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 AuMine, 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.

CCS Concepts

• Software and its engineering → Application specific development environments.

Keywords

graph pattern mining, programming framework, search space pruning, performance optimization

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICS '21, June 14–17, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s).
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 [4, 15, 17, 20]. One example is motif counting [6, 23, 41], 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 [20].

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 [12, 29, 38, 39, 55, 57, 62], and they can be categorized into high-level and low-level systems. Both simplify GPM programming compared to handwritten code, but they make different tradeoffs.

High-level systems such as AutoMine [39] and Peregrine [29] 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 [57] and Pangolin [12] 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.
2 Background

2.1 Problem Definition

The following definitions are standard [25]. 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 $G_1 \cong G_2$, if there exists a bijection $f: V_1 \rightarrow V_2$ such that any two vertices $u$ and $v$ of $G_1$ are adjacent in $G_1$ if and only if $fu$ and $fv$ are adjacent in $G_2$. Two graphs are structurally identical. When $f$ is a mapping of a graph onto itself, we say $G_1$ 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 $G$ and a pattern $\mathcal{P}$, an embedding $X$ of $\mathcal{P}$ in $G$ is a vertex- or edge-induced subgraph of $G$ s.t. $X \cong \mathcal{P}$.

Given an undirected graph $G$ and a set of patterns $S_p = \{P_1, \ldots, P_n\}$, GPM finds the vertex- or edge-induced embeddings of $P_i$ 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 [21] finds embeddings only for explicit pattern(s), whereas graph pattern mining [2] 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 $G$. A particular statistic for $\mathcal{P}$ in $G$ 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 $G$.

- **Triangle Counting (TC):** The problem is to count the number of triangles in $G$. 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 $G$ isomorphic to a pattern $P$.
- **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 $k$ 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 $G$, an integer $k$ and a threshold $\sigma_{min}$ for support, $k$-FSM finds all frequent patterns with $k$ or fewer edges where a pattern is frequent if its support is greater than $\sigma_{min}$. If $k$ is not specified, it is set to infinity, 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). 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 $G$ 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 $l$ 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 $G$ 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 $S_1$.

The subgraph tree is a property of the input graph $G$. 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 $P$ as a pattern vertex and a vertex in $G$ 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 [29, 32]. 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 $G$ into a directed acyclic graph (DAG), and dynamic automorphism checks are then not needed [12, 16]. 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 [16].

Given a pattern $P$, one can use pattern analysis [39] to generate a matching order [29] and a symmetry order [38]. To generate a matching order, the pattern analyzer first enumerates all the possible matching orders of $P$ and uses a set of rules to pick one that is likely to perform well in practice [32]. To generate a symmetry order, we take one of the matching orders $MO$ of $P$ and build a subgraph $S$ of $P$ incrementally by adding one vertex at a time in the order specified by $MO$. At each step, if the newly added vertex $w$ is symmetric 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 [29, 39] 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 [42] 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.

#### 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,
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.

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

\texttt{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 [19].

Some algorithms [3] 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 [16] 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 by hand.

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.

\(^3\)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 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).

4.1 Search Strategies

Given $G$ and $P$ with $k$ vertices, one can build the subgraph tree (Section 2) to a depth $k$ and test each subgraph $X$ at the leaves of the tree to see if $X \simeq P$. 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 $G$ 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 sub-graph 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 [59]; i.e., the embeddings are gathered to their pattern bins during extension and canonicity 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 subgraphs.

- 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 Figure 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 $G$. 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) [12].
- 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.

**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 \( u \) 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 [47]. Sandslash enables DF for all GPM problems.

**Memoizing of Neighborhood Connectivity (MNC):** When extending an embedding \( X = \{v_0, \ldots, v_k\} \) with a vertex \( u \), a common operation is to check the connectivity between \( u \) and each vertex in \( X \). 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.

**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.

**Figure 6:** An example of connectivity memoization. ▼, ◆ and ◼ are timestamps to show the order of actions.

Fig. 6 shows how the connectivity map is updated during vertex extension. At time ▼, 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 ◼, 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 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 ◼. 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 [16] and a hand-optimized \( k \)-MC solver, PGD [3]. 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 [7, 47]. Different from \( k \)-CL or \( k \)-MC, for an arbitrary pattern \( P \) 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 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 [3, 16, 28, 45] 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 it is possible to derive precise counts from counts of other patterns. Formally, the count of embeddings that match a pattern \( P \) may be calculated using the count of embeddings that match another pattern \( P' \). 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 \textit{local count} [28] (\textit{micro-level count} [3]) of embeddings associated with a single vertex or edge instead of a \textit{global count} (\textit{macro-level count}) of embeddings that match the pattern.

Given a pattern \( P \) and a vertex \( v \) (or an edge \( e \)) \( \in G \), let \( S \) be the set of all the embeddings of \( P \) in \( G \). The \textit{local count} of \( P \) 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 \( C_{wedge} \) can be calculated from the local count of \( e \) for triangles \( C_{triangle} \) using this formula:

\[
C_{wedge} = deg_u - C_{triangle} - 1 \quad deg_v - C_{triangle} - 1
\]

\( deg_u \) 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 counting can also be used for subgraph counting (SC). For example, to count edge-induced \textit{diamonds}, we first compute the local triangle count \( n_t \) for each edge \( e \) and then use the formula \( \binom{n_t}{2} = n_t \times (n_t - 1) - 12 \) to get the local \textit{diamond} count. The global diamond count is obtained by accumulating local counts.

Sandslash exposes \texttt{localReduce} (Listing 1) to let the user specify how local counts are accumulated. It also exposes \texttt{toExtend} and \texttt{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 \texttt{localReduce}.

**Listing 1:** Sandslash-L0 user code for 3-MC using local counting.

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

**Listing 2:** Sandslash-L0 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 [16]. 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
Table 3: Input graphs (symmetric, no loops, no duplicate edges, neighbor list sorted) and their properties ($\bar{d}$ is the average degree).

<table>
<thead>
<tr>
<th>Graph</th>
<th>Source</th>
<th># V</th>
<th># E</th>
<th>$\bar{d}$</th>
<th># Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pa</td>
<td>Patents</td>
<td>3M</td>
<td>28M</td>
<td>10</td>
<td>37</td>
</tr>
<tr>
<td>Yo</td>
<td>Youtube</td>
<td>1M</td>
<td>11M</td>
<td>18</td>
<td>29</td>
</tr>
<tr>
<td>Pdb</td>
<td>ProteinDB</td>
<td>49M</td>
<td>388M</td>
<td>8</td>
<td>25</td>
</tr>
<tr>
<td>Lj</td>
<td>LiveJournal</td>
<td>5M</td>
<td>86M</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>Or</td>
<td>Orkut</td>
<td>3M</td>
<td>234M</td>
<td>76</td>
<td>0</td>
</tr>
<tr>
<td>Tw4</td>
<td>Twitter40</td>
<td>42M</td>
<td>2.405M</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>Fr</td>
<td>Friendster</td>
<td>66M</td>
<td>3.612M</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>Uk</td>
<td>UK2007</td>
<td>100M</td>
<td>6.604M</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>Gsh</td>
<td>Gsh-2015</td>
<td>988M</td>
<td>51.381M</td>
<td>52</td>
<td>0</td>
</tr>
</tbody>
</table>

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 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 [12], 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 [16], 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 [39], Pangolin [12], and Peregrine [29]. We use all applications listed in Section 2.1. We also compare with the expert-optimized GPM applications: GAPBS [5] for TC, kClist [16] for k-CL, PGD [3] for k-MC, and DistGraph [53] for FSM (CECI [7] 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 (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.

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 [55], RStream [57], G-Miner [11], and Fractal [19].
Table 5: k-CL exec. time (sec) (OOM: out of memory; TO: timed out).

<table>
<thead>
<tr>
<th></th>
<th>Lj</th>
<th>Or</th>
<th>Tw4</th>
<th>Fr</th>
<th>Uk</th>
<th>Lj</th>
<th>Or</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pangolin</td>
<td>19.5</td>
<td>56.6</td>
<td>TO</td>
<td>564.1</td>
<td>TO</td>
<td>970.4</td>
<td>223.4</td>
</tr>
<tr>
<td>AutoMine</td>
<td>11.0</td>
<td>32.9</td>
<td>67168.4</td>
<td>209.6</td>
<td>44659.5</td>
<td>575.6</td>
<td>170.1</td>
</tr>
<tr>
<td>Peregrine</td>
<td>5.9</td>
<td>74.7</td>
<td>TO</td>
<td>397.4</td>
<td>55808.4</td>
<td>520.8</td>
<td>782.1</td>
</tr>
<tr>
<td>kClist</td>
<td>1.2</td>
<td>2.5</td>
<td>1174.0</td>
<td>84.0</td>
<td>OOM</td>
<td>22.3</td>
<td>5.8</td>
</tr>
<tr>
<td>Sandslash-Hi</td>
<td>0.6</td>
<td>2.4</td>
<td>1676.8</td>
<td>166.2</td>
<td>2481.2</td>
<td>13.9</td>
<td>7.4</td>
</tr>
<tr>
<td>Sandslash-Lo</td>
<td>0.7</td>
<td>1.9</td>
<td>681.8</td>
<td>60.4</td>
<td>2451.7</td>
<td>4.2</td>
<td>4.8</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Lj</th>
<th>Or</th>
<th>Tw4</th>
<th>Fr</th>
<th>diamond</th>
<th>Lj</th>
<th>Or</th>
<th>Fr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pangolin</td>
<td>10.8</td>
<td>96.5</td>
<td>TO</td>
<td>2460.1</td>
<td>23676.6</td>
<td>TO</td>
<td>TO</td>
<td></td>
</tr>
<tr>
<td>AutoMine</td>
<td>3.1</td>
<td>18.2</td>
<td>48901.7</td>
<td>352.8</td>
<td>4051.0</td>
<td>15529.7</td>
<td>90914.5</td>
<td></td>
</tr>
<tr>
<td>Peregrine</td>
<td>2.5</td>
<td>4.9</td>
<td>8414.7</td>
<td>165.3</td>
<td>3571.5</td>
<td>163.6</td>
<td>1701.4</td>
<td></td>
</tr>
<tr>
<td>PGD</td>
<td>11.2</td>
<td>42.5</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>192.8</td>
<td>4069.6</td>
<td></td>
</tr>
<tr>
<td>Sandslash-Hi</td>
<td>2.1</td>
<td>12.1</td>
<td>TO</td>
<td>723.7</td>
<td>4979.1</td>
<td>2366.2</td>
<td>30994.7</td>
<td></td>
</tr>
<tr>
<td>Sandslash-Lo</td>
<td>0.3</td>
<td>1.6</td>
<td>304.6</td>
<td>43.8</td>
<td>386.8</td>
<td>16.7</td>
<td>232.4</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Execution time (sec) of SL.

<table>
<thead>
<tr>
<th></th>
<th>Lj</th>
<th>Or</th>
<th>Tw4</th>
<th>Fr</th>
<th>diamond</th>
<th>Lj</th>
<th>Or</th>
<th>Fr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pangolin</td>
<td>92.3</td>
<td>884.1</td>
<td>TO</td>
<td>9308.1</td>
<td>553.5</td>
<td>13208.2</td>
<td>TO</td>
<td></td>
</tr>
<tr>
<td>Peregrine</td>
<td>5.4</td>
<td>10.2</td>
<td>20884.4</td>
<td>178.1</td>
<td>144.4</td>
<td>1867.2</td>
<td>32276.8</td>
<td></td>
</tr>
<tr>
<td>Sandslash-Hi</td>
<td>1.5</td>
<td>4.2</td>
<td>44659.5</td>
<td>284.2</td>
<td>6.3</td>
<td>79.0</td>
<td>20490.9</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Lj</th>
<th>Or</th>
<th>Tw4</th>
<th>Fr</th>
<th>diamond</th>
<th>Lj</th>
<th>Or</th>
<th>Fr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peregrine</td>
<td>0.05</td>
<td>0.5</td>
<td>1.6</td>
<td>2.2</td>
<td>8.7</td>
<td>158.8</td>
<td>245.8</td>
<td>16312.6</td>
</tr>
<tr>
<td>Sandslash</td>
<td>0.03</td>
<td>0.1</td>
<td>0.4</td>
<td>0.8</td>
<td>5.8</td>
<td>115.2</td>
<td>194.1</td>
<td>10187.4</td>
</tr>
</tbody>
</table>

Table 9: Execution time (sec) of SL.

<table>
<thead>
<tr>
<th></th>
<th>Lj</th>
<th>Or</th>
<th>Tw4</th>
<th>Fr</th>
<th>diamond</th>
<th>Lj</th>
<th>Or</th>
<th>Fr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pangolin</td>
<td>92.3</td>
<td>884.1</td>
<td>TO</td>
<td>9308.1</td>
<td>553.5</td>
<td>13208.2</td>
<td>TO</td>
<td></td>
</tr>
<tr>
<td>Peregrine</td>
<td>5.4</td>
<td>10.2</td>
<td>20884.4</td>
<td>178.1</td>
<td>144.4</td>
<td>1867.2</td>
<td>32276.8</td>
<td></td>
</tr>
<tr>
<td>Sandslash-Hi</td>
<td>1.5</td>
<td>4.2</td>
<td>44659.5</td>
<td>284.2</td>
<td>6.3</td>
<td>79.0</td>
<td>20490.9</td>
<td></td>
</tr>
</tbody>
</table>

Table 10: Execution time of subgraph counting (SC). Pattern: diamond.

<table>
<thead>
<tr>
<th></th>
<th>Lj</th>
<th>Or</th>
<th>Tw4</th>
<th>Fr</th>
<th>diamond</th>
<th>Lj</th>
<th>Or</th>
<th>Fr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peregrine</td>
<td>0.05</td>
<td>0.5</td>
<td>1.6</td>
<td>2.2</td>
<td>8.7</td>
<td>158.8</td>
<td>245.8</td>
<td>16312.6</td>
</tr>
<tr>
<td>Sandslash</td>
<td>0.03</td>
<td>0.1</td>
<td>0.4</td>
<td>0.8</td>
<td>5.8</td>
<td>115.2</td>
<td>194.1</td>
<td>10187.4</td>
</tr>
</tbody>
</table>

$k$-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 to 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 [3] 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 $27.2 \times, 53.6 \times, 8.6 \times$ and $17.9 \times$, 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 $29.5 \times$ and $5.6 \times$, 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 $2.1 \times$ speedup over SC in Peregrine. Compared to the high-level Sandslash, the local counting optimization brings an average $2.0 \times$ 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
We present the impact of optimizations in Sandslash that are customized pruning strategies. Figure 9 highlights the need to expose a low-level interface to express connectivity information in both the neighborhood and the 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 8.7× on average, respectively. This 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 $\sigma=30K$ 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 [53] 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. 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 8.7× 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 Fr and Or. The speedup for Fr increases as the pattern size $k$ increases. However, for Or, 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 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.

**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, \( G_{ah} \), requires 199GB in Compressed Sparse Row (CSR) format on disk, so we 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 [55] is a distributed GPM system that uses an embedding-centric programming paradigm. RStream [57] is an out-of-core GPM system on a single machine, using a relational algebra based model. Kaleido [62] is a single-machine system that uses a compressed sparse embedding (CSE) format to reduce memory consumption. G-Miner [11] is a distributed GPM system which uses task-parallel processing. Pangolin [12] is a shared-memory GPM system targeting both CPU and GPU. Instead of the BFS exploration used in the above systems, Fractal [19] 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 [39] 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 [38] improves AutoMine by introducing symmetry breaking to avoid over-counting. GraphPi [48] 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 [8, 23, 32] 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 [18, 49, 58, 61], distributed CPUs [22, 44, 52], and GPUs [26, 27, 43]. kClist [16] is a parallel k-CL algorithm derived from [14]. It constructs DAG using a core value based ordering to reduce search space. PGD [3] counts 3 and 4-motifs by leveraging proven formulas to reduce enumeration space. Escape [45] extends this approach to 5-motifs. Subgraph listing [1, 7, 8, 30–33, 35, 37, 40, 46, 47, 50, 51, 56] is another important application in which a matching order is applied to reduce search space and avoid graph isomorphism tests. gSpan [59] is a sequential FSM algorithm using a lexicographic order for symmetry breaking. DistGraph [53, 54] 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,
Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining

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

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. 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, 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
