to avoid the redundancy in the future. In the optimized code, we move the operation to the second loop and store the result in a vertex set $y0y1$. We next describe how to generalize this idea for arbitrary patterns.

We define a **prefix** of $\mathcal{F}^k$ as another function $\mathcal{F}_p^k$ where $2 < p < k$, which contain all $\mathcal{F}^k$'s operations on only vertices $[v_0, ..., v_{p-1}]$. If the prefix is pre-computed and its results stored, we only have to access the neighbor sets of $[v_p, ..., v_{n-1}]$ to complete the computation of $\mathcal{F}^k$. For the two schedules of the chordal cycle pattern shown in Figure 4, $\mathcal{N}(v_0) \cap \mathcal{N}(v_1)$ is a prefix of of the last schedule function computed in the innermost loop. If we precompute $\mathcal{N}(v_0) \cap \mathcal{N}(v_1)$, the only neighbor set accessed in the innermost loop is $\mathcal{N}(v_2)$.

During code generation for the loop at level k, we traverse each of $[\mathcal{F}^k, ..., \mathcal{F}^n]$ and try to generate code to compute and store the prefix that depends on $[v_0, ..., v_{k-1}]$. The code generation always succeeds as long as the corresponding $\mathbf{V}_T$ set (from Section 5) is non-empty, the same requirement for the use of intersection and subtraction. We aggressively apply this optimization because of two reasons. First, it reduces the computation redundancy as we previously discussed. Second, due to the anti-monotonic property of intersection and subtraction discussed in Section 4, the size of the resultant set of a prefix is typically much smaller than the size of its largest input set, which means the inner loops would access much less data. Parallelism is easy to apply within this model using OpenMP on the outermost loop.

6.2 Estimating Optimality

From the space of all possible schedules for a pattern, we will need to select one to use in practice. To achieve this goal, there must be a way to estimate the relative performance of each schedule. It is challenging because of the embedded structure and the complex set compositions used in the schedules. Moreover, the relative cost may even depend on

the topology of the input graph. We simplify the problem by leveraging a random graph of n vertices, in which any pair of vertices are neighbors with probability p. Hence, the expected size of a neighbor set is $n \times p$. The expected size of $\mathcal{N}(v_i) \cap \mathcal{N}(v_j)$ and $\mathcal{N}(v_i) - \mathcal{N}(v_j)$ is hence $n \times p^2$ and $n \times p \times (1-p)$, respectively, where v_i and v_j are two different vertices. With the estimate for the two basic operations, we can further estimate the size of the resultant set of any $\mathcal{F}^k$. The estimation works even if the prefix pre-computation optimization is applied, because a prefix also uses only intersection and subtraction operations. Given the estimate of the size of all the sets, we can derive the number of iterations of each loop and thus the number of neighbor set accesses in each loop level. By accumulating these estimates over the nested loop structure, we obtain the complexity for the schedule in n and p. When we compare the complexity of different schedules, n is always canceled out, so we only need to define p to properly rank all the schedules. We empirically choose 10^{-5} for p in our system to approximate the density of our chosen datasets.

6.3 Multi-Pattern Scheduling

When preparing a combined schedule for multiple patterns, we take the one for each pattern with the lowest data access complexity according to the prior analysis and combine them to form the merged schedule. Schedules for every pattern start with the same $\mathcal{F}^1$ and $\mathcal{F}^2$, and may remain the same for levels beyond that. Overlap of prefixes can also contribute to data reuse, so running schedules for multiple patterns at the same time is clearly desirable. Schedules always begin converged, and then diverge at some level k, as soon as $\mathcal{F}^k$ for the schedules differ. Note that once they diverge, they never re-converge. Even if later functions match again, they cannot be combined again, as the paths they took to get there are different. We refer to this as the *identity* problem, and it affects the way combined prefix storage is handled. Because divergent paths cannot share data, only the future function to be computed among paths that are still converged should be considered when selecting which prefixes to compute.

6.4 Supporting Infrastructure

Graph Data AutoMine stores graphs in the binary compressed sparse row format, in which the **vertex** array stores offsets into the **edge** array. A vertex v_i can find its sorted neighbor list at **edge**[**vertex**[i] : **vertex**[$i+1$]]. We use this format for both the in-memory and on-disk storage of graphs, making the graph data simple to handle, and enabling the option to process memory-resident or disk-resident graphs for out-core-processing.

Parallelization AutoMine uses OpenMP to parallelize mining tasks. Accesses to the graph are read-only, and inherently thread-safe. Accumulators are protected by OpenMP *reduce*(+) directives such that each thread accumulates results into thread-local memory until the parallel region ends. This makes the implementations easier to generate, as the parallelism is handled automatically.

Memory Management Two goals should be fulfilled by the memory management. First, graph data should not be copied, as it can be read directly. Second, the scratch space used for intermediate storage should be thread-local and reusable to avoid repeated allocation. We achieve these two goals using a **VertexSet** class which can either contain a read-only reference to graph data, or a writable reference to a scratch region. The scratch data is allocated at the beginning of execution according to the needs of the program. Since no composition of sets using intersection and subtraction can exceed the size of its largest operand, each region is allocated to hold *maximum degree* vertices. The regions are returned to the available memory pool when a memory-managed **VertexSet** goes out of scope.

Operators The **VertexSet** class also handles the intersection ($\cap$) and subtraction (-) operations as binary operators which, when called, return a memory-managed **VertexSet** containing the results. Note that the subtraction operation performs one check beyond its defined set operator scope. Since the edge pair $(v_0, v_1)(v_1, v_0)$ is not a wedge, but $v_0 \in N(v_1) - N(v_0)$, we must specifically exclude the vertex itself from subtractions when its neighbor list is a right-hand operand. The modification is trivial, but necessary for correctness.

7 Additional Features

7.1 Supporting out-of-core processing

The nested loop structure that AutoMine employs requires little memory on top of the graph representation, while previous work may produce multiple terabytes of intermediate data, stored either in distributed memory or on disk. In the case of the Motif-4 application on the MiCo graph with 1M edges, RStream generates 1.21TB of intermediate data, which it stores to disk. In our system, the graph representation consumes about 9MB of memory (755KB of vertex data, 8.3MB of edge data), and the intermediate data takes up an additional 1.7MB. For many graphs that trigger the out-of-core processing of existing systems, AutoMine can easily fit the entire workload into the main memory of a single machine.

For very large graphs, we may want to employ out-of-core processing to lighten the load of the graph data in memory, which dominates the memory requirement. In this case, AutoMine leverages memory-mapped files to support out-core-processing. The vertex data file and edge data file can be page faulted into physical memory as needed, and remain on disk when it is not. The key factor that makes this an efficient approach is that the total access costs to a given vertex are super-linear in the degree of the vertex. For triangle counting

the cost is quadratic, which continues to grow as the target pattern size increases. These super-linear costs produce large differences in access frequency between large-degree and small-degree vertices. The pages that contain the neighbor set of large-degree vertices become hot pages that occupy most of the available memory. We evaluate the efficiency of the out-of-core processing support in Section 8.

7.2 Supporting labeled graphs

Frequent Subgraph Mining (FSM) is unique from the other mining tasks that we consider in that it demands a labeled graph. Labeled patterns with the same topology, in this formulation, are different if their labels differ (i.e. the definition of isomorphism is expanded to include labels). Given a labeled pattern, AutoMine first generates a schedule of its unlabeled version and includes a lookup table to distinguish between instances of the labeled patterns. The FSM task also introduces a *support* parameter, which sets a threshold for the minimum number of embeddings of a labeled pattern to exist before it must be counted. This parameter has more selective power when considering a labeled graph, due to the lower average number of occurrences of each possible pattern.

In order to leverage this selectivity, however, the algorithm must proceed by growing patterns in the BSP style described in Section 2, which drives huge intermediate data requirements. But failing to maintain the intermediate data would make it impossible to determine if a computation could be avoided due to the support parameter. The implementation of labeled graph processing conceptually performs the nested loop schedule up to the next global synchronization point to generate and store the corresponding vertex sets. These sets are then pruned according to the support parameter, and execution resumes. This global synchronization must occur twice to process size-4 FSM.

8 Evaluation

In this section, we evaluate AutoMine's performance against three graph mining systems: Arabesque, RStream, and ASAP [23], specifically how well they scale to large graphs and patterns, as well as the optimization techniques proposed in AutoMine. The highlights of the results are as follows: 1) For 24 different mining workloads on real-world graphs, AutoMine is up to 4 orders of magnitude faster than Arabesque, running on 10 machines, and RStream. 2) ASAP uses approximation techniques to accelerate graph mining. Even when it uses 16 machines and 5% as the error target, ASAP takes on average 12.8X longer time to perform size-3 motif counting on 4 real-world graphs compared to AutoMine. 3) RStream runs out of disk space (2TB) for graphs with millions of edges. AutoMine, thanks to its efficient memory use and out-of-core processing capability, can successfully process a graph with more than 25 billion edges.

Graphs	#Vertices	#Edges	Description
CiteSeer [14]	3264	4536	Publication citation
MiCo [14]	96638	1080156	Co-authorship
Patents [26]	3.8M	16.5M	US Patents
LiveJournal-1 [7]	4.8M	42.9M	Social network
Orkut [2]	3.1M	117.2M	Social network
UK-2005 [9]	39.5M	783M	Web graph
Youtube [57]	1.1M	3M	Social network
LiveJournal-2 [57]	4M	34.7M	Social network
GSH-2015 [8]	988.5M	25.7B	Web graph

Table 1. Graph Datasets

8.1 Methodology

Graph mining applications AutoMine can generate programs to perform graph mining tasks for arbitrary patterns. We use its capability to provide high-level interfaces to run 4 popular graph mining applications on labeled or unlabeled graphs.

Triangle Counting (TC) is a simple mining task to count all the embeddings of the triangle pattern (i.e., size-3 clique) in an unlabeled graph. **Clique Counting (CC)** counts all the embeddings of the clique pattern given a specific size in an unlabeled graph. It only involves the 1-hop neighbors of each vertex but may incur heavy workload depending on the size. **Motif Counting (MC)** counts all the embeddings of each of the connected patterns of a particular size in an unlabeled graph. We consider 3-motifs (wedge and triangle) and 4-motifs (6 distinct patterns). **Frequent Subgraph Mining (FSM)** aims at discovering interesting patterns in a labeled graph. Given the *support* parameter and the pattern size, it counts embeddings of the patterns whose appearances exceed the threshold.

Datasets and settings Table 1 shows the 9 real-world graphs used in the experiments. Wang et al. [51] used the first 6 graphs to evaluate RStream to demonstrate that it outperforms multiple other mining systems, including Arabesque, by at least 1.7X. We hence also use these graphs to experiment with AutoMine and RStream. Since we do not have access to a private cluster, we use the performance numbers reported by Wang et al. for Arabesque, which was run on a 10-node cluster, each node equipped with a 8-core Intel E5-2640 v3 CPU and 32GB memory [51].

The ASAP system is not released, but the authors reported its performance on CiteSeer, MiCo, Youtube, and LiveJournal-2 [23]. They used a cluster of 16 Amazon EC2 r4.2xlarge instances, each having 8 virtual CPUs and 61GB memory. We also use the 4 graphs to compare AutoMine with ASAP.

We run experiments with AutoMine and RStream on a single machine with 2 10-core Intel Xeon E5-2630 (v4) CPUs (hyper-threading enabled), 64GB of memory, and 2TB of SSD. The machine runs on Ubuntu 16.04 with Linux kernel version 4.4.0-143. We use the GCC compiler with optimization

level O3 to compile RStream and the programs generated by AutoMine.

8.2 Comparisons with RStream and Arabesque

We run all 4 mining applications with AutoMine and RStream on CiteSeer, MiCo and Patents, because the RStream paper only shows the timing results on these graphs for Arabesque. Table 2 reports the running times of the three systems, which do not include the graph loading time.

For triangle counting, both RStream and Arabesque scale poorly to larger graphs. Patents is a medium-sized graph with less than 20 million vertices, but RStream takes 9.6 seconds to process it. Arabesque's performance is even worse. AutoMine's automatically generated triangle counting code produces 68.6X and 820.7X speedup over RStream and Arabesque, respectively. Observe that AutoMine also outperforms the single-threaded implementation by 24.3X on MiCo and by 44.3X on Patents, showing that its high-level abstraction does not sacrifice any performance for this application. Since CiteSeer is a tiny graph, AutoMine's parallelization adds non-trivial overhead and shows worse performance than the single-threaded implementation.

The same trend continues for motif counting and 5-clique counting. Motif counting is more compute-intensive than triangle counting, so all the systems take much longer time. Though MiCo and Patents can easily fit into the memory, RStream still heavily uses the disk and yields poor performance for 3-motif counting. When performing 4-motif counting, RStream's execution times out after 48 hours on MiCo and runs out of disk space on Patents. Arabesque can only process the smallest graph and runs out of memory with 10 machines for the other two. AutoMine only takes up to 22 seconds to run 4-motif counting on any of the graphs. AutoMine is particularly good at clique counting thanks to its aggressive application of the root-symmetry optimization. It only needs 0.17 seconds to run 5-CC Patents, leading to a speedup of 777X over RStream and 1011X over Arabesque.

FSM is a special application because of its *support* parameter. Since *support* is essentially a threshold to filter out infrequent patterns, the larger *support* is, the better the performance is for RStream and Arabesque. Because CiteSeer is a tiny graph, with $support \geq 300$ most patterns are filtered out, leading to trivial computation overhead. RStream is hence not much slower than AutoMine. For both MiCo and Patents, AutoMine is consistently faster than RStream and Arabesque. Notice that AutoMine does not benefit from the *support* parameter as much as RStream and Arabesque. A plausible reason is that the *support* parameter substantially reduces the memory consumption for RStream and Arabesque, but AutoMine has little memory space overhead even with *support* of 1. Filtering out the size-3 patterns does not affect the number of iterations of the two outermost loops AutoMine needs to execute.

App.	Sys.	CiteSeer	MiCo	Patents
TC	AM	0.01	0.04	0.14
	RS	0.01	2.5	9.6
	AR	38.1	43.1	114.9
	ST	0.003	0.97	6.2
3-MC	AM	0.016	0.12	0.5
	RS	0.13	1666.9	1149.1
	AR	40.6	51.7	116
4-MC	AM	0.024	22.0	20.0
	RS	2.2	T	F
	AR	F	F	F
5-CC	AM	0.024	11.4	0.17
	RS	0.075	F	134.1
	AR	42.8	132.0	174.5
3-FSM 300	AM	0.024	0.88	3.9
	RS	0.086	649.1	1453.2
	AR	F	F	F
3-FSM 500	AM	0.037	0.88	3.9
	RS	0.088	182.6	1002.8
	AR	F	F	F
3-FSM 1K	AM	0.033	0.87	4.1
	RS	0.09	2.5	81.5
	AR	35.6	5790.1	F
3-FSM 5K	AM	0.02	0.039	3.9
	RS	0.087	2.54	36.3
	AR	41.6	120.8	F

Table 2. Comparisons between AutoMine (AM), single-threaded triangle counting (ST), RStream (RS), and Arabesque (AR) on CiteSeer, MiCo, and Patents. 'T' indicates timeout after 48 hours of execution. 'F' indicates execution failure due to insufficient memory or disk space.

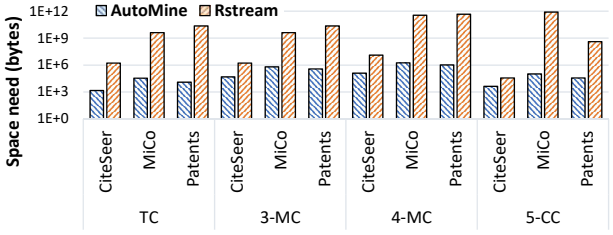

Figure 5. Needed space for RStream and AutoMine to fit all the data into memory.

Figure 5 shows the capacity needed by RStream and AutoMine to fit the entire workload (graph plus intermediates) into the main memory (note that the RStream numbers are a lower bound). When the space need exceeds the available memory, RStream stores the data into disk. Triangle counting, despite its simplicity, incurs on average 520MB space overhead for intermediates. AutoMine reduces the average space overhead to only 8.4KB. The results demonstrate the efficient memory use of the automatically generated schedules, which exploits the local graph structures and optimizes data reuse.

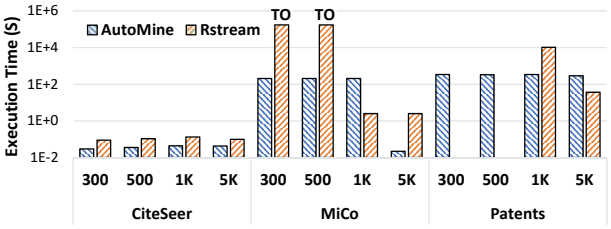

Figure 6. Size-4 FSM with different *support* parameters.

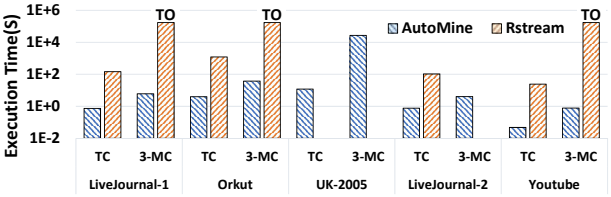

Figure 7. Results on larger graphs.

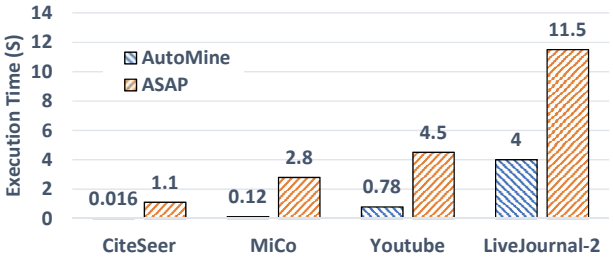

Figure 8. Results versus ASAP

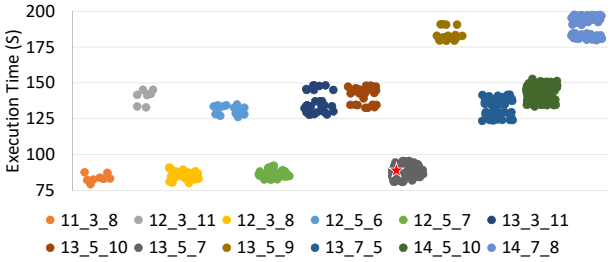

Figure 9. Candidate Schedules

We also run AutoMine and RStream on size-4 FSM with different *support* parameters. Figure 6 presents the results in log scale. The missing bars for RStream indicate execution failure due to insufficient disk space. "TO" on top of the bars for RStream shows that its execution times out after 48 hours. AutoMine successfully handles all the workloads. RStream needs tremendous disk space and fails to process Patents when *support* is small (i.e., less filtering). When *support* is large, most patterns are filtered out, so RStream can fit the data into main memory, producing better performance. Similar to the experiments on 3-FSM, AutoMine does not benefit much from using *support* except on MiCo with *support*=5K, which aggressively filters out most size-2 and size-3 patterns.

We run triangle counting and 3-motif counting on the 3 larger graphs used in the RStream paper as well as LiveJournal-2 and Youtube. RStream runs out of disk space or times out for all the 3-motif counting runs and TC on UK-2005. Figure 7 hence only shows the triangle counting results for RStream. AutoMine runs at least 140.5X faster than RStream, because it can easily fit the workload into memory even for the largest graph UK-2005, while RStream demands too much disk (e.g., at least 147TB for UK-2005).

8.3 Comparisons with ASAP

ASAP samples edges to produce approximate graph mining results. Though AutoMine always generates exact graph mining programs, we compare AutoMine with ASAP to show that without a high-performance baseline system, the approximation techniques fail to yield satisfactory performance. We point out that the ASAP paper uses 5% as the error target to report their results. This is an aggressive setting, because prior work shows that even with 1% error target, the approximation techniques can produce two orders of magnitude performance improvement [31]. Figure 8 shows the performance comparisons for 3-motif counting on 4 graphs used by the ASAP paper. Despite producing the exact counts using a single machine, AutoMine outperforms ASAP, running on 16 machines, by up to 68.8X (on average 12.8X). The reason is that ASAP follows Arabesque's basic approach to enumerate and store embeddings, hence inheriting the major weaknesses of inefficient algorithms and high memory consumption. It is possible to integrate the approximation techniques of ASAP into AutoMine when generating schedules, which has potential to produce much better performance if the user is willing to tolerate some accuracy loss.

8.4 Evaluating AutoMine's techniques

Schedule Selection AutoMine explores the schedule space and may generate many schedules for the same pattern. When there are multiple patterns (e.g., motif counting), the space is even larger with different combinations of the schedules for these patterns. To evaluate the effectiveness of AutoMine's automatic approach to produce an optimized combined schedule, we enumerate all the 560 possible combined schedules for size-4 motif counting, which operates on 6 patterns. Figure 9 shows the performance results of the schedules running on Patents. We use 3 parameters to group the schedules for a clear presentation of the data, namely the number of vertex sets, the number of intersection operations, and the number of subtraction operations in the generated static program. Each data point in the figure is represented by a tuple of these parameters. We make three observations. First, the schedules with the same parameters tend to perform similarly. Second, schedules with different parameters (e.g., 12_3_11 and 13_5_9) may also have similar performance.

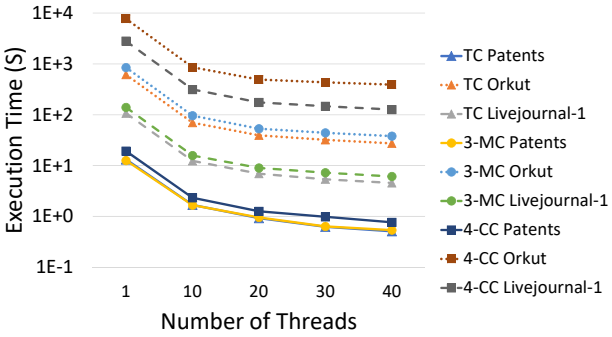

Figure 10. Threading scalability

Finally, the optimal schedule is about 2.4X faster than the slowest schedule. AutoMine's greedy approach described in Section 6.3 finds the schedule represented by the star symbol, which is 9.9% slower than the optimal schedule.

Multi-core Scalability AutoMine automatically generates parallel programs to leverage the multiple cores on the platform. Figure 10 shows the performance improvement with more threads. From 1 thread to 10 threads, AutoMine enjoys almost linear scalability, which becomes worse beyond 10 threads and further degrade beyond 20 threads. The reason is that the systems has 2 CPUs, each with 10 cores. AutoMine can efficiently utilize 1 CPU but using 2 CPUs triggers the NUMA effect, which is exacerbated by the irregular memory accesses inherent in graph applications. By launching more than 20 threads, AutoMine has to run more than one thread per core with hyper-threading, leading to diminishing returns. With low memory consumption and excellent scalability to physical cores, we would expect AutoMine to perform well in a distributed environment, even with trivial data replication, though such an evaluation is outside the scope of this paper.

Out-of-core Processing We use the largest graph, GSH-2015, to evaluate AutoMine's out-of-core processing capability. The graph has 25.6B edges, requiring 103.4GB disk space to store. To perform triangle counting on this graph, RStream needs at least 2.4PB disk space, and Arabesque requires at least 40,000 machines (each with 64GB of memory). Since AutoMine cannot fit the graph data into the main memory, it leverages the out-of-core support to perform triangle counting. AutoMine finishes triangle counting in 4966 seconds and triggers 3.8M page faults. It can even perform size-4 clique counting on this graph, which takes 45399 seconds (12.6 hours), triggering 35M page faults. This total amount of disk I/O is far more tolerable than what RStream requires, even using a mechanism as simple as paging.

Scalability to Larger Patterns We use Youtube and Orkut to evaluate AutoMine's scalability to larger patterns, specifically cliques of up to 8 vertices. Since the pattern matching problem is a variant of Subgraph Isomorphism, a well-known

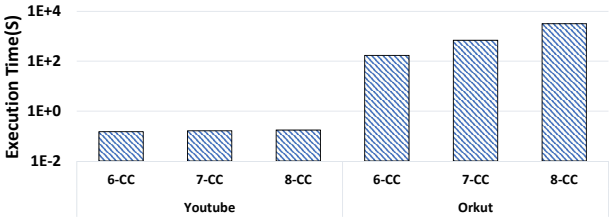

Figure 11. AutoMine Performance for Large Patterns

NP-Complete problem, scaling to large patterns is very difficult. As far as we are aware, these are the first 8-node pattern results published for graphs of this scale, which can be seen in Figure 11. RStream times out at 12 hours for all 6 of the experiments shown, even the ones that take AutoMine less than 1 second to complete. This showcases the real power of AutoMine to enable pattern mining at scales which were not possible with prior systems.

9 Related Work

Graph mining systems and algorithms. Arabesque [46] built on Giraph [1] is the first generic distributed graph mining system that spurs much interest in the community. G-thinker [55] and G-miner [10] address some of the performance issues of Arabesque with lower-level interfaces. Unfortunately, the current release of the systems does not support frequent subgraph mining and motif counting. DistGraph [45] is a distributed system to focus on FSM. It leverages pruning techniques to reduce the search space and provides optimized graph partitioning and collective communication operations. ScaleMine [3] is an MPI-based system to perform FSM, which uses approximation to optimize load balancing, prune the search space, and guide intra-task parallelism. ASAP [23] accelerates graph mining by sampling subgraph patterns but can only produce approximate results.

The distributed graph mining systems use expensive clusters and are difficult to debug. To address these issues, Wang et al. propose RStream [51], the first single-machine, out-of-core mining system. It supports a rich set of relational algebra operators, such as join, for programmers to compose mining applications which are executed by the underlying runtime through data streaming from and to disk. Though RStream efficiently implements out-of-core processing and the relational algebra operators, its abstraction, which is similar to that of Arabesque, leads to inefficient graph mining algorithms and high memory consumption by introducing severe, unnecessary synchronization. In contrast, AutoMine leverages a fundamentally different set-centric abstraction, so neither Arabesque nor RStream can implement our methods.

Motif counting has attracted significant attention in the data mining community [4, 22, 30, 38, 53]. Ahmed et al. carefully consider the combinatorial properties of the motifs to

reduce the complexity of the algorithms [4]. Our automatically generated algorithms are similar to their manually designed ones. Researchers also propose approximate motif counting algorithms [6, 13, 31, 36, 38] based on sampling, which may be implemented in AutoMine if exact counting is not required.

Graph computation systems. Most distributed graph computation systems [11, 15, 18, 19, 28, 29, 33, 39, 41, 54, 56, 60, 64] implement the vertex-centric or edge-centric programming model. The programmer has to implement low-level functions to run on each vertex or edge, which are difficult to write to identify subgraph patterns. Performance optimization of such systems mainly focuses on data reorganization [64], load balancing [11], communication reduction [18], or graph partitioning [17, 64]. AutoMine may implement some of these techniques (e.g., locality optimization) to further improve performance.

Many single-machine graph computation systems [34, 37, 42, 44, 52, 58, 59] assume that the input graph as well as the intermediate data can fit in the main memory. Because modern machines typically have large memory and the graph computation algorithms do not generate much intermediate data, these systems can practically handle most real-world graphs. They heavily optimize locality and scheduling, and achieve great performance for a broad set of graph computation problems.

Out-of-core graph computation on a single machine has attracted great attention since the introduction of GraphChi [25], the first of its kind. The idea is to stream edges and updates from and to disk as partitions if they do not fit in the memory. Many systems [5, 20, 21, 61, 63, 65] leverage this idea with different optimizations. X-stream [40] optimizes away random accesses on vertex data but may stream unuseful edges to the memory. Vora et al. [49] proposes a runtime to filter out edges that make no contribution to the update of vertices. GraphQ [50] can figure out the edge partitions that may be needed by the queries and only load these partitions. AutoMine leverages memory-mapped I/O to support out-of-core processing that efficiently exploits the locality of subgraph patterns and the neighbor set of hot vertices.

Compiler optimization for graph computation. Zhang et al. propose a DSL called GraphIt [62] for graph computations and a compiler to generate efficient code. GraphIt, similar to Halide [27] for image processing and TVM [12] for deep learning, separates the algorithm and its schedule. This decomposition enables relatively easy application of a set of compiler optimization techniques, such as edge traversal direction, locality improvement, and kernel fusion. While GraphIt has a sufficiently general programming interface, it cannot express the functional relationships between vertices. So its optimization phases cannot fundamentally restructure the loop ordering to resolve dependencies according to the ideas proposed in this paper. Pai and Pingali [35] propose a compiler to compile graph computation algorithms for GPUs. The compiler particularly addresses the challenges of optimizing throughput due to the special thread organization, the SIMD execution model, and the complex memory hierarchy of the GPU architecture. The Abelian [16] compiler takes a graph computation algorithm and generates code to execute on distributed systems with heterogeneous processors.

10 Conclusion and Future Work

We proposed a system to produce up to several-order-of-magnitude higher performance than existing systems for a variety of graph mining tasks on real-world graphs. The system provides high-level interfaces, which assume no knowledge of the user about graph mining algorithms, and automatically generates efficient mining programs. Though the system runs on a single machine, it can process very large graphs with tens of billions of edges that existing systems cannot handle. It is interesting to extend AutoMine for distributed processing when the input graphs cannot even fit into the disk of a single machine. In this case, AutoMine's local exploration complicates graph partitioning and load balancing. It is also critical to implement the basic set operators efficiently in a distributed manner.

Acknowledgements

We would like to thank Muthian Sivathanu (our shepherd) and the anonymous reviewers for their constructive comments. We also would like to thank Feng Yan for sharing his servers to conduct part of the experiments. This project was supported in part by NSF grant CCF-1823005 and an NSF CAREER Award (CNS-1750760).

References

[1] Apache giraph. http://giraph.apache.org/.

[2] Orkut social network. http://snap.stanford.edu/data/com-Orkut.html.

[3] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad 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 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pages 716–727, 2016.

[4] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick G. Duffield. Efficient graphlet counting for large networks. In *2015 IEEE International Conference on Data Mining, ICDM 2015, Atlantic City, NJ, USA, November 14-17, 2015*, pages 1–10, 2015.

[5] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 125–137, 2017.

[6] Maryam Aliakbarpour, Amartya Shankha Biswas, Themitstoklis Gouleakis, John Peebles, Ronitt Rubinfeld, and Anak Yodpinyanee. Sublinear-time algorithms for counting star subgraphs with applications to join selectivity estimation. *CoRR*, abs/1601.04233, 2016.

[7] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: Membership, growth,

and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 44–54, New York, NY, USA, 2006. ACM.

[8] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[9] Paolo Boldi and Sebastiano Vigna. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 595–602, 2004.

[10] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 32:1–32:12, 2018.

[11] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 1:1–1:15, 2015.

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 578–594, 2018.

[13] Talya Eden, Amit Levi, Dana Ron, and C. Seshadhri. Approximately counting triangles in sublinear time. In *FOCS*, 2015.

[14] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. GRAMI: frequent subgraph and pattern mining in a single large graph. *PVLDB*, 7(7):517–528, 2014.

[15] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. Parallelizing sequential graph computations. *ACM Trans. Database Syst.*, 43(4):18:1–18:39, December 2018.

[16] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. Abelian: A compiler for graph analytics on distributed, heterogeneous platforms. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, pages 249–264, 2018.

[17] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. A study of partitioning policies for graph analytics on large-scale distributed platforms. *PVLDB*, 12(4):321–334, 2018.

[18] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30, 2012.

[19] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, 2014. USENIX Association.

[20] Wei Han, Daniel Mawhirter, Matthew Buland, and Bo Wu. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, Oregon, USA, September 9-13*, 2017.

[21] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 77–85, 2013.

[22] Tomaz Hocevar and Janez Demsar. A combinatorial approach to graphlet counting. *Bioinformatics*, 30(4):559–565, 2014.

[23] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, approximate graph pattern mining at scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 745–761, Carlsbad, CA, 2018. USENIX Association.

[24] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy G. Mattson, and José E. Moreira. Mathematical foundations of the graphblas. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, pages 1–9, 2016.

[25] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, 2012.

[26] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pages 177–187, 2005.

[27] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Trans. Graph.*, 37(4):139:1–139:13, 2018.

[28] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[29] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, 2010.

[30] D. Marcus and Y. Shavitt. Rage - a rapid graphlet enumerator for large networks. *Comput. Netw.*, 56(2):810–819, February 2012.

[31] Daniel Mawhirter, Bo Wu, Dinesh Mehta, and Chao Ai. Approxg: Fast approximate parallel graphlet counting through accuracy control. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*, pages 533–542, 2018.

[32] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015.

[33] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 439–455, 2013.

[34] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 456–471, 2013.

[35] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 1–19, 2016.

[36] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and sampling triangles from a graph stream. *PVLDB*, 6(14):1870–1881, 2013.

[37] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018, Limassol, Cyprus, November 01-04, 2018*, pages 9:1–9:14, 2018.

[38] Mahmudur Rahman, Mansurul Bhuiyan, and Mohammad Al Hasan. Graft: An approximate graphlet counting algorithm for large graph analysis. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 1467–1471, 2012.

[39] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, 2015.

[40] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, 2013.

[41] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 317–332, 2016.

[42] Julian Shun and Guy 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 '13, pages 135–146, 2013.

[43] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 149–160, 2015.

[44] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11), July 2015.

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

[46] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 425–440, 2015.

[47] Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*, pages 1–7, 2017.

[48] Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*, pages 1–7, 2017.

[49] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 507–522, Denver, CO, 2016. USENIX Association.

[50] Kai Wang, Guoqing (Harry) Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement - scalable and programmable analytics over very large graphs on a single PC. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 387–401, 2015.

[51] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on A single machine. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 763–782, 2018.

[52] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 265–266, New York, NY, USA, 2015. ACM.

[53] Sebastian Wernicke and Florian Rasche. Fanmod: A tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, May 2006.

[54] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. Gram: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, pages 408–421, 2015.

[55] Da Yan, Hongzhi Chen, James Cheng, M. Tamer Özsu, Qizhen Zhang, and John C. S. Lui. G-thinker: Big graph mining made easier and faster. *CoRR*, abs/1709.03110, 2017.

[56] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[57] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[58] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. Finepar: Irregularity-aware fine-grained workload partitioning on integrated architectures. In *The International Symposium on Code Generation and Optimization*, 2017.

[59] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 183–193, 2015.

[60] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 285–300, 2016.

[61] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 608–621, 2018.

[62] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. Graphit: a high-performance graph DSL. *PACMPL*, 2(OOPSLA):121:1–121:30, 2018.

[63] Da Zheng, Disa Mhembere, Randal C. Burns, Joshua T. Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 45–58, 2015.

[64] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, Savannah, GA, 2016. USENIX Association.

[65] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, 2015. USENIX Association.