The VLDB Journal (2022) 31:253–286
https://doi.org/10.1007/s00778-021-00687-0

SPECIAL ISSUE PAPER

PrefixFPM: a parallel framework for general-purpose mining of frequent and closed patterns

Da Yan[1] · Wenwen Qu[2] · Guimu Guo[1] · Xiaoling Wang[2] · Yang Zhou[3]

Received: 30 August 2020 / Revised: 27 April 2021 / Accepted: 10 July 2021 / Published online: 9 August 2021

Abstract
A frequent pattern is a substructure that appears in a database with frequency (aka. support) no less than a user-specified threshold, while a closed pattern is one that has no super-pattern that has the same support. Here, a substructure can refer to different structural forms, such as itemsets, subsequences, subtrees, and subgraphs, and mining such substructures is important in many real applications such as product recommendation and feature extraction. Currently, there lacks a general programming framework that can be easily customized to mine different types of patterns, and existing parallel and distributed solutions are IO-bound rendering CPU cores underutilized. Since mining frequent and/or closed patterns are NP-hard, it is important to fully utilize the available CPU cores. This paper presents such a general-purpose framework called *PrefixFPM*. The framework is based on the idea of prefix projection which allows a divide-and-conquer mining paradigm. PrefixFPM exposes a unified programming interface to users who can readily customize it to mine their desired patterns. We have adapted the state-of-the-art serial algorithms for mining patterns including subsequences, subtrees, and subgraphs on top of PrefixFPM, and extensive experiments demonstrate an excellent speedup ratio of PrefixFPM with the number of CPU cores.

Keywords Frequent pattern · Closed pattern · Parallel

1 Introduction

Frequent patterns are substructures that appear in a database with frequency (called support) no less than a user-specified threshold. For example, a subsequence, such as buying first a laptop, then a mouse, and then a mouse pad, if it occurs frequently in a database of user shopping histories, is a frequent sequential pattern.

The previous pattern $\alpha = \langle$laptop, mouse, mouse pad$\rangle$, is checked against a database D of the so-called transactions, which are the individual elements of D. In our context, a transaction is a sequence of purchasing history of one user, such as $s = \langle$laptop, memory card, mouse, and mouse pad$\rangle$ which is a super-sequence of α and thus supports α (or, contributes 1 to α's support).

A closed pattern is one that has no super-pattern that has the same support. Referring back to the previous sequential pattern α again, if we have a subsequence $\beta = \langle$laptop, mouse pad$\rangle$ that has the same support as α, then β is not closed. There are also other variants of frequent patterns such as maximal patterns and high-utility patterns [14], which fall in the algorithmic category that we study. Without loss of generality, we simply use the term frequent pattern mining (FPM) to mean this category of problems.

FPM has been at the core of data mining research for over two decades [1], where numerous serial algorithms have been proposed for mining various types of substructures. The mined frequent substructures have also been widely used in

Da Yan and Wenwen Qu are parallel first authors.

✉ Da Yan
yanda@uab.edu

✉ Wenwen Qu
wenwenqu@stu.ecnu.edu.cn

Guimu Guo
guimuguo@uab.edu

Xiaoling Wang
xlwang@cs.ecnu.edu.cn

Yang Zhou
yangzhou@auburn.edu

[1] Department of Computer Science, The University of Alabama at Birmingham, Birmingham, AL, USA

[2] Shanghai Key Laboratory of Trustworthy Computing, East China Normal University (ECNU), Shanghai, China

[3] Department of Computer Science and Software Engineering, Auburn University, Auburn, AL, USA

many real applications. For example, FG-index [5] constructs a nested inverted index based on the set of frequent subgraphs, to speed up the finding of those graphs in a graph database that contains a query subgraph, while [21] uses frequent subgraphs as features for classifying labeled graphs modeling real-world data such as chemical compounds.

With the popularity of Big Data and Hadoop, and the need of FPM over Big Data, many parallel and distributed solutions to FPM emerge such as those based on MapReduce [2,24,32] and other dedicated ones [38,45]. However, as we shall explain in Sect. 2, all these works adopt an Apriori approach where frequent patterns of size $(i + 1)$ are generated from all frequent patterns of size i, leading to an iterative algorithm where Iteration i mines all frequent patterns of size i.

This simple approach has a catastrophic performance impact when implemented in a distributed environment. For example, data instances that contain size-i patterns need to be transmitted across the network to various machines for growing size-$(i + 1)$ patterns, and to get their frequency, another round of communication is needed for pattern frequency aggregation. This basically associates a data movement with each computing operation (e.g., growing a pattern or adding a frequency counter), but the former is the performance bottleneck rendering CPU cores underutilized. The performance is further exacerbated by the fact that the frequent patterns found in each iteration often need to be dumped to Hadoop Distributed File System (which replicates data) and then loaded back in Iteration $(i + 1)$.

As our experiments in Sect. 8.8 will show, Spark MLlib can be 5× to 60× slower than our proposed single-machine solution when running with one computing thread to mine frequent sequential patterns, while Sect. 8.9 will show that an existing MapReduce algorithm for mining frequent subgraph patterns [2] running with 10 machines is found to be 56× slower than even the serial program of gSpan [49]. Similar observation holds for non-distributed parallel solutions such as RStream [45], as we shall see in Sect. 8.5.

We propose a novel parallel framework called *PrefixFPM* for frequent pattern mining (FPM) that is able to fully utilize the CPU cores in a multicore machine. Instead of checking patterns in breadth-first order using Apriori algorithms, PrefixFPM adopts the prefix projection approach pioneered by PrefixSpan [31] and followed by many later works on FPM.

As we shall explain in Sect. 3, prefix projection partitions the workloads of pattern checking by divide and conquer, which naturally fits in a task-based parallel execution model; also, depth-first pattern-growth order allows a small memory footprint as one does not have to keep all size-i patterns for pattern growth, and it allows a pattern β that is grown from α to only examine the subset of data that contain α (called α's projected database). PrefixFPM also features a user-friendly programming model where users can customize it (by implementing some user-defined functions) to mine different kinds of patterns by adapting existing serial algorithms. The parallel execution details are handled by PrefixFPM itself and are transparent to users.

We remark that PrefixFPM currently focuses on mining frequent patterns against a database with many small to moderate-sized transactions, where pattern frequency is a natural criterion satisfying the famous antimonotonicity property. There also exists another problem setting in the literature where one mines frequent itemsets (aka. colocation patterns) from a large set of spatial objects [4], or mines frequent subgraphs in a big graph [10]. However, frequency alone does not satisfy antimonotonicity in such single-transaction scenario, and more advanced support measures are needed such as minimum image [3] for a big graph, and fraction score for colocation pattern mining [4]. As a result, different algorithms are needed with more complicated support evaluation. We remark that our T-thinker paradigm (foundation of *PrefixFPM*, see Sect. 2.1) still applies to this setting, such as the recursive procedure *SubgraphExtension* of GRAMI [10], but our G-thinker [18,46] framework for handling a single big graph is a more suitable platform for such parallelization than PrefixFPM which targets a database of transactions. We will explore this direction on top of G-thinker as a future work.

The main contributions of this work are summarized as follows.

- As far as we know, PrefixFPM is the first programming framework that unifies the mining of different types of frequent patterns; the resulted program is additionally efficient for parallel execution.
- We designed a general programming interface which can be easily customized by users for their desired pattern types; the interface also allows different FPM problems to share the same backend for parallel execution.
- PrefixFPM adopts prefix projection to enjoy the benefit of examining shrinking projected databases, and the divide-and-conquer nature of prefix projection also enables a task-based engine to concurrently examine different patterns to fully utilize CPU cores.
- We have provided the parallel implementations of 7 state-of-the-art serial pattern mining algorithms including PrefixSpan [31], CloSpan [48], Sleuth [56], TreeMiner [53], PrefixTreeSpan [59], gSpan [49], and Gaston [27] on top of PrefixFPM, which demonstrate an excellent speedup with the number of CPU cores.

This paper is a journal extension of our ICDE short paper [50] where we originally only briefly discussed PrefixFPM, and some results on PrefixFPM parallelization of PrefixSpan [31], gSpan [49], and Sleuth [56]. This journal extension now provides the complete PrefixFPM parallel algorithms

for all the 7 FPM algorithms listed above, including 4 new algorithms so that a richer set of pattern types are covered including closed patterns (i.e., CloSpan [48]), ordered (i.e., TreeMiner [53] and PrefixTreeSpan [59]) and unordered tree patterns (i.e., Sleuth [56]), and a mixture of path, free tree, and graph patterns (i.e., Gaston [27]).

This journal extension (i) presents implementations of the user-defined functions in PrefixFPM for each application with sufficient details; (ii) adds a new timeout mechanism for task decomposition in the system design for better load balancing compared with our ICDE version; and (iii) reports more extensive experiments using both real and synthetics datasets, including pros and cons of parallelizing different algorithms for the same problem (e.g., TreeMiner v.s. PrefixTreeSpan).

We have also open-sourced all our system and application codes on GitHub[1] to be readily used by other researchers and data mining practitioners.

The rest of this paper is organized as follows. Section 2 reviews existing Big Data solutions to various FPM problems and explains why their execution is IO-bound. Section 3 overviews the idea of prefix projection and how PrefixFPM utilizes it for task parallelism. Section 4 introduces our programming model for unifying FPM problems. Then, Sects. 5, 6, and 7 present our PrefixFPM algorithms to mine subsequence patterns, subtree patterns, and subgraph patterns, respectively. Finally, Sect. 8 reports the results of our extensive experimental studies and Sect. 9 concludes this paper.

2 Related work

In this section, we first review the IO-bound performance bottleneck issue that existing works have, as well as our T-thinker paradigm to address this issue for divide-and-conquer algorithms. We then review the existing works on parallel and distributed FPM and motivate the need of PrefixFPM.

2.1 IO bottleneck issue and the T-thinker paradigm

Researchers have begun to realize the issue that IO-bound Big Data frameworks can be ill-suited for compute-heavy problems like our FPM. In [8], McSherry indicates that "*the current excitement about distributed computation (e.g., Hadoop, Spark) produced implementations that improve when you give them more resources (they 'scale') but whose performance never quite gets to where you would be with a simple single-threaded implementation on a laptop*." Basically, the frameworks aggregate the IO bandwidth of disks and/or network (NICs) of many machines in a shared-nothing environment, but the aggregate throughput is still not beyond that of a single CPU core. Specific to graph processing, [25] found that even for IO-intensive problems such as computing PageRanks and connected components where the time complexity is low, the performance of existing graph-parallel systems is comparable and sometimes even slower than simple, single-threaded implementations using a high-end 2014 laptop.

This is of no exception in the context of FPM. For example, Arabesque [38] was proposed in SOSP 2015 as a distributed system that can handle frequent subgraph mining. Later in OSDI 2018, RStream [45] was developed following Arabesque's programming model, but it utilizes relational joins to run out-of-core on a single machine. Interestingly, RStream is found to even beat Arabesque in performance, which means that the mining throughput of Arabesque is even lower than the IO bandwidth on a single machine. Since FPM is NP-hard [51] and thus highly compute-intensive, it is critical to develop novel frameworks that scale with the number of CPU cores.

Recently, T-thinker [47] is proposed as a system paradigm that targets divide-and-conquer algorithms and adopts a task-based parallel execution design to achieve compute-intensive workloads. T-thinker effectively utilizes the CPU cores in a cluster by properly dividing a problem over a big dataset into tasks over smaller subsets of the dataset for load-balanced parallel computation. Two compute-intensive systems have been developed under the T-thinker paradigm, including *G-thinker* [18,46] for graph mining, and *PrefixFPM* [50] for general-purpose frequent pattern mining, the conference paper of which is extended to this journal paper.

2.2 Related work on parallel and distributed FPM

Since the advent of Hadoop and MapReduce, there have been many efforts to scale FPM problems using Big Data frameworks. In fact, dozens of papers have emerged for mining frequent itemsets and subgraphs using MapReduce, most of which just aim to make FPM work with MapReduce without caring much about the actual mining throughput, and adopt the straightforward approach of letting the i-th MapReduce job mine size-i patterns from those size-$(i-1)$ frequent patterns dumped to Hadoop Distributed File System (HDFS) in the previous MapReduce job, leading to excessive IO overheads.

Such breadth-first pattern generation and examination approach is inefficient even when a dedicated design is adopted other than using Hadoop MapReduce. For example, Arabesque [38] is a distributed system that supports frequent subgraph mining. Arabesque lets every machine load the entire input graph into memory and constructs subgraph instances of increasing size iteratively. In the i-th iteration, Arabesque expands the set of subgraphs with i edges (or vertices) by one more adjacent edge (or vertex), to construct

[1] https://github.com/wenwenQu/PrefixFPM

subgraphs with $(i+1)$ edges (or vertices) for processing. New subgraphs that pass a filtering condition (e.g., frequency check) are further processed and then passed to the next iteration. Obviously, Arabesque materializes all subgraph data instances that match frequent subgraph patterns, so it is IO-bound. As an in-memory system, Arabesque attempts to compress the numerous materialized subgraphs using a data structure called ODAG (Overapproximating Directed Acyclic Graph), but it does not fundamentally address the scalability limitation as the number of subgraphs grows exponentially.

RStream [45] is a single-machine system which proposes a so-called GRAS model to emulate Arabesque's filter-process model by utilizing relational joins. Their experiments show that RStream is several times faster than Arabesque even though it uses just one machine, but the improvement is mainly because of eliminating network overheads. Also, the execution of RStream is still IO-bound as it is an out-of-core system.

Realizing the drawbacks of breadth-first pattern examination approaches, a number of seminal works explored the use of depth-first pattern examination to reduce the IO overhead of examining data instances (by using projected databases), and the use of search space pre-partitioning to reduce the number of MapReduce job iterations which are found to be a major overhead due to the expensive shuffling stage.

For example, PFP [23] parallelizes the depth-first FP-Growth algorithm for frequent itemset mining over MapReduce. PFP partitions the computation in such a way that each machine executes an independent group of mining tasks. Such partitioning eliminates computational dependencies between machines, and thereby, communication between machines. However, there lacks such an efficient solution to other patterns like subsequences, subgraphs, and subtrees. The extension to these problems is non-trivial since the items in an itemset have no order, so the FP-Growth algorithm can define a total order over the items to build an FP-tree for compressing data and eliminating redundancy, but item order matters in sequences, graphs, and trees.

In the context of frequent subgraph mining (FSM), [24] proposes a two-step filter-and-refinement approach using MapReduce. The first step partitions the collection of graphs among worker nodes, so that each worker W_i obtains a local portion of graphs $\mathcal{G}_i$; this allows W_i to mine locally frequent subgraphs over $\mathcal{G}_i$, which we denote as $\mathcal{F}_i$. The union $\cup_i \mathcal{F}_i$ constitutes the input to the second step for refinement, since if a subgraph pattern is not frequent in every $\mathcal{G}_i$, it cannot be frequent in the entire graph database. This approach effectively decomposes an FSM problem over a big graph database into those over the smaller graph collections $\mathcal{G}_i$, but since it is based on MapReduce, there are still three weaknesses: (1) During the first step, all machines are mining local graph collections and the communication bandwidth is wasted; (2) the candidates $\cup_i \mathcal{F}_i$ are not tight and need to be dumped to Hadoop Distributed File System (HDFS) for use by the second step, which incurs expensive IO overheads; and (3) in the refinement step, each candidate is checked against every partition $\mathcal{F}_i$ to retain the globally frequent graphs, which incurs additional overheads and complication in algorithm design to reduce such overheads, such as the statistical threshold mechanism and lightweight graph compression mechanism designed by [24].

So far, we mainly discussed works in a distributed setting. Many efforts have also been made to mine frequent itemsets in a multicore environment, by parallelizing algorithms such as FP-growth [19] and Eclat [54] which generate frequent $(i+1)$-itemsets by intersecting the transactions of frequent i-itemsets. FP-Array [22], based on FP-Growth, utilizes a cache-conscious FP-Array built from a compact FP-Tree and a lock-free tree construction algorithm. MC-Eclat [35] is a parallel method based on Eclat. ShaFEM [43] is a parallel method that dynamically switches between FP-Growth and Eclat based on dataset characteristics. However, these works do not extend to more advanced patterns such as sequences, trees, and graphs.

Similarly, a number of works have explored the use of GPUs for mining frequent itemsets. GPUs have radically different characteristics than CPUs, including the SIMT model and the need of coalesced memory access, which add additional difficulty in parallelization. As a result, most GPU-based methods are based on Apriori: [12] represents a transaction database as an $n \times m$ binary matrix, where n is the number of itemsets and m is the number of transactions, so that intersection operations on rows can be conducted with a GPU to count support. GPApriori [57] generates a static bitmap that represents all the distinct 1-itemsets and their transaction ID sets. A GPU is only to parallelize the support counting step, while candidate generation step is performed using CPUs. In [36], the authors proposed a parallel version of the Dynamic Counting Itemset algorithm (DCI) [30] where two major DCI operations, intersection and computation, are parallelized using a GPU. The above three Apriori-based GPU methods cannot handle datasets larger than GPU memory. Frontier Expansion [58] can handle datasets larger than GPU memory. It is based on Eclat and utilizes multiple GPUs. GMiner [6] further improves efficiency by mining patterns from the first level of the enumeration tree rather than storing and utilizing the patterns at the intermediate levels of the tree.

Our PrefixFPM is different from the above existing works in that (1) it unifies the various FPM problems under one framework for parallel execution, and that (2) it adopts prefix projection for depth-first pattern examination to benefit from the shrinking projected databases, massive parallelism exposed by the divide-and-conquer algorithmic nature, and a smaller memory footprint. We overview these points in Sect. 3.

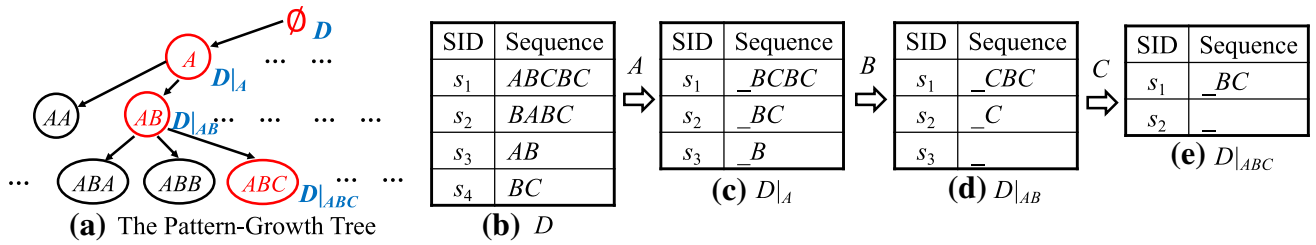


Fig. 1 Illustration of PrefixSpan

3 PrefixFPM solution overview

The PrefixFPM framework is designed based on the idea of *prefix projection*, which is pioneered by the PrefixSpan [31] algorithm for mining frequent sequential patterns from a sequence database.

To illustrate the idea of prefix projection and introduce our notations, we first briefly review PrefixSpan.

A tour of PrefixSpan A prefix-projection-based algorithm adopts a pattern-growth approach where a pattern of size i is extended from a sub-pattern of size $(i-1)$ by adding a new element e. Figure 1a shows a pattern-growth tree for PrefixSpan over a database D of 4 sequences shown in Fig. 1b, where each node in the tree corresponds to a pattern α. A serial prefix-projection algorithm usually traverses the pattern-growth tree in depth-first order, so that only one tree-path is active at any time. In Fig. 1, we assume that the current sequential pattern being checked is ABC, and the corresponding active path of its pattern-growth is highlighted by the red nodes in Fig. 1a.

For ease of presentation, throughout this paper, we use notation α to denote a pattern, e to denote an element to grow a pattern, and β to be a super-pattern grown from α by adding element e. In the context of PrefixSpan, a pattern is a sequence, and thus, we can denote $\beta = \alpha e$, i.e., by appending element e to α. For example, in Fig. 1, pattern $\beta = ABC$ is obtained by extending pattern $\alpha = AB$ with element $e = C$.

We also use the notation $\alpha \sqsubseteq s$ to denote that a pattern α occurs in transaction s in the database D. In the context of PrefixSpan, $\alpha \sqsubseteq s$ means that sequential pattern α occurs as a subsequence of sequence $s \in D$.

Given a pattern α and a transaction s, we use $s|_\alpha$ to denote the so-called α-projected transaction of s, which tracks what remains to be matched in s (by super-patterns grown from α) given that α has already been matched to the corresponding elements in s. In the context of PrefixSpan, $s|_\alpha$ is the suffix γ of s such that $s = s'\gamma$ with prefix s' being the shortest prefix of s satisfying $\alpha \sqsubseteq s'$. To highlight the fact that γ is a suffix, we write it as $\_\gamma$. To illustrate, when $\alpha = BC$ and $s = ABCBC$, we have $s' = ABC$ and $s|_\alpha = \_\gamma = \_BC$.

Given a pattern α and a sequence database D, the α-projected database $D|_\alpha$ is defined to be the set $\{s|_\alpha \mid s \in D \wedge \alpha \sqsubseteq s\}$. Note that if $\alpha \not\sqsubseteq s$, $s|_\alpha$ does not exist and thus s not considered in $D|_\alpha$.

Consider the sequence database D shown in Fig. 1b. The projected databases $D|_A$, $D|_{AB}$ and $D|_{ABC}$ are shown in Fig. 1c–e, respectively.

Let us define the support of a pattern α, denoted as $sup(\alpha)$, as the number of those transactions $s \in D$ where $\alpha \sqsubseteq s$. In PrefixSpan, the support of α is simply the cardinality of $D|_\alpha$ (i.e., the number of $s|_\alpha$ therein).

PrefixSpan finds the frequent patterns (with support at least τ_{sup}) by recursively checking the frequentness of patterns with growing lengths. In each recursion, if the current pattern α is checked to be frequent, it will recurse on each super-pattern β constructed by appending α with one more element.

PrefixSpan checks whether a pattern β is frequent using the projected database $D|_\beta$, which is constructed from $D|_\alpha$. Figure 1 presents one recursion path when $\tau_{sup} = 2$, where, for example, $s_1|_{ABC}$ in $D|_{ABC}$ is obtained by removing the element C from $s_1|_{AB}$ in $D|_{AB}$.

We remark that the PrefixSpan algorithm presented here is a simplified version where each element in a sequence can be only one item. In general, each element can be an itemset (e.g., goods in one supermarket transaction), and we refer readers to [31] for more details.

PrefixSpan is just the simplest form of prefix projection. We next explain how this idea on sequence data can be extended to work with trees and graphs.

Prefix projection The key idea to generalize prefix projection beyond sequence data is to establish a one-to-one correspondence between each subgraph/subtree pattern and its sequence encoding, so that we can examine the pattern encodings by a PrefixSpan-style algorithm. The challenge to address by each specific algorithm is that different patterns that are isomorphic to each other have different encodings, but they actually refer to the same pattern and growing larger patterns from them leads to a lot of redundant computation.

For example, consider the 3 trees shown in Fig. 2. The Sleuth algorithm [56] encodes a tree T by appending vertex labels to the sequence encoding in a depth-first preorder traversal of T, and by adding a unique symbol "$" whenever we backtrack from a child to its parent. For example, the

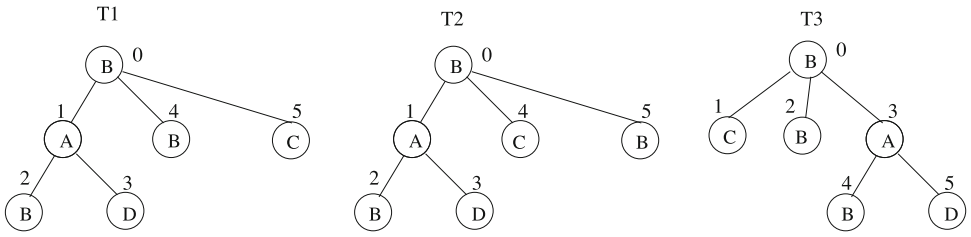


Fig. 2 Tree patterns from Fig. 4 of [56]

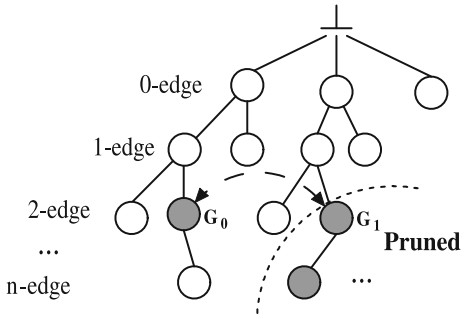


Fig. 3 Pattern search space from Fig. 1 of [49]

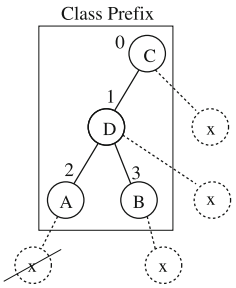


Fig. 4 Pattern extension from Fig. 5 of [56]

encoding of T_1 in Fig. 2 is BAB$D$$BC, while the encoding of T_2 is BAB$D$$CB. Even though the two encodings are different, if we consider unordered trees (meaning that the order of children nodes does not matter), T_1, T_2, and T_3 all refer to the same subtree pattern.

To avoid processing redundant patterns, existing serial algorithms define the *canonical* encoding of a pattern α as the minimum encoding of all automorphisms of α. For example, Lemma 4.2 of Sleuth [56] shows that if we reorder the children of each tree node in non-decreasing order of their labels (assuming "$" is larger than any label), then the encoding of the resulting reordered tree is canonical. In Fig. 2, all the three trees have the same canonical encoding which is that of T_1.

Let us denote the canonical encoding of a pattern α by $\min(\alpha)$, then we only examine a pattern α if its encoding equals $\min(\alpha)$. This method avoids redundant computation since only one pattern in its automorphism group will be checked to grow larger patterns.

Figure 3 illustrates the pattern-growth tree of the gSpan algorithm [49] for mining frequent subgraph patterns that are grown by adding adjacent edges. Each tree node in Fig. 3 represents a subgraph pattern α to examine, and the subtree under the node contains those patterns grown from α. Assume that G_0 and G_1 are isomorphic and since only G_0's encoding equals its canonical encoding, only G_0 is checked for frequentness and for further pattern growth, while G_1 (and its potential pattern-growth subtree) is pruned.

A pattern α is extended by one more element to generate a child-pattern β, and in PrefixSpan, we simply append α with all possible labels. However, for a subgraph or subtree pattern α, we cannot extend it with any adjacent edge since some extensions have been considered by a prior node in the depth-first pattern-growth tree. For example, in Fig. 4, we can only extend the subtree pattern in the box using an adjacent edge on its rightmost path CDB, since the extension from vertex A has a smaller encoding than the subtree pattern itself (CDAx$\cdots$ <CDA$\cdots$) and has thus been considered previously in the depth-first traversal order.

Task-based model of PrefixFPM PrefixFPM associates each pattern α with a task t_α which checks the frequentness of α using its projected database $D|_\alpha$, and which grows the pattern by one more element to generate the children patterns $\{\beta\}$ and their projected databases $\{D|_\beta\}$ (computed incrementally from $D|_\alpha$). These children patterns give rise to new tasks $\{t_\beta\}$ which can then be added to a shared task queue to be fetched by computing threads for further processing.

PrefixFPM runs a number of computing threads that fetch pattern-tasks from a shared task queue Q_{task} for concurrent processing. Since each task t_α needs to maintain $D|_\alpha$ to compute the projected databases of the child-patterns grown from α, a depth-first task fetching priority in the pattern-growth tree tends to minimize the memory footprint of patterns in processing. This is because we tend to grow those patterns that have been grown deeper, which are larger (and thus with smaller projected databases) and are closer to finishing their growth (due to the support becoming less than τ_{sup}).

We thus implement the task queue Q_{task} as a stack where newly pushed tasks are popped earlier for processing. Note that PrefixFPM processes the pattern-growth tree in a near-depth-first order but not strictly depth-first: When the leftmost pattern-node α is being processed as a task t_α by some computing thread, its next sibling pattern-node will be popped for processing by another available computing thread (rather than the children of pattern-node α). This allows idle computing threads to fetch patterns for processing ASAP to

keep CPU cores busy, but may require more memory than a serial depth-first solution.

In fact, the memory cost is the same as a serial algorithm if there is only one computing thread (as tasks are fetched in exact depth-first order), but it is expected to increase with the number of threads n_{thread}, though bounded by n_{thread} times the RAM consumption of a serial algorithm since nodes (and their projected databases) on the parent-to-root path could be shared. Queue Q_{task} is protected by a lock (mutex) so that only one thread can push or pop a task at a time.

Since fetching tasks from Q_{task} and adding tasks to Q_{task} incur locking overheads, interaction with Q_{task} is only worthwhile if each task contains sufficient computing workloads such that the locking cost of fetching it and inserting child-pattern tasks is negligible. Therefore, when processing a task t_α, we only add child-pattern tasks to Q_{task} if the number of projected transactions in $D|_\alpha$ is above a size threshold τ_{split}, so that the workloads of mining the pattern-growth subtree rooted at α can be divided by other computing threads; otherwise, t_α is not expensive and the current computing thread simply mines the entire pattern-growth subtree in depth-first order directly as in a serial algorithm.

However, τ_{split} only considers the number of projected transactions in $D|_\alpha$, and it is not sufficient to totally capture the running time of t_α if it mines its entire pattern-growth subtree, since the content of each $s|_\alpha \in D|_\alpha$ can affect the mining time a lot (e.g., how long is the suffix γ of $s|_\alpha$ in PrefixSpan). We find that setting τ_{split} small will over-partition a lot of tasks, adding burden to Q_{task}'s locking overhead, while setting τ_{split} large may cause some expensive tasks to mine their entire pattern-growth subtree for a long time, becoming the stragglers.

A better approach is to allow a task to mine its pattern-growth subtree in depth-first order until a timeout happens, after which we add the remaining tasks (or, nodes in the pattern-growth subtree) into Q_{task} for concurrent processing. We define the timeout threshold as τ_{time}. Note that long-running tasks are avoided since no task will mine for longer than τ_{time}, and this approach also guarantees that at last τ_{time} time is spent on the actual mining if timeout is triggered, so the additional cost of wrapping the remaining pattern-nodes as new independent tasks and inserting them into Q_{task} is not dominating (but rather, remains negligible).

We therefore set τ_{split} large to allow obviously big tasks to be split right away without time tracking, while leaving the task-splitting decision of the remaining tasks to our timeout mechanism. After tuning the parameters, we found that the default setting with $\tau_{split} = 100$ and $\tau_{time} = 0.1$ second works well in all scenarios (c.f. Sect. 8.7) and is thus adopted.

To generate initial patterns $\{\alpha\}$ along with their projected databases from D (e.g., the root-node splitting case in Fig. 1a), processing it as a single task in serial is not optimal since D often contains many transactions and it can be expensive to scan them to build $\{D|_\alpha\}$. In this initial case, we recommend users to implement this scan as a parallel for-loop, e.g., using OpenMP [29]. This effectively distributes the workloads among all computing threads, which is important in the initial stage when there are no other tasks.

Note that two different transactions s_1 and s_2 that are processed by two different OpenMP threads may both contain the same initial pattern α and thus insert $s_1|_\alpha$ and $s_2|_\alpha$ concurrently to the children table entry for $D|_\alpha$. To be thread-safe, one option is to let each table entry maintain a lock (mutex) so that only one thread can insert an item to $D|_\alpha$ at a time.

We find, however, that this strategy generates a lot of lock contention when there are many threads. We thus adopt another lock-free solution where each thread θ maintains its own local children table $children_\theta$ for inserting projected instances. At the end, we merge these local tables into the final children table $children$, in which each projected database $D|_\alpha$ is obtained by taking the union of those items in the corresponding projected databases inside thread-local tables $\{children_\theta\}$.

In general, for a task t_α, we can check if $D|_\alpha$ is large, and if so, we can let the scanning of $D|_\alpha$ be parallelized. However, this means that not only different tasks may execute concurrently, but each task itself may execute concurrently. Such complicated concurrency scheduling often backfires in our preliminary experiments where if we set the size threshold of $D|_\alpha$ as suboptimal, the execution time can be many times longer than a solution where only the scanning of D to generate root tasks is parallelized, due to the contention of the many threads for the limited number of CPU cores. Moreover, finding the optimal size threshold by our extensive trials reveals that (i) the optimal threshold can vary a lot and much larger than $|D|$ itself since each tree/graph transaction can have multiple projected transaction instances in $D|_\alpha$, and that (ii) even with the optimal size threshold found, the running time is only slightly reduced (always by less than 10%), which does not justify the expensive trial efforts. We, therefore, do not adopt this optimization.

4 PrefixFPM programming model

PrefixFPM is a programming framework written as a set of C++ header files defining some base classes and their virtual functions for users to inherit in their subclasses and to specify the application logic. We call these virtual functions as user-defined functions (UDFs). The base classes also contain C++ template arguments for users to specify with the proper data types (data structures) that fit their pattern mining applications.

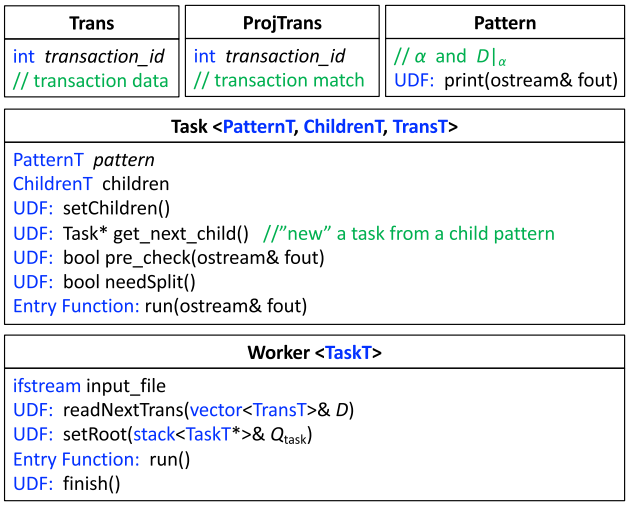


Fig. 5 PrefixFPM programming interface

We now introduce these base classes one by one, and Fig. 5 summarizes their key programming interfaces.

Trans The *Trans* class implements a transaction in the input database with a predefined transaction ID field. Users implement their transaction subclass by inheriting *Trans* and including additional fields to store the target data instance such as a sequence, a tree or a graph. Initially, the input dataset is read into an in-memory transaction database D which is simply an array of objects whose type is given by the *Trans* subclass.

ProjTrans The *ProjTrans* class implements a projected transaction $s|_\alpha$ in a projected database $D|_\alpha$. A *ProjTrans* object also has a transaction ID field indicating which transaction $s \in D$ this projected transaction corresponds to. The user-defined *ProjTrans* subclass should also indicate how $s|_\alpha$ is currently matched onto s, so that the matching status can be incrementally updated as the pattern α grows. We remark that a transaction $s \in D$ can have many matched instances (i.e., projected transactions) in $D|_\alpha$ for subgraph and subtree patterns, though at most one in PrefixSpan.

Pattern The *Pattern* class specifies the data structure of a pattern α and contains a (pure) virtual function *print*(*fout*) specifying how to output the object of a *Pattern* subclass into an output file stream *fout*. Recall that PrefixFPM runs multiple task computing threads, and each thread actually appends the frequent patterns found by it to a file of its own, and the file streaming handle is passed into *print*(.) as *fout*. When a job finishes, the frequent patterns are recorded by the collection of all files written by the task computing threads.

A *Pattern*-subclass object for pattern α usually also includes the projected database $D|_\alpha$ as a field (e.g., whose type can be an array of *ProjTrans*-subclass objects). In *print*(*fout*), users may choose to output $D|_\alpha$ along with α, to indicate the matched transactions.

Task The base classes we saw so far are to specify data structures, and let us next look at two other base classes whose aim is to specify the algorithm logic.

Recall that a task t_α checks the frequentness of pattern α using its projected database $D|_\alpha$ and grows α by one more element to generate the children patterns and their projected databases. Prefix-projection algorithms usually generate $\{D|_\beta\}$ for all children patterns $\{\beta\}$ together in one pass over $D|_\alpha$, since we can go through the elements of each $s|_\alpha \in D|_\alpha$ to incrementally generate $s|_\beta$ for every child-pattern β. For example, consider s_1 inside $D|_A$ of Fig. 1c where by scanning its 4 remaining elements, we can generate $s_1|_{AB} = \_CBC$ into $D|_{AB}$ and generate $s_1|_{AC} = \_BC$ into $D|_{AC}$.

Following this practice, base class *Task*<*PatternT, ChildrenT, TransT*> takes 3 C++ template arguments:

- *PatternT*: the user-defined *Pattern* subclass (which often also maintains $D|_\alpha$);
- *ChildrenT*: the type of a table $children$ that keeps the projected database of child-patterns $\{D|_\beta\}$;
- *TransT*: the user-defined *Trans* subclass.

A *Task* object t_α maintains 2 fields: a pattern α of type *PatternT* (often containing $D|_\alpha$), and the children table $children$ that keeps $\{D|_\beta\}$, which is typically implemented as std::map with $children[e] = D|_\beta$ if β is grown from α with element e.

TransT is needed since the *Task* class provides a function to access the global static transaction database D for users to call in their *Task* UDFs, which is useful since a projected transaction $s_i|_\alpha$ usually only keeps a compact matching status towards $s_i \in D$, and in order to extend it with one more element e in s_i to generate $s_i|_\beta$, we need to access s_i as $D[i]$ where i is the transaction ID field of $s_i|_\alpha$ (with type *ProjTrans*-subclass).

Task has an internal function *run*(*fout*) which executes the processing logic of the task t_α. The behavior of *run*(.) is specified by *Task* UDFs defined by users which are called in *run*(.), and Fig. 6 shows the details.

Specifically, upon the beginning of t_α, the task first records the initial time t_0 in Line 2 so that at any time t during its execution, it can obtain the task running time as $(t - t_0)$, to be used later in Line 8 to compare with the timeout threshold τ_{time}. Then, in Line 3, t_α first runs UDF *pre_check*(*fout*) in which t_α can check if α is frequent (and its encoding is canonical if applicable), and if so, to output α to *fout*. If α is not pruned, Line 5 then runs UDF *setChildren*($children$) to scan $D|_\alpha$ and compute $\{D|_\beta\}$ to be inserted into the table field $children$. At the end of UDF *setChildren*(.), we usually post-process $children$ to remove those infrequent child-patterns. This is because for each $s|_\alpha \in D|_\alpha$, we will grow $s|_\alpha$ into $\{s|_\beta\}$ with all applicable elements $\{e\}$, to be inserted to $\{D|_\beta\}$;

```
1   void run(ostream& fout){
2       {pseudocode: track task starting time here}
3       if(!pre_check(fout)) return;
4       //generate new patterns
5       setChildren(children);
6       //run new child tasks
7       while(Task* t= get_next_child()){
8           if(timeout){
9               q_mtx.lock();
10              queue().push(t);
11              while(Task* t= get_next_child())
12                  queue().push(t);
13              q_mtx.unlock();
14              return;
15          }
16          if(needSplit()){
17                q_mtx.lock();
18                queue().push(t);
19                q_mtx.unlock();
20            }
21            else{
22                t->run(fout);
23                delete t;
24            }
25        }
26  }
```

Fig. 6 The *run*(*fout*) function of base class *Task*

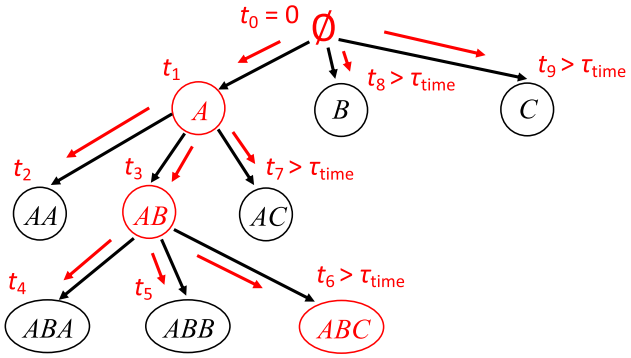


Fig. 7 Illustration of task split upon timeout

therefore, only when the last $s|_\alpha \in D|_\alpha$ is processed can we finalize the size of every $D|_\beta$ to determine whether β is frequent or not.

Line 7 then wraps each child-pattern β in table *children* as a task t_β for checking one by one. For each child-task t_β, we first check if timeout happens (i.e., $t - t_0 > \tau_{time}$), and if so, we process it in Lines 9–14 which we will explain shortly. Otherwise, we then call the UDF *needSplit*() in Line 16 which usually checks if $|D|_\beta|$ is large (and hence t_β is time-consuming), and if so, we add t_β to the task queue Q_{task} (Lines 17–19) to be fetched by available task-computing threads. (Recall that Q_{task} is a global last-in-first-out task stack protected by a mutex.) Otherwise, t_β is considered inexpensive and we recursively call t_β's *run*(*fout*) in Line 22 to process the entire checking and extension of β by the current thread, which avoids contention on Q_{task}. Finally, note that Line 23 deletes t_β including its maintained $D|_\beta$, so only the projected databases of those patterns being actively mined are kept in memory.

Now, let us get back to the timeout processing in Lines 9–14. Specifically, we will lock the task queue Q_{task} once (see Lines 9 and 13) and insert all remaining child-tasks (see Lines 11–12) and then exit the current *run*(.) function which runs t_α. Since *run*(.) is recursive (see Line 22), the thread running t_α will then return to the task of the parent pattern of α, denoted by $pa(\alpha)$, for which its *run*(.) will exit the call of $t_\alpha - >run(.)$ at Line 22 and move on to the next child-pattern of $pa(\alpha)$ (see Line 7).

Figure 7 illustrates this process assuming that we run the root task using the timeout mechanism in PrefixSpan, and that timeout happens when we enter the processing of task t_{ABC} in Line 8. Before timeout occurs, the pattern-growth tree is traversed in depth-first order where tree-nodes are processed in the time order $t_0, t_1, \ldots, t_5$. Since $t_6 > \tau_{time}$, task t_{ABC} gets added to Q_{task} as the last child-task of t_{AB}. Then, the execution backtracks to t_{AB}, and in its level there is only one more task t_{AC}. Since the visit time $t_7 > \tau_{time}$, task t_{AC} is also added to Q_{task} before the execution backtracks to t_A. Finally, the remaining two tasks in this level, t_B and t_C, are added to Q_{task}. Note how the timeout mechanism only expands a task to its necessary granularity, e.g., t_B will not be split until it is fetched again from Q_{task} for further processing.

Existing frameworks such as Arabesque [38] and RStream [45] over-emphasize simplicity of programming, but the performance becomes IO-bound. Consider, for example, the so-called embedding-centric level-by-level subgraph growth API of [38]. In contrast, our PrefixFPM framework focuses on the ease of adapting an existing prefix-projection algorithm (e.g., in a day) for CPU-bound parallel execution, not the simplicity of prefix-projection algorithms themselves. In fact, those serial algorithms are designed to be sophisticated to exploit the superior performance over simplistic methods, but with such a serial program at hand, sorting out how to reorganize its code into PrefixFPM is easy and quick.

Worker A PrefixFPM program is executed by subclassing the *Worker*<*TaskT*> base class, and call its *run*() function. Here, *TaskT* refers to the user-defined *Task* subclass, from which *Worker* derives the other necessary types such as *TransT*.

The *run*() function (1) keeps calling UDF *getNextTrans*() to read transactions from an input file and appends them to the transaction database D, (2) calls the UDF *setRoot*() to generate root tasks (where α contains only one element) into Q_{task}, and (3) creates k task-computing threads to process the tasks in Q_{task}.

Note that the element type is different for different pattern mining problems. For example, an element in sequential pattern mining is just a label, while that for subgraph/subtree pattern mining is an edge since patterns are grown with adjacent edges. As a result, we do not provide a base class *Element* and let users freely define their own types.

Also, note that implementing *Worker*::*setRoot*(.) is similar to implementing *Task*::*setChildren*(.): Instead of constructing $\{D|_\beta\}$ from $D|_\alpha$, we construct $\{D|_e\}$ from D. Each seed task $t_e = \langle e, D|_e\rangle$ is then added to Q_{task} to initiate the parallel task computation.

Since D is large, UDF *Worker*::*setRoot*(.) can scan its transactions in parallel, i.e., users can use OpenMP parallel for-loop in their implementation. There are 2 ways: (1) to let each OpenMP thread lock the container $children[e]$ for storing $D|_e$ when appending a new $s|_e$ projected from $s \in D$ with element e, which causes a lot of lock contentions among the k OpenMP threads, and (2) to create an array of k children-tables, one for use by each OpenMP thread, so that there is no contention for appending $s|_e$ during the parallel scanning of D, but the computing thread needs to merge the k tables into the final table $children$ which leads to additional overheads. We tested both choices and find that the lock-free method "(2)" beats "(1)" consistently and the gap is larger with increasing k, as the table merging is found to be very fast compared with the parallel scanning of D. We thus recommend users to always implement *Worker*::*setRoot*(.) using Method "(2)."

At the beginning of *Worker*::*setRoot*(.), we also need to get the element frequency statistics and eliminate infrequent elements (i.e., they are not considered when growing patterns), which is a commonly used and effective pruning technique. Depending on the mining problem, computing the statistics may need multiple passes on D. For example, when transactions are labeled graphs, one pass is needed to filter out infrequent vertex and edge labels, followed by another pass over transactions in D to (1) eliminate data edges containing those infrequent edge labels, and data edges whose end vertices have those infrequent vertex labels, and to (2) count the frequency of pattern edges (decided by the edge label and the labels of an edge's end vertices).

During parallel task computation, each computing thread keeps fetching a task t_α from Q_{task} to call its *run*(*fout*) function, and it gets suspended to release the occupied CPU core when it cannot find a task in Q_{task}. Note that while Q_{task} is currently empty, another thread may be processing a task and could add more child-tasks back to Q_{task}. The job terminates only if all k task computing threads are suspended and no task is found in Q_{task}.

Every 100 ms, the *Worker*::*run*() main thread checks if all threads are suspended. If not, it wakes up all computing threads to continue task processing; otherwise, it wakes up all computing threads to terminate.

Finally, the main thread runs *Worker*::*finish*() where some post-processing operations can be conducted before the job terminates.

We use C++11 condition variable's *wait* and *notify_all* functions to implement the thread suspension and main thread wakeup notification, and the number of threads suspended is maintained with an atomic counter (atomic<int> in C++11).

5 Sequential pattern mining with PrefixFPM

Starting from this section, we will describe how to implement popular pattern-mining algorithms on top of PrefixFPM. This section considers frequent and close sequential pattern mining from a sequence database.

5.1 Mining frequent sequential patterns

We have reviewed PrefixSpan in Sect. 3. We now describe how we adapt the serial PrefixSpan implementation of [34] to PrefixFPM for parallel mining.

Transaction A transaction object here is simply implemented as a sequence.

Projected Transaction Instead of keeping a suffix, a projected transaction object here only keeps the position of the last match (in addition to the transaction ID of class *ProjTrans*), i.e., "_" in Fig. 1, to minimize the memory consumption occupied by projected databases.

Pattern A pattern object here maintains a sequence α along with the projected database $D|_\alpha$.

Task The most important UDF is *setChildren*(). Given a pattern α, to extend a projected transaction $s|_\alpha$ in $D|_\alpha$ by an element e, we first (1) get the sequence in D using the transaction ID of $s|_\alpha$ and then (2) continue to scan the remaining elements starting from the next position after "_" as follows. Recall that *PrefixSpan looks for the minimal (i.e., earliest) prefix match*. Therefore, whenever we scan to an element e that has not appeared before during our scan of $s|_\alpha$, we insert $s|_\beta$ to $children[e]$ (i.e., $D|_\beta$) where $\beta = \alpha e$. Here, $s|_\beta$ keeps the position of e scanned for future expansion.

At the end of *setChildren*(), another pass over the table $children$ is conducted to delete those table entries $\langle\beta, D|_\beta\rangle$ where $|D|_\beta| < \tau_{sup}$. Since infrequent child-patterns have been removed, every extended pattern is guaranteed to be frequent, and therefore, UDF *pre_check*(*fout*) can directly output pattern β.

Finally, UDF *needSplit* simply checks if $|D|_\beta| \geq \tau_{split}$ where τ_{split} is a user-defined threshold beyond which the task is added to Q_{task} rather than recursively processed by the current thread.

Worker The implementation of UDF *setRoot*(Q_{task}) is the same as *Task*::*setChildren*(.) except that D is scanned using OpenMP parallel-for loop (rather than $D|_\alpha$ scanned in serial), and it generates each task $t_e = \langle e, D|_e\rangle$ and adds it to Q_{task} only if e is frequent (i.e., $|D|_\beta| \geq \tau_{sup}$).

Besides, infrequent singleton patterns e which are also infrequent elements are marked so that in future task process-

SID	Sequence
0	(*af*)*dea*
1	*eab*
2	*e*(*abf*)(*bde*)

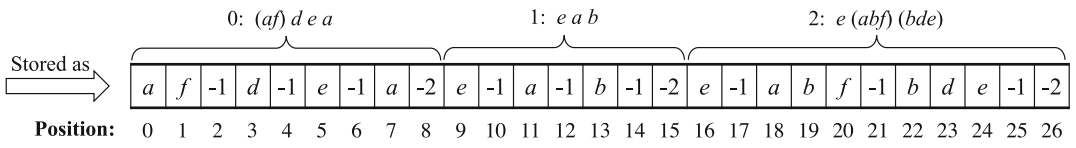


Fig. 8 Illustration of transaction database in CloSpan

ing (e.g., the scan of the suffix of $s|_\alpha$ in *Task*::*setChildren*(.)), such elements are directly skipped.

5.2 Mining closed frequent sequential patterns

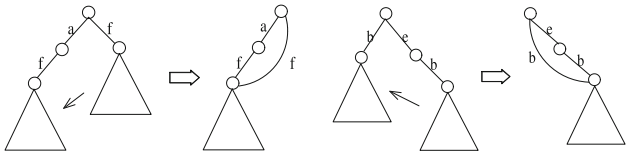
PrefixFPM can also be used to parallelize frequent pattern mining algorithms with other constraints, such as finding closed patterns, maximal patterns, and patterns with gap constraints. This subsection illustrates how we can implement the CloSpan algorithm [48] for mining closed sequential patterns in PrefixFPM, by parallelizing its implementation at [7].

We remark that so far, we have been assuming that each element in a sequence can be only one item for ease of presentation, but in the literature, each element can itself be an itemset, which is adopted by the implementation at [7]. For example, to extend a sequential pattern $\alpha = AB$ with an item $e = C$, we can either do the so-called I-Step extension [48] to obtain $\beta = A(BC)$ which means the second element is an itemset $\{B, C\}$, or S-Step extension [48] to obtain $\beta = ABC$. To be compatible with prefix projection, a lexicographic order is defined to order the child-patterns in the pattern-growth tree, where an item-extended sequence is considered less than sequence-extended sequence if their prefixes are the same. For example, $A(BC) < ABC$.

Unlike in Sect. 5.1 where a transaction database D is implemented as an array (std::vector) of sequences which are themselves item (or itemset) arrays, CloSpan keeps the entire D as one big array where transactions are separated by special labels "-2" (regular item labels are encoded as 0, 1, 2, ...), while itemsets in the same sequence are separated by "-1," as Fig. 8 illustrates. This representation is used to compute hash keys needed in CloSpan to identify non-closed sequences (to be introduced soon). We next explain the idea of CloSpan, followed by our parallel implementation in PrefixFPM.

Closed sequential patterns A sequential pattern α is closed if there does not exist a super-sequence β whose support is the same as $sup(\alpha)$. For example, Fig. 1c, d shows that $sup(A) = sup(AB) = 3$, and thus, AB invalidates pattern A from being closed.

We usually mine closed patterns that are also frequent, i.e., with support at least τ_{sup}. The benefit is that the number of closed frequent patterns is much smaller than the full set of frequent patterns, while the expressive power remains the same [48].



(a) backward sub-pattern **(b)** backward super-pattern

Fig. 9 Transplanting descendants, from Fig. 3 of [48]

Early termination rule The efficiency of CloSpan compared with PrefixSpan lies in its early termination rule. The insight is illustrated in Fig. 9.

Specifically, (Case 1) let us assume that a pattern $\alpha = af$ has been mined in the pattern-growth tree as shown in Fig. 9a. When we reach another pattern $\gamma = f$ and find that $D|_\alpha = D|_\gamma$, then the pattern growth process is exactly the same in the subtrees under both α and γ. We can thus hook γ directly to α (meaning that the children list of γ is pointed to that of α) to avoid repeated subtree pattern-growth. Moreover, since α is a super-sequence of γ, γ is not closed and thus pruned.

In Fig. 9b, (Case 2), we assume that a pattern $\alpha = b$ has been mined in the pattern-growth tree, and we then reach another pattern $\gamma = eb$ and find that $D|_\alpha = D|_\gamma$. In this case, we can similarly hook γ directly to α to reuse the constructed tree under α. The difference is that now γ is a super-sequence of α, and thus, α is not closed and hence should be pruned with its pattern-growth subtree taken over by γ.

Note that since the pattern-growth tree is mined by the serial CloSpan algorithm in depth-first order, the subtree under α must have been fully constructed when we reach γ. However, since different tree branches could be mined in parallel in PrefixFPM, in our parallel scenario when we reach γ, the subtree under α may still be under construction. But we can still hook γ to α as before to avoid constructing the subtree under γ: When the subtree under α is finally constructed, it is also reused by γ anyway.

Efficient rule checking Note that a key requirement in the above rule is $D|_\alpha = D|_\gamma$, which is inefficient to check directly. Fortunately, without loss of generality if we assume γ is a super-sequence of α, then we only need to check if $len(D|_\alpha) = len(D|_\gamma)$ to ensure $D|_\alpha = D|_\gamma$, where $len(D|_\alpha)$ is the length of $D|_\alpha$ defined as the number of items in $D|_\alpha$. For example, for the database in Fig. 8, $D|_{(af)} = \{s_0|_{(af)}, s_2|_{(af)}\} = \{\_dea, \_(bde)\}$, and thus $len(D|_{(af)}) = 6$.

To see why the above theorem is correct, note that since γ is a super-sequence of α, more items are prefix-matched leaving suffixes shorter, i.e., $s|_\gamma$ must be a suffix of $s|_\alpha$, and $len(s|_\gamma) \leq len(s|_\alpha)$. If $len(D|_\alpha) = len(D|_\gamma)$, then we must have $len(s|_\alpha) = len(s|_\gamma)$ for any s in the projected databases. This is only possible if $s|_\alpha = s|_\gamma$ since $s|_\gamma$ is a suffix of $s|_\alpha$.

The above proof still holds if we assume α is a super-sequence of γ, i.e., if $len(D|_\alpha) = len(D|_\gamma)$, then $D|_\alpha = D|_\gamma$. Therefore, when we reach γ in the pattern-growth tree, we just need to find all patterns α already processed in the tree where (i) $len(D|_\alpha) = len(D|_\gamma)$, and (ii) α is either a super-sequence or a sub-sequence of γ, which gives the following 2 cases. **(Case A):** if some patterns $\{\alpha\}$ are found to be sub-sequences of the current pattern γ, they are not closed and should be pruned, and their subtrees are the same and only one copy is retained for γ to hook to for reuse. **(Case B):** if any pattern α is found to be a super-sequence of γ, γ is not closed and thus should be pruned, while we hook γ to α to avoid repeated subtree pattern-growth.

Hash-based fetching of α Now that we know we can simply check $len(D|_\alpha) = len(D|_\gamma)$ rather than $D|_\alpha = D|_\gamma$, one problem remains: how can we fetch all such patterns $\{\alpha\}$ efficiently from the set of all patterns currently processed? There are 2 implications here: (1) We need to maintain all currently processed patterns in memory for later fetching, rather than streaming each frequent pattern found to an output file stream *fout* as in our PrefixFPM algorithm for PrefixSpan; (2) we do not want to scan all currently processed patterns, as only those with $len(D|_\alpha) = len(D|_\gamma)$ are relevant.

For implication (1), we respect the assumption of CloSpan [48] that "based on today's technology and our experience, it is easy to maintain a million sequences in memory." Although another algorithm BIDE [44] has been proposed later that is shown to perform better than CloSpan when a database is large without need of keeping all patterns in memory, our goal here is to show the generality of PrefixFPM's API, and so we respect and reuse all the original in-memory design of CloSpan. BIDE follows the same recursive algorithmic framework as PrefixSpan (though with additional pruning rules) and is straightforward to parallelize in PrefixSpan.

Another note is that while CloSpan's early termination rule prunes patterns that are not closed, there is no guarantee that all such patterns are captured by the rule and pruned [48]. Therefore, a post-processing step over the pattern-growth tree (or more accurately, "lattice" since subtrees are merged by the pruning rule) recursively to enumerate all closed patterns [48]. We conduct this post-processing in UDF *Worker*::*finish*() to write closed patterns to the output file which is executed by the main thread at the end. While this recursive process can be parallelized like the recursive serial PrefixSpan algorithm, we find it to be very lightweight compared with the pattern-growth tree mining process, and thus we leave it to run simply by the main thread.

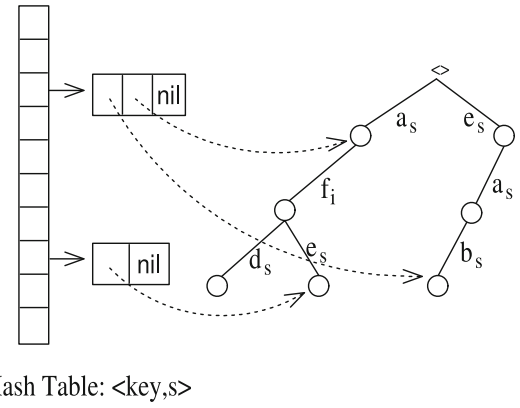


Fig. 10 Hashing for fast rule checking, from Fig. 5 of [48]

For implication (2), a straightforward solution is to maintain patterns using a hash table where the key is $len(.)$: when we reach γ, we use the key $len(\gamma)$ to obtain only those already processed patterns α with $len(\alpha) = len(\gamma)$ for rule checking (i.e., Cases A and B mentioned before). More specifically, Fig. 10 shows the hash table T where the table entry for key k saves a list of patterns $T[k] = \{\alpha\}$ with $len(\alpha) = k$, and each list element is actually a pointer to the node of α in the pattern-growth tree (or more accurately, lattice) for ease of node hooking. In **Case A** where some patterns $\{\alpha\}$ are found to be sub-sequences of the current pattern γ, their subtrees are the same and merged for γ to hook to for reuse, and they are removed from the list $T[k]$ as they are not closed. In **Case B** where if any α is found to be a super-sequence of γ, γ is not closed and thus not inserted to $T[k]$. If no such super-sequence α is found after scanning $T[k]$, we will add γ to $T[k]$.

However, directly using $len(.)$ as the hash key may still lead to a long $T[k]$ list. To allow hash keys to span a larger value range, CloSpan [48] actually adopts a key that adds $len(\gamma)$ with a summation $\mathcal{L}(D|_\gamma)$ that sums up the distance between the start position of a projected sequence $s|_\gamma$ and the end position of the whole database D. For the example shown in Fig. 8, we have $\mathcal{L}(D|_{\langle af \rangle}) = (26 - 1) + (26 - 20) = 31$. Obviously, if $D|_\alpha = D|_\gamma$, then the hash keys $(len(\alpha) + \mathcal{L}(D|_\alpha))$ and $(len(\gamma) + \mathcal{L}(D|_\gamma))$ must be equal. However, since the reverse may not be true, for each $\alpha \in T[k]$ we should first check whether $len(\alpha) = len(\gamma)$, and skip α if not, before checking Cases A and B to update T and the pattern-growth lattice.

We next describe our implementation in PrefixFPM.

Transaction A transaction object here is implemented as an array representing each transaction sequence. As an illustration, Fig. 8 shows how we represent each of the 3 sequences on the right-hand side of the arrow, and the transaction database D is simply an array that concatenates those transaction sequence arrays.

Projected transaction Due to the existence of itemset element, for each transaction, we now need to track all the prefix-matched positions rather than only the first one, or we may miss matched projected transactions. To see this, consider a transaction $s = (ac)(abc)b$. If we only consider the first match for pattern a, i.e., $s|_a = (\_c)(abc)b$, then we will miss $s|_{(ab)}$ when growing from $s|_a$ since the matched a is in the first itemset of $s|_a$ but b is in the second itemset. We thus need to keep track of all matches $s|_a = \langle(\_c)(abc)b, (\_bc)b\rangle$.

We define a *ProjTrans* object directly as the projected database $D|_\alpha$ of a pattern α, rather than a projected transaction $s|_\alpha$ in $D|_\alpha$ as we did for PrefixSpan.

Specifically, our *ProjTrans*-subclass object keeps an array of the matched positions in D to minimize the memory consumption occupied by projected databases. To illustrate using D in Fig. 8, for $D|_{(af)}$ we keep an array of two positions 1 and 20 that match the last pattern-element f to extend. To avoid repeatedly incrementing support for multiple pattern-matches in the same transaction, we only increment the support counter each time when our scan in D reaches a "-2."

Pattern A pattern object here maintains a sequence α along with the projected database $D|_\alpha$ whose type is defined by the above-mentioned *ProjTrans* subclass.

Task Given the pattern α of task object t_α, the UDF *pre_check*(*fout*) first tries to add α to the pattern hash table T (that tracks the current set of mined patterns) for later output, which will trigger our early termination rule checking. If α is determined to be not closed, *pre_check*(*fout*) returns *false* to skip the pattern growth from α; note that *run*(*fout*) will exit directly as shown in Line 3 of Fig. 6. Otherwise, *pre_check*(*fout*) returns *true* so that the UDF *setChildren*() will then be called to grow α and set the $children$ table.

Specifically, we will use the position pointers in our *ProjTrans*-subclass object for $D|_\alpha$ to scan D to identify frequent child-patterns $\{\beta\}$ (i.e., $sup(\beta) \geq \tau_{sup}$). For each such β, we have two cases which are differentiated based on a concept of "growth-support" defined as follows: if a projected transaction $s|_\beta \in D|_\beta$ is fully matched to pattern β, i.e., the suffix becomes empty, then we do not increment β's growth-support; while if the suffix is not empty, we increment β's growth-support. Intuitively, β's growth-support defines an upper bound on $sup(\gamma)$ for any pattern γ that is extended from β.

In UDF *setChildren*(), therefore, we have two cases for each frequent child-pattern β. (i) If β's growth-support is less than τ_{sup}, then β cannot be extended to generate a frequent pattern and is thus not added to table $children$. However, since β itself is frequent, we try to add it to the pattern hash table T for later output, which will trigger our early termination rule checking. (ii) Otherwise, we add the table entry $children[\beta] = D|_\beta$ so that β will be extended to grow longer patterns.

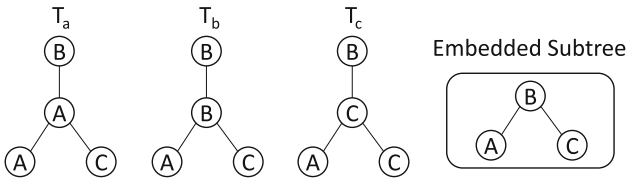


Fig. 11 Embedded subtree pattern illustration

Finally, UDF *needSplit* simply checks if the array of position pointers maintained by $D|_\beta$ (of type being our *ProjTrans* subclass) is larger than τ_{split}, and returns *true* if so, and returns *false* otherwise.

Worker UDF *setRoot*(Q_{task}) scans D and generates a task $t_e = \langle e, D|_e\rangle$ for each frequent item-label e. If the growth-support of such a single-item pattern e is at least τ_{sup}, we add t_e to Q_{task} for pattern growth; otherwise, we cannot extend e to generate frequent patterns, but since e is still frequent, we try to add it to the pattern hash table T for later output, which will trigger our early termination rule checking. Note that the infrequent singleton patterns e can be marked so that they can be ignored during pattern growth.

Finally, UDF *finish*() outputs closed frequent patterns at the end of the job, by visiting nodes of our constructed pattern-growth lattice.

6 Parallel embedded subtree pattern mining

Mining frequent subtree patterns in a tree database (or, forest) is useful in domains such as bioinformatics and mining semi-structured data. This section describes how to implement popular subtree mining algorithms, PrefixTreeSpan [59] and Sleuth [56], on PrefixFPM.

We consider the problem of mining embedded subtrees in a database of rooted and labeled trees. Here, "rooted" means that the tree root matters, and "embedded" means that the tree edge in a subtree pattern only needs to capture the ancestor-descendant relationship (i.e., can skip nodes in the middle) rather than a direct parent-child edge (the latter is called "induced").

We illustrate the concept of an embedded subtree pattern using Fig. 11, which shows a database of three trees. The subtree shown in the box is considered frequent since it appears in all the three trees T_a, T_b and T_c, obtained by skipping the "middle" node in each tree, even though this subtree is the induced subgraph of only T_b alone.

Overview of the algorithms For rooted and embedded subtree patterns, there are still two cases: ordered and unordered. Here, "ordered" (resp. "unordered") means that the order of children nodes matters (resp. does not matter). Referring to

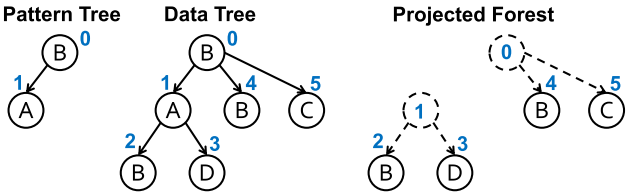


Fig. 12 Forests after pattern projection

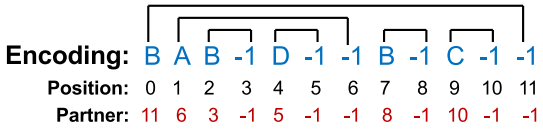


Fig. 13 PrefixTreeSpan tree encoding

the 3 trees shown in Fig. 2 again, they are considered different (resp. the same) if they are ordered (resp. unordered).

Among the algorithms that we will review, PrefixTreeSpan [59] and TreeMiner [53] are for mining ordered subtree patterns, while Sleuth [56] extends TreeMiner to mine unordered subtree patterns.

All these algorithms adopt prefix projection to enumerate patterns in the pattern-growth tree. Recall that in Fig. 2, if we encode each tree pattern by appending "$" whenever we backtrack from a child to its parent, we obtain encoding BAB$D$$BC for T_1, BAB$D$$CB for T_2, and BCBAB$D$$ for T_3. For the Sleuth algorithm which considers trees as unordered, T_1, T_2, and T_3 are considered the same and Sleuth only checks and grows pattern T_1 whose encoding is canonical (i.e., the smallest alphabetically among the three). In contrast, PrefixTreeSpan and TreeMiner do not perform such canonical encoding test to prune patterns.

As the Sleuth paper [56] indicates, finding the canonical encoding of a tree only requires reordering the children of each tree node by their labels. So a baseline approach called "canonical extension" [56] is proposed which extends each canonically encoded pattern α with one more adjacent edge on its rightmost path. In other words, if we denote the frequent size-i pattern set by $\mathcal{F}_i$, then the canonical extension approach simply obtains $\mathcal{F}_{i+1}$ by "joining" $\mathcal{F}_i$ and $\mathcal{F}_1$, where "join" means that we are joining the matched transaction instances in their projected databases. This is also the pattern-extension approach followed by PrefixTreeSpan [59].

Sleuth joins the projected databases of two patterns $\mathcal{F}_i$ that share the same size-$(i-1)$ edges to obtain $\mathcal{F}_{i+1}$ and its projected database, which is more selective and thus efficient since two matched instances have to share $(i-1)$ edges to be joinable.

Unlike Apriori algorithms that explore the pattern-growth tree in breadth-first order, Sleuth still explores in depth-first order where each tree node represents the shared size-$(i-1)$ pattern "prefix." This allows it to be implemented in our PrefixFPM framework for easy parallelization. We next describe our PrefixFPM adaptions of PrefixTreeSpan and Sleuth.

6.1 Mining ordered subtree patterns in PrefixFPM

Recall that the data tree in Fig. 12 is encoded (by Sleuth) as BAB$D$$BC following preorder traversal that finally returns back to root B. To facilitate prefix projection, PrefixTreeSpan encodes this tree in a similar manner as shown in Fig. 13, where backtracking is encoded with "−1" which is like "$." However, PrefixTreeSpan lets each node be paired with a corresponding partner "−1" in the encoding so that the first B is now also paired with a "−1" at last. The part between a node and its partner is called the node's scope, which enables a quick checking of ancestor-descendant relationships. For example, in Fig. 12, after prefix projection by the pattern tree, the data tree gets split into a so-called "postfix-forest" with two trees, the node of which can be used to further extend the current pattern.

To see how this is achieved, PrefixTreeSpan requires the scanning of the data tree (i.e., its preorder encoding) to start from right after the position that matches the last node in the pattern subtree. For the example in Fig. 12, we should start from after "A" at the second position of the above encoding. Based on A's scope we can obtain the first tree in the projected forest as shown in Fig. 12, encoded as B-1D-1 which is hooked to Node 1 in the pattern tree (1 is encoded due to preorder traversal of the pattern). Continuing the scanning, we will obtain the second projected postfix-tree encoded as B-1C-1 which is hooked to Node 0 in the pattern tree.

We remark that by scanning the data tree encoding from the last matched position, we effectively extend a pattern along its rightmost path. Referring to Fig. 4, for example, we will not consider extending Node 2 since the last pattern node matched to a data tree is Node 3.

Note that we do not materialize the postfix-forest of a tree transaction T_i since it is implicitly maintained by the T_i's encoding. Next, we present the implementation of PrefixTreeSpan in PrefixFPM.

Transaction A transaction object here is simply an encoding array as illustrated in Fig. 13. Besides, we also maintain another array called *partner* also shown in Fig. 13. For a data-tree node encoded in the i-th position of the encoding array, $partner[i]$ keeps the position of its paired "−1" in the encoding array for getting the node's scope in $O(1)$ time. The *partner* array can be easily pre-constructed by scanning the encoding array.

Projected transaction Besides the ID of the matched tree transaction T_i, each projected transaction object also keeps an array M that saves the positions of previously matched nodes in the encoding array of T_i.

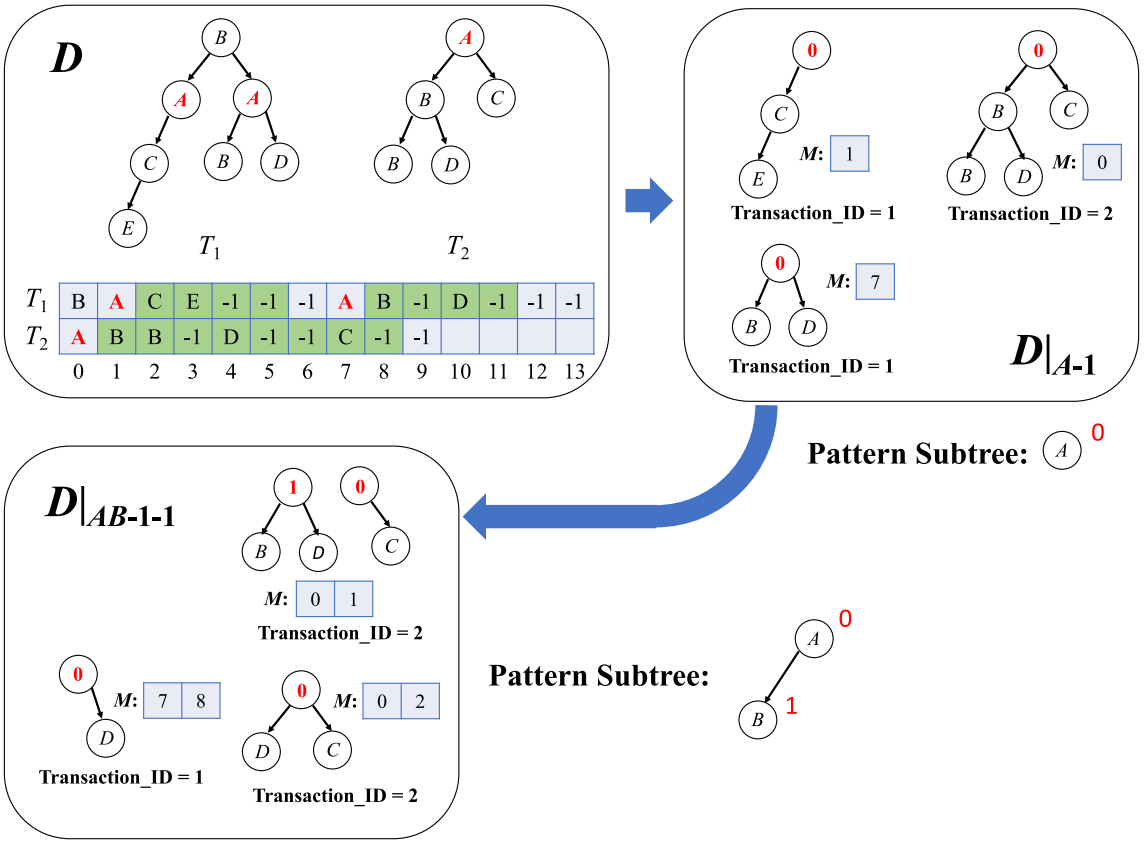


Fig. 14 PrefixTreeSpan prefix projection

Consider the example in Fig. 14 with two tree transactions, and a two-node subtree pattern encoded by α = AB-1-1 where if we code the tree nodes by preorder traversal, Node A (resp. B) in the pattern tree α are with ID 0 (resp. 1). Then, for the projected transaction that is matched to T_1, we have $M[0] = 7$ (resp. $M[1] = 8$) indicating that Node A (resp. B) in the pattern tree α are matched to T_1's data node at position 7 (resp. 8) in T_1's encoding array. Recall that pattern extension should continue from after position 8 of T_1's encoding array, and hence the projected transaction only has one child D left in postfix-tree.

Pattern A pattern object here maintains an encoded pattern tree α along with the projected database $D|_\alpha$. For example, Fig. 14 shows the pattern object for α = A-1 and α = AB-1-1.

Let us consider pattern tree α = A-1, and we have marked the three nodes with label A in D in red. Each of these three nodes generates a projected transaction in $D|_{\text{A-1}}$ with transaction ID along with an array M saving the position of match to A in the encoding array.

Task The most important UDF is *setChildren*(). Given a tree pattern α, we extend it by an edge (or, element) e that is represented by a pair $\langle pa, X\rangle$ where pa is the ID of the parent node in α, and X is the node label. For example, in Fig. 14, pattern AB-1-1 is obtained by extending pattern A-1 with edge $e = (0, B)$.

To extend a projected transaction $s|_\alpha$ in $D|_\alpha$ where $transaction\_id(s|_\alpha) = i$ by an edge, we only need to scan the encoding of T_i starting from right after the last matched position (i.e., the last element of array M of $s|_\alpha$).

For example, consider $D|_{\text{A-1}}$ in Fig. 14. Here, the first projected transaction of T_1 has the root node matched to $T_1[1]$ in the encoding, and it can be extended with $e = (0, C)$ or $(0, E)$. (Recall that we consider embedded patterns.) Similarly, the second projected transaction of T_1 has the root node matched to $T_1[7]$ in the encoding, and it can be extended with $e = (0, B)$ or $(0, D)$.

When scanning each element $T_i[o]$ in the postfix-forest of $s|_\alpha$, we can identify its parent node $p \in M$ in α by checking whether data node $T_i[M[p]]$'s scope contains o (i.e., p is o's ancestor) and p is the last such node in M; if so, we need to append the new projected transaction (extended by o) into $D|_\beta$ where β is the tree pattern extending α with $e = (p, T_i[o])$.

In UDF *setChildren*(), for each $s|_\alpha$ in $D|_\alpha$, (i) we conduct the above postfix-forest element scanning, where for each element o, we append the newly extended projected transac-

tion $s|_\beta$ to the projected database $D|_\beta$ that is maintained in $children[e]$.

Then, when all $s|_\alpha \in D|_\alpha$ are processed, (ii) we scan the table $children$ to remove the entry for every infrequent child-pattern β (i.e., $sup(\beta) < \tau_{sup}$). Note that there could be multiple projected transactions with the same transaction ID, $sup(\beta)$ only counts them once.

Worker UDF $setRoot(Q_{task})$, instead, (i) scans each tree transaction $T_i \in D$ using OpenMP parallel-for loop. For each tree node X at position j of T_i's encoding array, we create a projected transaction $s|_{\text{X-1}}$ with $M[0] = j$ and append $s|_{\text{X-1}}$ to $D|_{\text{X-1}}$ which is recorded by $children[X]$.

Then, when all $T_i \in D$ are processed, (ii) we scan the table $children$ to remove the entry for every infrequent tree pattern X-1 (i.e., label X occurs in less than τ_{sup} transactions). Besides, (iii) we also mark every infrequent label X so that in future task processing (e.g., the scan of the postfix-forest elements in projected transactions), elements with X are directly skipped.

6.2 Mining unordered subtree patterns in PrefixFPM

Sleuth [56] mines tree patterns that are frequent, node-labeled, rooted, unordered, and embedded. Since the order of children nodes does not matter, we need to prune non-canonical pattern encodings. Recall that in Fig. 2, only the first pattern T_1 is considered canonical.

To capture the ancestor–descendant relationship among nodes similarly as the scope-based encoding of PrefixTreeSpan, Sleuth assigns each node v an interval $[\ell, r]$ called v's *scope*, where ℓ is the rank of v in a preorder traversal of the tree, and r is the rank of the rightmost node in the subtree rooted at v (i.e., the largest node rank in the subtree). For example, for tree T_0 in Fig. 15, Node 0 has scope [0, 3], Node 1 has [1, 1], and Node 3 has [3, 3]. Then, the ancestor–descendant relationship can be judged by the scope containment relationship. For example, Nodes 1 and 3 are both the descendant of Node 0 since $[1, 1], [3, 3] \subset [0, 3]$, but Node 3 is not a child of Node 1 since $[1, 1] \cap [3, 3] = \emptyset$.

Since a pattern α can have multiple matches in a tree transaction, Sleuth represents each projected transaction in $D|_\alpha$ as a pair (*tid*, *scope*) where *tid* is the transaction ID of the tree transaction T_i whose subtree matches α, and *scope* is the scope of last matched node in T_i that matches the last extended node in pattern α (which is also α's rightmost node as edge extension is along the rightmost path). For example, Fig. 15 shows the vertical representation of initial patterns $\alpha =$ A, B, C, D, and E. The rectangle for pattern B, which is called its *scope list*, contains 3 matched instances in tree T_1, corresponding to Nodes 0, 2, and 4, respectively.

In Sect. 3, we saw that Sleuth encodes a tree T by adding vertex labels to the encoding in a depth-first preorder traversal of T, and by adding "$" whenever we backtrack from a child to its parent. This sequence encoding of T is also called its horizontal format as shown in Fig. 15. Sleuth adopts prefix projection to enumerate patterns by their horizontal encodings.

To avoid redundancy, a pattern is checked and extended only if its horizontal encoding is canonical. Recall from Sect. 3 that finding the canonical encoding of a tree only requires reordering the children of each tree node by their labels, so a simple approach to implement Sleuth is to follow PrefixTreeSpan's framework to grow patterns by one edge along the rightmost path, but with canonical pattern filtering, which is called *canonical extension* in [56].

Equivalence class-based extension Compared with *canonical extension*, [56] finds that another approach called *equivalence class-based extension* is more efficient. While *canonical extension* extends a pattern α with one edge, *equivalence class-based extension* generates a size-$(k+1)$ pattern from two size-k patterns that share the same size-$(k-1)$ prefix encoding. Obviously, the latter is more selective and thus reduces mining workloads.

This is where scope-lists come into play. Referring back to Fig. 4 again, we have a size-3 prefix encoding $P = CDA\$B$ (as there are 3 solid edges), from which we can grow size-4 patterns (i.e., using each of the 4 dashed edges). Let each dashed edge be denoted by (i, x) where i is the hooked node ID in P, and x is a node label. Let us denote the new pattern extended with (i, x) by $\beta = P_x^i$, then all $\{P_x^i\}$ constitute an equivalence class where patterns share the prefix P, denoted by $[P]$.

To build the equivalence class $[P_x^i]$ where patterns share the prefix P_x^i, we can extend P_x^i using another edge $(j, y) \in [P]$.

Sleuth keeps a projected database $D|_\beta$ for each $\beta = P_x^i$, which is represented as a scope list described before. To incrementally compute $D|_\gamma$ for the pattern γ obtained by extending P_x^i with (j, y), we can join the scope list of P_x^i with the scope list of every $P_y^j \in [P]$.

The idea of join is simple (see [56] for details): Two scopes $(tid_1, scope_1)$ and $(tid_2, scope_2)$ can be joined only if $tid_1 = tid_2$ (i.e., the match is from the same transaction T), the matched prefix occurrences (i.e., their node IDs in T which are also saved) are the same, and y's matched node in T is a descendant or cousin of x's matched node (need to check $scope_1$ and $scope_2$). Since we always order scope list items by tid, the joining of two scope lists requires only one pass over the two lists similar to the merge operation in merge sort.

However, the joined new patterns may not be canonical and such patterns need to be pruned. There are other details such as a smart algorithm for canonical form checking (illustrated in Fig. 6 in [56]), but these are implemented in [37]

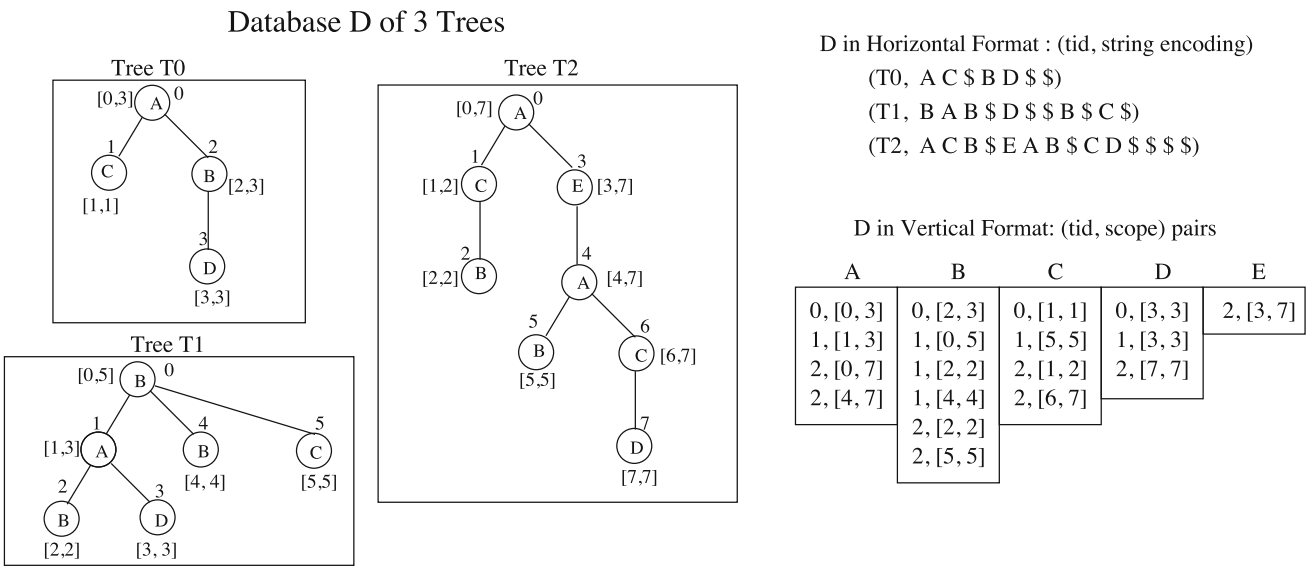


Fig. 15 Scope lists from Fig. 3 of [56]

already and we can directly reuse them when adapting the implementation to PrefixFPM. We omit these details and refer interested readers to [56].

TreeMiner TreeMiner [53] mines ordered trees like in PrefixTreeSpan rather than unordered ones, so it does not perform canonical pattern filtering. Otherwise, the algorithm is exactly the same as Sleuth, where equivalence class-based extension is adopted.

We next adapt the serial Sleuth implementation of [37] *to PrefixFPM.* We remark that unlike the other algorithms we describe where a *Pattern* object maintains just one pattern α and its projected database $D|_\alpha$, in Sleuth, a *Pattern* object now maintains a prefix encoding P along with a list of extending edges of the form $(i, x) \in [P]$, each associated with the scope list for $\beta = P_x^i$ (i.e., projected database $D|_\beta$). Note that a *Pattern* object now maintains multiple projected databases $\{D|_\beta\}$ instead of one $D|_\alpha$.

Transaction A transaction object here is simply implemented as an array that keeps the horizontal encoding of a tree transaction.

Projected transaction Refer to the vertical encodings in Fig. 15 again: For the singleton-node pattern $\alpha = A$, its scope list $D|_\alpha$ has 4 pairs where the first pair is $(tid, scope) = (0, [0, 3])$. In general, besides $(tid, scope)$, each projected transaction $s|_\alpha$ also keeps an array M tracking the prefix-matched nodes in T_{tid} just like in PrefixTreeSpan.

To summarize, a projected transaction object keeps three fields: (i) the transaction ID tid, (ii) the array M of prefix-matched nodes, and (iii) the scope which is an interval defining the range of the subtree rooted at the last-matched node (i.e., the last element in M).

Eqnode This class defines the edge (i, x) used to extend a prefix P. An *Eqnode* object maintains three fields: (i) the node i in prefix-tree P to hook the edge, (ii) the label x of the hooked node, and (iii) the scope list (or, projected database) $D|_\beta$ where $\beta = P_x^i$.

```
1: for each (i, x) ∈ [P]
2:     if P_x^i is not canonical:  continue
3:     L1 ← scope list of P_x^i
4:     for each (j, y) ∈ [P]
5:         if i < j: continue
6:         L2 ← scope list of P_y^j
7:         Q_y^j ← join(L1, L2)  // note that Q = P_x^i
8:         if Q_y^j is frequent:  children[Q].append(Q_y^j)
```

Fig. 16 Algorithm of *SleuthTask*::*setChildren*(.) in PrefixFPM

Pattern A pattern object here defines an equivalent class $[P]$ of patterns that share the same prefix P. It maintains two fields: (i) the prefix P and (ii) an array of *Eqnode* objects each of which is for one edge (i, x) and keeps $D|_\beta$ where $\beta = P_x^i$.

Task Recall that each task maintains a pattern object. Thus, UDF *setChildren*(.) is to compute every pattern object $[Q]$ where $Q = P_x^i$. (We use Q in replace of β for the notation to be consistent with P.)

Note that to compute a pattern object $[Q]$, we need to compute its array of *Eqnode* objects, or equivalently, the extending edges $(j, y) \in [Q]$ and their scope lists.

Figure 16 shows the algorithm for *setChildren*(.). Note that each children table entry $children[Q]$ to construct is a pattern object $[Q]$. Specifically, for each extending edge (i, x) in $[P]$ (Line 1), we check if $Q = P_x^i$ is canonical (Line 2). If not, Q is pruned; otherwise, we continue to build $[Q]$ to be added to $children[Q]$ (Line 8).

Specifically, Lines 4–7 join the scope list of Q with the scope list of every $P_y^j \in [P]$ to generate the scope list of the

new pattern Q_y^j, which are then appended to $children[Q]$ one by one (if Q_y^j is frequent which is judged using its scope list). After all $(j, y) \in P$ are processed (Line 4), $children[Q]$ now contains the complete $[Q]$ with all *Eqnode* objects $\{Q_y^j\}$ constructed.

UDF *get_next_child*(.) (recall Line 7 of Fig. 6) then wraps each $children[Q]$ into a task t_Q that processes $[Q]$ for further processing. Since t_Q needs to join the scope lists of $\{Q_y^j\}$ and join operations are merge-like and thus with a linear time complexity, we estimate the task computation cost of t_Q as the sum of the lengths of scope lists $\{Q_y^j\}$. In UDF *needSplit*(), we return *true* only if this sum is at least τ_{split} for divide and conquer.

Worker The UDF *setRoot*(.) first scans D to count label frequencies and remove infrequent node labels. Let the set of frequent labels be $\mathcal{F}_1$, the UDF then counts the frequencies of edges $e = (X, Y)$ with a counter array of size $|\mathcal{F}_1| \times |\mathcal{F}_1|$ by scanning D. Note that in subsequent edge extension, only frequent (labeled) edges (denoted by $\mathcal{F}_2$) will be considered. The UDF then builds the pattern object $[e]$ for each $e \in \mathcal{F}_2$ and wraps them as the set of initial tasks to be added to Q_{task}.

7 Subgraph pattern mining with PrefixFPM

This section first reviews the state-of-the-art gSpan [49] algorithm for mining frequent subgraph patterns from a graph database and then describes how we implement a parallel version of gSpan in PrefixFPM.

Finally, we wrap up this section with a description of the GrAph/Sequence/Tree extractiON (Gaston) algorithm [27] that aims a quickstart in frequent subgraph pattern mining by searching first for frequent paths, then frequent free trees, and finally, frequent cyclic graphs, followed by our parallelization of Gaston with PrefixFPM.

7.1 gSpan review

gSpan [49] considers a database of labeled graph transactions $D = \{G_i \mid i = 1, 2, \cdots, n\}$ where each vertex and each edge can have a label. For simplicity, we assume that each transaction G_i is an undirected and simple graph. The goal is to find all subgraph patterns whose support is at least τ_{sup}, where we say that G_i supports a subgraph pattern α if α is isomorphic to a subgraph of transaction G_i. Note that even if G_i has multiple subgraph instances that match α, it still only contributes 1 to $sup(\alpha)$. However, these subgraph instances should all be extended when generating the projected database of child-patterns.

To encode a graph G into a sequence for prefix projection, gSpan builds a so-called DFS code tree over G as follows: Depth-first search (DFS) is conducted on the nodes of G starting from some initial node, and each node is assigned an ID that equals the order that it is first visited in G by DFS. Figure 17a shows a graph G, and Fig. 17b–d shows 3 different DFS visits of G with different starting nodes and different edge selection orders.

For example, Fig. 17b shows a DFS visit of nodes v_0, $v_1, \ldots, v_4$. Edges (v_0, v_1), (v_1, v_2), (v_2, v_3), and (v_1, v_4) are traversed in order, and they are called *forward edges* which is defined as (v_i, v_j) with $i < j$. There are also two *backward edges* (v_2, v_0) and (v_3, v_1) which are not in the DFS code tree.

gSpan grows subgraph patterns by edges, so each graph G is encoded into an edge sequence. To define a unique mapping from G to its edge sequence given a DFS code tree T, gSpan orders edges such that whenever we reach a vertex v_i during DFS, we first traverse all backward edges starting from v_i before moving forward with (v_i, v_{i+1}). Moreover, for two backward edges (v_i, v_j) and (v_i, v_k), the former is before the latter if $k < j$. For example, Fig. 17b gives the edge sequence (v_0, v_1), (v_1, v_2), (v_2, v_0), (v_2, v_3), (v_3, v_1), (v_1, v_4). In general, given any two edges (v_{i_1}, v_{j_1}) and (v_{i_2}, v_{j_2}), the previous edge total order can be defined based on the value comparison of i_1, i_2, j_1 and j_2 (see Theorem 1 of [17]).

We have not considered labels so far. If we incorporate labels, each edge (v_i, v_j) can be expanded into a quintuple $(i, j, L(v_i), L(v_i, v_j), L(v_j))$ where $L(.)$ means the label. This will expand the previous sequence for Fig. 17b into $(0, 1, X, a, Y), (1, 2, Y, b, X), \cdots$. Such a sequence of edge-quintuples is called the DFS code of G given DFS code tree T.

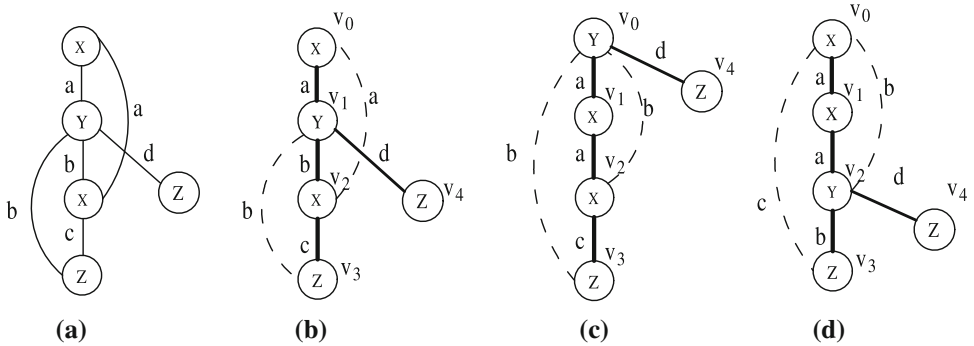


Fig. 17 DFS tree and its forward/backward edges [17]

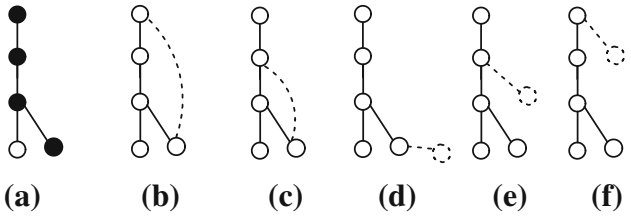


Fig. 18 gSpan pattern growth [17]

Note that a graph G can have many different DFS codes: For the same graph in Fig. 17a, c gives another DFS code $(0, 1, Y, a, X)$, $(1, 2, X, a, X)$, $\cdots$. We can compare different DFS codes in lexicographic order, where each element (i.e., edge-quintuple) is also compared in lexicographic order. For example, for the previous two DFS codes, since $X < Y$, we have $(0, 1, \underline{X}, a, Y) < (0, 1, \underline{Y}, a, X)$, and since the first element already breaks the tie, DFS code $(0, 1, X, a, Y)$, $(1, 2, Y, b, X)$, $\cdots$ is smaller than $(0, 1, Y, a, X)$, $(1, 2, X, a, X)$, $\cdots$.

So far, we have defined a way of encoding a graph G given a DFS code tree T of it. Given a subgraph pattern α, different DFS orders over α generate different DFS codes and we define the minimum DFS code among them as the canonical encoding of α. This allows prefix projection to be applicable by enumerating DFS codes representing subgraph patterns, but to avoid redundancy, we only check and extend a pattern α if its DFS code is canonical. (Recall Fig. 3.)

Due to the DFS nature of a DFS code tree, to grow the DFS code of pattern β from the DFS code of pattern α by adding one adjacent edge e, e must grow from vertices on the rightmost path of α's DFS code tree (to avoid pattern redundancy). As Fig. 18b–c shows, backward edges can only extend from the rightmost vertex, while Fig. 18d–f shows that forward edges can extend from any vertex on the rightmost path.

7.2 gSpan implementation in PrefixFPM

We next adapt the serial gSpan implementation of [16] *to PrefixFPM.* We remark that the previous review of gSpan is kept simple and there are actually a lot of additional algorithmic details, such as an enumerating engine to find all projected instances $G_i|_\beta$ from $G_i|_\alpha$ (in $D|_\alpha$) by edge backtracking in G_i, and the pruning rules in Sect. 5 of [17]. However, we can directly use their implementation in [16] and so only details relevant to adaption were reviewed.

Transaction A transaction object here keeps a graph G_i as a list of labeled vertices, each of which maintains an adjacency list of labeled adjacent edges.

Projected transaction Given a pattern α with i edges ordered as $e_1, e_2, \ldots, e_i$, a projected transaction $G_i|_\alpha$ that projects α over G_i keeps an array M of edge pointers, each pointing to a matching edge in G_i. Specifically, $M[i]$ points to the matching edge of e_i in G_i.

Pattern A pattern object keeps the pattern α as a sequence of DFS codes (i.e., edge-quintuples), as well as the projected database $D|_\alpha$. The DFS code sequence contains the whole information of a pattern graph α, so we can recover graph α for printing. Note that $D|_\alpha$ may contain multiple projections $G_i|_\alpha$ (i.e., subgraph instances of G_i that match α), but they should count only once to α's support.

In contrast, since every instance needs to be edge-extended when building child-patterns' projected databases $\{D|_\beta\}$, we should compare the size of $D|_\alpha$ (i.e., every instance counts) with τ_{split} in UDF *Task*::*needSplit*() to decide whether to process a task t_β itself or to add it to Q_{task}.

Task The UDF *setChildren*(.) extends each instance $G_i|_\alpha$ by checking the potential edges in G_i to extend α as shown in Fig. 18. If such an edge e (on the new rightmost path) is found in G_i, we append the extended instance to table entry $children[e]$ (which keeps $D|_\beta$ where child-pattern β is obtained by extending α with e). Note that the children table's entry keys depend on the edges in the matched transactions, so no entry is created if such an edge (v_i, a, v_j) (i.e., $L(v_i, v_j) = a$) does not exist in the transactions.

Finally, for each children table entry $\langle e, D|_\beta \rangle$ we check if β is frequent (computed using $D|_\beta$), and infrequent child-patterns are removed from the table.

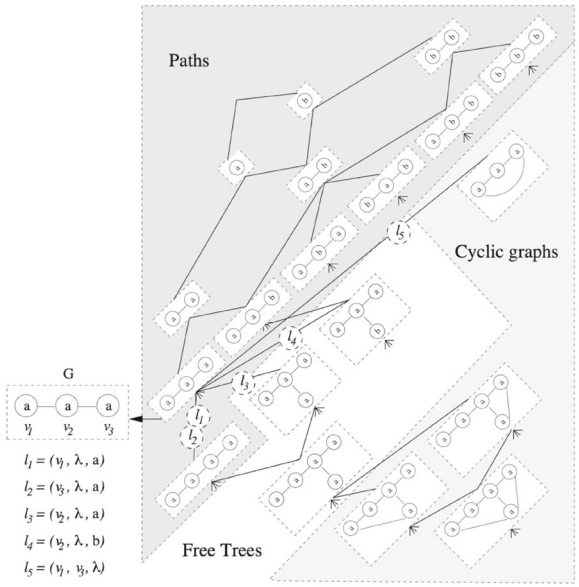
In UDF *get_next_child*(.), we then create a task from a children table entry $\langle e, D|_\beta \rangle$ (where β is guaranteed to be frequent) by constructing β's DFS code sequence. Given this sequence, we can decide if it is canonical. A task t_β is created only if the β is canonical.

Worker The UDF *setRoot*(Q_{task}) scans D to collect unique labeled edges (X, a, Y) and their supports. The set of frequent edges are found as $\mathcal{F}$. Another pass over D then removes all infrequent edges (i.e., not in $\mathcal{F}$) from the graph transactions. It also handles other details like giving each edge a unique ID so that later, a visited edge can be marked without repeated edge growth.

The UDF *setRoot*(.) then scans every $G_i \in D$, and for every vertex $v \in G_i$, it inserts every forward edge $e = (X, a, Y)$ originating from v into a root-level entry $children[(X, a, Y)]$ that keeps $D|_e$. Note that the processing is done by parallel for-loop. Finally, each table entry generates a root-level task t_e that is added to Q_{task} to initiate the mining.

7.3 Gaston and its PrefixFPM implementation

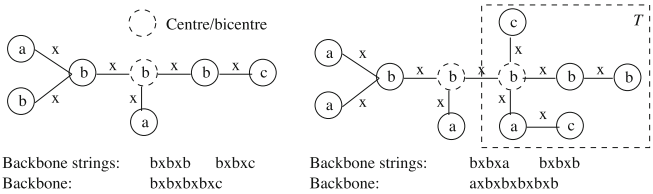
As [28] indicates, *experiments with molecular databases reveal that the largest numbers of frequent substructures in such databases are actually free trees.* Simpler structures such as paths and free trees can be mined with more efficient algorithms compared with subgraphs, which motivates the



(a) Pattern Growth

DFGraphMiner (A graph code C, a leg l, a set of legs L)
(1) $C' := \rho(C, l)$
(2) **if** C' is not canonical **then return**
(3) output graph C'
(4) $L' := \{l' | l'$ is a necessary leg of $C', support(\rho(C', l'), D) \geq minsup,$
$l' \in L$, or l' connects to the node introduced by l,
if l is a node refinement $\}$
(5) **for all** $l' \in L'$ **do** DFGraphMiner (C', l', L')

(b) Depth-First Graph Mining Workflow



(c) Free Trees, (Bi)centers and Backbones

Fig. 19 Gaston algorithm overview (figures are directly obtained from [27] and [28])

Gaston algorithm that divides the frequent graph discovery process into phases, from paths to free trees, and finally to subgraphs. In each phase, a dedicated algorithm is used to mine the respective patterns only to the extent of algorithm complexity needed for such patterns, so simpler patterns can be mined much more efficiently, achieving an overall performance much better than gSpan.

Studying Gaston is interesting since it unifies all the applications that we discussed so far: frequent sequence (aka. path), subtree, and subgraph pattern mining, into one mining algorithm. It is also easy to parallelize since the serial code of Gaston is available at [15]. The details of the entire algorithm are complicated and require much space to present, so this section focuses on a high-level overview of the Gaston algorithmic workflow.

Gaston algorithm overview As shown in Fig. 19a, b, Gaston grows patterns by one edge at a time, similar to gSpan. In Gaston terminology, there are two types of edge extensions:

- **Node refinement**, where a subgraph G is extended with a new (forward) edge $e = (u, v)$ with u being an existing vertex in G and v being a new vertex.
- **Cycle closing refinement**, where a subgraph G is extended with a (backward) edge $e = (u, v)$ with both u and v being existing vertices in G.

The extending edges are called legs, as illustrated by l_1, l_2, l_3, and l_4 in Fig. 19a. Figure 19b provides a simplified view of the depth-first graph mining algorithm, where in Line 1, the refinement operator $\rho(G, l)$ refines a graph G by adding leg l, and Line 2 skips non-canonical patterns to avoid redundancy. Line 4 then computes leg candidates for child-pattern extension in Line 5: Specifically, only frequent candidate legs from the previous parent-pattern and those connecting to the lastly added vertex are considered as leg candidates for pattern extension.

A pattern is grown in 3 phases, initially as paths, then free trees (i.e., unrooted trees), and finally cyclic graphs. (Note that paths and free trees are acyclic.) This allows simpler patterns to use more efficient algorithms for pattern growth. The algorithms are similar to those we have seen so far with some necessary changes.

For example, a path can be regarded as a sequence only when the path orientation is determined, such as $axaxb$ and $bxaxa$. (a, b are vertex labels, and x is an edge label.) To avoid generating this path from both axb (by leg $l_1 = bxa$ on the left) and bxa (by leg $l_2 = axb$ on the right), Gaston only allows the lexicographically smaller leg l_2 to extend.

As another example, we previously only considered rooted trees, but a path or free tree can be extended by node refinement into another free tree, and it is important to determine the pattern canonicality of a free tree. As Fig. 19c shows, Gaston converts a free tree into one rooted tree (aka. a centered tree), or two rooted trees with roots connected (aka. a bicentered tree). This is utilizing the property that one can use the center(s) of a free-tree path with maximal length to make a free tree uniquely rooted. As a result, a backbone string can be defined by concatenating the two lexicographically lowest paths from the root(s). Gaston restricts that free trees can only grow from a free tree with the same backbone, and

that all free trees of a backbone grow from the path that corresponds to that backbone. Gaston utilizes a duplication-free enumeration approach to avoid generating redundant patterns for canonicality checking, which makes it more efficient than gSpan for free trees.

Gaston implementation on PrefixFPM As we can see from Fig. 19b, the mining procedure of Gaston directly follows the pattern-growth procedure that PrefixFPM requires for parallel adaptation, so we hereby only briefly discuss how we adapted the serial code [15].

Specifically, even though Fig. 19b provides a unified algorithmic workflow, the actual data structures and algorithms used in each stage are different; therefore, we maintain the stage number as a field in a *Pattern* object which is used by the UDFs of our Gaston's *Task* subclass to branch to the appropriate operations.

As Fig. 19a shows, a path is allowed to be extended by a node refinement into either a path or a free tree, and by a cycle closing refinement into a cycle (which is a graph). Similarly, a free tree is allowed to be extended by a node refinement into another free tree, and by a cycle closing refinement into a graph; finally, a graph is allowed to only be extended by a cycle closing refinement into another graph.

For projected databases, we follow Gaston to keep for each pattern an embedding list that keeps the pattern embeddings (i.e., projected transactions) of the input graph transactions. The serial code [15] keeps embeddings incrementally, where each projected transaction (i.e., embedding) only keeps the matching information for the last leg, followed by a pointer to the embedding of the parent-pattern, so that the projected databases of all predecessor patterns are needed to recover the information of an embedding. While this approach is space-efficient, a task t_α in PrefixFPM generates child-tasks and may then be released before the computation of a child-task t_β, so we cannot assume that t_β is able to access the projected databases of the predecessor patterns. We, therefore, materialize the embeddings into the projected database of every child-pattern instead.

8 Experiments

We evaluate the performance of PrefixFPM using the 7 FPM applications described in Sects. 5, 6, and 7, i.e., the parallel versions of PrefixSpan [31], CloSpan [48], Sleuth [56], TreeMiner [53], PrefixTreeSpan [59], gSpan [49], and Gaston [27] on top of PrefixFPM. *For ease of presentation, we refer to these PrefixFPM algorithms by directly using the names of their serial algorithms.*

The PrefixFPM framework and the 6 applications on top are open-sourced at:

https://github.com/wenwenQu/PrefixFPM

Table 1 Sequence datasets

Dataset	#{transactions}	#{labels}	AVG seq length
S10	16,000,000	10,000	10
S50	400,000	10,000	50
Plan	202,071	366	199.3

To thoroughly test the scale-up capability of PrefixFPM, we ran our PrefixFPM programs on the BlueBlaze server donated by IBM to the Department of Computer Science, the University of Alabama at Birmingham, which is equipped with 5 IBM POWER8 CPU (32 cores, 3491 MHz), 1 TB RAM, and 2TB HDD. Since there are $5 \times 32 = 160$ cores, we are able to thoroughly test the scalability with 1, 2, 4, 8, 16, 32, 64, and 128 cores. Every experiment was repeated for 5 times, and the reported results were averaged over the 5 runs; we find that the difference of different runs is small and consistently within 5%.

For each dataset, we set the frequency threshold τ_{sup} such that running with 16 computing threads takes a few minutes. This is because the mining time grows quickly as τ_{sup} decreases, and our focus is on evaluating the multicore speedup ratio and we want to keep the total running time of each experiment tractable.

Recall that a task t_α of PrefixFPM puts a child-task t_β to Q_{task} rather than processes t_β by itself if $|D|_\alpha| > \tau_{split}$, and meanwhile, if a task runs for more than τ_{time} seconds, a timeout happens and all remaining tasks are added to Q_{task}. Unless otherwise stated, we use the default setting where $\tau_{split} = 100$ and $\tau_{time} = 0.1$ second. We will also present experiments that tune the two parameters to justify our default choice later in this section.

The rest of this section first describes the scalability results of each of our 7 applications presented in order, followed by the study of the effects of τ_{split} and τ_{time}, and finally by a comparison with existing distributed solutions on Apache Hadoop and Spark.

8.1 PrefixSpan scalability on PrefixFPM

Data For sequential pattern mining, we use two synthetic sequence datasets and one real sequence dataset for experiments. Table 1 summarizes the 3 datasets.

The first two datasets are generated using the sequence generator of IBM Synthetic Data Generator [20], which mimics real-world transactions where people buy a sequence of items. We generate two datasets, one with 400,000 sequences of average length 50, denoted by *S50*; the other with 16,000,000 sequences of average length 10, denoted by *S10*. We use the default value of 10,000 by [55] for the number of labels. Note that the mining time grows quickly with

Table 2 Scalability results of PrefixSpan

#{cores}	S10 (s)	Speedup	RAM (GB)	S50 (s)	Speedup	RAM (GB)	PLAN (s)	Speedup	RAM (GB)
Serial	1335.96	–	3.0	1455.60	–	0.4	387.56	–	1.2
1	1,447.87	1	3.1	1,646.77	1	0.4	434.06	1	1.0
2	739.74	1.96	3.2	830.73	1.98	0.4	229.50	1.89	1.2
4	389.55	3.72	3.2	419.36	3.93	0.4	117.53	3.69	1.7
8	211.86	6.83	4.2	211.44	7.79	0.4	84.22	5.15	1.7
16	119.13	12.15	4.2	109.39	15.05	0.4	40.24	10.79	2.0
32	85.11	17.01	4.2	71.04	23.18	0.6	25.54	16.99	2.5
64	68.64	21.09	5.2	56.84	28.97	0.8	18.86	23.02	2.9
128	64.61	22.41	5.2	48.22	34.15	1.4	17.71	24.51	3.8

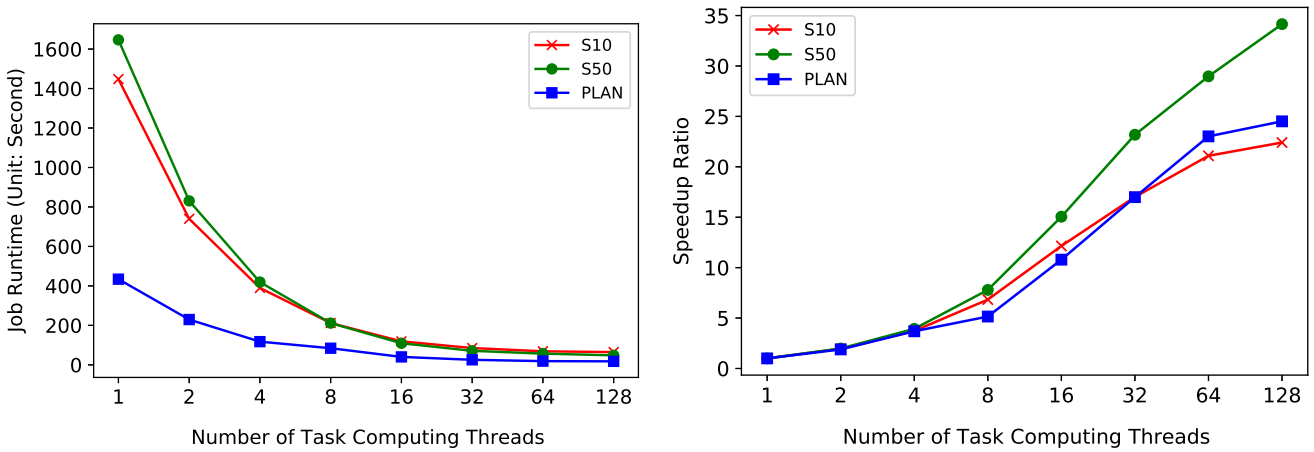


Fig. 20 Scalability of PrefixSpan

sequence length but linearly with the number of sequences, so we let *S10* have many more sequences than *S50*, while *S50* is used to test the performance with long sequences.

One real dataset, denoted by *Plan*, is from [33], which is obtained from a planning domain. The input consists of a database of plans for evacuating people from one city to another. Each plan represents a sequence of events, and the dataset only contains 202,071 bad plans that lead to failure.

Scalability results We adapted the serial PrefixSpan implementation of [34] to PrefixFPM for parallel mining. For all the three datasets, we fix $\tau_{sup} = 50$ in this set of experiments.

Table 2 and Fig. 20 show the scalability results on our three datasets with an increasing number of task computing threads. Specifically, Table 2 reports the running time, speedup ratio and peak memory consumption, while Fig. 20 plots the running time and speedup ratio curves. Moreover, the first row of Table 2 also reports the results when running the serial program of [34] as a comparison, the performance of which is similar to running our PrefixFPM program with a single computing thread. We can see that the speedup ratio is very good for up to 16 computing threads and keeps improving significantly with more threads.

Also, even though using more threads consumes more memory, the space does not increase much as the number of threads increases. This is because the sequence transactions occupy most of the space, and the space consumed by the tasks' projected databases is limited since each projected transaction only keeps the position of the last match and the transaction ID (recall Sect. 5.1) rather than the actual sequence.

The speedup is better on *S50* and *Plan* than on *S10*, because *S50* and *Plan* have much longer sequences that allow the search space tree to grow deeper, giving more task parallelism opportunities.

8.2 CloSpan scalability on PrefixFPM

We also use the three datasets in Table 1 to evaluate the scalability of CloSpan which finds closed frequent sequential patterns.

We adapted the serial CloSpan code from [7] to PrefixSpan. The serial CloSpan code of [7] supports sequence elements that are itemsets, and for this purpose, it has to extend every projected transaction of a transaction $s \in D$

Table 3 Scalability results of CloSpan

#{cores}	S10 (s)	Speedup	RAM (GB)	S50 (s)	Speedup	RAM (GB)	PLAN (s)	Speedup	RAM (GB)
Serial	3258.69	–	1.8	1736.94	–	0.5	2147.42	–	0.9
1	3301.20	1	2.5	1775.45	1	0.4	2609.16	1	2.7
2	1807.10	1.83	2.7	909.32	1.95	0.4	1184.45	2.20	5.3
4	943.14	3.50	3.1	509.03	3.49	0.7	645.53	4.04	7.8
8	604.71	5.46	3.7	329.06	5.40	1.5	377.13	6.92	16.8
16	421.26	7.84	5.1	283.90	6.25	2.8	286.17	9.12	29.8
32	371.04	8.90	5.2	299.04	5.94	4.2	243.62	10.71	57.2
64	418.70	7.88	12.9	313.39	5.67	11.0	204.86	12.74	100.5
128	435.50	7.58	25.4	339.62	5.23	21.5	217.65	11.99	221.7

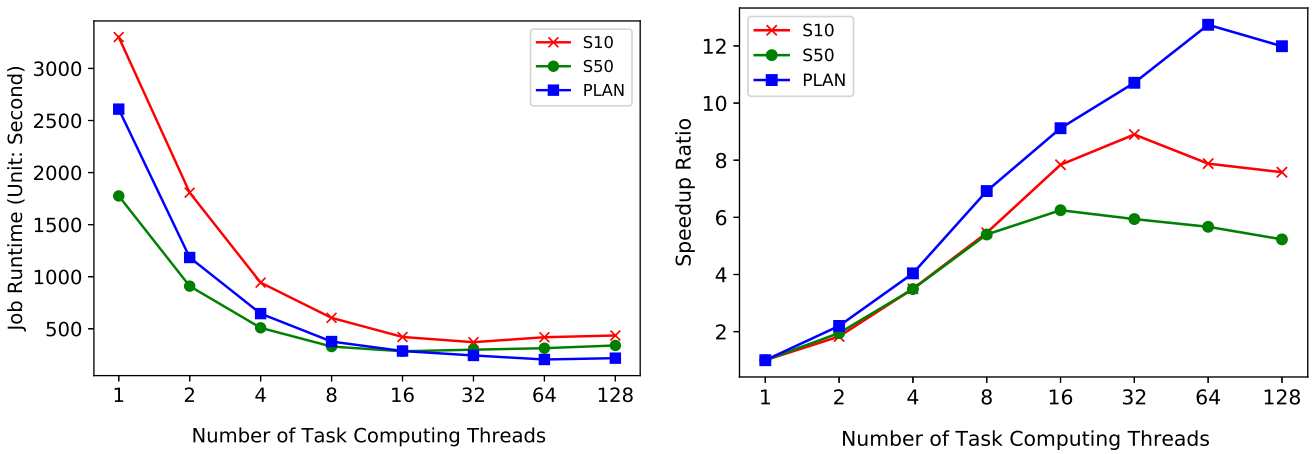


Fig. 21 Scalability of CloSpan

rather than only the minimal (i.e., earliest) prefix match as in our PrefixSpan program.

Surprisingly, this small difference creates a lot of mining overheads, making the CloSpan program much more time-consuming. Therefore, we use larger support thresholds to avoid a very long running time. For *S50* and *Plan*, we use $\tau_{sup} = 1000$, and for *S10*, we use $\tau_{sup} = 5000$.

Table 3 and Fig. 21 show the scalability results on our three datasets with an increasing number of task computing threads. Specifically, Table 3 reports the running time, speedup ratio, and peak memory consumption, while Fig. 21 plots the running time and speedup ratio curves. Moreover, the first row of Table 3 also reports the results when running the serial program of [7] as a comparison, the performance of which is similar to running our PrefixFPM program with a single computing thread. We can see that the speedup ratio is good for only up to 4 computing threads with further noticeable improvement all the way up to 16 computing threads, and there is very limited performance improvement (if not degradation) beyond 16 threads.

The poor scalability when the number of CPU cores becomes large is due to the lock contention on a shared hash table for early termination rule checking, as we have described in Sect. 5.2. In fact, the performance degrades on all three datasets when the number of cores increases from 64 to 128 due to the big overheads for hash table contention. On the other hand, the memory consumption does increase with the number of threads since more tasks and hence projected databases are maintained, but the memory v.s. CPU cores (i.e. threads) ratio is reasonable, bounded by 1.73 for *PLAN* when using 128 cores, while being one order of magnitude smaller on the other 2 datasets.

Also, *S10* and *Plan* have more transactions than *S50*, and so, *S10* and *Plan* tend to have more frequent patterns, and hence, their speedup ratios are better.

8.3 Sleuth scalability on PrefixFPM

Data For subtree pattern mining, we use one synthetic tree dataset and one real tree dataset for experiments. Table 4 summarizes the two datasets.

TreeGen is a synthetic tree transaction database generated using a synthetic data generator [40] that creates a database of artificial website browsing behavior: A website browsing

Table 4 Tree datasets

Dataset	#{trees}	#{labels}	AVG node fan-out	AVG non-leaf fan-out
TreeGen	10,000,000	10	0.6	1.3
TreeBank	52,581	189	1	2.2

Table 5 Scalability results of Sleuth

#{cores}	TreeGen (s)	Speedup	RAM (GB)	TreeBank (s)	Speedup	RAM (GB)
Serial	2689.49	–	6.4	1667.54	–	7.6
1	2175.41	1	12.2	1409.43	1	5.8
2	1228.36	1.77	72.2	863.35	1.63	8.7
4	701.48	3.10	124.2	542.62	2.60	11.8
8	390.11	5.58	133.2	540.12	2.61	13.9
16	224.20	9.70	135.2	521.89	2.70	14.5
32	157.10	13.85	143.2	539.13	2.61	15.5
64	143.64	15.14	145.2	535.24	2.63	14.1
128	151.36	14.37	145.2	545.00	2.59	16.0

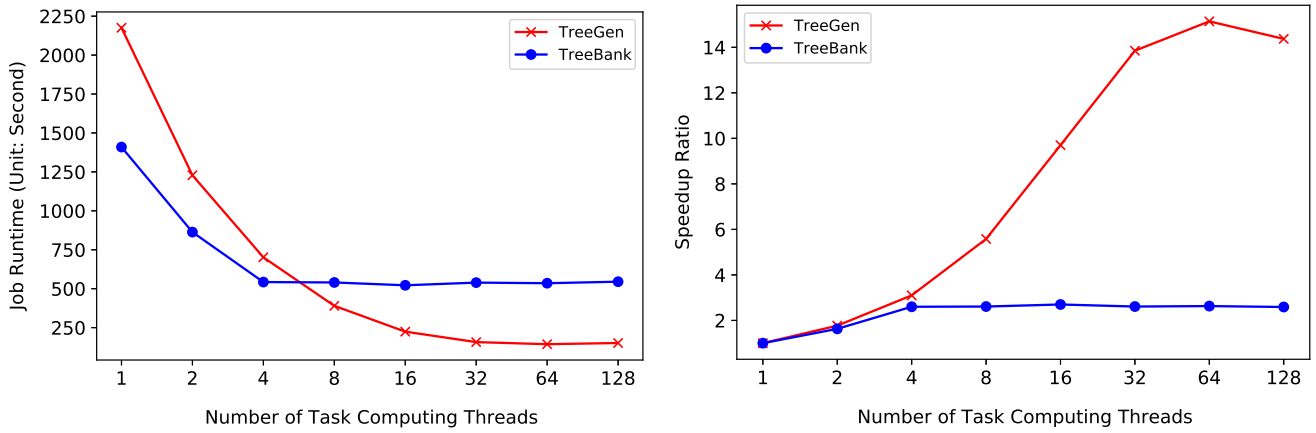


Fig. 22 Scalability of Sleuth

"master tree" is first created based on parameters supplied by the users; then, one can generate random subtrees of the master tree as the tree transactions for mining. The details of data generation can be found in [53].

We use the default parameters for master tree: depth = 5, fan-out factor = 5, number of labels = 10, and we set the number of nodes in the master tree as 50 to generate 10,000,000 subtree transactions.

The other dataset, *TreeBank* [41], is a real XML dataset derived from computation linguistics. It models the syntactic structure of English text and provides a hierarchical representation of the sentences in the text by breaking them into syntactic units based on part of speech. The dataset is deep and comprises highly recursive and irregular structures.

Scalability results We adapted the serial Sleuth implementation of [37] to PrefixFPM for the parallel mining of unordered tree patterns. For *TreeGen*, we use $\tau_{sup} = 50$, and for *TreeBank*, we use $\tau_{sup} = 30{,}000$.

Table 5 and Fig. 22 show the scalability results on our datasets with an increasing number of task computing threads. Specifically, Table 5 reports the running time, speedup ratio, and peak memory consumption, while Fig. 22 plots the running time and speedup ratio curves. Moreover, the first row of Table 5 also reports the results when running the serial program of [37] as a comparison, the performance of which is similar to running our PrefixFPM program with a single computing thread. (PrefixFPM with 1 thread is even faster since we avoided some unnecessary deep-copy overheads in [53]'s implementation.) We can see that the speedup ratio is good for up to 4 computing threads, but the improvement beyond 16 cores is not significant. This is because Sleuth adopts equivalence class-based extension, where each task is coarse-grained and handles an equivalence class $[P]$ of multiple patterns P_x^i each with its own projected database (i.e., scope list). This coarse task granularity limits the potential of parallelism.

Table 6 Scalability results on *TreeGen*

#{cores}	TreeMiner (s)	Speedup	RAM (GB)	PrefixTreeSpan (s)	Speedup	RAM (GB)
Serial	699.66	–	4.8	–	–	–
1	746.24	1	7.9	2877.87	1	5.9
2	405.97	1.84	8.9	1632.68	1.76	5.7
4	228.59	3.26	11.8	890.28	3.23	8.3
8	134.49	5.55	14.6	468.08	6.15	11.9
16	91.18	8.18	20.4	259.35	11.1	16.3
32	79.79	9.35	16.1	187.68	15.33	21.3
64	88.02	8.48	15.3	137.30	20.96	19.5
128	96.86	7.7	16.1	119.76	24.03	18.6

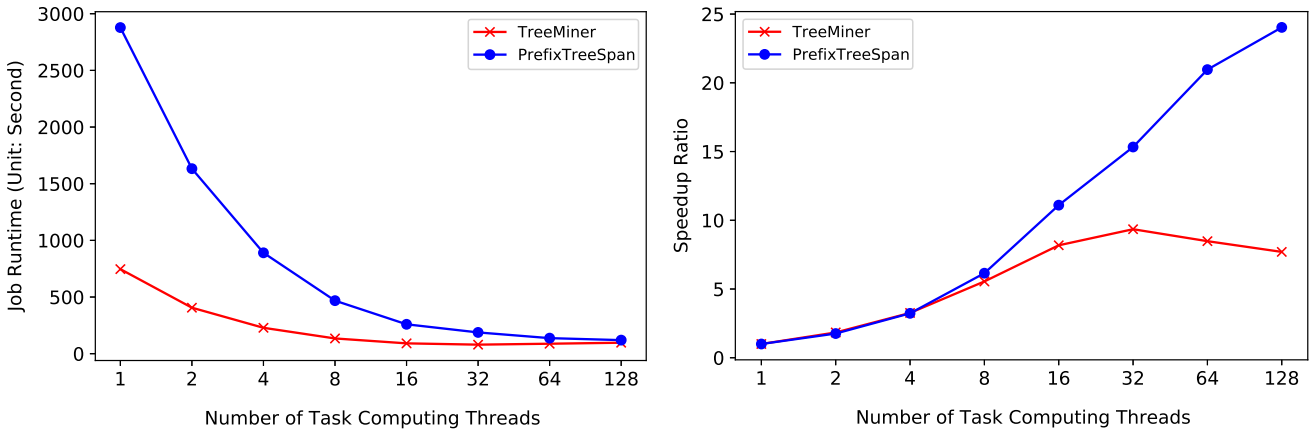


Fig. 23 Scalability on *TreeGen*

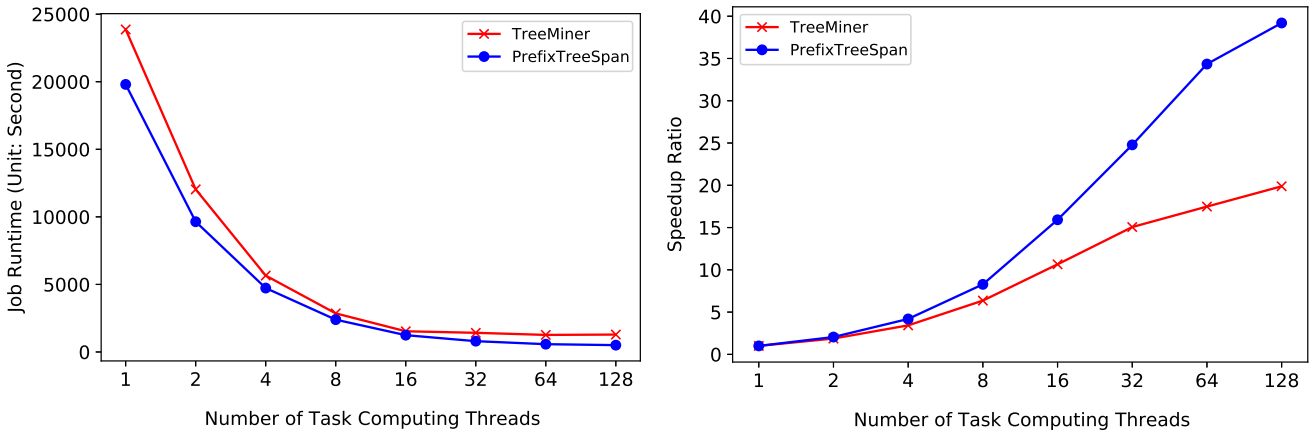


Fig. 24 Scalability on *TreeBank*

In fact, Fig. 22 shows that Sleuth's performance saturates on *TreeBank* with merely 4 CPU cores. This is because with $\tau_{sup} = 30{,}000$, we find that the mining focuses only on several branches in the pattern-growth tree, leaving only 2–4 tasks in Q_{task} for most of the time, hence leading to poor speedup. We also tried to reduce τ_{sup} to 10,000 and found that there are enough tasks in Q_{task} to keep computing threads busy, but since there are now many more frequent patterns each requiring an expensive canonical encoding judgement, the running time is too long (> 24 h) and thus we have to cut the execution of the program.

However, as we shall see in the next subsection, TreeMiner is able to finish with $\tau_{sup} = 10{,}000$ since it is mining ordered subtree patterns and thus canonical encoding judgement is not needed.

Table 7 Scalability results on *TreeBank*

#{cores}	TreeMiner (s)	Speedup	RAM (GB)	PrefixTreeSpan (s)	Speedup	RAM (GB)
Serial	24,937	–	3.8	–	–	–
1	23,870	1	11.5	19,802	1	2.6
2	12,034	1.87	18.2	9640.55	2.05	4.2
4	5654.08	3.43	21.4	4729.50	4.19	6.7
8	2856.68	6.36	25.1	2391.62	8.28	9.3
16	1532.53	10.65	34.4	1243.89	15.92	15.5
32	1418.09	15.06	49.6	799.09	24.78	26.3
64	1262.75	17.48	76.7	576.68	34.34	43.7
128	1287.82	19.88	67.6	505.15	39.20	79.3

8.4 Scalability of TreeMiner and PrefixTreeSpan

We also use the two datasets in Table 4 to evaluate the scalability of TreeMiner and PrefixTreeSpan, both of which find ordered subtree patterns. For *TreeGen*, we use $\tau_{sup} = 50$, and for *TreeBank*, we use $\tau_{sup} = 10{,}000$. We adapted the serial TreeMiner implementation of [42] to PrefixFPM, but since PrefixTreeSpan does not have a code release, we directly implemented our PrefixFPM version of it according to the algorithm description in [59]. For *TreeGen*, we use $\tau_{sup} = 50$, and for *TreeBank*, we use $\tau_{sup} = 30{,}000$.

Recall that in PrefixTreeSpan, each task is associated with one subtree pattern, while TreeMiner is like Sleuth, where each task is associated with an equivalent class $[P]$ of multiple patterns that share the prefix P.

Scalability on TreeGen Table 6 and Fig. 23 show the scalability results on our datasets with an increasing number of task computing threads. Specifically, Table 6 reports the running time, speedup ratio, and peak memory consumption, while Fig. 23 plots the running time and speedup ratio curves. Moreover, the first row of Table 6 also reports the results when running the serial TreeMiner program of [42] as a comparison, the performance of which is similar to running our PrefixFPM program for TreeMiner with a single computing thread. We can see that *TreeMiner* is consistently faster than *PrefixTreeSpan* which is because *TreeMiner* joins two length-k projected transactions to generate a length-$(k+1)$ projected transaction, which is more selective than *PrefixTreeSpan*'s growth by frequent edges and hence has much less mining workloads. On the contrary, the speedup ratio is good for *PrefixTreeSpan* but not good for *TreeMiner*, which is because *TreeMiner* has a larger task granularity which reduces the opportunity for parallelism.

Scalability on TreeBank Table 7 and Fig. 24 show the scalability results on our datasets with an increasing number of task computing threads. Surprisingly, *PrefixTreeSpan* is consistently faster than *TreeMiner* and, meanwhile, achieves a higher speedup ratio. After looking into the mining process, we find that the projected databases (i.e., scope lists) of patterns in *TreeBank* are big, making the scope-list join operations very expensive, which slows down *TreeMiner* significantly.

Also, note that *TreeMiner* can now finish the mining in a reasonable amount of time even though we use $\tau_{sup} = 10{,}000$, which was intractable in Sleuth due to the expensive canonical encoding checking that is not needed by *TreeMiner*.

Table 8 Graph datasets

Dataset	#{graphs}	AVG|E|	AVG|V|
GraphGen	1000,000	50	50
OpenNCI	265,242	41.76	40.48
Enamine	735,147	21.81	22.93
DBLP	317,080	7.62	27.67
Yeast	79,601	21.54	22.84

8.5 gSpan Scalability on PrefixFPM

Data For subgraph pattern mining, we use one synthetic graph dataset and four real graph datasets for experiments. Table 8 summarizes the 5 datasets.

GraphGen is a synthetic graph database generated as follows: We first randomly generate a master graph with 100 vertices and 100 edges, where the label of each vertex is randomly sampled from $0, 1, \cdots, 9$; we then create graph transactions by sampling each edge with a probability of 50%. Altogether 1,000,000 graph transactions are generated by this edge sampling method. The use of a master graph allows the graph database to have patterns rather than being totally random.

Four real graph datasets are also shown in Table 8, including three real-world molecular structure datasets *OpenNCI* [26], *Enamine* [11], and *Yeast* [52], and a graph database *DBLP* of author ego-networks built according to the procedures mentioned in [2]. Compared with [2], we use a more up-to-date DBLP collaboration network from [9] consider-

ing publications in the year range 1992–2012 to build *DBLP*. Each graph in *DBLP* is the ego-network of a distinct author u; vertices in this graph are u and his/her collaborators, and an edge between a pair of vertices represents one or more co-authorship events between the corresponding authors. Mining frequent subgraphs from this dataset is interesting, because it generates subgraph patterns that correspond to a group of authors who collaborate more frequently.

Scalability results We adapted the serial gSpan implementation of [16] to PrefixFPM. We choose τ_{sup} for each graph to avoid very long execution time as follows: $\tau_{sup} = 50$ for *GraphGen*, $\tau_{sup} = 60{,}000$ for *OpenNCI*, $\tau_{sup} = 200{,}000$ for *Enamine*, $\tau_{sup} = 153$ for *DBLP*, and $\tau_{sup} = 16{,}000$ for *Yeast*.

Table 9 and Fig. 25 show the scalability results on our five datasets with an increasing number of task computing threads. Specifically, Table 9 reports the running time, speedup ratio, and peak memory consumption, while Fig. 25 plots the running time and speedup ratio curves. Moreover, the first row of Table 9 also reports the results when running the serial gSpan program of [16] as a comparison.

Note that unlike previous applications, the serial program here is much faster than our PrefixFPM program with a single computing thread. This is because in the serial algorithm, a projected transaction $G_i|_\alpha$ only keeps the last matched edge (actually a pointer to that edge in the transaction $G_i \in D$) since the previously matched edges can be obtained from the predecessor patterns. In contrast, PrefixFPM has to materialize all edges (actually their pointers to edges in G_i) in a *Pattern* object's projected database, since the corresponding task's parent-task could have finished with its pattern object released. As a result, the materialization of projected transactions causes additional overheads compared with the serial program. We can eliminate the materialization cost during recursive processing (i.e., Line 3 of Fig. 6) and only conduct materialization of projected transactions when new tasks need to be added to the task queue (i.e., Lines 10 and 18 of Fig. 6), but additional implementation in the application code would be needed to realize such an optimization.

In Fig. 25, we show the running time on *DBLP* and *Yeast* with separate plots since their running times are very different from the other 3 datasets. In particular, mining on *DBLP* is much more time-consuming, since it has some dense clique-like structures for publication collaborations that cause the number of patterns to grow exponentially. We can see that the speedup ratio on *GraphGen*, *OpenNCI* and *DBLP* is good for up to 8 computing threads, and continues to improve beyond 16 threads. In contrast, the speedup ratio on *Enamine* and *Yeast* is fair and there is no improvement beyond 16 threads. After examining the mined patterns, we find that they share common ancestor structures causing mining to focus on only a few branches of the pattern-growth tree, leaving many threads idle without tasks to process.

Table 9 Scalability results of gSpan

#{cores}	GraphGen (s)	Speedup	RAM (GB)	OpenNCI (s)	Speedup	RAM (GB)	Enamine (s)	Speedup	RAM (GB)	DBLP (s)	Speedup	RAM (GB)	Yeast (s)	Speedup	RAM (GB)
Serial	2764.29	–	7.2	1239.76	–	6.0	474.07	–	6.3	14,004.10	–	2.3	48.51	–	0.6
1	3837.63	1	11.5	4,281.99	1	11.4	1,230.30	1	7.2	76,886.00	1	0.6	92.52	1	0.9
2	2048.12	1.87	11.8	2113.00	2.03	18.0	645.51	1.91	8.2	38,395.50	2.00	0.6	45.33	2.04	1.1
4	1119.75	3.43	12	1,053.26	4.07	24.0	382.63	3.22	11.2	19,590.10	3.92	0.7	25.00	3.70	1.7
8	603.68	6.36	12.4	563.94	7.59	37.7	231.79	5.31	15.2	9,977.37	7.71	0.8	17.93	5.16	2.5
16	360.48	10.65	12.4	369.67	11.58	54.1	173.03	7.11	25.2	5,074.77	15.15	1	17.56	5.27	2.7
32	254.87	15.06	13.9	235.80	18.16	94.5	169.18	7.27	43.2	3,193.69	24.07	1.3	17.50	5.29	2.5
64	219.58	17.48	17.5	202.14	21.18	136.9	173.10	7.11	54.2	2,430.83	31.63	2	17.90	5.17	4.2
128	193.03	19.88	20.9	201.06	21.3	178.5	177.29	6.94	61.2	2,475.95	31.05	3.3	17.76	5.21	5.2

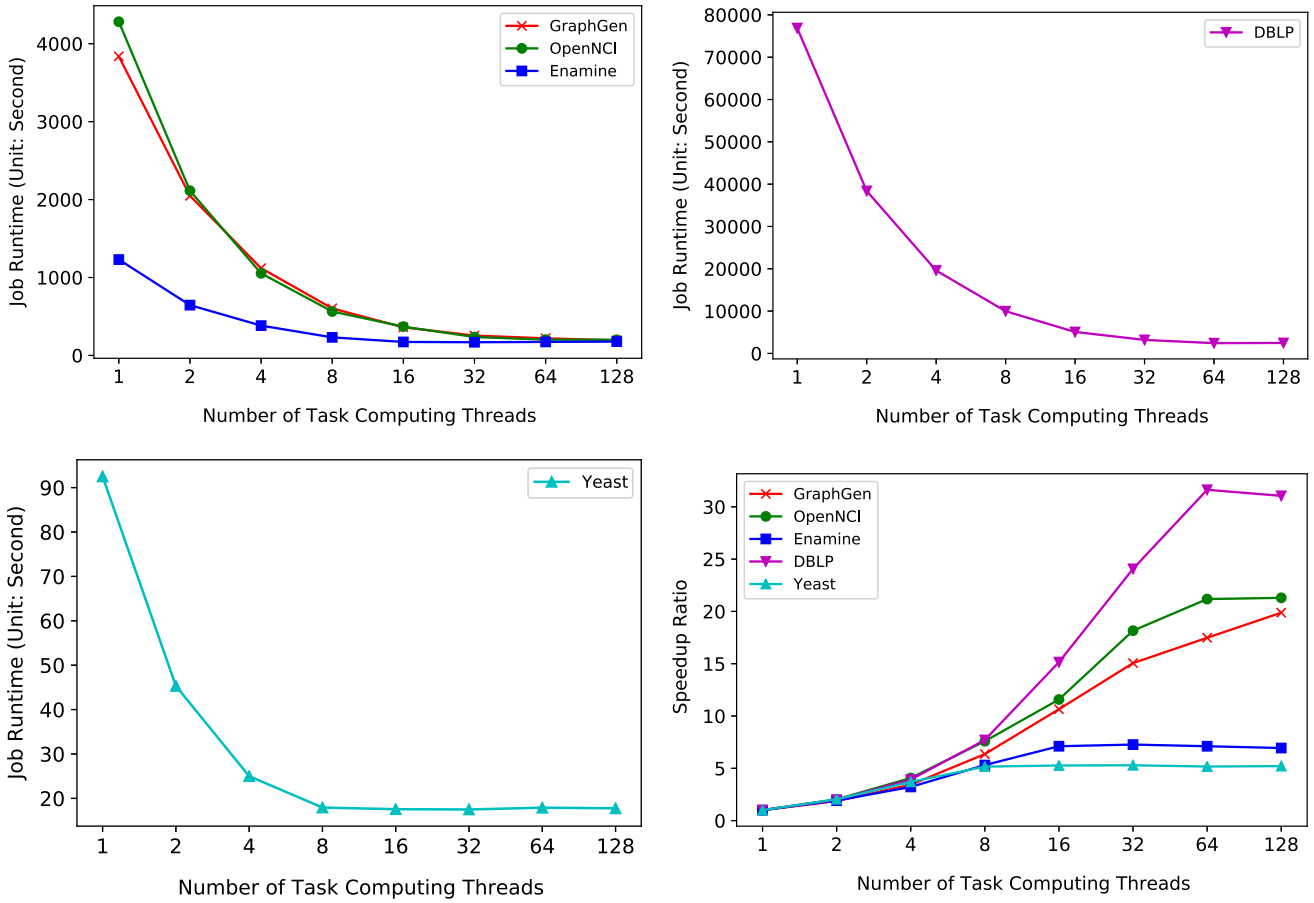


Fig. 25 Scalability of gSpan

Comparison with RStream [45] Recall from Sect. 2 that RStream also supports frequent subgraph mining (FSM), which runs in iterations where the i-th iteration constructs subgraph patterns with i edges. However, RStream mines patterns from a big graph rather than a collection of small graph transactions, and therefore, we consider each graph dataset in Table 8 as a big graph made up of all graph transactions (with vertex IDs recoded to avoid ID conflict), which can then be input to RStream. Note that the frequent subgraphs found as such by RStream are different from those from a transaction database, since subgraph frequency is not anti-monotonic on a big graph and "minimum image-based support" metric is used instead [45], but this is the closest setting that we can compare with RStream.

The RStream program [39] takes the maximum pattern size i_{max} as an input and builds patterns for i_{max} iterations (may not yet find all frequent subgraph patterns). As an out-of-core system, RStream also partitions the data so that only necessary data need to be loaded to memory, but since our BlueBlaze server has 1TB RAM space, we simply set the number of partitions to be 1 to avoid this data movement overhead. This allows RStream to report the fastest performance since the IO overhead is minimized.

We run RStream on the small *Enamine* for a comparison, as RStream is very slow on big datasets. In our PrefixFPM experiments on *Enamine*, the average pattern size we find is 5.6 and the maximum pattern size is 11. We run RStream with $i_{max} = 6$, which takes 32.344 GB RAM and 1708 s to run. This is already longer than 1230 s taken by PrefixFPM with just 1 task computing thread (c.f. Table 9). If we use $i_{max} = 10$, RStream takes 211.46 GB RAM and 11,138 s to complete. In contrast, the RAM consumption by PrefixFPM is only 7.2, 8.2, 11.2, 15.2, 25.2, 43.2, 54.2 and 61.2 GB, respectively, for 1, 2, 4, 8, 16, 32, 64, 128 threads. Moreover, PrefixFPM finds all frequent patterns rather than those with length up to i_{max}.

8.6 Gaston scalability on PrefixFPM

We also use the five datasets in Table 8 to evaluate the scalability of Gaston, with the same τ_{sup} setting as in our gSpan experiments.

Scalability results We adapted the serial Gaston implementation of [15] to PrefixFPM. Table 10 and Fig. 26 show the scalability results on our five datasets with an increasing number of task computing threads. Specifically, Table 10

Table 10 Scalability results of Gaston

#{cores}	GraphGen (s)	Speedup	RAM (GB)	OpenNCI (s)	Speedup	RAM (GB)	Enamine (s)	Speedup	RAM (GB)	DBLP	Speedup	RAM (GB)	Yeast (s)	Speedup	RAM (GB)
Serial	2069.81	–	4.6	619.30	–	4.3	403.24	–	3.8	2936.79	–	0.3	42.20	–	0.5
1	2353.74	1	14.3	1171.62	1	16.4	509.90	1	13.7	3680.84	1	0.3	51.39	1	1.3
2	1350.00	1.74	15.3	675.36	1.73	24.8	290.65	1.75	19.3	1874.78	1.96	0.3	27.11	1.90	1.8
4	778.22	3.02	20.2	421.08	2.78	31.1	195.86	2.60	22.3	961.49	3.83	0.3	16.77	3.06	2.2
8	448.44	5.25	24.2	237.37	4.94	44	136.71	3.73	18.3	484.99	7.59	0.4	14.34	3.58	2
16	255.94	9.20	35.2	178.02	6.58	33.2	141.94	3.59	18.6	247.58	14.87	0.4	15.07	3.41	2.2
32	179.77	13.09	34.9	169.35	6.92	31.6	135.58	3.76	19.2	147.35	24.98	0.5	15.28	3.36	2.6
64	149.83	15.71	31.3	172.06	6.81	36.8	136.91	3.72	19.8	104.39	35.26	0.6	15.01	3.42	2.7
128	151.18	15.57	23.1	166.78	7.02	36.2	136.56	3.73	20.1	81.64	45.09	0.7	14.83	3.46	3.2

reports the running time, speedup ratio, and peak memory consumption, while Fig. 26 plots the running time and speedup ratio curves. Moreover, the first row of Table 10 also reports the results when running the serial gSpan program of [16] as a comparison.

Comparing Table 10 with Table 9, we can see that even though the speedup ratios of Gaston on some datasets such as *OpenNCI* and *Enamine* are not as good as gSpan (due to a smaller task computing workloads to be distributed to the threads), the absolute time used by Gaston is always smaller than that by gSpan, showing that Gaston is a safely better choice than gSpan in all scenarios, thanks to its ability to use simpler and hence more efficient algorithms when mining simpler structures such as paths and free trees.

Another important observation from Table 10 is that unlike gSpan, Gaston's serial program is only slightly faster than the PrefixFPM program running with one computing thread, showing that the system overhead for task scheduling is small for Gaston. This demonstrates the superiority of Gaston for parallelization with PrefixFPM. In fact, the speedup ratio is also good for up to 8 threads which is a common setting in most modern PCs/laptops, though more cores beyond 8 often do not help for Gaston due to the smaller mining workloads of Gaston, where most tasks are fast to complete and will not trigger a timeout for decomposition.

An exception is on *DBLP* where our Gaston program achieves 45.09× speedup with 128 CPU cores, even higher than that of gSpan in Table 9, thanks to the more superior performance of Gaston when the number of labels (i.e., authors in the *DBLP* database) is large. In fact, Gaston is a clear winner for such datasets: When running 128 threads on *DBLP*, Table 10 shows that Gaston takes merely 81.64 s, while Table 9 shows that gSpan needs 2,475.95 s, which is over 30 times slower.

Also, note from Table 10 and Table 9 that Gaston tends to use smaller amount of RAM thanks to its use of simpler operations when mining simpler patterns. However, the RAM consumption of Gaston's PrefixFPM program running with 1 computing thread is still a few times higher than the serial Gaston program, which is for the same reason as in gSpan: In the serial program, a projected transaction $G_i|_\alpha$ only keeps the last matched edge (actually a pointer to that edge in the transaction $G_i \in D$) since the previously matched edges can be obtained from the predecessor patterns; in contrast, PrefixFPM has to materialize all edges (actually their pointers to edges in G_i) in a *Pattern* object's projected database, since the corresponding task's parent-task could have finished with its pattern object released.

In Fig. 26, we show the running time of different graphs with 3 separate plots since their running times are very different. We can see that the speedup ratio on *DBLP* is great, followed by *GraphGen*, and the speedup ratios on the other 3 datasets are limited: Running with 4-8 threads is enough to

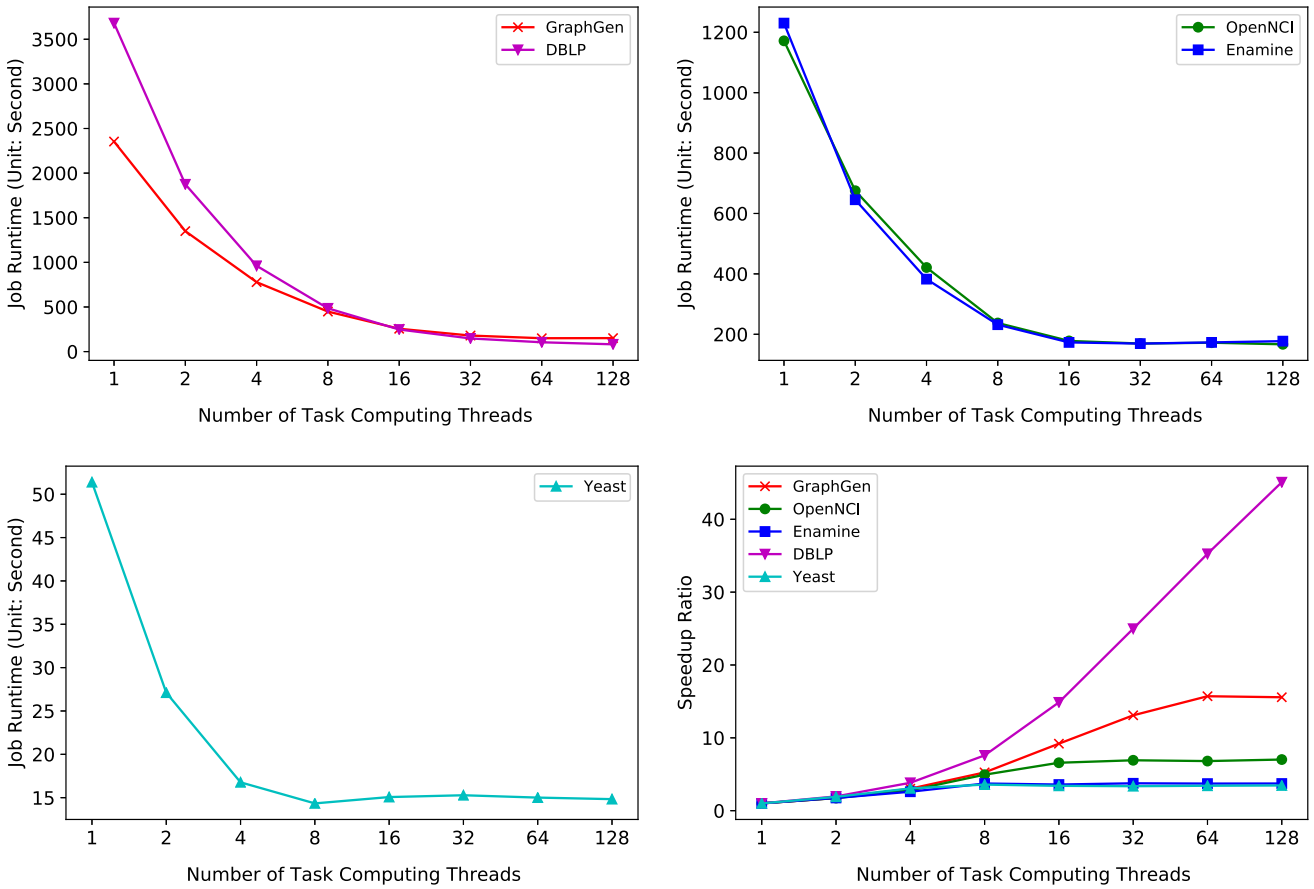


Fig. 26 Scalability of Gaston

achieve a few times speedup, but no benefit can be achieved when running more computing threads.

8.7 Effect of system parameters

Recall that we have been using the default setting where $\tau_{split} = 100$ and $\tau_{time} = 0.1$ second so far. We have actually extensively tested different values of both parameters to achieve this default setting that works consistently well in all cases. In fact, the performance is more sensitive to the value of τ_{time}.

In this section, we report the effect of τ_{time} on the running time using three datasets of different transaction types: *Plan* (sequence transactions), *Enamine* (graph transactions), and *TreeGen* (tree transactions). We vary the value of τ_{time} as 0.01, 0.1, 1, 10, 100 s while keeping $\tau_{split} = 100$ and observe how the performance is impacted.

Table 11 and Fig. 27 show the results on *Plan* with varying τ_{time} and varying number of task computing threads.

Table 12 and Fig. 28 show the results on *Enamine* with varying τ_{time} and varying number of task computing threads.

Table 13 and Fig. 29 show the results on *TreeGen* with varying τ_{time} and varying number of task computing threads.

Table 11 Effect of τ_{time} on *Plan*: running time

	τtime				
#{cores}	0.01	0.1	1	10	100
1	532.99 s	467.99 s	486.00 s	408.44 s	462.97 s
2	255.02 s	226.88 s	227.11 s	265.13 s	260.07 s
4	138.58 s	140.00 s	127.04 s	134.84 s	125.22 s
8	62.91 s	81.26 s	72.94 s	89.89 s	111.98 s
16	53.88 s	42.21 s	41.57 s	49.38 s	105.58 s
32	24.76 s	23.96 s	34.47 s	38.06 s	110.84 s
64	20.41 s	19.91 s	21.61 s	36.19 s	103.18 s
128	21.77 s	17.49 s	17.67 s	36.59 s	103.19 s

From the above results, we can obtain the following observations:

- The impact of τ_{time} is not significant when there are only 1 to 2 threads, which is within expectation since there are enough tasks to keep the 1 to 2 computing threads busy even without timeout-triggered task decomposition.
- The impact of τ_{time} is significant when the number of computing threads goes beyond 4. In particular, when

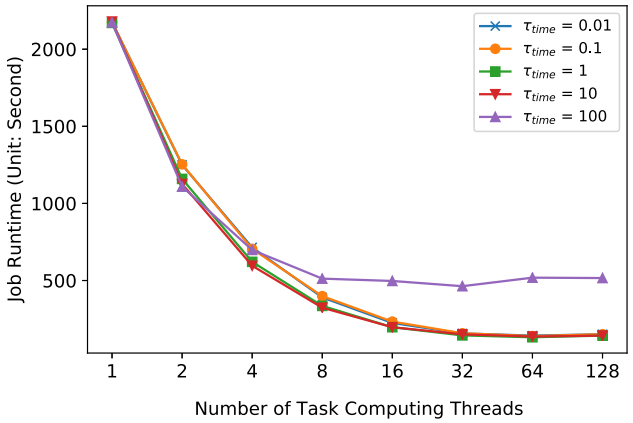


Fig. 27 Effect of τ_{time} on *Plan*

Table 12 Effect of τ_{time} on *Enamine*: running time

	τtime				
#{cores}	0.01	0.1	1	10	100
1	1241.53 s	1234.02 s	1241.71 s	1205.23 s	1192.53 s
2	617.04 s	615.51 s	625.95 s	731.43 s	663.09 s
4	329.55 s	329.52 s	331.77 s	348.99 s	456.42 s
8	225.88 s	217.90 s	205.35 s	218.68 s	406.24 s
16	164.84 s	171.59 s	162.72 s	183.11 s	383.28 s
32	165.04 s	163.27 s	171.01 s	173.22 s	388.16 s
64	171.68 s	168.27 s	172.57 s	174.07 s	392.04 s
128	178.59 s	179.00 s	177.38 s	179.13 s	386.03 s

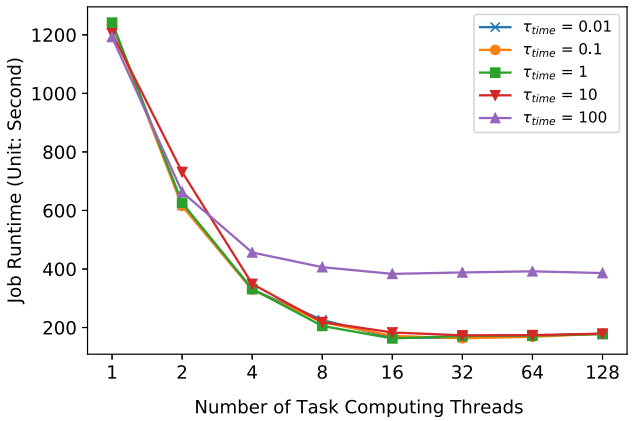


Fig. 28 Effect of τ_{time} on *Enamine*

$\tau_{time} = 100$ s, the running time becomes a few times longer on all our datasets. This is because some tasks can run for up to 100 s before being decomposed as smaller tasks to be added into Q_{task}, leaving many other threads idle and hence a load balancing issue.

– In all experiments, the best performance is achieved either when $\tau_{time} = 0.1$ or 1 second, which justifies our default choice of $\tau_{time} = 0.1$ second.

Table 13 Effect of τ_{time} on *TreeGen*: running time Numbers

	τtime				
#{cores}	0.01	0.1	1	10	100
1	2175.25 s	2175.39 s	2170.36 s	2179.11 s	2173.71 s
2	1250.46 s	1253.25 s	1158.80 s	1125.88 s	1108.65 s
4	714.33 s	705.10 s	621.07 s	595.86 s	698.25 s
8	390.63 s	398.71 s	335.32 s	323.74 s	512.17 s
16	224.99 s	233.50 s	197.29 s	197.13 s	497.10 s
32	156.81 s	158.08 s	144.08 s	150.86 s	463.13 s
64	142.88 s	139.63 s	132.01 s	136.59 s	518.45 s
128	151.55 s	151.77 s	142.81 s	143.03 s	515.96 s

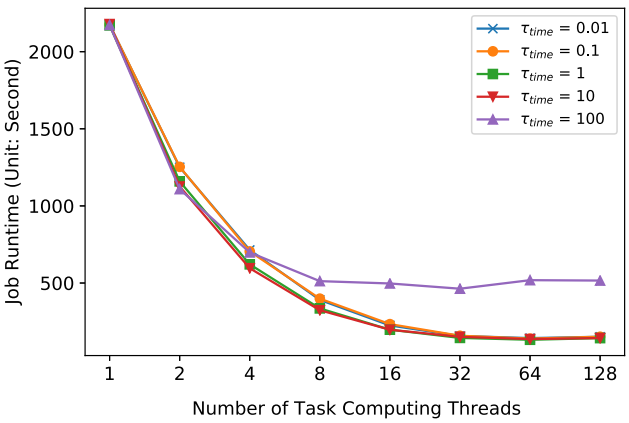


Fig. 29 Effect of τ_{time} on *TreeGen*

8.8 Comparison with Apache Spark

Spark MLlib is currently the de facto standard for parallel and distributed data mining, but it only has limited support for frequent pattern mining, currently just FP-Growth (for itemsets) and PrefixSpan. We, therefore, can only compare with Spark for the application of PrefixSpan, using our three sequence datasets *Plan*, *S50* and *S10* (c.f., Table 1).

We install Spark 3.0.0 on an on-premise cluster of 16 machines each with 64 GB RAM, AMD EPYC 7281 CPU, and 22TB disk, where each machine runs a Spark executor. We vary the number of machines (i.e., executors) as 1, 2, 4, 8, and 16 and compare the results with PrefixFPM's PrefixSpan with 1, 2, 4, 8, and 16 CPU cores, respectively. The results are shown in Table 14.

As we can see, when the number of transactions is not large as in *Plan* and *S50*, (1) running PrefixFPM with one thread is around 5 times faster than running MLlib with one machine; (2) while PrefixFPM achieves very good speedup up to 16 cores, MLlib's speedup plateaus beyond 2 machines likely due to the communication bottleneck, though the performance with 2 machines is a bit more than 2× that with 1 machine, likely because the overhead of the master node

Table 14 Scalability comparison with spark

#{cores}	PrefixFPM (s)	#{machines}	Spark MLlib (s)
(a) Scalability comparison on Plan			
1	434.06	1	2410.46
2	229.50	2	1078.55
4	117.53	4	962.86
8	84.22	8	1055.78
16	40.24	16	1167.96
(b) Scalability Comparison on S50			
1	1646.77	1	7941.22
2	830.73	2	3373.08
4	419.36	4	3255.51
8	211.44	8	3323.52
16	109.39	16	3048.56
(c) Scalability Comparison on S10			
1	1447.87	1	90,731.62
2	739.74	2	44,548.87
4	389.55	4	25,364.66
8	211.86	8	15,831.12
16	119.13	16	9403.22

occupies an amount of machine resource that is not negligible.

In contrast, when the number of transactions is large as in *S10*, MLlib's scalability with the number of machines is good as shown in Table 14(c). However, the problem is that MLlib running on one machine is over 60× more expensive than PrefixFPM running with one core, and this large performance gap makes MLlib even less competitive than PrefixFPM even when the number of cores/machines is 16.

8.9 Comparison with subgraph mining on Hadoop

Since Spark MLlib does not have support for more advanced frequent patterns such as subgraphs, we explored other research on Hadoop for frequent subgraph pattern mining (FSM). While there are some works, many of them either do not have open-source code or cannot run through smoothly. We finally identified one work, FSM-H [2], which uses an iterative MapReduce-based framework for FSM, which claims to be *efficient as it applies all the optimizations that the latest FSM algorithms adopt*, and which has open-source code at [13].

To replicate the exact environment of [2], we used Google Cloud to create 10 machine nodes, one as the Hadoop master and the other 9 as worker nodes. The machine type of each node is e2-standard-8 with 8 vCPUs, 32GB RAM, and 500GB SSD upon which the Hadoop Distributed File System (HDFS) was built.

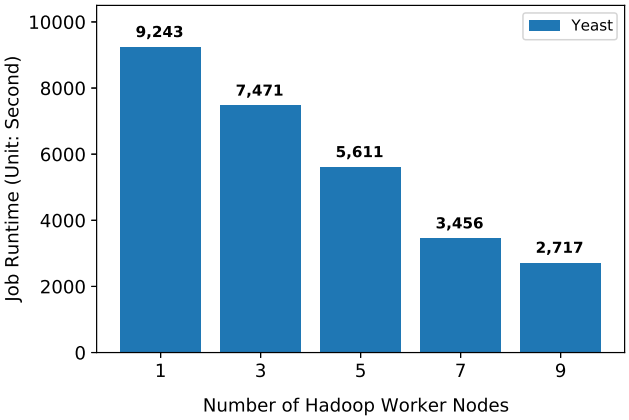


Fig. 30 Horizontal scalability of FSM-H on *Yeast*

All experimental settings were strictly following those of [2]: The graph database was partitioned so that each file partition contains 100 graphs; the number of reducers is set to be 90% of the number of vCPUs available. (Remaining 10% are for the operating system.) We vary the number of worker nodes as 1, 3, 5, 7, and 9 to obtain the job running time. Unfortunately, among the 5 datasets in Table 8, only *Yeast* (which is the smallest dataset) can finish within 24 h. This is within expectation since FSM-H adopts an IO-bound iterative Apriori algorithm where size-$(i+1)$ patterns are mined after size-i ones, rather than using our PrefixFPM's prefix projection method.

On *Yeast*, we obtain 796 file partitions, and we use $\tau_{sup} = 16{,}000$ as before. Figure 30 shows the horizontal scalability of FSM-H on *Yeast*, where we can see that the running time reduces as the number of worker nodes increases. However, even with 9 worker nodes, the mining still takes 2717 s. In contrast, recall from Table 9 that our parallel gSpan program on PrefixFPM only needs 92.52 s with just 1 computing thread on *Yeast*, and merely 17.93 s with 8 threads; even the serial gSpan code only needs 48.51 s (56× faster than FSM-H). Also, recall from Table 10 that parallel Gaston program only needs 51.39 s with just 1 computing thread, and merely 14.34 s with 8 threads; even the serial Gaston code only needs 42.20 s (over 64× faster than FSM-H).

The experiments in Sects. 8.8 and 8.9 verify that existing Big Data solutions for FPM on Hadoop and Spark merely scale (i.e., can run even when data volume goes beyond what one machine can accommodate), but the performance can be much worse than even a single-threaded program, not to mention vertical scalability to effectively utilize multicore hardware. PrefixFPM effectively fills this gap.

9 Conclusion

We proposed the PrefixFPM framework for general-purpose frequent pattern mining, which is based on the idea of *prefix*

projection and *divide and conquer* to exploit massive parallelism in a modern multicore environment.

A key advantage of PrefixFPM is that it provides a general programming interface which can be easily customized by users for their desired pattern types; the interface also allows different FPM problems to share the same backend for parallel execution.

We have shown that PrefixFPM demonstrates excellent vertical scalability with the number of task computing threads and scales well in a common modern multicore machine with 8–16 CPU cores. Further improvements can also be achieved when more CPU cores are available.

As a future work, we plan to migrate PrefixFPM to a distributed environment, where there is still potential to fully utilize CPU cores since when $D|_\alpha$ is small enough, we can send the data to some machine (with linear IO cost) which will then mine it to check and extend pattern α, the computing cost of which is exponential to the size of $D|_\alpha$. As long as the mining tasks (i.e., computation) are well overlapped with the data requests (i.e., communication), the CPU cores in the cluster can be kept busy.

Another further work is to further extend the applications on top of PrefixFPM beyond the 6 applications that we have already developed in this paper, making it a comprehensive library for parallel FPM.

A third further work is to consider the different problem context of a big dataset such as a large social network or knowledge graph, rather than a database of individual transactions. An example is the GRAMI algorithm for FSM on a big graph using the antimonotonic support measure of minimum image [3]. GRAMI follows a similar pattern-growth recursive procedure and is perfect to address using our task-based G-thinker [18,46] framework for handling a single big graph, on top of which we can reuse many classes of the PrefixFPM API that we laid out in Fig. 5 (though some components such as class *Trans* are no longer useful).

Acknowledgements Da Yan and Guimu Guo were supported by NSF OAC-1755464 and NSF DGE-1723250. Guimu Guo acknowledges financial support from the Alabama Graduate Research Scholars Program (GRSP) funded through the Alabama Commission for Higher Education and administered by the Alabama EPSCoR. Wenwen Qu and Xiaoling Wang were supported by NSFC grants (No. 61972155), the Science and Technology Commission of Shanghai Municipality (20DZ1100300), and the Open Project Fund from Shenzhen Institute of Artificial Intelligence and Robotics for Society.

References

1. Aggarwal, C.C., Han, J. (eds.): Frequent Pattern Mining. Springer, Berlin (2014)
2. Bhuiyan, M., Hasan, M.A.: An iterative mapreduce based frequent subgraph mining algorithm. IEEE Trans. Knowl. Data Eng. **27**(3), 608–620 (2015)
3. Bringmann, B., Nijssen, S.: What is frequent in a single graph? In: Washio, T., Suzuki, E., Ting, K.M., Inokuchi, A., (eds) PAKDD, vol. 5012 of Lecture Notes in Computer Science, pp. 858–863. Springer, Berlin (2008)
4. Chan, H.K., Long, C., Yan, D., Wong, R.C.: Fraction-score: a new support measure for co-location pattern mining. Presented at the (2019)
5. Cheng, J., Ke, Y., Ng, W., Lu, A.: Fg-index: towards verification-free query processing on graph databases. In: SIGMOD, pp. 857–872 (2007)
6. Chon, K., Hwang, S., Kim, M.: Gminer: a fast gpu-based frequent itemset mining method for large-scale data. Inf. Sci. **439–440**, 19–38 (2018)
7. CloSpan Package. https://sites.cs.ucsb.edu/~xyan/software/Clospan.htm
8. COST in the Land of Databases. https://github.com/frankmcsherry/blog/blob/master/posts/2017-09-23.md
9. DBLP Collaboration Network. http://networkrepository.com/ca-dblp-2012.php
10. Elseidy, M., Abdelhamid, E., Skiadopoulos, S., Kalnis, P.: GRAMI: frequent subgraph and pattern mining in a single large graph. Proc. VLDB Endow. **7**(7), 517–528 (2014)
11. Enamine Dataset. https://enamine.net/
12. Fang, W., Lu, M., Xiao, X., He, B., Luo, Q.: Frequent itemset mining on graphics processors. In: DaMoN, pp. 34–42. ACM (2009)
13. FSM-H Code. http://dmgroup.cs.iupui.edu/Mansurul_FSMH.php
14. Gan, W., Lin, J.C., Fournier-Viger, P., Chao, H., Tseng, V.S., Yu, P.S.: A survey of utility-oriented pattern mining. IEEE Trans. Knowl. Data Eng. **33**(4), 1306–1327 (2021)
15. Gaston Implementation. https://liacs.leidenuniv.nl/~nijssensgr/gaston/
16. gSpan Implementation. https://github.com/rkwitt/gboost/tree/master/src-gspan
17. gSpan Technical Report. https://sites.cs.ucsb.edu/~xyan/papers/gSpan.pdf
18. Guo, G., Yan, D., Özsu, M.T., Jiang, Z., Khalil, J.: Scalable mining of maximal quasi-cliques: an algorithm-system codesign approach. Proc. VLDB Endow. **14**(4), 573–585 (2020)
19. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. Presented at the (2000)
20. IBM Synthetic Data Generator. https://github.com/zakimjz/IBMGenerator
21. Kudo, T., Maeda, E., Matsumoto, Y.: An application of boosting to graph classification. In: NIPS, pp. 729–736 (2004)
22. Li, E., Liu, L.: Optimization of frequent itemset mining on multiple-core processor. In: VLDB, pp. 1275–1285. ACM (2007)
23. Li, H., Wang, Y., Zhang, D., Zhang, M., Chang, E.Y.: Pfp: parallel fp-growth for query recommendation. In: Proceedings of the 2008 ACM Conference on Recommender Systems, RecSys 2008, Lausanne, Switzerland, October 23–25, 2008, pp. 107–114 (2008)
24. Lin, W., Xiao, X., Ghinita, G.: Large-scale frequent subgraph mining in mapreduce. In: ICDE, pp. 844–855 (2014)
25. McSherry, F., Isard, M., Murray, D.G.: Scalability! but at what cost? In: HotOS, USENIX Association (2015)
26. NCI Dataset. https://cactus.nci.nih.gov/download/nci/
27. Nijssen, S., Kok, J.N.: A quickstart in frequent structure mining can make a difference. In: KDD, pp. 647–652. ACM (2004)
28. Nijssen, S., Kok, J.N.: The gaston tool for frequent subgraph mining. Electron. Notes Theor. Comput. Sci. **127**(1), 77–87 (2005)
29. OpenMP. https://www.openmp.org/
30. Orlando, S., Lucchese, C., Palmerini, P., Perego, R., Silvestri, F.: kdci: a multi-strategy algorithm for mining frequent sets. In: FIMI, vol. 90 of CEUR Workshop Proceedings. CEUR-WS.org (2003)
31. Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: In: Prefixspan: Mining Sequential Patterns by Prefix-Projected Growth, pp. 215–224, Heidelberg, Germany (2001)

32. Peng, Z., Wang, T., Lu, W., Huang, H., Du, X., Zhao, F., Tung, A.K.H.: Mining frequent subgraphs from tremendous amount of small graphs using mapreduce. Knowl. Inf. Syst. **56**(3), 663–690 (2018)
33. Plan Dataset. https://www.cs.rpi.edu/~zaki/software/plandata.gz
34. PrefixSpan Implementation. http://chasen.org/~taku/software/prefixspan/prefixspan-0.4.tar.gz
35. Schlegel, B., Karnagel, T., Kiefer, T., Lehner, W.: Scalable frequent itemset mining on many-core processors. In: DaMoN, p. 3. ACM (2013)
36. Silvestri, C., Orlando, S., gpudci.: Exploiting gpus in frequent itemset mining. In: PDP, pp. 416–425. IEEE (2012)
37. Sleuth Implementation. https://github.com/zakimjz/SLEUTH
38. Teixeira, C.H.C., Fonseca, A.J., Serafini, M., Siganos, G., Zaki, M.J., Aboulnaga, A.: Arabesque: a system for distributed graph mining. In: SOSP, pp. 425–440 (2015)
39. The RStream system. https://github.com/rstream-system
40. Tree Generator. http://www.cs.rpi.edu/~zaki/software/TreeGen.tar.gz
41. TreeBank. http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html#treebank
42. TreeMiner Implementation. https://github.com/zakimjz/TreeMiner
43. Vu, L., Alaghband, G.: Novel parallel method for mining frequent patterns on multi-core shared memory systems. In: DISCS@SC, pp. 49–54. ACM (2013)
44. Wang, J., Han, J.: BIDE: efficient mining of frequent closed sequences. In: Özsoyoglu, Z.M., Zdonik, S.B., (eds) ICDE, pp. 79–90 (2004)
45. Wang, K., Zuo, Z., Thorpe, J., Nguyen, T.Q., Xu, G.H.: Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In: OSDI, pp. 763–782 (2018)
46. Yan, D., Guo, G., Chowdhury, M.M.R., Özsu, M.T., Ku, W.-S., Lui, J.C.: G-thinker: a distributed framework for mining subgraphs in a big graph. ICDE (2020)
47. Yan, D., Guo, G., Chowdhury, M.M.R., Özsu, M.T., Lui, J.C.S., Tan, W.: T-thinker: a task-centric distributed framework for compute-intensive divide-and-conquer algorithms. Presented at the (2019)
48. Yan, X., Han, J., Afshar, R.: Clospan: mining closed sequential patterns in large datasets. In: SDM, pp. 166–177. SIAM (2003)
49. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: ICDM, pp. 721–724 (2002)
50. Yan, D., Qu, W., Guo, G., Wang, X.: Prefixfpm: a parallel framework for general-purpose frequent pattern mining. In: ICDE (2020)
51. Yang, G.: The complexity of mining maximal frequent itemsets and maximal frequent patterns. Presented at the (2004)
52. Yeast Dataset. https://sites.cs.ucsb.edu/~xyan/dataset.htm
53. Zaki, M.J.: In: Efficiently mining frequent trees in a forest, pp. 71–80. , Edmonton, Alberta, Canada (2002)
54. Zaki, M.J.: Scalable algorithms for association mining. IEEE Trans. Knowl. Data Eng. **12**(3), 372–390 (2000)
55. Zaki, M.J.: SPADE: an efficient algorithm for mining frequent sequences. Mach. Learn. **42**(1/2), 31–60 (2001)
56. Zaki, M.J.: Efficiently mining frequent embedded unordered trees. Fundam. Inform. **66**(1–2), 33–52 (2005)
57. Zhang, F., Zhang, Y., Bakos, J.D.: Gpapriori: Gpu-accelerated frequent itemset mining. In: CLUSTER, pp. 590–594. IEEE Computer Society (2011)
58. Zhang, F., Zhang, Y., Bakos, J.D.: Accelerating frequent itemset mining on graphics processing units. J. Supercomput. **66**(1), 94–117 (2013)
59. Zou, L., Lu, Y., Zhang, H., Hu, R.: Prefixtreeespan: a pattern growth algorithm for mining embedded subtrees. In: Aberer, K., Peng, Z., Rundensteiner, E.A., Zhang, Y., Li, X., (eds) WISE, vol. 4255 of Lecture Notes in Computer Science, pp. 499–505. Springer (2006)