Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining

Xuhao Chen
MIT CSAIL
Cambridge, Massachusetts, USA
cxh@mit.edu

Roshan Dathathri, Gurbinder Gill
Katana Graph
{roshan,gill}@katanagraph.com

Loc Hoang, Keshav Pingali
The University of Texas at Austin
Austin, Texas, USA
{loc,pingali}@cs.utexas.edu

Abstract

Graph pattern mining (GPM) is a key building block in diverse applications, including bioinformatics, chemical engineering, social network analysis, recommender systems and security. Existing GPM frameworks either provide high-level interfaces for productivity at the cost of expressiveness or provide expressive low-level interfaces at the cost of increased programming complexity. They also lack the flexibility to explore combinations of optimizations to achieve performance competitive with hand-optimized applications.

We present Sandslash, an in-memory graph pattern mining framework that uses a novel programming interface to support productive, expressive, and efficient GPM on large graphs. Sandslash provides a high-level API that only needs a specification of the GPM problem from the user, and the system implements fast subgraph enumeration, provides efficient data structures, and applies high-level optimizations automatically. To achieve performance competitive with expert-optimized implementations, Sandslash also provides a low-level API that allows users to express algorithm-specific optimizations. This enables Sandslash to support both high-productivity and high-efficiency without losing expressiveness. We evaluate Sandslash using five GPM applications and a wide range of real-world graphs. Experimental results demonstrate that applications written using Sandslash's high-level or low-level API outperform those in state-of-the-art GPM systems AutoMine, Pangolin, and Peregrine on average by 13.8×, 7.9×, and 5.4×, respectively. We also show that these Sandslash applications outperform expert-optimized GPM implementations by 2.3× on average with less programming effort.

ACM Reference Format:
Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Loc Hoang, Keshav Pingali. 2021. Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3447818.3460359

ICS '21, June 14–17, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8335-6/21/06.
https://doi.org/10.1145/3447818.3460359

1 Introduction

Graph pattern mining (GPM) problems exist in many application domains [5, 16, 18, 22]. One example is *motif counting* [7, 25, 43], which counts the number of occurrences of certain structural *patterns* in a given graph (Fig. 1). These numbers are often different for graphs from different domains, so they can be used as a "signature" to infer, for example, the probable origin of a graph [22].

GPM problems are solved by searching the input graph for patterns of interest. Programming efficient parallel solutions for GPM problems is challenging. For efficiency, the search space needs to be pruned aggressively without compromising correctness. Complex book-keeping data structures are needed to avoid repeating work during the search process; maintaining them efficiently in a parallel program can be challenging. A number of GPM frameworks exist that reduce these burdens on the programmer [13, 31, 40, 41, 57, 59, 64], and they can be categorized into *high-level* and *low-level* systems. Both simplify GPM programming compared to hand-optimized code, but they make different tradeoffs.

High-level systems such as AutoMine [41] and Peregrine [31] take specifications of patterns as input and leverage static analysis techniques to automatically generate GPM programs for those patterns. These systems promote productivity, but they do not provide enough flexibility to allow the programmer to express more efficient algorithms. Low-level systems such as RStream [59] and Pangolin [13] provide low-level API functions for the user to control the details of mining process, and they can be used to implement solutions for a wider variety of GPM problems, but they require more programming effort. Programming in these low-level APIs may prevent those systems from applying other optimizations as the problem may become over-specified or unnecessarily constrained. Moreover, both high-level and low-level systems lack the ability to explore combinations of optimizations that have been implemented in handwritten GPM solutions for different problems.

We present *Sandslash*, an in-memory GPM system that provides high productivity and efficiency without compromising generality. Sandslash provides a novel programming interface that separates problem specifications from algorithmic optimizations. The Sandslash high-level interface requires the user to provide only the specification of the pattern(s) of interest. Sandslash analyzes the specification and automatically enables efficient search strategies, data representations,

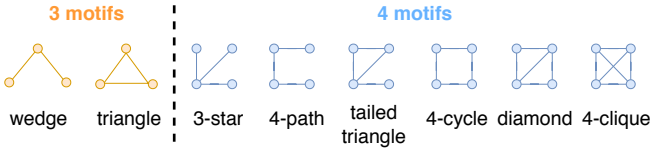


Figure 1: 3-vertex (left) and 4-vertex (right) motifs.

and optimizations. This is achieved by augmenting the vertex/edge extension model with pattern-awareness. We automate existing optimizations used in specific problems, such that they can be applied more generally to other GPM problems. Moreover, users can leverage the Sandslash low-level API to customize their algorithms and improve performance further. Note that this low-level interface is not exposed in prior high-level systems, and only partially exposed in prior low-level systems.

Evaluation on a 56-core CPU demonstrates that applications written using Sandslash high-level API outperform those in the state-of-the-art GPM systems, AutoMine [41], Pangolin [13], and Peregrine [31] by 7.7×, 6.2× and 3.9× on average, respectively, due to the high-level optimizations that are not enabled in prior systems. Applications using Sandslash low-level API which contain low-level optimizations outperform AutoMine, Pangolin, and Peregrine by 22.6×, 27.5× and 7.4× on average, respectively. Sandslash applications are also 2.3× faster on average than expert-optimized GPM applications, mainly due to the flexible combination of optimizations explored by Sandslash.

This work makes the following contributions:

- We present Sandslash, an in-memory graph pattern mining system that supports productive, expressive, and efficient pattern mining on large graphs.
- We propose a high-level programming model in which the choice of efficient search strategies, subgraph data structures, and high-level optimizations are automated, and we expose a low-level programming model to allow the programmers to express algorithm-specific optimizations.
- We holistically analyze the optimization techniques available in hand-tuned applications and separately enabled them in the two levels. We show that existing optimizations in the literature applied to specific problems/applications can be applied more generally to other GPM problems. The user is allowed to flexibly control and explore the combination of optimizations.
- Experimental results show that Sandslash substantially outperforms existing GPM systems. They also show the impact of (and the need for) Sandslash's low-level API. Compared to expert-optimized applications, Sandslash achieves competitive performance with less programming effort.

2 Background

2.1 Problem Definition

The following definitions are standard [27]. Given a graph $G(V,E)$, a *subgraph* $G'(V',E')$ of G is a graph where $V' \subseteq V$, $E' \subseteq E$. If G_1 is a subgraph of G_2, we denote it as $G_1 \subseteq G_2$.

Given a vertex set $W \subseteq V$, the *vertex-induced subgraph* is the graph G' whose (1) vertex set is W and whose (2) edge set contains the edges in E whose endpoints are in W. Given an edge set $F \subseteq E$, the *edge-induced subgraph* is the graph G' whose (1) edge set is F and whose (2) vertex set contains the endpoints in V of the edges in F.

The difference between the two types of subgraphs can be understood as follows. Suppose v_1 and v_2 are connected by an edge e in G. If v_1 and v_2 occur in a vertex-induced subgraph, then e occurs in the subgraph as well; in an edge-induced subgraph, edge e will be present only if it is in the given edge set F. Any vertex-induced subgraph can be formulated as an edge-induced subgraph.

Two graphs $G_1(V_1,E_1)$ and $G_2(V_2,E_2)$ are *isomorphic*, denoted $G_1 \simeq G_2$, if there exists a bijection $\mathbf{f}$: $V_1 \rightarrow V_2$, such that any two vertices u and v of G_1 are adjacent in G_1 if and only if $\mathbf{f}u$ and $\mathbf{f}v$ are adjacent in G_2. In other words, G_1 and G_2 are structurally identical. When $\mathbf{f}$ is a mapping of a graph onto itself, i.e., G_1 and G_2 are the same graph, G_1 and G_2 are *automorphic*, i.e. $G_1 \cong G_2$.

A *pattern* is a graph defined explicitly or implicitly. An **explicit** definition specifies the vertices and edges of the graph while an **implicit** definition specifies its desired properties. Given a graph $\mathcal{G}$ and a pattern $\mathcal{P}$, an *embedding* $\mathcal{X}$ of $\mathcal{P}$ in $\mathcal{G}$ is a vertex- or edge-induced subgraph of $\mathcal{G}$ s.t. $\mathcal{X} \simeq \mathcal{P}$.

Given an undirected graph $\mathcal{G}$ and a set of patterns $S_p = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, GPM finds the vertex- or edge-induced embeddings of $\mathcal{P}_i$ in G. For explicit-pattern problems, the solver finds embeddings of the given pattern(s). For implicit-pattern problems, S_p is described using some high-level rules. Therefore, the solver must find the patterns as well as the embeddings. If the cardinality of S_p is 1, we call this problem a *single-pattern* problem. Otherwise, it is a *multi-pattern* problem. Note that *graph pattern matching* [23] finds embeddings only for explicit pattern(s), whereas *graph pattern mining* [3] solves both explicit- and implicit-pattern problems. In some GPM problems, the required output is the list of embeddings. However, in other GPM problems, users want statistics such as a count of the occurrences of the pattern(s) in $\mathcal{G}$. A particular statistic for $\mathcal{P}$ in $\mathcal{G}$ is called its *support*. The support is *anti-monotonic* if the support of a supergraph does not exceed the support of a subgraph.

Graph Pattern Mining Problems We consider the following GPM problems with the input graph $\mathcal{G}$.

- *Triangle Counting* (TC): The problem is to count the number of triangles in $\mathcal{G}$. It uses vertex-induced subgraphs.
- *k-Clique Listing* (k-CL): A subset of vertices W of G is a *clique* if every pair of vertices in W is connected by an edge in G. If the cardinality of W is k, this is called a k-clique (triangles are 3-cliques). The problem of listing k-cliques is denoted k-CL, and it uses vertex-induced subgraphs.
- *Subgraph Listing* (SL): The problem is to enumerate all edge-induced subgraphs of $\mathcal{G}$ isomorphic to a pattern $\mathcal{P}$.
- *Subgraph Counting* (SC): same as SL but does counting.
- *k-Motif Counting* (k-MC): It counts the number of occurrences of the different patterns that are possible with k

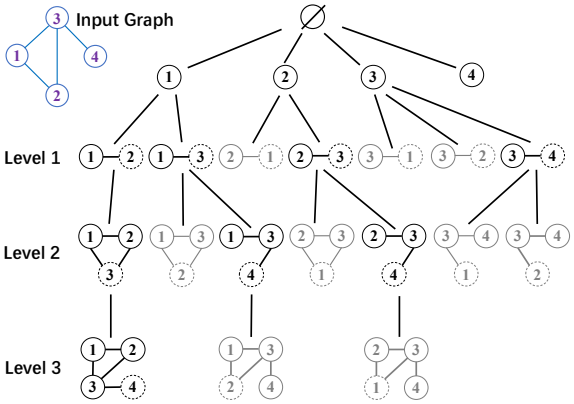


Figure 2: A portion of a vertex-induced subgraph tree with 3 levels. Lightly colored subgraphs are removed from consideration by automorphism checks.

vertices. In the literature, each pattern is called a *motif*. Fig. 1 shows all 3-motifs and 4-motifs. This problem uses vertex-induced subgraphs.

- *k-Frequent Subgraph Mining* (k-FSM): Given a labeled input graph $\mathcal{G}$, an integer k and a threshold σ_{min} for support, k-FSM finds all *frequent* patterns with k or fewer edges where a pattern is frequent if its support is greater than σ_{min}. If k is not specified, it is set to ∞, meaning it considers all possible values of k. FSM is a implicit-pattern problem, and it finds edge-induced subgraphs. The support in FSM is often defined as the *minimum image-based* (MNI) support (*a.k.a. domain support*), where MNI is the minimum number of distinct mappings for any vertex in the pattern over all embeddings of the pattern. MNI support is anti-monotonic.

Note that *Counting* and *listing* may have different search spaces because listing requires enumerating every embedding, but counting does not. Therefore counting allows more aggressive pruning in many cases where the total count can be computed by only enumerating a small portion of the search tree (see Section 5.1).

2.2 Subgraph Trees and Vertex/Edge Extension

The vertex-induced subgraphs of a given input graph $\mathcal{G}$ can be ordered naturally by containment (*i.e.*, if one is a subgraph of the other). It is useful to represent this partial order as a *vertex-induced subgraph tree* whose vertices represent the subgraphs. Level l of the tree represents subgraphs with l 1 vertices. The root vertex of the tree represents the empty subgraph. Intuitively, subgraph S_2=W_2, E_2 is a child of subgraph S_1=W_1, E_1 in this tree if S_2 can be obtained by extending S_1 with a single vertex $v \notin W_1$ that is connected to some vertex in W_1 (v is in the *neighborhood* of subgraph S_1); this process is called *vertex extension*. Formally, this can be expressed as W_2=$W_1 \cup \{v\}$ where $v \notin W_1$ and an edge $v, u \in E$ exists for some $u \in W_1$. It is useful to think of the edge connecting S_1 and S_2 in the tree as being labeled by v. Fig. 2 shows a portion of a vertex-induced subgraph tree with three levels (for lack of space, not all subgraphs are shown). Note that a specific subgraph can occur in multiple places in this tree. For example, in Fig. 2, the subgraph containing vertices 1 and 2 occurs in two places, i.e., [1, 2] and [2, 1]. These identical subgraphs are called automorphisms, i.e., they are automorphic with each other. Automorphisms can cause *over-counting* or *multiplicity*, i.e., the same subgraph is counted multiple times. The *edge-induced subgraph tree* for $\mathcal{G}$ can be defined in a similar way. *Edge extension* extends an edge-induced subgraph S_1 with a single edge u, v provided at least one of the endpoints of the edge is in S_1.

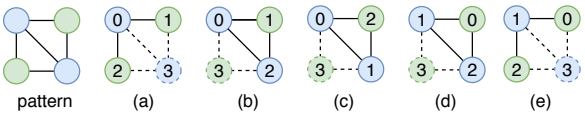


Figure 3: Possible matching orders for pattern `diamond`. The number in each vertex is not a vertex ID but the order being matched. Colors show the symmetric positions. The matching process can start from blue vertices (a & b & c) or green vertices (d & e). Among them, (a) and (e) match a wedge first, and then form a diamond; (b) (c) and (d) discover a triangle first, and then form a diamond.

The subgraph tree is a property of the input graph $\mathcal{G}$. When solving a specific GPM problem, a solver uses the subgraph tree as a search tree and builds a *prefix* of the subgraph tree that depends on the problem, pattern, and other aspects of the implementation (e.g., if the size of the pattern is k, subgraphs of larger size are not explored). We use the term *embedding tree* to refer to the prefix of the subgraph tree that has been explored at any point in the search. Finally, since a pattern is a graph, its connected subgraphs form a tree as well. These subgraphs are called *sub-patterns*, and the tree formed by them is called the *sub-pattern tree*.

2.3 Pattern-Aware GPM Solutions

A straightforward approach to solving a GPM problem is to build the search tree and perform a graph isomorphism test on each tree leaf to check if the subgraph matches the pattern. This approach is oblivious to the given pattern during the search and is general enough to solve a wide range of GPM applications, including both *explicit* and *implicit* pattern problems. A more efficient approach is to use pattern analysis to generate a *matching order* and a *symmetry order* to prune the search space, avoid isomorphism tests, and perform symmetry breaking. We describe matching order and symmetry order below. To avoid notational confusion, we denote a vertex in $\mathcal{P}$ as a *pattern vertex* and a vertex in $\mathcal{G}$ as a *data vertex*.

Matching Order is a total order among pattern vertices that defines the order to match each data vertex to a pattern vertex. Fig. 3 shows the 5 possible matching orders for `diamond`. Matching orders have different compute/memory requirements. For example, for `diamond`, we can find the triangle first, and then add the fourth vertex connected to two of the endpoints of the triangle. Another possibility is to find a wedge first and then find the fourth vertex that is connected to all three vertices of the wedge. In real-world sparse graphs, the number of wedges is usually orders-of-magnitude larger

than the number of triangles, so it is more efficient to find the triangle first. Using matching order avoids isomorphism tests if the patterns are explicit. However, for implicit-pattern problems, this approach needs to *enumerate all the possible patterns before search*. For example, FSM in AutoMine generates a matching order for each unlabeled pattern and includes a lookup table to distinguish between labeled patterns. This table is massive when the graph has many distinct labels. Peregrine also suffers significant overhead for FSM since there are many patterns that it matches one by one.

Symmetry Order is a specific partial order over data vertices in an enumerated subgraph [31, 34]. This order is used to avoid automorphic enumerations (a.k.a overcounting), by finding a *canonical* representation from identical subgraphs, i.e., only subgraphs that satisfy the partial order are counted. This technique is also known as *symmetry breaking*. A valid order guarantees each subgraph is enumerated once and only once. In Fig. 2, lightly colored subgraphs are removed by symmetry breaking, leaving a unique canonical subgraph for each set of automorphisms. Symmetry breaking can significantly prune the search tree: *e.g.*, the subgraph [2,1] is not extended in Fig. 2 because it is automorphic to the subgraph [1,2]. For a k-clique whose every pair of vertices are symmetric, the partial order becomes a total order. In these cases, one can convert $\mathcal{G}$ into a directed acyclic graph (DAG), and dynamic automorphism checks are then not needed [13, 17]. The search done over the DAG instead of the original graph: this reduces the search space. The technique of constructing the DAG for k-cliques is known as *orientation* [17].

Given a pattern $\mathcal{P}$, one can use *pattern analysis* [41] to generate a matching order [31] and a symmetry order [40]. To generate a matching order, the pattern analyzer first enumerates all the possible matching orders of $\mathcal{P}$ and uses a set of rules to pick one that is likely to perform well in practice [34]. To generate a symmetry order, we take one of the matching orders $\mathcal{MO}$ of $\mathcal{P}$ and build a subgraph S of $\mathcal{P}$ incrementally by adding one vertex at a time in the order specified by $\mathcal{MO}$. At each step, if the newly added vertex w is *symmetric*[1] to any of the existing vertices v in S, we add a partial order among w and v.

Using matching order and symmetry order to mine a graph is a *pattern-aware* approach: the pattern guides the search. Automated pattern analysis [31, 41] can improve both performance and productivity. However, to generate matching order and symmetry order, the patterns of interest must be known explicitly. Therefore, for implicit pattern problems, existing systems need to enumerate all the possible (but not necessarily interesting) patterns which results in non-trivial performance overhead and memory consumption.

3 Sandslash

To provide flexibility of user-defined optimizations while retaining productivity and expressiveness, Sandslash provides a *two-level* programming interface to separate functional specification with optimizations. Fig. 4 shows the overview of Sandslash, which is built on top of the Galois [44] parallel system. High-level Sandslash accepts a GPM problem specification and performs search without further guidance from the user (Section 3.1). Low-level Sandslash allows the user to customize the search strategy to boost performance using the low-level API functions (Section 3.2). The low-level implementation is optional. Sandslash constructs a solver for the problem and parallelizes the solver.

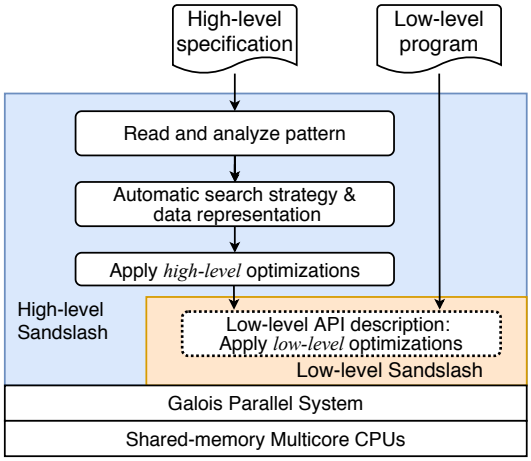


Figure 4: Overview of Sandslash. Dash indicates optional.

3.1 High-Level API

Table 1 shows Sandslash high-level API. The first two required flags define whether the embeddings are vertex-induced or edge-induced and whether the matched embeddings need to be listed or counted. The third required flag defines if the set of patterns is explicit or implicit. If they are explicit, then the patterns must be defined using `getExplicitPatterns()`. Otherwise, the rule to select implicit patterns must be defined using `isImplicitPattern()`. `process()` is a function for customized output. `terminate()` specifies an optional early termination condition (useful to implement pattern existence queries). The default *support* for each pattern in Sandslash is count (number of embeddings). The *support* can be customized using three functions: `getSupport()` defines the support of an embedding, `isSupportAntiMonotonic()` defines if the support has the anti-monotonic property, and `reduce()` defines the reduction operator (e.g., sum) for combining the support of different embeddings of the same pattern. In addition, Sandslash has a runtime parameter k to denote the maximum size (vertices or edges) of the (vertex- or edge-induced) embeddings to find.

The problem specifications for TC, CL, SL, and MC are straightforward. They all specify an explicit set of patterns[2]. Each pattern is specified using an edge-list. For example, in TC, the user provides an edge-list of {(0,1) (0,2) (1,2)}. Table 1's right column shows the problem specification for FSM.

[1] w and v are symmetric if any match of $\mathcal{P}$ in $\mathcal{G}$ will be found twice by permuting w and v when no order is enforced between w and v.

[2] Sandslash provides helper functions to generate a clique or all patterns of a given size k. It also allows reading the patterns from files.

Flag	Required	Example: spec for FSM
isVertexInduced	yes	false
isListing	yes	false
isExplicit	yes	false
Function	**Required**	**Example: user code for FSM**
getExplicitPatterns()	no	-
isImplicitPattern(Pattern pt)	no	pt.support $\geq$ MIN_SUPPORT
process(Embedding emb)	no	-
terminate(Embedding emb)	no	-
isSupportAntiMonotonic()	no	true
getSupport(Embedding emb)	no	getDomainSupport(emb)
reduce(Support s1, Support s2)	no	mergeDomainSupport(s1, s2)

Table 1: Left column lists Sandslash high-level API flags and functions. Right column is the spec and user code of FSM using the API.

```
1 bool toExtend(Embedding emb, Vertex v);
2 bool toAdd(Embedding emb, Vertex u);
3 bool toAdd(Embedding emb, Edge e);
4 Pattern getPattern(Embedding emb);
5 void localReduce(int depth, vector<Support> &sups);
6 void initLG(Graph gg, Vertex v, Graph lg);
7 void initLG(Graph gg, Edge e, Graph lg);
8 void updateLG(Graph lg);
```

Listing 1: Sandslash low-level API functions.

`isImplicitPattern()` is used to specify that only *frequent* patterns (i.e., those with support greater than `MIN_SUPPORT`) are of interest. FSM uses the *domain* (MNI) support and its associated reduce operation. Sandslash provides helper functions `getDomainSupport` and `mergeDomainSupport` for the standard definition of domain support which is anti-monotonic.

3.2 Low-Level API

Sandslash low-level API shown in Listing 1 is designed to give fine-grained control to the user. This includes customizing (1) the graph to search (`initLG`, `updateLG`), (2) the extension candidates and their selection (`toAdd`, `toExtend`), and (3) the reduction operations to perform (`getPattern`, `localReduce`). The low-level API is expressive enough to support sophisticated algorithms.

`toExtend()` determines if a vertex v in embedding *emb* must be extended. `toAdd` decides if extending embedding *emb* with vertex u (or edge e) is allowed. `toExtend` and `toAdd` can be used to do fine-grained pruning to reduce search space (Section 5.3).

`getPattern()` returns the pattern of an embedding. This function can be used to replace the default graph isomorphism test with a custom method to identify patterns (Section 5.3). Note that `Pattern` can be user-defined; therefore, Sandslash can support custom aggregation-keys like Fractal [20].

Some algorithms [4] do *local counting* for a vertex or edge instead of global counting. Sandslash provides `localReduce` to support this; Listing 2 shows 3-MC using this. Some algorithms [17] search local (sub-)graphs instead of the (global) input graph. `initLG` and `updateLG` can be defined for search on local graphs.

Note that in prior low-level frameworks, e.g. Fractal, specification of the problem in the low-level API may prevent the system from applying high-level optimizations as the problem is over-specified or unnecessarily-constrained. Sandslash's low-level API, however, is designed such that it can apply any high-level optimization.

3.3 Discussion on System Tradeoffs

Sandslash's high-level API provides the same productivity as existing high-level systems. For example, for explicit-pattern GPM problems, the programmer only needs to provide the pattern of interest. High-level API is much easier to use as opposed to the existing low-level systems which require user code to select or filter patterns [3]. However, the ease of use of high-level Sandslash does not come at the cost of the performance: high-level Sandslash already outperforms all existing high- and low-level GPM systems (Section 6) without loss in programmer productivity.

For expert programmers, Sandslash's low-level API exposes a lower level interface that is not present in existing high-level systems: therefore, high-level systems lack the low-level optimizations. Using the low-level API requires extra programming effort, but it boosts performance on top of high-level Sandslash as it enables the user to express custom algorithms while allowing the system to apply high-level optimizations and explore different traversal orders. In contrast, some low-level systems (e.g., Fractal) expose an even lower level interface to give the user full control of the mining process at the cost of preventing the system from applying high-level optimizations. For example, to implement local graph search in Sandslash, the user only implements initialization and modification functions for the local graph; Sandslash still applies all possible high-level optimizations in Table 2a during exploration. In Fractal, the user must change the entire exploration which includes implementing all the optimizations by hand: the system does not apply optimizations automatically.

Furthermore, Sandslash is expressive enough to support all features that fit in its pattern-aware vertex/edge extension abstraction (Section 4.1) and can be extended to support new features that fit in the abstraction (e.g., anti-edges and anti-vertices in Peregrine). Sandslash's vertex/edge extension is more efficient than the set intersection/difference model used in prior high-level systems: Peregrine and AutoMine (set model) must enumerate patterns before the search which leads to non-trivial overhead for FSM.

4 High-Level Sandslash

We describe high-level Sandslash which uses efficient search strategies (Section 4.1), data representations (Section 4.2), and automatic application of high-level optimizations (Section 4.3).

[3] Note Fractal supports high-level API but only for **single**, explicit-pattern problems. For implicit-pattern problems like FSM, Fractal requires the user to write code for iterative expand-aggregate-filter, which is also more complex than high level systems.

4.1 Search Strategies

Given $\mathcal{G}$ and $\mathcal{P}$ with k vertices, one can build the subgraph tree (Section 2) to a depth k and test each subgraph $\mathcal{X}$ at the leaves of the tree to see if $\mathcal{X} \simeq \mathcal{P}$. This pattern-oblivious approach works effectively for any pattern (even implicit patterns). In Sandslash, we augment this model with pattern-awareness. If the user defines an explicit pattern problem, Sandslash uses matching order (Section 2.3) for vertex extension to prune the search tree and avoids isomorphism tests. Sandslash also uses the standard *symmetry breaking* technique (Section 2.3) to avoid over-counting. Sandslash performs a depth-first search (DFS) parallel exploration as follows:

- Each vertex v in $\mathcal{G}$ corresponds to a vertex-induced subgraph for the vertex set $\{v\}$ and corresponds to a search tree vertex t_v.
- The subtree below each such t_v is explored in DFS order. This is a *task* executed serially by a single thread. When the exploration reaches the pattern size k, the support is updated appropriately.
- Multiple threads execute different tasks in parallel. The runtime does work-stealing of tasks for thread load-balancing.

Pattern filtering for implicit-pattern problems that use anti-monotonic support. The search strategy described above mines implicit-pattern problems by enumerating all embeddings, binning them according to their patterns, and checking the support for each pattern. This can be optimized by exploiting the sub-pattern tree when the support is anti-monotonic: if a sub-pattern does not have enough support, then its descendants in the sub-pattern tree will not have enough support and can be ignored. Instead of pruning sub-patterns during post-processing, one can prune after generating all the embeddings for a given sub-pattern. This is easy in breadth first search (BFS): embeddings are generated level by level, and in each level, the entire list of the embeddings is scanned to aggregate support for each sub-pattern. However, this does not work for DFS (what Sandslash uses) of the sub-graph tree as DFS is done by each thread independently.

To handle pattern-wise aggregation, Sandslash performs a DFS traversal on the sub-pattern tree instead of the subgraph tree. This ensures that the embeddings for a given sub-pattern are generated by a single thread using the same approach for *pattern extension* in hand-optimized gSpan [61]: i.e., the embeddings are gathered to their pattern bins during extension and canonicality is checked for each sub-pattern to avoid duplicate pattern enumeration. When the thread finishes extension in each level, the support for each sub-pattern can be computed using its own bin of embeddings.

4.2 Representation of Tree

Since subgraphs are created incrementally by vertex/edge extension in the subgraph tree, the representation of subgraphs should allow structure sharing between parent and child subgraphs. We describe the information stored in the search tree and the concrete representation of the tree for vertex-induced sub-graphs.

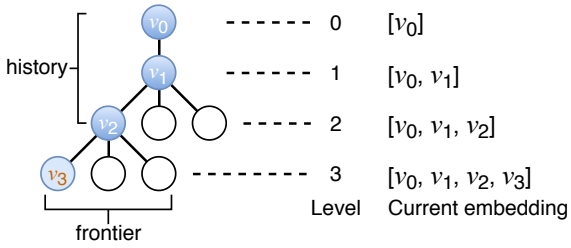


Figure 5: Embedding data structure (vertex-induced).

- Each non-root vertex in the tree points to its parent vertex.
- Each non-root vertex in the tree corresponds to a subgraph obtained from its parent subgraph by vertex extension with some vertex v of the input graph. The vertex set of a subgraph can be obtained by walking up the tree and collecting the vertices stored on the path to the root. These vertices are the *predecessors* of v in the embedding; they correspond to vertices discovered before v in that embedding. As shown in Fig. 5, the leaf containing v_3 represents the subgraph with a vertex set of $\{v_3, v_2, v_1, v_0\}$, which are the vertices stored on the path to the root from this leaf.
- Given a set of vertices $W = \{v_0, \ldots, v_n\}$ in a subgraph, the edges among them are obtained from the input graph $\mathcal{G}$. To avoid repetitive look-ups, edge information is cached in the embedding tree. Each time performing vertex extension by adding a vertex u, the edges between u and its predecessors in the embedding tree are determined and stored in the tree together with u. The connectivity (i.e. edges) of an embedding can be represented compactly using a bit-vector of length l for vertices at level l of the tree. We call this bit-vector the *connectivity code*. For example, if u is connected with the first and the third vertices in the embedding, but disconnected with the second, the code is '101'. This technique is called *Memoization of Embedding Connectivity* (MEC) [13].
- For a sub-pattern tree, embeddings of each sub-pattern are gathered as an embedding list (bin of embeddings). The search tree is constructed with sub-patterns as vertices, and each sub-pattern has an embedding list associated with it. Embedding connectivity is not needed as the sub-pattern contains this information.

For edge-induced extension, a set of edges instead of vertices is stored for each embedding. There is no need to store connectivity for embeddings since the set of edges are already recorded.

4.3 High Level Optimizations

Sandslash automatically performs high-level optimizations without guidance from the user. Table 2a (left) lists which of these optimizations are applied to each application. Table 2b (left) lists which of them are supported by other GPM systems.

	High-level					Low-level			
	SB	DAG	MO	DF	MNC	FP	CP	LG	LC
TC	✓	✓		✓					
k-CL	✓	✓	✓	✓	✓			✓	
SL	✓		✓	✓	✓			✓	
SC	✓		✓	✓	✓			✓	✓
k-MC	✓			✓	✓	✓	✓		✓
k-FSM	✓			✓					

(a) Optimizations applied to GPM applications.

	High-level					Low-level			
	SB	DAG	MO	DF	MNC	FP	CP	LG	LC
AutoMine [41]			✓						
Pangolin [13]	✓	✓	✓			✓	✓		
Peregrine [31]	✓		✓						
Sandslash-Hi	✓	✓	✓	✓	✓				
Sandslash-Lo	✓	✓	✓	✓	✓	✓	✓	✓	✓

(b) Optimizations supported by GPM frameworks.

Table 2: Optimizations enabled in Sandslash. *High level optimizations*: SB: Symmetry breaking; DF: Degree Filtering; DAG: orientation; MO: Matching Order; MNC: Memoization of Neighborhood Connectivity. *Low-level optimizations*: FP: Fine-grained Pruning; CP: Customized Pattern classification; LG: search on Local Graph; LC: Local Counting. ✓: supported.

Symmetry Breaking (SB), Orientation (DAG), and Matching Order (MO): Sandslash applies symmetry breaking for all GPM problems by default, i.e., it enumerates only canonical embeddings, unless the user specifies a custom automorphism check. Orientation (DAG) is enabled when it is a single explicit-pattern problem and if the pattern is a clique. Sandslash enables MO for explicit-pattern problems. We use a greedy approach to automatically generate a good matching order: at each step, (1) we choose a sub-pattern which has more internal partial orders for symmetry breaking, (2) if there is a tie, we choose a denser sub-pattern, i.e., one with more edges. In Fig. 3, (c) is the matching order chosen by the system since there is a partial order between vertex 0 and 1 for symmetry breaking (2.3). The intuition is that applying partial ordering as early as possible can better prune the search tree. Similarly, matching denser sub-pattern first can possibly prune more branches at early stage.

Degree Filtering (DF): When searching for a pattern in which the smallest vertex degree is d, it is unnecessary to consider vertices with degree less than d. When MO is enabled, at each level, only one vertex v of the pattern is searched for, so all vertices with degree less than that of v can also be filtered. This optimization (DF) has been used in a hand-optimized SL implementation, PSgL [49]. Sandslash enables DF for all GPM problems.

Memoizing of Neighborhood Connectivity (MNC): When extending an embedding $\mathcal{X} = \{v_0, \ldots v_n\}$ with a vertex u, a common operation is to check the connectivity between u and each vertex in $\mathcal{X}$. To avoid repeated lookups in the input graph, we memoize connectivity information in a *connectivity map* during embedding construction. The map takes a vertex ID v and returns the positions in the embedding of the vertices connected to v. In Fig. 6, v_3 is connected to v_0 and v_2, so when v_3 is looked up in the map, the map returns 0 and 2, the embedding positions of v_0 and v_2. Whenever a new vertex (w) is added to an embedding, the map for the neighbors of w that are not in the embedding are updated with the position of w in the embedding; when backing out of this step in the DFS walk, this information is removed.

Fig. 6 shows how the connectivity map is updated during vertex extension. At time ❶, depth of v_0 is sent to the map to update the entries of v_1, v_2 and v_3 since v_1, v_2 and v_3 are neighbors of v_0 and they are not in the current embedding. At time ❷, depth of v_2 is sent to the map, and the entry of v_3 is updated. Note that although v_0 is also a neighbor

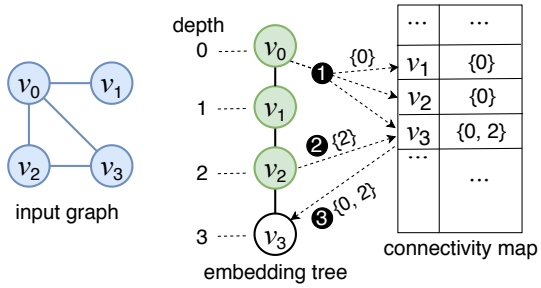


Figure 6: An example of connectivity memoization. ❶, ❷ and ❸ are timestamps to show the order of actions.

of v_2, there is no need to update the entry of v_0 since v_0 already exists in the current embedding. When v_3 is added to the embedding, the map performs look up with v_3, and the positions {0, 2} are returned at time ❸. Therefore, we know that v_3 is connected to the 0-th and 2-th vertices in the embedding, which are v_0 and v_2. For parallel execution, the map is thread private, and each entry is represented by a bit-vector.

MNC does not exist in any prior GPM systems, though it has been used in a hand-optimized k-CL implementation, kClist [17] and a hand-optimized k-MC solver, PGD [4]. Sandslash can enable MNC for any vertex-induced problem: in particular, Sandslash enables MNC for SL. MNC is missing in all hand-optimized SL implementations [8, 49]. Different from k-CL or k-MC, for an arbitrary pattern $\mathcal{P}$ we do not need to update the map for every vertex added into the embedding. For example, for `4-cycle`, the fourth vertex v_4 is a common neighbor of the second vertex v_2 and the third vertex v_3. Since v_4 is extended from v_3, we only need to check if v_4 is connected with v_2. Therefore, we only need to update the map for v_2's neighbors. Sandslash uses pattern analysis to detect in which level the vertex's neighbor connectivity are useful and sets the corresponding flag to notify the runtime update of the map.

Note that MNC is different from the vertex set buffering (VSB) technique used in Peregrine and AutoMine. To remove redundant computation, VSB buffers the vertex sets computed for a given embedding. However, for multi-pattern problems, different patterns may require buffering different vertex sets. Peregrine's solution is to match one pattern at a time, which is inefficient for a large number of patterns. AutoMine's solution is to only buffer one vertex set, which

leads to recomputation of unbuffered vertex sets. The other alternative is to buffer multiple vertex sets for a large pattern, but this does not scale well memory-wise. Unlike these solutions, MNC works well for multi-pattern problems since the information in the map can be used for both set intersection and set difference. More importantly, MNC fits naturally in Sandslash's vertex/edge extension model. This results in an augmented model which maintains its expressiveness and improves productivity.

5 Low-Level Sandslash

Hand-optimized GPM applications [4, 17, 30, 47] use algorithmic insight to prune the search tree. Table 2a (right) lists optimizations applicable to each application, and Table 2b (right) lists those that are supported by GPM systems. Sandslash's low-level API enables users to express such optimizations without implementing everything from scratch. The API allows Sandslash to perform all possible high-level optimizations which may be missing in hand-optimized applications (e.g., MNC is missing in hand-optimized SL). To use the low-level API, the user only needs to understand the subgraph tree abstraction and how to prune the tree. They do not need to understand Sandslash's implementation.

5.1 Local Counting (LC)

For GPM problems that count matched embeddings, there is no need to enumerate all matched embeddings if is possible to derive precise counts from counts of other patterns. Formally, the count of embeddings that match a pattern $\mathcal{P}$ may be calculated using the count of embeddings that match another pattern $\mathcal{P}'$. This is useful when both patterns are being searched for or when one pattern is more efficient to search for than the other. This typically requires a *local count* [30] (*micro-level count* [4]) of embeddings associated with a single vertex or edge instead of a *global count* (*macro-level count*) of embeddings that match the pattern.

Given a pattern $\mathcal{P}$ and a vertex v (or an edge e) $\in G$, let S be the set of all the embeddings of $\mathcal{P}$ in G. The *local count* of $\mathcal{P}$ on v (or e) is defined as the number of subgraphs in S that contains v (or e). Fig. 7 shows an example of local counting on edge e. Given an edge $e : u, v$, the local count of e for wedges $C_{wdg}e$ can be calculated from the local count of e for triangles $C_{tri}e$ using this formula:

$$C_{wdg} = deg_u - C_{tri} - 1 \;\; deg_v - C_{tri} - 1 \qquad (1)$$

deg_u and deg_v are the degrees of u and v.

Since wedge counts can be computed from triangle counts, enumerating wedges is avoided when using local counting for 3-MC. Similar formulas can be applied for k-MC. Local

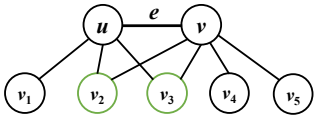


Figure 7: An example of local counting. Given the local triangle count of edge e is $C_{tri} = 2$, and $deg_u = 4$, $deg_v = 5$, we can get local wedge count of e as $C_{wdg} = 4 - 2 - 1 \;\; 5 - 2 - 1 = 3$.

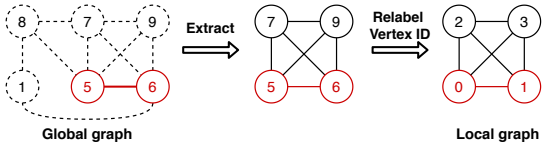


Figure 8: Local graph induced by edge v_5, v_6 and common neighbors of v_5 and v_6 from the global graph.

counting can also be used for subgraph counting (SC). For example, to count edge-induced `diamond`, we first compute the local triangle count n_t for each edge e and then use the formula $\binom{n_t}{2} = n_t \times n_t - 12$ to get the local `diamond` count. The global diamond count is obtained by accumulating local counts.

Sandslash exposes `localReduce` (Listing 1) to let the user specify how local counts are accumulated. It also exposes `toExtend` and `toAdd` to permit the user to customize the subgraph tree exploration so that the user can determine which patterns need to be enumerated. Listing 2 shows the user code for 3-MC using local counting. Local counting is activated when the user implements `localReduce`.

```
1  void localReduce(int depth, vector<Support> &sups){
2    if (depth == 0) { //local wedge count for each v
3      Vertex v = getHistory(depth);
4      int n = getDegree(v);
5      int pid = getWedgePid();
6      sups[pid] += n * (n-1) / 2;
7    }
8  }
9  Pattern p = generateTriangle();
10 Support tri_count = enumerate(p);
11 int pid = getTrianglePid();
12 sups[pid] = tri_count; // global triangle count
13 pid = getWedgePid();
14 sups[pid] -= 3 * tri_count; // global wedge count
```

Listing 2: Sandslash-Lo user code for 3-MC using local counting.

5.2 Search on Local Graph (LG)

For very dense patterns such as k-clique or k-clique-minus, one pruning scheme is to build a local graph for search instead of searching on the original input graph, i.e., the global graph [17]. Fig. 8 illustrates constructing a local graph induced by an edge (v_5, v_6) and common neighbors of v_5 and v_6. This graph is much smaller than the original graph because every vertex's neighborhood is limited to the common neighbors of v_5 and v_6.

This optimization leverages the property of a dense pattern. For example, when mining a 5-clique-minus (i.e. one edge less than a 5-clique), at level 2 we are extending an embedding $\{v_0, v_1, v_2\}$, and the candidate vertices of the forth vertex v_3 should be in the intersection set of v_0 and v_1. v_2's neighbors that are not in this intersection set do not need to be considered. This is also true for v_4, so v_3's neighbors that are not in this intersection set can also be removed from consideration. Based on this observation, it is safe to search a 5-clique-minus or a k-clique from the induced graph in Fig. 8 because any match starting from edge (v_5, v_6) is covered by the induced local graph. Note that the local graph is different from the vertex set buffering (VSB): the

Graph	Source	# V	# E	$\overline{d}$	# Labels
Pa	Patents [26]	3M	28M	10	37
Yo	Youtube [14]	7M	114M	16	29
Pdb	ProteinDB [55]	49M	388M	8	25
Lj	LiveJournal [38]	5M	86M	18	0
Or	Orkut [38]	3M	234M	76	0
Tw4	Twitter40 [36]	42M	2,405M	29	0
Fr	Friendster [62]	66M	3,612M	28	0
Uk	UK2007 [10]	106M	6,604M	31	0
Gsh	Gsh-2015 [11]	988M	51,381M	52	0

Table 3: Input graphs (symmetric, no loops, no duplicate edges, neighbor list sorted) and their properties ($\overline{d}$ is the average degree).

benefit of local graphs comes from shrinking neighbor lists in the embedding, not from memoizing intersection results.

Sandslash supports searching on either global or local graph, depending on the user's need. To enable local graphs, the user specifies how to initialize the local graph using `initLG()` and how to update it at the end of each DFS level using `updateLG()` (optional). When `initLG()` is defined, Sandslash enables LG to get the neighborhood information during extension using the local graph.

5.3 Fine-Grained Pruning (FP) and Customized Pattern Classification (CP)

FP and CP are existing low-level optimizations [13], so we only describe when and how to enable them.

Fine-Grained Pruning For explicit-pattern problems, Sandslash exposes `toExtend` and `toAdd` (Listing 1) to allow user-defined matching order and symmetry order. `toExtend` specifies the next vertex to extend in each level. Connectivity and partial orders are checked in `toAdd`. For example, in k-CL [17], since an i-clique can only be extended from an (i-1)-clique, `toExtend` and `toAdd` can be used to only extend the last vertex in the embedding and check if the newly added vertex is connected to all previous vertices in the embedding, respectively. Sandslash generates these functions automatically for explicit-pattern problems if not defined.

Customized Pattern Classification To recognize the pattern of a given embedding, a straightforward approach is the graph isomorphism test, which is expensive. If FP is not enabled, Sandslash uses matching order for explicit patterns to avoid isomorphism test. When FP is enabled, CP allows the user to replace isomorphism test with a custom method. CP is also useful for implicit pattern problems. For example, in FSM, the labeled `wedge` patterns can be differentiated by hashing the labels of the three vertices (the two endpoints of the wedge are symmetric). To enable CP, the user specifies a custom `getPattern`.

6 Evaluation

We present experimental setup in Section 6.1, compare Sandslash with state-of-the-art GPM systems and expert-optimized implementations in Section 6.2, analyze it in Section 6.3.

6.1 Experimental Setup

We evaluate two variants of Sandslash: Sandslash-Hi, which only enables high-level optimizations, and Sandslash-Lo, which enables both high-level and low-level optimizations. We compare Sandslash with the state-of-the-art GPM systems[4]: AutoMine [41], Pangolin [13], and Peregrine [31]. We use all applications listed in Section 2.1. We also compare with the expert-optimized GPM applications: GAPBS [6] for TC, kClist [17] for k-CL, PGD [4] for k-MC, and DistGraph [55] for FSM (CECI [8] for SL is not publicly available). For fair comparison, we modified DistGraph and PGD so that they produce the same output as Sandslash. We added a parameter k in DistGraph to stop exploration when the pattern size reaches k. For PGD, we disabled counting disconnected patterns.

Table 3 lists the input graphs. The first 3 graphs (`Pa`, `Yo`, `pdb`) are vertex-labeled graphs which can be used for FSM. We also include widely used large graphs (`Lj`, `Or`, `Tw4`, `Fr`, `Uk`), and a very large web-crawl [11] (`Gsh`). These graphs do not have labels and are only used for TC, k-CL, SL, k-MC.

Our experiments were conducted on a 4 socket machine with Intel Xeon Gold 5120 2.2GHz CPUs (56 cores in total) and 190GB RAM. All runs use 56 threads. For the largest graph, `Gsh`, we used a 2 socket machine with Intel Xeon Cascade Lake 2.2 Ghz CPUs (48 cores in total) and 6TB of Intel Optane PMM (byte-addressable memory technology).

Peregrine preprocesses the input graph to reorder vertices based on their degrees, which can improve the performance of GPM applications. In our evaluation, Sandslash does not reorder vertices to be fair to other systems and hand-optimized applications which do not perform such preprocessing. We use a time-out of 30 hours excluding graph loading and preprocessing time and report results as an average of three runs.

	Lj	Or	Tw4	Fr	Uk
Pangolin	0.4	2.3	75.5	55.1	45.8
AutoMine	1.1	6.4	9849.4	126.6	565.9
Peregrine	1.6	7.3	8492.4	100.3	3640.9
GAP	0.3	2.7	65.8	77.0	48.1
Sandslash-Hi	**0.3**	**1.8**	**57.2**	**44.9**	**24.5**

Table 4: Execution time (sec) of TC.

	4-CL					5-CL		
	Lj	Or	Tw4	Fr	Uk	Lj	Or	Fr
Pangolin	19.5	56.6	TO	564.1	TO	970.4	223.4	1704.4
AutoMine	11.0	32.9	67168.4	209.6	44666.6	575.6	170.1	389.0
Peregrine	15.9	73.7	TO	397.3	55808.4	520.8	782.1	957.6
kClist	1.2	2.5	1174.0	84.0	OOM	22.3	5.8	87.5
Sandslash-Hi	**0.6**	2.4	1676.8	166.2	2481.2	**13.9**	7.4	194.9
Sandslash-Lo	0.7	**1.9**	**681.8**	**60.4**	**2451.7**	14.2	**4.8**	**64.3**

Table 5: k-CL exec. time (sec) (OOM: out of memory; TO: timed out).

6.2 Comparisons with Existing Systems

Recall that Tables 2a and 2b list the optimizations applicable for each GPM application and enabled by each GPM system.

Triangle Counting (TC): Note that BFS and DFS are similar for enumerating triangles. As shown in Table 4, Sandslash achieves competitive performance with Pangolin and

[4]These GPM systems are orders of magnitude faster than previous GPM systems such as Arabesque [57], RStream [59], G-Miner [12], and Fractal [20].

	3-MC					4-MC	
	Lj	Or	Tw4	Fr	Uk	Lj	Or
Pangolin	10.8	96.5	TO	2460.1	23676.6	TO	TO
AutoMine	3.1	18.2	48901.7	352.8	4051.0	15529.7	90914.5
Peregrine	2.5	4.9	8447.4	165.3	3571.5	163.6	1701.4
PGD	11.2	42.5	OOM	OOM	OOM	192.8	4069.6
Sandslash-Hi	2.1	12.1	TO	723.7	4979.1	2366.2	30394.7
Sandslash-Lo	**0.3**	**1.6**	**304.6**	**43.8**	**386.8**	**16.7**	**232.4**

Table 6: k-MC exec. time (sec) (OOM: out of memory; TO: timed out).

	diamond				4-cycle		
	Lj	Or	Tw4	Fr	Lj	Or	Fr
Pangolin	92.3	884.5	TO	9301.6	553.5	13208.2	TO
Peregrine	5.4	10.2	**20898.4**	**178.1**	144.4	1867.2	32276.8
Sandslash-Hi	**1.5**	**4.2**	44659.5	284.2	**6.3**	**79.0**	**20490.9**

Table 7: Execution time (sec) of SL.

(sec)	Mi	Pa	Yo	Lj	Or	Fr	Tw2	Tw4
Peregrine	0.05	0.5	1.6	2.2	8.7	158.8	245.8	16312.6
Sandslash	**0.03**	**0.1**	**0.4**	**0.8**	**5.8**	**115.2**	**194.1**	**10187.4**

Table 8: Exec. time of subgraph counting (SC). Pattern: `diamond`.

expert-implemented GAP [6]. Both Pangolin and Sandslash outperform Peregrine and AutoMine because they use DAG which is more efficient than on-the-fly symmetry breaking. Sandslash is slightly faster than Pangolin because Pangolin use edge-parallel (i.e. each task is an edge, not a vertex) by default (vertex-parallel has better locality). On average, Sandslash outperforms AutoMine, Pangolin, Peregrine, and GAP by 10.1×, 1.4×, 13.8×, and 1.4×, respectively, for TC.

k-Clique Listing (k-CL): Table 5 presents k-CL results. Pangolin (BFS-only) performs poorly as memoizing of neighborhood connectivity (MNC) can only be enabled in DFS. Peregrine does on-the-fly symmetry breaking, but it does not construct and use DAG unlike Pangolin and Sandslash. Therefore, Peregrine performs similarly to Pangolin although it is a DFS based system. AutoMine is slower than Sandslash because it does not do symmetry breaking. We observe that Sandslash-Hi is already significantly faster than all the previous GPM systems. Moreover, Sandslash-Lo achieves better performance than even expert-implemented kClist [17] by enabling search on a local graph. There are some cases where Sandslash-Lo underperforms Sandslash-Hi. This is because searching on local graph requires computing/maintaining local graphs. When the search space is not reduced significantly, the overhead might outweigh the benefits (we explain this in detail in Section 6.3). On average, Sandslash-Lo outperforms AutoMine, Pangolin, Peregrine, and kClist by 21.0×, 35.1×, 31.1×, and 1.4×, respectively, for k-CL.

k-Motif Counting (k-MC): Table 6 compares k-MC performance. Sandslash-Hi outperforms AutoMine due to symmetry breaking, and Sandslash-Lo is orders of magnitude faster than Sandslash-Hi due to the local counting optimization. Pangolin is particularly inefficient for 4-MC as it cannot memoize neighborhood connectivity (MNC). Sandslash-Hi and Sandslash-Lo count all patterns simultaneously, whereas Peregrine does counting for each pattern/motif one by one; this allows it apply optimizations for each pattern. Unlike Sandslash, Peregrine reorders vertices during preprocessing. Peregrine is faster than Sandslash-Hi likely due to these reasons. Sandslash-Lo is faster than Peregrine due to the formula-based local counting optimization, which cannot be supported in the Peregrine API. All optimizations in expert-implemented PGD [4] are enabled in Sandslash-Lo. Sandslash-Lo outperforms PGD because PGD does not apply symmetry breaking and has much larger enumeration space. On average, Sandslash-Lo outperforms AutoMine, Pangolin, Peregrine, and PGD by 27.2×, 53.6×, 8.6× and 17.9×, respectively.

Subgraph Listing (SL): Table 7 presents SL results (AutoMine is omitted since it does vertex-induced, not edge-induced, SL). Sandslash outperforms all other systems, except Peregrine for the `diamond` pattern on `Lj` and `Fr`, which is likely because Peregrine reorders vertices during preprocessing. Pangolin is much slower than the other systems as it does not support memoization of neighborhood connectivity (MNC) optimization. The MNC approach in Sandslash is more efficient than the vertex set buffering (VSB) in Peregrine as explained in Section 4.3: Peregrine must do neighborhood intersections to determine connectivity while Sandslash does not. MNC is especially important for patterns like `4-cycle`, for which VSB has no benefit because there is no reusable vertex set to buffer. On average, Sandslash outperforms Pangolin and Peregrine by 29.5× and 5.6×, respectively.

Subgraph Counting (SC): Low-level Sandslash can support local counting in subgraph counting (SC) using formulas for specific patterns, e.g. `diamond`. Table 8 shows the SC performance on `diamond` compared to Peregrine. SC in Peregrine implements similar optimization in a hard-coded fashion as is does not provide any API for that. Because Sandslash allows high-level optimizations applied automatically while the user implements low-level optimizations, SC in Sandslash is faster than Peregrine due to MNC. On average, we observe 2.1× speedup over SC in Peregrine. Compared to the high-level Sandslash, the local counting optimization brings an average 2.0× speedup for `diamond`.

k-Frequent Subgraph Mining (k-FSM): Table 9 presents k-FSM results (AutoMine is omitted because it does not use domain support for FSM). Although Peregrine uses DFS exploration, it does global synchronization among threads for each DFS iteration in FSM which results in BFS-like exploration. In contrast, Sandslash uses DFS exploration on the sub-pattern tree and filters patterns without synchronization. Peregrine is the fastest for `Yo` due to better load balance and relatively small number of frequent patterns. We observe that for graphs with a large number of frequent patterns (`Pa`), Peregrine becomes very inefficient as its pattern-centric approach enumerates all the possible patterns first and then enumerates embeddings for each pattern one by one; this is detrimental to performance for larger graphs and patterns (e.g., it times out for `Pdb`). Sandslash is similar or faster than Pangolin in most cases, but is slower for `Pa` at σ=30K mainly because the BFS based approach has high parallelism for that case. For 4-FSM, Sandslash outperforms both Pangolin and Peregrine. It also performs better than expert-implemented DistGraph [55] as it enables all optimizations that are used in DistGraph, but with a better parallel implementation. Sandslash is the only system that can run 4-FSM on `Pdb`.

	3-FSM									4-FSM					
	Pa			Yo			Pdb			Pa			Pdb		
σ_{min}	500	1K	5K	500	1K	5K	500	1K	5K	10K	20K	30K	500	1K	5K
Pangolin	17.0	19.1	27.4	86.8	88.3	91.5	57.6	66.1	117.3	OOM	146.2	29.4	OOM	OOM	OOM
Peregrine	103.8	118.4	94.3	**52.8**	**69.9**	**60.8**	928.7	837.1	943.7	28301.0	4240.6	397.3	TO	TO	TO
DistGraph	13.1	13.0	14.1	OOM	OOM	OOM	253.9	278.8	239.8	120.7	**58.01**	**25.1**	OOM	OOM	OOM
Sandslash	**3.5**	**3.8**	**6.1**	81.0	80.8	82.8	**46.5**	**40.0**	**44.5**	**102.3**	108.4	43.7	**200.2**	**198.0**	**195.1**

Table 9: Execution time (sec) of k-FSM:- σ_{min}: minimum support (OOM: out of memory; TO: timed out).

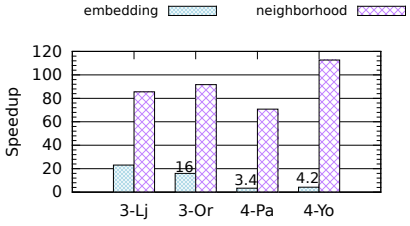


Figure 9: k-MC speedup with memoization of embedding/neighborhood connectivity.

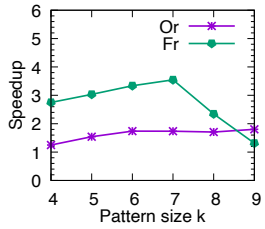


Figure 10: k-CL speedup by applying search on local graph.

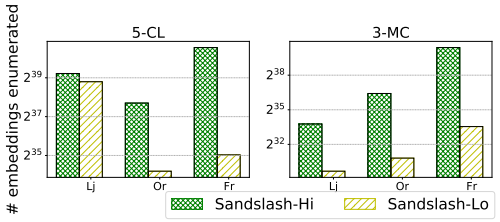


Figure 11: Comparing search space (# of enumerated embeddings) of high- and low-level Sandslash.

On average, Sandslash outperforms Pangolin, Peregrine, and DistGraph by 1.2×, 4.6× and 2.4×, respectively, for FSM.

6.3 Analysis of Sandslash

We present the impact of optimizations in Sandslash that are missing in other systems (Table 2b).

High-Level Optimizations: We observe 2% to 16% improvement for k-CL due to the degree filtering (DF) optimization. Fig. 9 shows speedup due to memoization of embedding connectivity (MEC) and memoization of neighborhood connectivity (MNC) optimizations for k-MC. For k-MC, the connectivity information in both the neighborhood and the embedding is memoized. MEC and MNC improve performance by 7.4× and 87× on average, respectively.

Low-Level Optimizations: Formula-based local counting (LC) reduces compute time by avoiding unnecessary enumeration of patterns. Table 6 shows Sandslash-Lo is 38× faster than Sandslash-Hi due to LC. As the pattern gets larger, pruning becomes more important. LC improves performance of 3-MC and 4-MC by 25× and 136× on average, respectively. This highlights the need to expose a low-level interface to express customized pruning strategies.

Fig. 10 illustrates the performance improvement on k-CL using the local graphs (LG) optimization on large patterns. Shrinking the local graph can reduce the search space compared to using the original graph. This improves performance by 1.2× to 3.5× for `Or` and `Fr`. The speedup for `Or` increases as the pattern size k increases. However, for `Fr`, the speedup peaks at k = 7, indicating that further shrinking becomes less effective as k grows. This trend depends on the input graph topology, but in general, this optimization is effective for supporting large patterns.

Both LC and LG optimizations prune the enumeration search space. We compare the search spaces of Sandslash-Hi and Sandslash-Lo to explain how they improve performance. Fig. 11 shows the number of enumerated embeddings for k-CL and k-MC. We observe a significant reduction for `Or` and `Fr` in Sandslash-Lo, explaining the performance differences between Sandslash-Hi and Sandslash-Lo in Tables 5 and 6. However, the pruning is less effective for `Lj` in k-CL, and given the overhead of local graph construction, Sandslash-Lo performs similar to Sandslash-Hi for `Lj` as shown in Table 5.

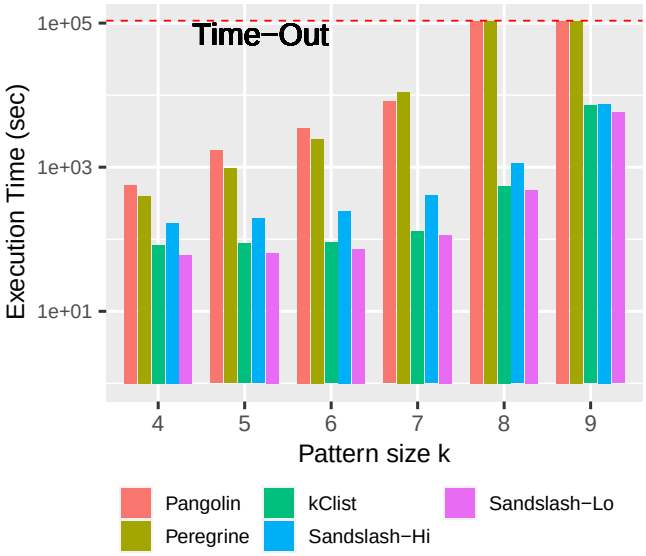
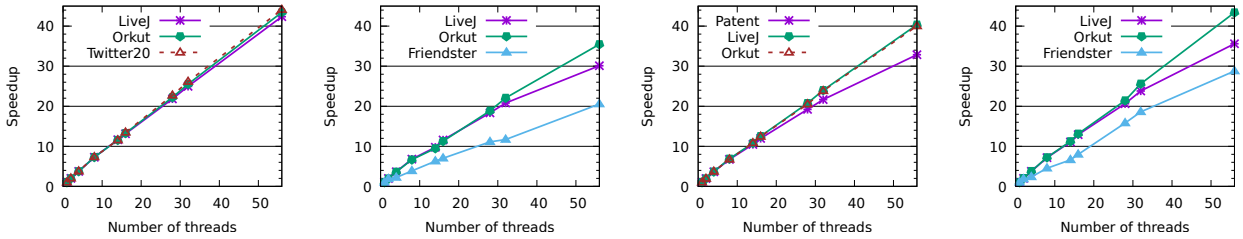


Figure 12: Execution time (sec in log scale) of k-CL on `Fr` graph.

Large Patterns. Fig. 12 shows k-CL on `Fr` graph with the pattern size k from 4 to 9. Pangolin and Peregrine timed out for k = 8 and k = 9. Existing systems cannot efficiently mine large patterns due to a much larger enumeration search space or significant amount of redundant computation. In contrast, Sandslash can effectively handle these large patterns, and in all cases Sandslash-Lo is faster than expert-implemented kClist. More importantly, the performance gap between Sandslash and prior systems becomes larger as k increases, indicating the importance of including all the high-level and low-level optimizations that are missing in prior systems.

Large Inputs. The large input graph, `Gsh`, requires 199GB in Compressed Sparse Row (CSR) format on disk, so we



(a) TC (b) 4-CL (c) SL (4-cycle) (d) 3-MC

Figure 13: Strong scaling of Sandslash.

evaluate it using 96 threads on the Optane machine. We were not able to run AutoMine and Peregrine on this large input. For 4-CL, Pangolin takes 6.5 hours, whereas Sandslash-Hi takes only 0.9 hours. Sandslash-Hi's memory usage is low as well: peak memory usage for Sandslash-Hi is 436 GB, while Pangolin, a BFS-based system, uses 3.5 TB memory. kClist and Sandslash-Lo run out of memory because maintaining the local graphs consumes more than 6 TB memory.

Strong Scaling. Fig. 13 shows the strong scaling of Sandslash applications. We observe the performance scales linearly for most of the applications as we increase threads. The average speedups of Sandslash on 56-thread over 1-thread are 43×, 28×, 39×, 35×, and 8× for TC, k-CL, SL, k-MC, and k-FSM, respectively (FSM not shown in the figure due to limited space). The speedup for k-FSM is lower than that for other applications due to constrained parallelism in traversing the sub-pattern tree in FSM. We also observe that Sandslash balances work well because the number of grains/vertices is large enough. Orthogonal techniques like fine-grained work-stealing in Fractal and vertex reordering in Peregrine can be added to Sandslash to further improve load balance.

7 Related Work

Low-level GPM Systems: Arabesque [57] is a distributed GPM system that uses an embedding-centric programming paradigm. RStream [59] is an out-of-core GPM system on a single machine, using a relational algebra based model. Kaleido [64] is a single-machine system that uses a compressed sparse embedding (CSE) format to reduce memory consumption. G-Miner [12] is a distributed GPM system which uses task-parallel processing. Pangolin [13] is a shared-memory GPM system targeting both CPU and GPU. Instead of the BFS exploration used in the above systems, Fractal [20] uses DFS to enumerate subgraphs on distributed platforms. Compared to these low-level systems, Sandslash improves productivity and performance with automated optimizations. Some of these GPM systems use distributed, out-of-core, or GPU platforms, which are orthogonal to our work.

High-level GPM Systems: AutoMine [41] is a DFS based system targeting a single-machine. It provides a high-level programming interface and employs a compiler to generate high performance GPM programs. GraphZero [40] improves AutoMine by introducing symmetry breaking to avoid over-counting. GraphPi [50] further improves GraphZero with a better performance model for redundancy elimination. Both GraphZero and GraphPi support only pattern matching, while Sandslash supports a wider range of GPM problems and also enhances performance without compromising productivity. Peregrine is the state-of-the-art high-level GPM system. It includes efficient matching strategies from well-established techniques [9, 25, 34] and improves performance compared to previous systems. Nevertheless, Sandslash with only its high-level API outperforms Peregrine. Furthermore, Sandslash provides a low-level API to trade-off programming effort for better performance.

GPM Algorithms: There are numerous hand-optimized GPM applications targeting various platforms. For TC, there are parallel solvers on multicore CPUs [19, 51, 60, 63], distributed CPUs [24, 46, 54], and GPUs [28, 29, 45]. kClist [17] is a parallel k-CL algorithm derived from [15]. It constructs DAG using a core value based ordering to reduce search space. PGD [4] counts 3 and 4-motifs by leveraging proven formulas to reduce enumeration space. Escape [47] extends this approach to 5-motifs. Subgraph listing [2, 8, 9, 32–35, 37, 39, 42, 48, 49, 52, 53, 58] is another important application in which a matching order is applied to reduce search space and avoid graph isomorphism tests. gSpan [61] is a sequential FSM algorithm using a lexicographic order for symmetry breaking. DistGraph [55, 56] parallelizes gSpan with a customized load balancer.We did holistic analysis on the optimizations introduced in these expert-written solvers and implemented them in Sandslash.

8 Conclusion

In this work, we revisit GPM system design tradeoffs on multicore CPU, based on a holistic investigation on optimizations in hand-tuned applications. We present Sandslash, a two-level GPM programming system targeting shared-memory CPUs. The Sandslash programming interface is split into two levels, which provides high productivity in the high-level and high performance in the low level, while retaining expressiveness. The user can easily compose GPM applications with the system support of automated optimizations and transparent parallelism. The system also gives the user flexibility to optionally express advanced optimizations to boost performance further. With two-level optimizations, Sandslash significantly outperforms existing systems and even hand-optimized implementations. This work demonstrates that a GPM programming system can provide both high productivity and high efficiency, without compromising expressiveness.

Acknowledgments

The research was supported by NSF grants 1406355, 1618425, 1705092, and 1725322, DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563, NSFC grant 61802416, and XSEDE grant ACI-1548562 through allocation TG-CIE-170005. We thank Intel for providing the Intel Optane DC PMM machine.

References

[1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. Scalemine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 61, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014986

[2] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (Oct. 2017), 44 pages. https://doi.org/10.1145/3129246

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

[4] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick Duffield. 2015. Efficient Graphlet Counting for Large Networks. In *ICDM*. 1–10.

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

[6] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). http://arxiv.org/abs/1508.03619

[7] Austin R. Benson, David F. Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (2016), 163–166. https://doi.org/10.1126/science.aad9029 arXiv:https://science.sciencemag.org/content/353/6295/163.full.pdf

[8] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 1447–1462. https://doi.org/10.1145/3299869.3300086

[9] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1199–1214. https://doi.org/10.1145/2882903.2915236

[10] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A Large Time-Aware Graph. *SIGIR Forum* 42, 2 (2008), 33–38.

[11] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[12] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: An Efficient Task-Oriented Graph Mining System. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 32, 12 pages. https://doi.org/10.1145/3190508.3190545

[13] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proc. VLDB Endow.* 13, 8 (Aug. 2020). https://doi.org/10.14778/3389133.3389137

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

[15] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14, 1 (Feb. 1985), 210–223. https://doi.org/10.1137/0214017

[16] Young-Rae Cho and Aidong Zhang. 2010. Predicting Protein Function by Frequent Functional Association Pattern Mining in Protein Interaction Networks. *Trans. Info. Tech. Biomed.* 14, 1 (Jan. 2010), 30–36. https://doi.org/10.1109/TITB.2009.2028234

[17] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing K-cliques in Sparse Real-World Graphs*. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 589–598. https://doi.org/10.1145/3178876.3186125

[18] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis. 2005. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering* 17, 8 (Aug 2005), 1036–1050. https://doi.org/10.1109/TKDE.2005.127

[19] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18)*. Association for Computing Machinery, New York, NY, USA, 393–404. https://doi.org/10.1145/3210377.3210414

[20] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 1357–1374. https://doi.org/10.1145/3299869.3319875

[21] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph. *Proc. VLDB Endow.* 7, 7 (March 2014), 517–528. https://doi.org/10.14778/2732286.2732289

[22] Katherine Faust. 2010. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Social Networks* 32, 3 (2010), 221 – 233. https://doi.org/10.1016/j.socnet.2010.03.004

[23] Brian Gallagher. 2006. Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*.

[24] I. Giechaskiel, G. Panagopoulos, and E. Yoneki. 2015. PDTL: Parallel and Distributed Triangle Listing for Massive Graphs. In *2015 44th International Conference on Parallel Processing*. 370–379. https://doi.org/10.1109/ICPP.2015.46

[25] Joshua A. Grochow and Manolis Kellis. 2007. Network Motif Discovery Using Subgraph Enumeration and Symmetry-Breaking. In *Proceedings of the 11th Annual International Conference on Research in Computational Molecular Biology* (Oakland, CA, USA) *(RECOMB'07)*. Springer-Verlag, Berlin, Heidelberg, 92–106.

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

[27] F. Harary. 1969. *Graph theory*. Addison-Wesley Pub. Co. https://books.google.com/books?id=QNxgQZQH868C

[28] Loc Hoang, Vishwesh Jatala, Xuhao Chen, Udit Agarwal, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2019. DistTC: High Performance Distributed Triangle Counting. In *HPEC 2019 23rd IEEE High Performance Extreme Computing, Graph Challenge*.

[29] Y. Hu, H. Liu, and H. H. Huang. 2018. TriCore: Parallel Triangle Counting on GPUs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 171–182. https://doi.org/10.1109/SC.2018.00017

[30] Shweta Jain and C. Seshadhri. 2020. The Power of Pivoting for Exact Clique Counting. In *Proceedings of the 13th International Conference on Web Search and Data Mining* (Houston, TX, USA) *(WSDM '20)*. Association for Computing Machinery, New York, NY, USA, 268–276. https://doi.org/10.1145/3336191.3371839

[31] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: A Pattern-Aware Graph Mining System. In *Proceedings of the Fifteenth EuroSys Conference (EuroSys '20)*.

[32] Madhav Jha, C. Seshadhri, and Ali Pinar. 2015. Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy) *(WWW '15)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 495–505. https://doi.org/10.1145/2736277.2741101

[33] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1695–1698. https://doi.org/10.1145/3035918.3056445

[34] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H.A. Jarrah. 2016.

DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 1231–1245. https://doi.org/10.1145/2882903.2915209

[35] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. ACM, New York, NY, USA, 411–426. https://doi.org/10.1145/3183713.3196917

[36] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[37] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable Subgraph Enumeration in MapReduce. *Proc. VLDB Endow.* 8, 10 (June 2015), 974–985. https://doi.org/10.14778/2794367.2794368

[38] J. Leskovec. 2013. SNAP: Stanford Network Analysis Platform. http://snap.stanford.edu/data/index.html

[39] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. 2012. Distributed Graph Pattern Matching. In *Proceedings of the 21st International Conference on World Wide Web* (Lyon, France) *(WWW '12)*. ACM, New York, NY, USA, 949–958. https://doi.org/10.1145/2187836.2187963

[40] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2019. GraphZero: Breaking Symmetry for Efficient Graph Mining. arXiv:1911.12877 [cs.PF]

[41] Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing High-level Abstraction and High Performance for Graph Mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. ACM, New York, NY, USA, 509–523. https://doi.org/10.1145/3341301.3359633

[42] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (July 2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[43] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. 2002. Network Motifs: Simple Building Blocks of Complex Networks. *Science* 298, 5594 (2002), 824–827. https://doi.org/10.1126/science.298.5594.824 arXiv:https://science.sciencemag.org/content/298/5594/824.full.pdf

[44] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[45] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. 2019. H-INDEX: Hash-Indexing for Parallel Triangle Counting on GPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7. https://doi.org/10.1109/HPEC.2019.8916492

[46] Roger Pearce, Trevor Steil, Benjamin W Priest, and Geoffrey Sanders. 2019. One Quadrillion Triangles Queried on One Million Processors. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–5.

[47] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. 2017. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1431–1440. https://doi.org/10.1145/3038912.3052597

[48] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. 2019. Fast and robust distributed subgraph enumeration. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1344–1356.

[49] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel Subgraph Listing in a Large-scale Graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. ACM, New York, NY, USA, 625–636. https://doi.org/10.1145/2588555.2588557

[50] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 100, 14 pages.

[51] J. Shun and K. Tangwongsan. 2015. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*. 149–160. https://doi.org/10.1109/ICDE.2015.7113280

[52] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. 2019. Efficient Parallel Subgraph Enumeration on a Single Machine. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 232–243.

[53] Shixuan Sun and Qiong Luo. 2019. Scaling Up Subgraph Query Processing with Efficient Subgraph Matching. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 220–231.

[54] Siddharth Suri and Sergei Vassilvitskii. 2011. Counting Triangles and the Curse of the Last Reducer. In *Proceedings of the 20th International Conference on World Wide Web* (Hyderabad, India) *(WWW '11)*. ACM, New York, NY, USA, 607–614. https://doi.org/10.1145/1963405.1963491

[55] N. Talukder and M. J. Zaki. 2016. A Distributed Approach for Graph Mining in Massive Networks. *Data Min. Knowl. Discov.* 30, 5 (Sept. 2016), 1024–1052. https://doi.org/10.1007/s10618-016-0466-x

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

[57] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. ACM, New York, NY, USA, 425–440. https://doi.org/10.1145/2815400.2815410

[58] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (Jan. 1976), 31–42. https://doi.org/10.1145/321921.321925

[59] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. 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* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 763–782. http://dl.acm.org/citation.cfm?id=3291168.3291225

[60] Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. 2017. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[61] Xifeng Yan and Jiawei Han. 2002. gSpan: graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*. 721–724. https://doi.org/10.1109/ICDM.2002.1184038

[62] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. *CoRR* abs/1205.6233 (2012). arXiv:1205.6233 http://arxiv.org/abs/1205.6233

[63] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, Jonathan Berry, and Ümit V Çatalyürek. 2018. Fast triangle counting using cilk. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[64] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng Guo. 2020. Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine. In *Proceedings of the 2020 IEEE International Conference on Data Engineering (ICDE 2020) (ICDE '20)*.