Parallel Mining of Frequent Subtree Patterns

Wenwen Qu[2], Da Yan[1][0000−0002−4653−0408], Guimu Guo[1], Xiaoling Wang[2], Lei Zou[3], and Yang Zhou[4]

[1] The University of Alabama at Birmingham
{yanda, guimuguo}@uab.edu
[2] East China Normal University
wenwenqu@sei.ecnu.edu.cn, xlwang@cs.ecnu.edu.cn
[3] Peking University
zoulei@pku.edu.cn
[4] Auburn University
yangzhou@auburn.edu

Abstract. Mining frequent subtree patterns in a tree database (or, forest) is useful in domains such as bioinformatics and mining semi-structured data. We consider the problem of mining embedded subtrees in a database of rooted, labeled, and ordered trees. We compare two existing serial mining algorithms, PrefixTreeSpan and TreeMiner, and adapt them for parallel execution using PrefixFPM, our general-purpose framework for frequent pattern mining that is designed to effectively utilize the CPU cores in a multicore machine. Our experiments show that TreeMiner is faster than its successor PrefixTreeSpan when a limited number of CPU cores are used, as the total mining workloads is smaller; however, PrefixTreeSpan has a much higher speedup ratio and can beat TreeMiner when given enough CPU cores.

Keywords: Tree · Parallel · Frequent pattern mining · Prefix projection.

1 Introduction

Frequent patterns are substructures that appear in a dataset with frequency no less than a user-specified threshold. A substructure can refer to different structural forms, such as itemsets, sequences, trees and graphs. Frequent pattern mining (FPM) has been at the core of data mining research for over two decades [3], and numerous serial mining algorithms have been proposed for various types of substructure patterns. The mined frequent substructures have also been widely used in 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 [8] uses frequent subgraphs as features for classifying labeled graphs modeling real-world data such as chemical compounds.

This paper focuses on tree patterns, or more specifically, to mine frequent "embedded" subtrees in a database of "rooted", "labeled", and "ordered" trees. Here, "rooted" means that the tree root matters, "ordered" means that the

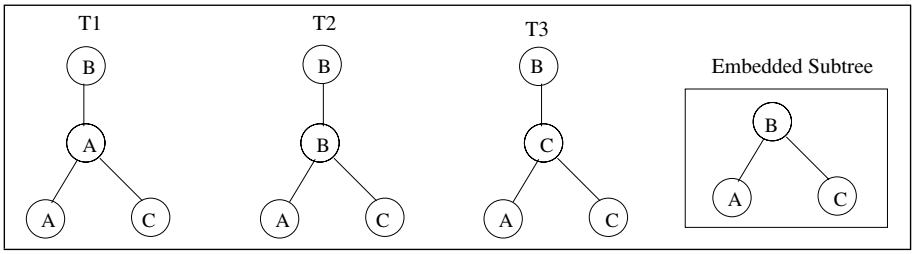


Fig. 1. Embedded Subtree Pattern Illustration

order of children nodes 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. 1, which shows a database of three trees. The subtree shown in the box is considered frequent as it appears in all 3 the trees T_1, T_2 and T_3, obtained by skipping the "middle" node in each tree, even though the subtree is the induced subgraph of only T_2 alone.

This problem is useful in many applications. In bioinformatics, researchers have collected vast amounts of RNA structures, which are essentially trees. To get information about a newly sequenced RNA, they compare it with known RNA structures, looking for common topological patterns, which provide important clues to the function of the RNA [10]. In web usage mining [7], given a database of web access logs at a popular site, one can mine the tree-structured browsing history of users to find frequently accessed subtrees (where nodes are webpages) at the site for prioritized investment. In web applications, tree-structured XML/JSON documents are popular for data transmission and storage, and discovering the commonly occurring subtrees that appear in these documents can help locate frequent user queries and data responses to be cached for faster access.

Tree mining has been well studied in the serial algorithm domain by a number of algorithms such as TreeMiner [13], FREQT [4], CMTreeMiner [6], Chopper [11], Xspanner [11] and PrefixTreeSpan [14]. We select PrefixTreeSpan for parallelization since it was reported to beat all the other algorithms. However, [14] treats TreeMiner to be an Apriori-like algorithms that check patterns of size-i only when all patterns of size-$(i-1)$ are found, while TreeMiner is actually a PrefixSpan[9]-like similar to PrefixTreeSpan, therefore we also select TreeMiner for parallelization to compare with PrefixTreeSpan.

We parallelize PrefixTreeSpan and TreeMiner using the PrefixFPM framework [12], which is found to be able to fully utilize the available CPU cores in a multi-core machine as long as the implemented algorithm provides sufficient opportunity for concurrent execution. PrefixFPM is designed for writing a general frequent pattern mining algorithm following the prefix-projection paradigm

pioneered by PrefixSpan [9], and PrefixTreeSpan and TreeMiner naturally fit in this paradigm. The main contributions and insights of this paper are as follows:

- We developed the parallel PrefixFPM algorithms for both PrefixTreeSpan and TreeMiner, and empirically compared them under different conditions.
- We find that TreeMiner is more effective in reducing the total mining workloads and thus faster when using up to only a moderate number of CPU cores. This is in contrary to the finding in PrefixTreeSpan's paper [14], which could be due to [14]'s treating TreeMiner as an Apriori-like algorithm.
- We find that, in contrast, PrefixTreeSpan is more amenable to parallel execution with a higher speedup ratio, and can beat TreeMiner when given enough CPU cores. This is a new finding since prior works have not considered parallel mining, and can shed light on the architecture-aware algorithm choice.

The rest of this paper is organized as follows. Section 2 reviews the related work including the idea of prefix projection illustrated with the pioneering PrefixSpan algorithm, and the PrefixFPM programming paradigm for parallelizing a PrefixSpan-like algorithm. Section 3 introduces the PrefixTreeSpan algorithm and its parallel implementation in PrefixFPM, and Section 4 describes the TreeMiner algorithm and its parallel implementation in PrefixFPM. Finally, Section 5 reports the results of our experimental comparison and Section 6 concludes this paper.

2 Preliminaries

A Tour of PrefixSpan. To understand the idea of prefix projection, let us first briefly review the pioneering PrefixSpan [9] algorithm for mining frequent sequential patterns from a sequence database.

We denote $\alpha\beta$ to be the sequence resulted from concatenating sequence α with sequence β. We also use $\alpha \sqsubseteq s$ to denote that sequence α occurs as a subsequence of sequence s in the database. Given a sequential pattern α and a sequence s, the α-projected sequence $s|_\alpha$ is defined to be the suffix γ of s such that $s = \beta\gamma$ with β being the minimal 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 $\beta = ABC$ and $s|_\alpha = \_\gamma = \_BC$.

Given a sequential 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 $\alpha \not\sqsubseteq s$, then the minimal prefix β of s satisfying $\beta \sqsubseteq s$ does not exist, and therefore s is not considered in $D|_\alpha$.

Consider the sequence database D shown in Fig. 2(a). The projected databases $D|_A$, $D|_{AB}$ and $D|_{ABC}$ are shown in Fig. 2(b), (c) and (d), respectively. Let us define the support of a pattern α as the number of sequences in D that contain α as a subsequence, then the support of α is simply the size of $D|_\alpha$. 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

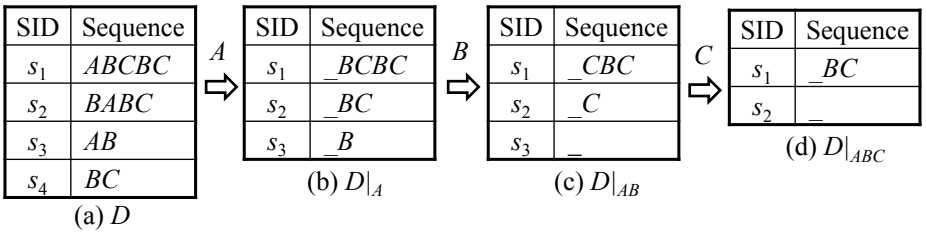


Fig. 2. Illustration of PrefixSpan

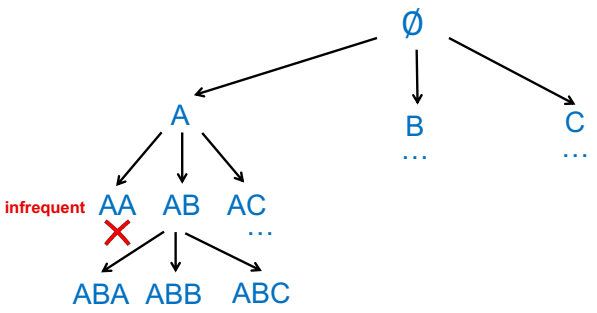


Fig. 3. Depth-First Search Space Tree

current pattern α is checked to be frequent, it will recurse on all the possible patterns α' constructed by appending α with one more element. PrefixSpan checks whether a pattern α is frequent using the projected database $D|_\alpha$, which is constructed from the projected database of the previous iteration. Fig. 2 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., the purchase of multiple goods in one supermarket transaction), and we refer readers to [9] for more details.

Prefix Projection. We can summarize the PrefixSpan algorithm's pattern (which is also the prefix) search space by a tree as illustrated in Fig. 3. The idea actually generalizes to other patterns including the embedded subtrees that we consider. The key insight is that we can establish a one-to-one correspondence between each subtree pattern and its sequence encoding, so that we can examine the pattern encodings by a PrefixSpan-style algorithm.

For example, consider the 3 subtrees shown in Fig. 4. We can encode a tree T by adding vertex labels to the encoding in a depth-first preorder traversal of T, and by adding a unique label "$" whenever we backtrack from a child to its parent. For example, the encoding of T_1 in Fig. 4 is BAB$D$$BC, the encoding of T_2 is BAB$D$$CB, while the encoding of T_3 is BCBAB$D$$.

If we consider "$" as the smallest label, and combined with the other node labels in the alphabet where label ordering is defined, then we can check through

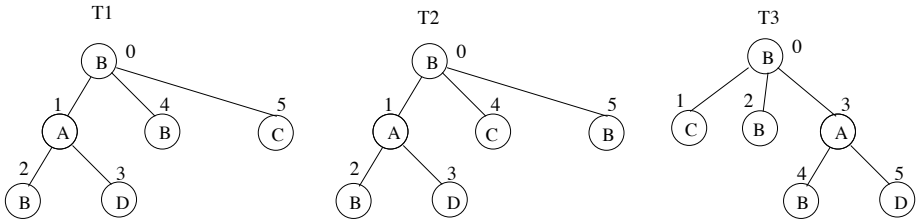


Fig. 4. Illustrative Tree Patterns

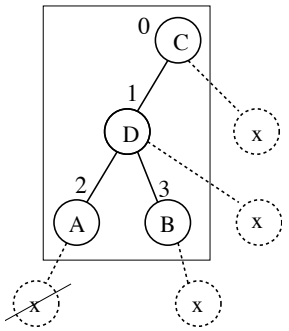


Fig. 5. Pattern Extension Along Rightmost Path

the subtree patterns similarly as in Fig. 3, imagining that the sequence encoding at each node is obtained by the depth-first preorder traversal of its corresponding subtree pattern. Recall that there is a one-to-one correspondence between a subtree pattern and its sequence encoding. This is exactly the mining workflow adopted by PrefixTreeSpan and TreeMiner.

In this case, the root node $\emptyset$ in the search tree of Fig. 3 basically finds node labels that are frequent in the tree database (e.g., A, B and C). Then, at the next layer node A basically finds frequent edges where the source node is A (e.g., AA, AB and AC). In the next layer, node AB (whose pattern only contains one edge AB) is basically extending the pattern with one more edge, which can give child-patterns like ABA or ABA$$ that corresponds to different subtrees for frequentness checking (Fig. 3 is for PrefixSpan so only a sequence ABA is shown). In a nutshell, each pattern as a node α in the search tree is extended by one more edge to generate a child-pattern β.

It is not difficult to see that to avoid redundant pattern examination from different subtrees, we should only extend a pattern using an adjacent edge on its rightmost path. For example, in Fig. 5, 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 an encoding CDAx$\cdots$ which does not match the pattern prefix CDA$B and should have been covered in the other search space subtree rooted at node CDA (i.e., the child CDAx rather than CDA$B).

In frequent subtree mining, the difference from PrefixSpan lies in the maintenance of projected database, where each tree data after prefix projection can give rise to multiple instances (for example, pattern B-B can map to node pairs 0-2 and 0-4 in T_1 of Fig. 4) that lead to different future extension trajectories; also, special encodings need to be maintained to facilitate the checking of ancestor-descendant relationship between a matched node in a data tree T and another node in T to extend the current pattern.

PrefixFPM Review. PrefixFPM associates each pattern α (which corresponds to a node in the search tree of Fig. 3) with a task t_α that 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 $D|_\alpha$ rather from the entire D). These children patterns give rise to new tasks $\{t_\beta\}$ which are added to a shared task queue for concurrent processing. PrefixFPM runs a number of mining 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 search 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}).

Since fetching tasks from a shared task queue and adding new child-tasks to Q_{task} incur locking overheads, this is only worthwhile if each task contains sufficient computing workloads such that the locking overhead is negligible. We therefore only add child-pattern tasks to Q_{task} if the number of projected data instances in $D|_\alpha$ is above a size threshold τ_{split}, so that the workloads can be divided by other computing threads; otherwise, t_α is not expensive and the current computing thread simply processes its entire search space subtree in depth-first order directly.

PrefixFPM Programming Interface. PrefixFPM is 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 the target FPM application. We refer readers to [12] for the complete API. Here, we briefly review the key UDFs that users need to specify in order to implement a parallel mining algorithm.

The most important base class is *Task*. A *Task* object t_α maintains 2 fields: a pattern α (along with its relevant data such as $D|_\alpha$), and a children table *children* that keeps $\{D|_\beta\}$: specifically, $children[e] = D|_\beta$ if β is grown from α with element e. *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.

```
1   void run(ostream& fout){
2         if(!pre_check(fout)) return;
3         //generate new patterns
4         setChildren(children);
5         //run new child tasks
6         while(Task* t= get_next_child()){
7               if(needSplit()){
8                     q_mtx.lock();
9                     queue().push(t);
10                    q_mtx.unlock();
11              }
12              else{
13                    t->run(fout);
14                    delete t;
15              }
16        }
17  }
```

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

Specifically, in Line 2, t_α first runs UDF *pre_check*(*fout*) to see if α is frequent and if so, to output α to an output file stream *fout*. If α is frequent and thus not pruned by *pre_check*(.), Line 4 then runs UDF *setChildren*(*children*) to scan $D|_\alpha$ and compute $\{D|_\beta\}$ into the table field *children*. In this step, every infrequent child pattern β should be removed from the table *children* as a postprocessing step after $\{D|_\beta\}$ are constructed.

Line 6 then wraps each child pattern β in table *children* as a task t_β, and calls the UDF *needSplit*() to predict if t_β is time-consuming (e.g., $D|_\beta$ is big). If so, we add t_β to the task queue Q_{task} (Lines 8–10) to be fetched by available task computing threads (Q_{task} is a global last-in-first-out task stack protected by a mutex to prioritize depth-first task processing order), which divides the computing workloads by multithreading. Otherwise, we recursively call t_β's *run*(*fout*) to process the entire checking and extension of β by the current thread, which avoids contention on Q_{task}. Since *needSplit*() just estimates if t_β is time-consuming and could have false negatives that become stragglers, we also count the time elapsed since t_α begins, and if it is larger than a timeout threshold, we also add t_β to Q_{task} for concurrent processing as in Lines 8–10.

The other important base class is *Worker*, which is the main thread that loads the database and creates the computing threads to process tasks. A *Worker* object is responsible for generating the initial tasks into Q_{task} from the database, the logic of which is specified by UDF *setRoot*().

Implementing *Worker*::*setRoot*(.) is similar to implementing *Task*::*setChildren*(.) (Line 4 of Fig. 6): 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 added to Q_{task} to initiate the parallel task computation.

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

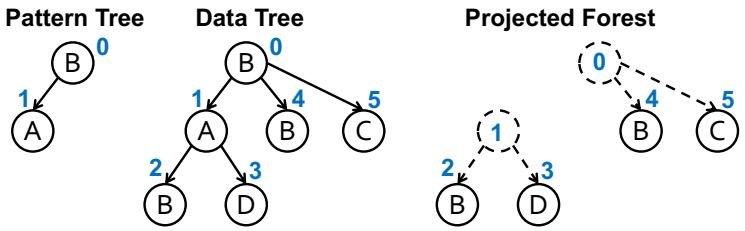


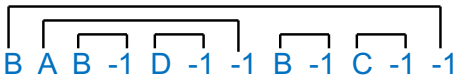
Fig. 7. Forests After Pattern Projection

when growing patterns), which is a common and effective pruning. In frequent subtree mining, one pass over the database is needed to filter out infrequent edges (determined by labels of its end-nodes), followed by another pass to (1) delete data edges that match those infrequent pattern-edges (in terms of end-node labels) and to (2) count the frequency of pattern-edges.

To summarize, to implement a parallel frequent pattern mining algorithm in PrefixFPM, we need to specify 2 key UDFs: *Worker*::*setRoot*(.) and *Task*::*setChildren*(*children*).

3 PrefixTreeSpan in PrefixFPM

Recall from Section 2 that the data tree in Fig. 7 can be encoded as BAB$D$$BC following preorder traversal that finally returns back to root B. To facilitate prefix projection, PrefixTreeSpan encodes this tree instead as:



Here, backtracking is encoded with -1 which is basically the same as $. However, PrefixTreeSpan lets each node to 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.

The definition of scope allows a quick checking of ancestor-descendant relationships. For example, in Fig. 7, after prefix projection by the pattern tree, the data tree now 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 be from right after the position that matches the last node in the pattern subtree. For the example in Fig. 7, 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. 7, encoded as B-1D-1 which is hooked to Node 1 in the pattern tree (1 is encoded by 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.

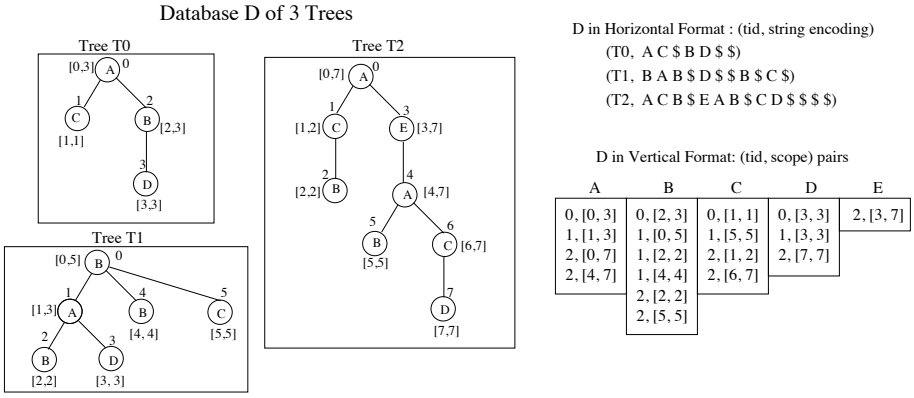


Fig. 8. Scope Lists

We remark that by scanning the data tree encoding from the last matched position, we effectively extend a pattern along its rightmost path. For example, referring to Fig. 5 again, we will not consider extending Node 2 since the last pattern node matched to a data tree is Node 3.

The implementation of this algorithm in PrefixFPM is straightforward, where in *Task*::*setChildren*(.) each task t_α scans its projected postfix-forest database once to determine the frequent edges (called growth elements) to extend pattern α, and then scans the projected database for another pass to create the projected postfix-forest database in $children[e]$ for each frequent edge e. Each child pattern β that extends α with e is then wrapped as a child task t_β for further processing.

Worker::*setRoot*(.) is slightly different, where after frequent nodes (in terms of labels) are identified to create singleton-node patterns, it is only matched to the so-called "independent-occurrences" of the node in each data tree, i.e., the node does not have an ancestor that is also matched. This is to avoid redundancy [14].

4 TreeMiner in PrefixFPM

TreeMiner Review. TreeMiner [13] captures the ancestor-descendant relationship among nodes by assigning each node v a so-called *scope* $[\ell, r]$, 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. 8, 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 data tree, TreeMiner represents each projected transaction in $D|_\alpha$ as a pair (*tid*, *scope*) where *tid* is the transaction ID of the data tree 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. For example, Fig. 8 shows the vertical representation of initial patterns α = 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.

Recall from Section 2 that a tree T is encoded by listing vertex labels in a depth-first preorder traversal of T, and by adding a unique symbol "$" whenever we backtrack from a child to its parent. This sequence encoding of T is also called its horizontal format as shown in Fig. 8.

TreeMiner adopts prefix projection to enumerate patterns by their horizontal encodings. One way is to always extend a pattern by a frequent edge from its rightmost path to avoid redundant pattern checking, which is similar to PrefixTreeSpan as we have reviewed in Section 3.

TreeMiner adopts a different approach called "equivalence class-based extension": instead of extending a pattern α with frequent edges, TreeMiner 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 faster.

This is where the scope-list comes into play. Refer back to Fig. 5 in Section 2 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 3 valid dashed edges long the rightmost path). 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 $P_y^j \in [P]$.

While we refer readers to [13] for the details of the join, the idea is simple: 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) 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.

Implementation on PrefixFPM. To adapt the serial TreeMiner algorithm to PrefixFPM, a task t_P 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 (i.e., projected database) for P_x^i. Note that task object here maintains a list of projected databases, which is different from the PrefixFPM algorithm for PrefixTreeSpan where each task object only maintains one project database.

UDF *Task*::*setChildren*(.) computes every task object related to $Q = P_x^i$, including the extending edges $(j, y) \in [Q]$ and their scope lists, as detailed in the algorithm shown in Fig. 9. Note that each children table entry $children[Q]$

```
1: for each (i, x) ∈ [P]
2:     L1 ← scope list of P_x^i
3:     for each (j, y) ∈ [P]
4:         if i < j: continue
5:         L2 ← scope list of P_y^j
6:         Q_y^j ← join(L1, L2)   // note that Q = P_x^i
7:         if Q_y^j is frequent:   children[Q].add(Q_y^j)
```

Fig. 9. Algorithm of *TreeMinerTask*::*setChildren*(.) in PrefixFPM

to construct maintains the content a task object t_Q, including a list of Q_y^j each associated with its scope list.

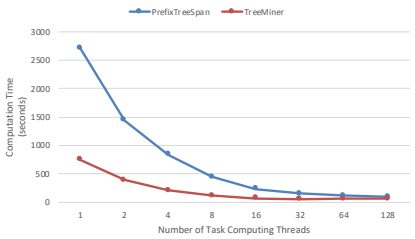
In the UDF *Task*::*setChildren*(.) of task t_P, for each extending edge (i, x) in $[P]$ (Line 1), we build $[Q]$ ($Q = P_x^i$) to be added to $children[Q]$ (Line 7). Specifically, Lines 3–6 join the scope list of Q with the scope list of every $P_y^j \in [P]$ to generate the scope list of the new pattern Q_y^j, which are then added to $children[Q]$ one by one (if Q_y^j is frequent which is judged using its scope list). This allows UDF *Task*::*get_next_child*(.) (recall Line 6 of Fig. 6) to then wrap each $children[Q]$ into a task t_Q that processes $[Q]$ (containing $\{Q_y^j\}$) for further processing.

One tricky issue is to estimate the cost of task t_Q as needed by Line 7 of Fig. 6 to determine whether to add t_Q to Q_{task} for concurrent processing or to directly process it recursively. Unlike in PrefixTreeSpan where we simply check the size of a child-task's projected database, here, we need to sum the lengths of the scope lists of $[Q] = \{Q_y^j\}$ to reflect the total task workloads, and if the sum is above threshold τ_{split}, the child task t_Q is added to Q_{task} rather than processed by the current computing thread.

Worker::*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, and only considers frequent (labeled) edges (denoted by $\mathcal{F}_2$) for subsequent edge extension. The UDF then builds the pattern object $[X]$ for each $X \in \mathcal{F}_1$, constructs its scope list with all edge-patterns $(X, Y) \in \mathcal{F}_2$, and then wraps them as the set of initial tasks to be added to Q_{task}.

5 Experiments

Summary of Algorithm Differences. One difference is that TreeMiner joins the **rightmost node** (using scope list) of two size-k frequent patterns to generate a size-$(k+1)$ pattern, which tends to have a smaller candidate set size than if we extend size-k frequent patterns with one frequent edge, as is done by the encoding scanning method of PrefixTreeSpan which basically extends the residual forest along the **rightmost matched path**. As a result, the total mining workload of TreeMiner tends to be smaller than that of PrefixTreeSpan.



	PrefixTreeSpan		TreeMiner	
Thread#	**Time (S)**	**Speedup**	**Time**	**Speedup**
1	2706	1	746.1	1
2	1441	1.88	393.6	1.90
4	840.7	3.22	214.1	3.49
8	447.3	6.05	118.7	6.29
16	241.6	11.20	72.25	10.33
32	155.5	17.41	56.07	13.30
64	117.0	23.14	59.62	12.51
128	99.27	27.26	61.01	12.23

Fig. 10. Results on the Synthetic Data

However, each task t_P in TreeMiner is essentially an equivalent class $[P]$, or a cluster of prefix projections $\{P_x^i\}$ along with their projected databases. In contrast, each task t_P in PrefixTreeSpan is simply pattern P along with its projected postfix-forest database, so that task granularity is finer than that of a task in TreeMiner, making it more amenable to concurrent processing.

Experimental Setup. We evaluate the performance of PrefixTreeSpan and TreeMiner on top of PrefixFPM, and all our codes are open-sourced at

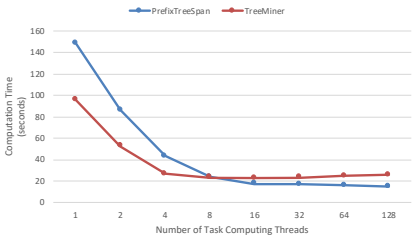
https://github.com/wenwen-Q/PrefixFPM

To thoroughly test the scale-up capability of both algorithms, we ran our programs on the BlueBlaze server donated by IBM to UAB CS Department, which has 160 CPU cores and 1TB RAM. The CPU model is IBM POWER8 with 3491 MHz. The large number of CPU cores allows us to test the scalability with 1, 2, 4, 8, 16, 32, 64 and 128 cores, and the 1TB RAM is more than enough and we actually only use a tiny fraction.

Results on a Synthetic Dataset. We follow [13] and generate a tree transaction database using a synthetic data generator [1] that creates a database of artificial website browsing behavior: a website browsing "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 [13].

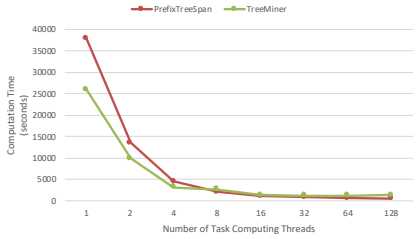
We use the default parameters for master tree: depth = 5, fan-out factor = 5, number of labels = 10, and we set the number of nodes in the master tree as 50 to generate $10,000,000$ subtree transactions. We call this dataset as *TreeGen*. We set $\tau_{sup} = 50$ and the timeout threshold as 0.01s.

Fig. 10 shows the scalability results where good speedup ratio is achieved all the way up to 16 threads, but TreeMiner does not show significant further improvement and even becomes slower beyond 32 threads. This is because TreeMiner operates on the big unit of equivalent class $[P]$ which can only keep less than 32 cores busy, and using more threads only incurs more lock contention and backfires. PrefixTreeSpan has a better scaleup ratio but due to its larger total workloads, it cannot beat TreeMiner in all settings in this experiment.



	PrefixTreeSpan		TreeMiner	
Thread#	Time (S)	Speedup	Time	Speedup
1	148.7	1	95.78	1
2	86.3	1.72	52.49	1.82
4	43.0	3.46	26.45	3.62
8	23.9	6.22	23.41	4.09
16	17.3	8.60	22.77	4.21
32	17.0	8.75	23.31	4.11
64	16.1	9.23	24.67	3.88
128	14.8	10.03	25.47	3.76

Fig. 11. Results on Treebank ($\tau_{sup} = 30,000$)



	PrefixTreeSpan		TreeMiner	
Thread#	Time (S)	Speedup	Time	Speedup
1	37778.8	1	25918.6	1
2	13502.5	1.99	9909.9	2.62
4	4472.0	6.02	3194.6	8.11
8	2032.5	13.24	2636.3	9.83
16	1089.7	24.70	1302.7	19.90
32	837.7	32.13	1157.5	22.39
64	607.5	44.31	1269.1	20.44
128	510.6	52.72	1285.0	20.12

Fig. 12. Results on Treebank ($\tau_{sup} = 10,000$)

Results on a Real Dataset. Treebank [2] is a parsed text corpus that annotates syntactic or semantic sentence structure. The XML file contains 52,851 trees. We first set $\tau_{sup} = 30,000$ and the timeout threshold as 0.1s. Since τ_{sup} is large, most patterns will be pruned early leading to limited workload for parallel mining. Fig. 11 shows the scalability results where we can see that TreeMiner's performance saturates with merely 4 CPU cores. PrefixTreeSpan scales better but the speedup is still quite limited. Interestingly, despite more workload, PrefixTreeSpan breaks a tie with TreeMiner when there are 8 CPU cores, and PrefixTreeSpan beats TreeMiner when the number of CPU cores increases further, already significantly faster than TreeMiner even when 16 CPU cores are used. This is thanks to the finer task granularity of PrefixTreeSpan which allows more CPU cores to be utilized.

We then set $\tau_{sup} = 10,000$ so that most patterns will be valid allowing for more parallelism. Fig. 12 shows the scalability results where we can see that TreeMiner's performance now saturates at up to 32 CPU cores thanks to the more parallelism provided by a lower τ_{sup}. PrefixTreeSpan scales even better and achieves an impressive 52.72× speedup with 128 cores, and ultimately beats TreeMiner by more than 2.5×.

To summarize, while the total mining workload of TreeMiner can be much smaller than that of PrefixTreeSpan due to the scope list join technique, it does limit TreeMiner's capability for massively parallel execution due to the larger task granularity. When there are enough CPU cores, PrefixTreeSpan can be a better choice that is worth trying out, and can ultimately beat TreeMiner by several times. We remark that these conclusions are made assuming that the underlying parallel execution engine is able to utilize as much parallelism as is

available, which is ideally provided by PrefixPFM as explained in [12] which is recently proposed to overcome the IO-bound execution bottleneck of a few prior systems and solutions.

6 Conclusion

This paper implemented the parallel versions of two frequent embedded subtree mining algorithms, PrefixTreeSpan and TreeMiner, on top of the PrefixFPM system that follows a prefix-projection programming paradigm and that is able to fully carry out the parallelism potential of the algorithms on top.

A few new insights are obtained: (i) PrefixTreeSpan does not beat TreeMiner in the serial setting as what was claimed in PrefixTreeSpan's paper [14], likely because [14] implemented TreeMiner as an Apriori-like algorithm rather than a PrefixSpan-like one. However, TreeMiner's workload optimization requires a larger task granularity which limits its potential for parallel execution, and could be beaten by PrefixTreeSpan when enough CPU cores are available.

Acknowledgments. This work was partially supported by NSF OAC-1755464 and DGE-1723250.

References

1. Tree Generator. https://github.com/zakimjz/TreeGen.
2. Treebank. http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html#treebank.
3. C. C. Aggarwal and J. Han, editors. *Frequent Pattern Mining*. Springer, 2014.
4. T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In R. L. Grossman, J. Han, V. Kumar, H. Mannila, and R. Motwani, editors, *SDM*, pages 158–174. SIAM, 2002.
5. J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872, 2007.
6. Y. Chi, Y. Xia, Y. Yang, and R. R. Muntz. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Trans. Knowl. Data Eng.*, 17(2):190–202, 2005.
7. R. Cooley, B. Mobasher, and J. Srivastava. Web mining: Information and pattern discovery on the world wide web. In *ICTAI*, pages 558–567. IEEE Computer Society, 1997.
8. T. Kudo, E. Maeda, and Y. Matsumoto. An application of boosting to graph classification. In *NIPS*, pages 729–736, 2004.
9. J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 215–224, 2001.
10. B. A. Shapiro and K. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Comput. Appl. Biosci.*, 6(4):309–318, 1990.
11. C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi. Efficient pattern-growth methods for frequent tree pattern mining. In H. Dai, R. Srikant, and C. Zhang, editors, *PAKDD*, volume 3056 of *Lecture Notes in Computer Science*, pages 441–451. Springer, 2004.

12. D. Yan, W. Qu, G. Guo, and X. Wang. Prefixfpm: A parallel framework for general-purpose frequent pattern mining. In *(ICDE)*, 2020.
13. M. J. Zaki. Efficiently mining frequent trees in a forest. In *SIGKDD*, pages 71–80, 2002.
14. L. Zou, Y. Lu, H. Zhang, and R. Hu. Prefixtreeespan: A pattern growth algorithm for mining embedded subtrees. In K. Aberer, Z. Peng, E. A. Rundensteiner, Y. Zhang, and X. Li, editors, *WISE*, volume 4255 of *Lecture Notes in Computer Science*, pages 499–505. Springer, 2006.