IFIN$^+$: A Parallel Incremental Frequent Itemsets Mining in Shared-Memory Environment

Van Quoc Phuong Huynh[1](✉), Josef Küng[1], Markus Jäger[1],
and Tran Khanh Dang[2]

[1] Faculty of Engineering and Natural Sciences (TNF),
Institute for Application Oriented Knowledge Processing (FAW),
Johannes Kepler University (JKU), Linz, Austria
`{Vqphuynh, jkueng, mjaeger}@faw.jku.at`
[2] Faculty of Computer Science and Engineering,
HCMC University of Technology, HCM City, Vietnam
`khanh@hcmut.edu.vn`

Abstract. In an effort to increase throughput for IFIN, a frequent itemsets mining algorithm, in this paper we introduce a solution, called IFIN$^+$, for parallelizing the algorithm IFIN with shared-memory multithreads. The inspiration for our motivation is that today commodity processors' computational power is enhanced with multi physical computational units; and therefore, exploiting full advantage of this is a potential solution for improving performance in single-machine environments. Some portions in the serial version are changed in means which increase efficiency and computational independence for convenience in designing parallel computation with Work-Pool model, be known as a good model for load balance. We conducted experiments to evaluate IFIN$^+$ against its serial version IFIN, the well-known algorithm FP-Growth and other two state-of-the-art ones FIN and PrePost$^+$. The experimental results show that the running time of IFIN$^+$ is the most efficient, especially in the case of mining at different support thresholds in the same running session. Compare to its serial version, IFIN$^+$ performance is improved significantly.

Keywords: Incremental · Parallel · Frequent itemsets mining · Data mining · Big Data · IPPC-Tree · IFIN · IFIN$^+$

1 Introduction

Frequent itemsets mining can be briefly described as follows. Given a dataset of n transactions $D = \{T_1, T_2, \ldots, T_n\}$, the dataset contains a set of m distinct items $I = \{i_1, i_2, \ldots, i_m\}$, $T_i \subseteq I$. A k-itemset, IS, is a set of k items ($1 \leq k \leq m$). Each itemset IS possesses an attribute, *support*, which is the number of transactions containing IS. The problem is featured by a support threshold ε which is the percent of transactions in the whole dataset D. An itemset IS is called frequent itemset iff $IS.support \geq \varepsilon * n$. The problem is to discover all frequent itemsets existing in D.

T.K. Dang et al. (Eds.): FDSE 2017, LNCS 10646, pp. 121–138, 2017.
https://doi.org/10.1007/978-3-319-70004-5_9

Discovering frequent itemsets in a large dataset is an important problem in data mining. In Big Data era, this problem as well as other mining techniques has been being challenged by very large volume and high velocity of datasets. Fortunately, nowadays, RAM memory has larger capacity and becomes much cheaper, and commodity processors' computational power is enhanced considerably with multi physical computational units. To take this advantage and confront with the challenge, we propose an algorithm, named IFIN$^+$, as a solution for parallelizing our previous work IFIN [18] (Incremental Frequent Itemsets Nodesets) algorithm with shared-memory multithreads. The purpose is to improve the performance IFIN by increasing the throughput in single-machine environments. In general, IFIN algorithm encompasses four phases: (1) IPPC-Tree (Incremental Pre-Post-Order Coding Tree) construction, (2) Frequent 2-itemsets generation, (3) Nodesets for frequent 2-itemsets generation, (4) Frequent k-itemsets generation ($k > 2$). In that the first three phases take most of the mining time and can be divided into small independent chunks of work, these three phases are separately parallelized and synchronized at the end of each phase. These synchronizations will delay the next processing step and result in longer mining time if load balance is not guaranteed. To avoid this problem, therefore, all these three processing phases are designed in Work-Pool model, a well-known model for load balance, in which all workers continuously fetch and process small chunks of tasks until there are no more tasks in the work pool. Besides, the second and third phases are changed to increase the independence for parallelization. By that solution, the running time of IFIN$^+$ is improved significantly compared to its serial version IFIN.

The rest of the paper is organized as follows. In Sect. 2, some related works are presented. Section 3 introduces the IPPC-Tree structure, some relevant algorithms and parallel solution for loading the IPPC-Tree. The algorithm IFIN$^+$ is mentioned in Sect. 4 based on preliminaries in Sect. 5 and followed with experiments in Sect. 6. Finally, conclusions are given in Sect. 7.

2 Related Works

Problem of mining frequent itemsets was started up by Agrawal and Srikant with algorithm Apriori [1]. This algorithm generates candidate ($k + 1$)-itemsets from frequent k-itemsets at the $(k + 1)^{th}$ pass and then scans dataset to check whether a candidate ($k + 1$)-itemsets is a frequent one. Many previous works were inspired by this algorithm. Algorithm Partition [8] aim at reducing I/O cost by dividing dataset into non-overlapping and memory-fitting partitions which are sequentially scanned in two phases. In the first phase, local candidate itemsets are generated for each partition, and then they are checked in the second one. DCP [9] enhances Apriori by incorporating two dataset pruning techniques introduced in DHP [10] and using direct counting method for storing candidate itemsets and counting their support. In general, Apriori-like methods suffer from two drawbacks: a deluge of generated candidate itemsets and/or I/O overhead caused of repeatedly scanning dataset. Two other approaches, which are more efficient than Apriori-like methods, are also proposed to solve the problem: (1) frequent pattern growth adopting divide-and-conquer with FP-Tree structure and FP-Growth [2], and (2) vertical data format strategy in Eclat [11].

FP-Growth and algorithms based on it such as [12, 13] are efficient solutions as unlike Apriori, they avoid many times of scanning dataset and generation-and-test. However, they become less efficient when datasets are sparse. While algorithms based on FP-Growth and Apriori use a horizontal data format; Eclat and some other algorithms [8, 14, 15] apply vertical data format, in which each item is associated a set of transaction identifiers, Tids, containing the item. This approach avoids scanning dataset repeatedly, but a huge memory overhead is expensed for sets of Tids when dataset becomes large and/or dense. Recently, two remarkably efficient algorithms are introduced: FIN [4] with POC-Tree and PrePost$^+$ [5] with PPC-Tree. These two structures are prefix trees and similar to FP-Tree, but the two mining algorithms use additional data structures, called Nodeset and N-list respectively, to significantly improve mining speed.

To better deal with the challenge of high volume in Big Data, in addition to the ideas of parallel mining for existing algorithms such as [16] for Eclat, incremental mining approaches are also considered as a potential solution. Some typical algorithms in this approach are algorithm FELINE [3] with CATS-Tree structure and IM_WMFI [17] for mining weighted maximal frequent itemsets from incremental datasets. These methods are both based on the well-known FP-Tree for its efficiency.

3 IPPC Tree Construction

IPPC-Tree is a prefix tree and possesses two properties, Properties 1 and 2. IPPC-Tree includes one root labeled "*root*" and a set of prefix sub trees as its children. Each node in the sub trees contains the following attributes:

- *item-name*: the name of an item in a transaction that the node registered.
- *support* (or *local support* of an item): the number of transactions containing the node's *item-name*. Conversely, *global support* of an item, without concerning nodes, is the number of transactions containing the item.
- *pre-order* and *post-order*: two global identities in the IPPC-Tree which are sequent numbers generated by traversing the tree with pre and post order respectively.

Property 1: For a given IPPC-Tree, there exist no duplication nodes with the same item in a path of nodes from the root to a leaf node.

Property 2: In a given IPPC-Tree, the *support* of a parent node must be greater than or equal to the sum of all its children's *support*.

IPPC-Tree is a combination of (1) the idea of flexible and local order of items in a path from the root to a leaf node in CATS-Tree [3] and (2) the PPC-Tree [5] which each node in PPC-Tree is identified by a pair of codes: *pre-order* and *post*-order. The construction of the IPPC-Tree does not require a given support threshold. The tree is a compact and information-lossless structure of the whole items of all transactions in a given dataset *D*. Local order of items in a path of nodes from the root to a leaf is flexible and can be changed to improve compression while remaining Property 2. To guarantee this, two conditions for swapping are as follows.

Child Swapping: A node can be swapped with its child node if it has only one child node, its *support* is equal to its child's *support*, and the number of child nodes of its child is not greater than one.

Descendant Swapping: Given a path of k nodes $N_1 \rightarrow N_2 \rightarrow \cdots \rightarrow N_k (k > 2)$, is parent node of N_j $(i < j)$; if every node N_i $(i < k)$ satisfies the Child Swapping condition, node N_1 can be swapped with descendant node N_k.

To demonstrate the building process of an IPPC-Tree, the Fig. 1 records transaction by transaction in Table 1 inserted into an empty IPPC-Tree. Initially, the tree has only the root node, and transaction 1 (*b*, *e*, *d*, *f*, *c*) is inserted as it is in Fig. 1(a). The Fig. 1 (b) is of the tree after transaction 2 (*d*, *c*, *b*, *g*, *f*, *h*) is added. The item *b* in transaction 2 is merged with node *b* in the tree. Although transaction 2 does not contain item *e*, but its common items *d*, *f* and *c* can be merged with the corresponding nodes. The item *d* is found common, so it is merged with node *d* after node *d* is swapped[1] with node *e* to guarantee the Property 2. Similarly, items *f* and *c* are merged with node *f* and *c* respectively; and the remaining items *g* and *h* are inserted as a child branch of node *c*. In Fig. 1(c), transaction 3 (*f*, *a*, *c*) is processed. Common item *f* is found that can be merged with node *f*, so node *f* is swapped with node *b*. Item *c* is also a common one, but it is not able to be merged with node *c* as node *d* does not satisfy the Descendant

Table 1. Example transaction dataset

ID	Items in transactions	ID	Items in transactions
1	*b*, *e*, *d*, *f*, *c*	4	*a*, *b*, *d*, *f*, *c*, *h*
2	*d*, *c*, *b*, *g*, *f*, *h*	5	*b*, *d*, *c*
3	*f*, *a*, *c*		

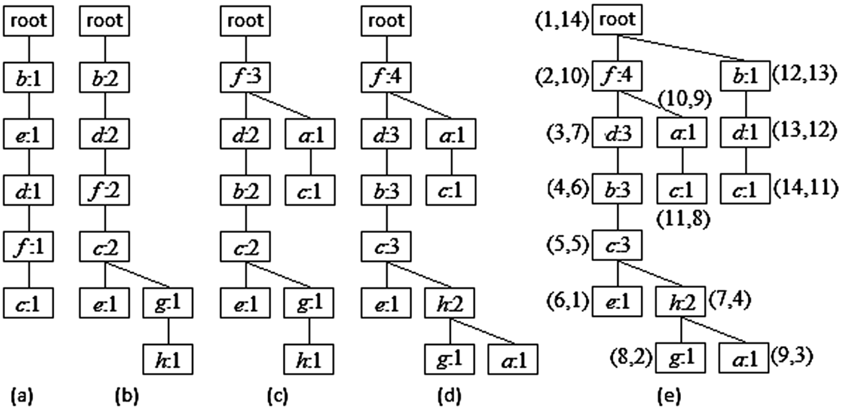


Fig. 1. An illustration for constructing an IPPC-Tree on example transaction dataset

---

[1] Swapping two nodes is simply exchanging one's item name to that of the other.

Swapping condition with node *c*. Then the items *a* and *c* are added as a branch from node *f*. When transaction 4 (*a*, *b*, *d*, *f*, *c*, *h*) is added in Fig. 1(d), common items *f*, *d*, *b* and *c* are merged straightforwardly with corresponding nodes *f*, *d*, *b* and *c*. The remaining items *a* and *h* are then inserted into the sub tree having root node *c*. The item *h* is found common with node *h* in the second branch. Node *h* and item *h*, therefore, are merged together after node *h* is swapped with node *g*. The last item *a* is then inserted as a new child branch from node *h*. Insertion of transaction 5 (*b*, *d*, *c*) is depicted in Fig. 1 (e). All items in transaction 5 are common, but they cannot be merged with nodes *b*, *d* and *c* as node *f* does not guarantee the Child Swapping condition. Thus, transaction 5 is added as a new child branch of root node.

After the dataset has been processed, each node in the IPPC-Tree is attached with a pair of sequent numbers (*pre-order*, *post-order*) by scanning the tree with pre order and post order traversals through procedure **AssignPrePostOrder**. For an example, node (4, 6) is identified by *pre-order* = 4 and *post-order* = 6, and it registers item *b* with *support* = 3. Above are all concepts of IPPC-Tree construction; for a formal and detail description, refer to IFIN algorithm [18].

```
Procedure AssignPrePostOrder (Node R)
    // PreOrder and PostOrder are initialized at 1.
    R.pre-order ← PreOrder; PreOrder++;
    For Each child node N of R Do AssignPrePostCode(N);
    R.post-order ← PostOrder; PostOrder++;
```

As the IPPC-Tree construction is independent to the support threshold and the global order of items in a dataset, a built IPPC-Tree from a dataset D is reusable for different support thresholds and changed dataset $D' = D \pm \Delta D$. To complete providing incremental ability for the IPPC-Tree, methods of storing and loading for the tree and item list $\mathcal{L}$ must be proposed, in which the data format and algorithms are their two features. For the simplicity of storing and loading for $\mathcal{L}$, this detail will not be mentioned here for concision. Besides *item-name*, *support*, etc., the important information for loading a node is its parent's information to identify where the node was in the built tree. By utilizing the *pre-order* (or *post-order*), the global identity, the requirement is resolved. The data format for a single node record is as follows.

<parent's pre-order>:<pre-order>:<post-order>:<item-name>:<support>

We employ Breadth-First-Search traversal to store the IPPC-Tree. In fact, the storing phrase can utilize other strategies such as pre order traversal, but the sequence of node records generated by Breadth-First-Search traversal is more convenient in loading phrase. The reason is that the records of all child nodes with the same parent node are continuous together. By storing the data record of each single node on a line, the stored data for the example tree in Fig. 1(e) is in right column of Table 2. The algorithm for loading the IPPC-Tree, procedure **LoadIPPCTree**, is presented in Table 2.

When the dataset becomes larger with progress of additional data accumulated, the stored data for the built tree is also bigger; and the tree loading takes most of the

Table 2. Loading algorithm and data format for IPPC-Tree

```
Procedure LoadIPPCTree(File $F$, Root $R$, $\mathcal{L}$)
  Load item list $\mathcal{L}$; TransCount ← 0;
  Load sequentially TransCount and R from F;
  ParentNode ← R; NodeList ← ∅;
  For Each line L in data file F
    Create a node N from L;
    parentID ← <parent's pre-order>;
    Add N into the end of NodeList;
    While(parentID <> ParentNode.pre-order){
      ParentNode ← NodeList[0];
      Remove ParentNode from NodeList;}
    Add N as a child of ParentNode;
  End For
```

```
<No. Trans.>
-1:1:14:root:0
1:2:10:f:4
1:12:13:b:1
2:3:7:d:3
2:10:9:a:1
12:13:12:d:1
3:4:6:b:3
10:11:8:c:1
13:14:11:c:1
4:5:5:c:3
5:6:1:e:1
5:7:4:h:2
7:8:2:g:1
7:9:3:a:1
```

tree construction time. Therefore, improving efficiency for procedure **LoadIPPCTree** is necessary. The IPPC-Tree loading in serial version comprises three tasks for each line of data: (1) read a line, (2) parse the line and build a corresponding node, (3) connect the node to the tree. We realize that the second task takes approximately 75% of the total time; and fortunately the second task is performed in main memory and not interrupted by waiting for I/O. The parallelization design for the IPPC-Tree loading is depicted in the Fig. 2. The file of a built IPPC-Tree is divided into n chunks of l lines and processed by k threads ($k \ll n$). The last chunk's number of lines may be lesser than l. Each time, a thread reads a chunk into its local buffer and sequentially creates a node for each data line. A shared reference array *FArray* is maintained for all created nodes, and connections between nodes for the IPPC-Tree will be established after node creation stage has finished. We can see that the access address spaces of individual threads in the *FArray* are different. Hence, independence between threads is guaranteed. For tracing the parent-child relationship between nodes in connection stage, a shared integer array, *IndexIDArray*, is used to map from a node's index to its parent node's ID. The separation of address spaces of threads in this array is the same as that of *FArray*. The parallelization is given in procedure **ParallelLoadIPPCTree**.

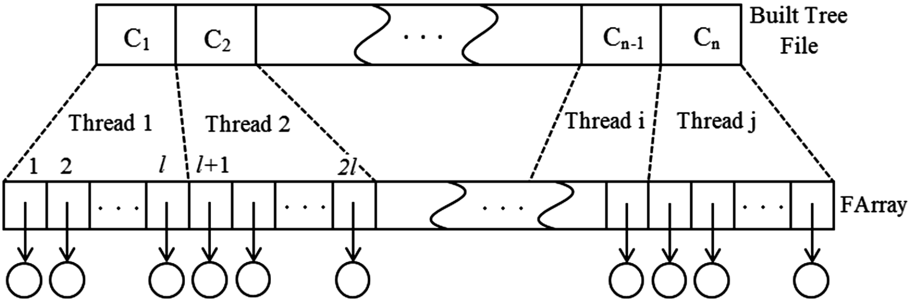


Fig. 2. The concept of parallelization for IPPC-Tree loading

```
ParallelLoadIPPCTree(FileReader F, Root R, $\mathcal{L}$, ThreadCount)
Load item list $\mathcal{L}$; TransCount ← 0;
Load sequentially TransCount and R from F;
Initialize FArray and IndexIDArray with length R.post-order
FArray[0] ← R; lineIndex ← 0;
For i From 1 To ThreadCount
  Start LoadingThread(F, FArray, IndexIDArray, lineIndex);
After all Threads finished, main execution continues.
parentIndex ← 0; parentNode ← FArray[parentIndex];
For i From 1 To FArray.length {
  While(IndexIDArray[i] <> parentNode.pre-order){
    parentIndex++;
    parentNode ← FArray[parentIndex];}
  FArray[i] ← parentNode;
  parentNode.childList.add(FArray[i]);
}
```

```
LoadingThread(FileReader F, FArray, IndexIDArray, lineIndex)
startIndex ← 0; lineCount ← 0;
While(Work-Pool <> ∅){
  Mutually-exclusive-region {
    startIndex ← lineIndex;
    Load a chunk from F into Buffer;
    lineCount ← number of loaded lines;
    lineIndex += lineCount; }
  For i From startIndex To (startIndex + lineCount){
    Parse the next line in Buffer to generate a node N and
    its parent ID;
    FArray[i] ← N; IndexIDArray[i] ← (parent ID of N);
  }
}
```

4 Preliminaries

In this subsection, some IPPC-Tree related definitions and lemmas are introduced as preliminaries for IFIN algorithm. In addition to the IPPC-Tree, another output of Algorithm 1 is the increasingly ordered list of items based on their frequencies $\mathcal{L} = \{I_1, I_2, \ldots, I_n\}$. For the convenience of expressing the relative order between two items, we denote $I_i \prec I_j$ to indicate that I_i is in front of I_j in $\mathcal{L}$ $(1 \le i < j \le n)$. There are two premises of traversing a tree with pre order and post order as follows:

Premise 1: Traversing a tree to process a work at each node with pre order, it must be that (1) N_1 is an ancestor of N_2 or (2) N_1 and N_2 stay in two different branches (N_1 in the left and N_2 in the right) iff the work is done at N_1 before N_2.

Premise 2: Traversing a tree to process a work at each node with post order, it must be that (1) N_1 is an ancestor of N_2 or (2) N_1 and N_2 stay in two different branches (N_1 in the right and N_2 in the left) iff the work is done at N_2 before N_1.

By applying a work which assigns an increasingly global number at each node on Premises 1 and 2, two following lemmas are directly deduced.

Lemma 1: For any two different nodes N_1 and N_2 in the IPPC-Tree, N_1 is an ancestor of N_2 iff N_1*.pre-order* < N_2*.pre-order* and N_1*.post-order* > N_2*.post-order.*

Lemma 2: For any two nodes N_1 and N_2 in two different branches of the IPPC-Tree, N_1 is in the left branch and N_2 in the right one iff N_1*.pre-order* < N_2*.pre-order* and N_1*.post-order* < N_2*.post-order.*

Definition 1 (Nodeset of an item): Given an IPPC-Tree, the *nodeset* of an item *I*, denoted by NS_I, is a set of all nodes in the IPPC-Tree with ascending order of *pre-order* and *post-order* in which all the nodes register the same item *I*.

In case N_1 and N_2 register the same item, N_1 and N_2 must be in two different branches because of Property 1. By traversing the IPPC-Tree with pre order, all nodes with the same item *I*, sequentially from the left-most branch to the right-most one, are added into the end of the list of nodes reserved for the item *I*. Hence, according to Lemma 2, the increasing orders of both *pre-order* and *post-order* are guaranteed. Finally, we have *nodesets* for all items in $\mathcal{L}$. For an instance, the *nodeset* for item *c* in the example IPPC-Tree Fig. 1(e) will be $NS_c = \{(5,5,3),(11,8,1),(14,11,1)\}$. Here, each node *N* is depicted by a triplet of three numbers (*N.pre-order*, *N.post-order*, *N.support*).

Lemma 3: Given an item *I* and its nodeset is $NS_I = \{N_1, N_2, \ldots N_l\}$, the *support* (or *global support*) of item *I* is $\sum_{i=1}^{l} N_i$*.support.*

Rationale: Refer to IFIN [18].

Definition 2 (Nodeset of a *k*-itemset, $k \geq 2$): Given two ($k-1$)-itemsets $P_1 = p_1p_2\ldots p_{k-2}p_{k-1}$ with *nodesets* NS_{P_1} and $P_2 = p_1p_2\cdots p_{k-2}p_k$ with *nodeset* NS_{P_2} ($p_1 \prec p_2 \prec \cdots \prec p_k$), the *nodeset* of *k*-itemset $P = p_1p_2\ldots p_{k-2}p_{k-1}p_k$, NS_P, is defined as follows.

$$NS_P = \left\{ D_k \,\middle|\, \left[\underset{D_k \in NS_{p_1} \wedge D_k \in NS_{P_2}}{D_k = Descendant(N_i,\, M_j)\; with\, N_i \in NS_{P_1} \wedge M_j \in NS_{p_2}} \right. \right\}$$

Function $Descendant(N_i, M_j)$ means that there has been an ancestor-descendant relationship between N_i and M_j, and the output is the descendant node.

Lemma 4: Given a *k*-itemset *P* and its nodeset is $NS_P = \{N_1, N_2, \ldots N_l\}$, the *support* of the itemset *P* is $\sum_{i=1}^{l} N_i$*.support.*

Proof. Refer to IFIN [18].

Given two ($k-1$)-itemsets $P_1 = p_1p_2\ldots p_{k-2}p_{k-1}$ and $P_2 = p_1p_2\ldots p_{k-2}p_k$ with their *nodesets* $NS_{P_1} = \{N_1, N_2, \ldots, N_{l1}\}$ and $NS_{P_2} = \{M_1, M_2, \ldots, M_{l2}\}$; at first glance, the computational complexity of generating *nodeset* NS_P for *k*-itemset $P = p_1p_2\ldots p_k$ is O($l1 * l2$). In fact this complexity can be reduced significantly to O($l1 + l2$), a linear cost, by utilizing Lemmas 1 and 2. For each pair of nodes N_i and M_j $(1 \leq i \leq l1, 1 \leq j \leq l2)$, there are the following five cases:

1. (N_i.*pre-order* > M_j.*pre-order*) ∧ (N_i.*post-order* > M_j.*post-order*): The relationship between N_i and M_j is not an ancestor-descendant relationship, so no node is added to NS_P. Certainly, M_j also does not have this relationship with remaining nodes in NS_{P_1} as increasing orders of both *pre-order* and *post-order* in *nodesets*. Therefore, M_{j+1} is selected as the next node for the next comparison.
2. (N_i.*pre-order* > M_j.*pre-order*) ∧ (N_i.*post-order* < M_j.*post-order*): N_i is added to NS_P as N_i is the descendant node of M_j. Consequently, N_{i+1} is selected as the next node for the next comparison.
3. (N_i.*pre-order* < M_j.*pre-order*) ∧ (N_i.*post-order* > M_j.*post-order*): Similar to the case 2, M_j is added to NS_P, and M_{j+1} is the next node for the next comparison.
4. (N_i.*pre-order* < M_j.*pre-order*) ∧ (N_i.*post-order* < M_j.*post-order*): This case is similar to the case 1; and N_{i+1}, therefore, is the next node for the next comparison.
5. $N_i \equiv M_j$: This identical node N_i is added to NS_P. Two new nodes N_{i+1} and M_{j+1} are selected for next comparison.

Based on analyses above, the algorithm for generating a *nodeset*, the procedure **NodesetGenerator**, is as follows.

```
Procedure NodesetGenerator(Nodeset NS1, Nodeset NS2)
i ← 1; j ← 1; NS;
While((i < NS1.size) ∧ (j < NS2.size))
  If(NS1[i].pre-order > NS2[j].pre-order)
    If(NS1[i].post-order > NS2[j].post-order) j++;
    Else {NS ← NS ∪ NS1[i]; i++;}
  Else If(NS1[i].pre-order < NS2[j].pre-order)
    If(NS1[i].post-order < NS2[j].post-order) i++;
    Else {NS ← NS ∪ NS2[j]; j++;}
  Else {NS ← NS ∪ NS1[i]; i++; j++;}
End While
Return NS;
```

It is easy to see that the increasing order of nodes in *NS* is guaranteed as these nodes are inserted to the end of *NS* in that order. Therefore, *NS* is also a *nodeset*.

Lemma 5 (Superset equivalence): Given an item I and an itemset P ($I \notin P$), if the *support* of P is equal to the *support* of $P \cup \{I\}$, the *support* of $A \cup P$ is equal to the *support* of $A \cup P \cup \{I\}$. Here $(A \cap P = \emptyset) \wedge (I \notin A)$.

Proof. Refer to IFIN [18].

5 Algorithm IFIN$^+$

In this section, we present the algorithm IFIN$^+$ based on its serial version IFIN and the preliminaries introduced in the previous section. There are three running modes in algorithm IFIN: (1) **Just-Building-Tree**, just build an IPPC-Tree from a dataset *D*; (2) **Incremental**, load an IPPC-Tree from a previously stored tree Tree-*D* and build up the loaded IPPC-Tree with an incremental dataset *D*; (3) **Just-Loading-Tree**, just load an IPPC-Tree from a previously stored tree Tree-*D*. Each mode can be performed with different support thresholds (lines 5–32) with only one time of constructing the IPPC-Tree (lines 1–4). Lines 9–16 generate list of candidate 2-itemsets *C2* as well as

```
Algorithm 2: IFIN
Input: Stored tree Tree-D, incremental dataset D, $\varepsilon$
Output: Set of frequent k-itemsets L
1.  Create the root node R; $\mathcal{L} \leftarrow \emptyset$;
2.  If(Tree-D <> null) LoadIPPCTree(Tree-D, R, $\mathcal{L}$);
3.  If(D <> null) BuildIPPCTree(D, R, $\mathcal{L}$);
4.  HasMap<itemset, support> C2 $\leftarrow \emptyset$;
5.  LOOP:
6.  Ask for a new support threshold $\varepsilon$ or exit;
7.  Filter frequent items in $\mathcal{L}$ based on $\varepsilon$ and add to L1;
8.  If(C2 <> $\emptyset$) Goto SKIP;
9.  Scan Each node N in IPPC-Tree with pre order traversal
10.   $I_N \leftarrow$ N.item-name;
11.   For Each ancestor A of N
12.     $I_A \leftarrow$ A.item-name;
13.     If($I_N \prec I_A$) C2.add($I_N I_A$, $I_N I_A$.support + N.support);
14.     Else C2.add($I_A I_N$, $I_A I_N$.support + N.support);
15.   End For
16. End Scan
17. SKIP:
18. L2' $\leftarrow$ L2; L2 $\leftarrow \emptyset$;
19. Filter frequent itemsets in C2 based on $\varepsilon$ and add to L2;
20. Scan Each node N in IPPC-Tree with pre order traversal
21.   $I_N \leftarrow$ N.item-name;
22.   For Each ancestor A of N
23.     $I_A \leftarrow$ A.item-name;
24.     If($I_N \prec I_A$) IS $\leftarrow I_N I_A$;
25.     Else IS $\leftarrow I_A I_N$;
26.     If((IS $\in$ L2)$\wedge$(IS $\notin$ L2')) $nodeset_{IS}$.add(N);
27.   End For
28. End Scan
29. L $\leftarrow$ L $\cup$ L1; L $\leftarrow$ L $\cup$ L2;
30. For Each $I_i I_j$ $\in$ L2
31.   GenerateFrequentItemsets($I_i I_j$, $\{I | I \in L1, I_j \prec I\}$, $\emptyset$);
32. Goto LOOP;
```

their respective *supports*. This task is ignored if the current running session performs for following times of mining with other support thresholds. Lines 20–28 create the corresponding *nodeset* for each frequent 2-itemsets in *L2*. From the second time of mining, just new frequent 2-itemsets' *nodesets* are generated. In lines 30–31, each frequent 2-itemset in *L2* will be extended by the recursive procedure **GenerateFrequentItemsets** to discover longer frequent itemsets.

The second phase, frequent 2-itemsets generation, is performed by lines 9–19. Remaining the encoding for each 2-itemset as an ordered string of item names and set of 2-itemsets *C2* as a hash map in the parallelized second phase will cause the running time is not improved, even worse, because of sharing and synchronization between threads when updating 2-itemsets' *supports* in *C2*. To overcome this, each item is encoded with an integer which is its position in the item list $\mathcal{L}$ ($|\mathcal{L}| = m$); and instead of a shared hash map *C2*, a $m \times m$ matrix of integers M_t is reserved for t^{th} thread. Two elements $M_t(i, j)$ and $M_t(j, i)$ partially indicate support for a 2-itemset comprising two items I_i and I_j at positions i and j in $\mathcal{L}$ respectively. In this phase, the work pool is the built IPPC-Tree, and tasks in the work pool are the built tree's direct sub-trees. When a

```
Algorithm 3: IFIN+
Input: Stored tree Tree-D, incremental dataset D, $\varepsilon$, ThreadCount
Output: Set of frequent k-itemsets L
1.  Create the root node R; $\mathcal{L} \leftarrow \emptyset$;
2.  If(Tree-D <> null) LoadIPPCTree(Tree-D, R, $\mathcal{L}$, ThreadCount);
3.  If(D <> null) BuildIPPCTree(D, R, $\mathcal{L}$);
4.  Scan Each node N in IPPC-Tree with pre order traversal
5.    $Nodeset_N$.add(N);
6.  LOOP:
7.  Ask for a new support threshold $\varepsilon$ or exit;
8.  Filter frequent items in $\mathcal{L}$ based on $\varepsilon$ and add to L1;
9.  If($M_1$ <> null) Goto SKIP;
10. Initialize matrixes $M_{t=[1,\ ThreadCount]}$;
11. childIndex $\leftarrow$ 0;
12. For t From 1 To ThreadCount
13.   Start ItemsetGenThread(R, childIndex, $M_t$);
    After all threads finished, main execution continues.
14. $M_1[i,j]_{i,j=[0,\ |\mathcal{L}|-1];\ i<j} = \sum_{t=[1,\ ThreadCount]} M_t[i,j]_{i,j=[0,\ |\mathcal{L}|-1];\ i<j}$;
15. SKIP:
16. L2' $\leftarrow$ L2; L2 $\leftarrow \emptyset$;
17. For each $M_1[i,\ j] \geq \varepsilon * number\_of\_transactions, (i < j)$ L2.add($\mathcal{L}[i]\mathcal{L}[j]$);
18. index $\leftarrow$ 0;
19. For t From 1 To ThreadCount
20.   Start NodesetGenThread(L2\L2', index);
    After all threads finished, main execution continues.
21. L $\leftarrow$ L $\cup$ L1; L $\leftarrow$ L $\cup$ L2;
22. For Each $I_iI_j$ $\in$ L2
23.   GenerateFrequentItemsets($I_iI_j$, $\{I | I \in L1, I_j \prec I\}$, $\emptyset$);
24. Goto LOOP;
```

```
ItemsetGenThread(R, childIndex, Matrix)
  While(childIndex < R.childList.length){
    Mutually-exclusive-region {
      subTree = R.childList[childIndex];
      childIndex++;
    }
    Scan Each node N of subTree with pre order traversal
      i = mapToIndex(N.item-name);
      For Each ancestor A of N {
        j = mapToIndex(A.item-name);
        Matrix[i,j] = Matrix[i,j] + N.support;}
    End Scan
  }
  For i From 0 To Matrix.with-1
    For j From i+1 To Matrix.with-1
      Matrix[i,j] = Matrix[i,j] + Matrix[j,i];
```

```
NodesetGenThread(New2Itemsets, index)
  While(index < New2Itemsets.length){
    Mutually-exclusive-region {
      IJ = New2Itemsets[index]; index++;
    }
    $Nodeset_{IJ}$ = NodesetGenerator($Nodeset_{I}$, $Nodeset_{J}$);
  }
```

thread has no longer sub-trees to process, it calculates local supports for 2-itemsets I_iI_j through Eq. (1). After threads have completed their works, aggregation and filter operators are performed to achieve the global supports for all 2-itemsets following Eq. (2) and to extract frequent 2-itemsets.

$$Local\_Support_t(I_iI_j) = M_t(i,j) + M_t(j,i), (i<j) \tag{1}$$

$$Support(I_iI_j) = \sum\nolimits_t Local\_Support_t(I_iI_j), (i<j) \tag{2}$$

The third phase, nodesets generation for frequent 2-itemsets, is executed in lines 20–28. The same problems of sharing and synchronization in the second phase happen as threads may concurrently update the same nodeset of a certain frequent 2-itemset. For the purpose of independent execution between threads, nodesets for items need to be generated in advance, and nodesets for frequent 2-itemsets are produced from two nodesets of componential items. The work pool is now a list of frequent 2-itemsets, and threads independently retrieve items' nodesets and generate nodesets for frequent 2-itemsets. Base on explanations above, the algorithm IFIN$^+$ is designed as follows.

By the same means as IFIN for generating frequent *k*-itemsets ($k > 2$), the procedure **GenerateFrequentItemsets** searches on a space of itemsets which is demonstrated by a set-enumeration tree [6] constructing from the list of ordered frequent items $L1$. An example of the search space for the dataset in Table 1 with support threshold $\varepsilon = 0.6$ is visualized in Fig. 3. The procedure employs two pruning strategies to greatly narrow down the search space. The first strategy is that if P is not a frequent itemset, its supersets are not either, and the second one is the superset equivalence introduced in Lemma 5. There are three input parameters for procedure **GenerateFrequentItemsets**: (1) FIS is a frequent itemset which will be extended; (2) CI is a list of candidate items used to expand the FIS with one more item; (3) Parent_FISs is the set of frequent itemsets generated at the parent of FIS in the set-enumeration tree. The detail procedure is as follows.

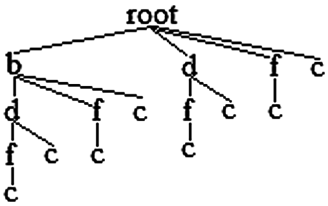


Fig. 3. Set-enumeration tree for example dataset Table 1, support threshold $\varepsilon = 0.6$

```
Procedure GenerateFrequentItemsets(FIS, CI, Parent_FISs)
nextCI ← ∅; eqItems ← ∅; extFISs ← ∅;
For Each item I ∈ CI
  IS = (FIS\{FIS.last_item}) ∪ {I};
  extIS = FIS ∪ {I};
  extIS.nodeset ← NodesetGenerator(FIS.nodeset,IS.nodeset);
  If(extIS.support = FIS.support) eqItems.add(I);
  Else If(extIS is an frequent itemset){
    nextCI.add(I); extFISs.add(extIS); F.add(extIS);}
End For
If(eqItems <> ∅)
  SoS ← set of all subsets of eqItems, excluding ∅;
  For Each IS ∈ SoS Do F.add(FIS ∪ IS);
  If(Parent_FISs <> ∅)
    Production ← {P| P = P1∪P2, P1 ∈ SoS, P2 ∈ Parent_FISs};
    For Each IS ∈ Production Do F.add(FIS ∪ IS);
    Parent_FISs ← Parent_FISs ∪ Production;
  End If
  Parent_FISs ← Parent_FISs ∪ SoS;
End If
If(Parent_FISs <> ∅)
  Production ← {P| P = P1∪P2, P1 ∈ extFISs, P2 ∈ Parent_FISs};
  F ← F ∪ Production;
End If;
For Each itemset IS ∈ extFISs
  GenerateFrequentItemsets(IS, nextCI, Parent_FISs);
```

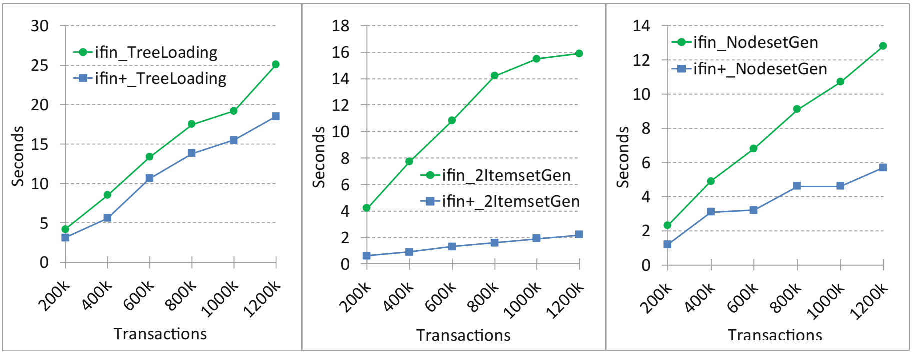
6 Experiments

All experiments were conducted on a 1.86 GHz Intel Core(MT) i3-4030U processor, and 4 GB memory computer with Window 8.1 operating system. To evaluate the performance, we used the Market-Basket Synthetic Data Generator [7], based on the IBM Quest, to prepare a dataset of 1.2 million transactions. The average transaction length and number of distinguishing items are 10 and 1000 respectively.

For emulating incremental scenario, the dataset was divided into six equal parts, 200 thousand transactions for each one. The experiments start mining on the first part and then part by part from the second one is accumulated and mined.

The algorithm IFIN^{+} was compared with its original version IFIN, two state-of-the-art algorithms FIN and PrePost^{+}, and the well-known one FP-Growth. All the five algorithms were implemented in Java. Experimental values of running time and used memory are the average values from three individual ones.

Figure 4 depicts partially the running time for the three processing phases in parallel version IFIN^{+} with two threads and in the serial one IFIN. The processor possesses two physical computational units, and we found that the performance achieved its best with two threads in parallel version. The IFIN^{+}'s execution time in each phase is reduced significantly compared to its original version IFIN. The performance improvement of loading a stored built tree (Fig. 4a) achieves its best with 6 s for datasets of from 200 k to 1200 k transactions. More contrast in Fig. 4c, IFIN^{+}'s execution is speeded up by two times over the original; and especially in Fig. 4b, an approximate 8× speed-up is achieved in the phase of frequent 2-itemsets generation. The reason for such high speed-up is that the efficient parallelization is synergized with additional improvements in data representation.



(a) (b) (c)

Fig. 4. Comparisons on the running time of partial processing phases between IFIN^{+} and IFIN: (a) Loading the stored built tree, (b) Generate frequent 2-itemset, (c) Built Nodeset for each frequent 2-itemset

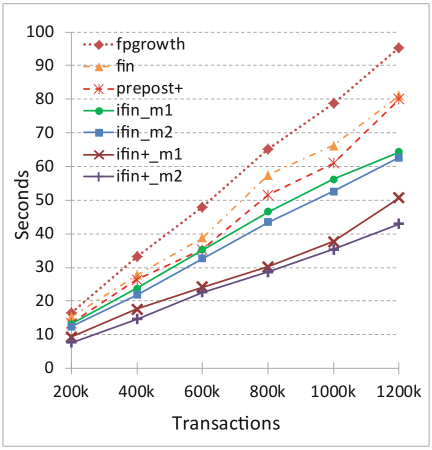


Fig. 5. Running time on incremental datasets

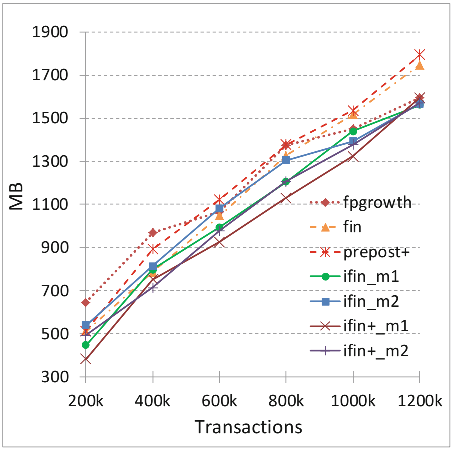


Fig. 6. Peak memory on incremental datasets

Figures 5 and 6 sequentially demonstrate the running time and peak used memory for the five algorithms on incremental datasets at the support threshold $\varepsilon = 0.1\%$. Two running modes are performed by the IFIN and IFIN+ algorithms: **Incremental** (ifin_m1 and ifin+_m1) and **Just-Loading-Tree** (ifin_m2 and ifin+_m2).

For all algorithms, both running time and peak memory increase linearly when the dataset is accumulated. Follow the increasing of the dataset size, while there is not much difference in used memory of the five algorithms; the running time of IFIN and IFIN+ become more discrepant compared with that of the remaining algorithms.

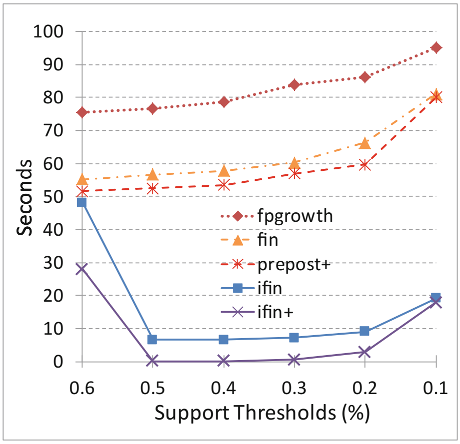


Fig. 7. Running time with different support thresholds

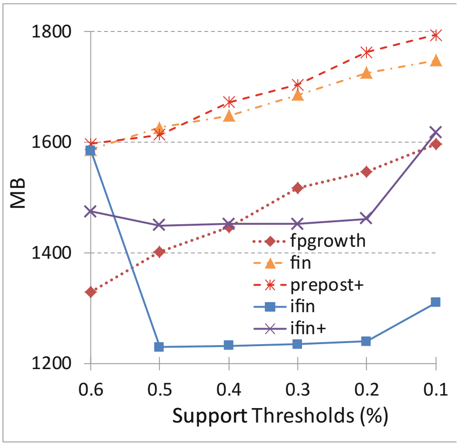


Fig. 8. Peak memory with different support thresholds

Especially the parallelized characteristics of IFIN^+ demonstrate a significant improvement, also compared to the IFIN. One of the reasons is that with the same dataset, loading a stored built IPPC-Tree (IFIN^+ and IFIN) is faster than constructing the corresponding trees in PrePost^+ and FP-Growth. In addition, parallelized characteristics and structural improvement for more efficient and independent execution of IFIN^+ make its effectiveness in running time reduction. The larger the dataset is accumulated, the more the running time difference is. While the reduction in used memory per transaction of IFIN^+ is not considerably, the processing time is reduced remarkably although IFIN^+ must compensate the execution time for generating nodesets for items. The running time of IFIN^+ is slightly less than a half the running time of FP-Growth; and comparing to its original version IFIN, the time ratio is two third for IFIN^+.

In the Figs. 7 and 8, the running time and peak used memory are visualized for the five algorithms mining on the dataset of 1.2 million transactions with different ε values. At $\varepsilon = 0.6\%$, IFIN and IFIN^+ perform two tasks: building an IPPC-Tree and mining; but for other ε values, the two algorithms only run their mining tasks since the built tree is completely reused. Besides, according to the algorithms IFIN and IFIN^+, only a portion of its mining is performed. Consequently, with $\varepsilon \neq 0.6\%$, the running time of IFIN^+ and IFIN take an overwhelming advantage against that of the three remaining algorithms. Algorithm IFIN^+ has an improvement in execution time compared to IFIN, whereas its peak used memory is worse than IFIN's. The explanation for this result is that IFIN^+ allocates memory for nodesets of items and retains these nodesets for following mining cycles at other support thresholds. The mining tasks of IFIN and IFIN^+ at all ε values are in the same running session. Consequently, the peak used memory in the case $\varepsilon = 0.6\%$ and in the cases $\varepsilon \neq 0.6\%$ and are fairly the same. Hence, the "worse" usage in memory of IFIN^+ against IFIN is not really important.

The algorithm FP-Growth uses memory more efficient than the two algorithms FIN and PrePost^+. However, its running time is considerably longer than that of FIN and PrePost^+. Algorithm PrePost^+ is more efficient than FIN in both running time and used memory, but this dominance of PrePost^+ is not significant.

7 Conclusions

In this paper, we proposed a solution, IFIN^+, for parallelizing the frequent itemsets mining algorithm IFIN. Some portions in the serial version were changed in ways that increase efficiency and computational independence for convenience in designing parallel computation with the load balance model, Work-Pool. Hence, computational throughput and efficiency are increased significantly in single-machine environment. Besides, the IFIN^+ algorithm also possesses incremental property as its original version which allows no wasting time to rebuild the IPPC-Tree when new data is added and to re-mine when support threshold is changed.

The aim of shared-memory based parallelized algorithm IFIN$^+$ is to increase the throughput for its serial version IFIN by utilizing as much as possible the computational power of commodity multi-cores processors. In fact, it is just a minor solution to deal with the running time problem in Big Data. For a major and much preferred one, a parallel solution for IFIN$^+$ on distributed environment will be proposed to better confront with the running time and memory scalability problems of Big Data. Besides, conducting experiments on other real datasets will be also performed as our next steps.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceedings of 20th International Conference on VLDB, pp. 487–499 (1994)
2. Han, J., Pei, J., Yin, Y.: Mining frequent itemsets without candidate generation. ACM Sigmod Rec. **29**(2), 1–12 (2000)
3. Cheung, W., Zaïane O.R.: Incremental mining of frequent patterns without candidate generation or support constraint. In: Proceedings of the 7th International Database Engineering and Applications Symposium, pp. 111–116. IEEE (2003)
4. Deng, Z.-H., Lv, S.-L.: Fast mining frequent itemsets using nodesets. Expert Syst. Appl. **41** (10), 4505–4512 (2014)
5. Deng, Z.-H., Lv, S.-L.: PrePost$^+$: an efficient N-lists-based algorithm for mining frequent itemsets via children-parent equivalence pruning. Expert Syst. Appl. **42**(13), 5424–5432 (2015)
6. Rymon, R.: Search through systematic set enumeration. In: Proceedings of the 1st International Conference Principles of Knowledge Representation and Reasoning, pp. 539–550 (1992)
7. Market-Basket Synthetic Data Generator. https://synthdatagen.codeplex.com/
8. Savasere, A., Omiecinski, E., Navathe, S.: An efficient algorithm for mining association rules in large databases. In: VLDB, pp. 432–443 (1995)
9. Perego, R., Orlando, S., Palmerini, P.: Enhancing the *Apriori* algorithm for frequent set counting. In: Kambayashi, Y., Winiwarter, W., Arikawa, M. (eds.) DaWaK 2001. LNCS, vol. 2114, pp. 71–82. Springer, Heidelberg (2001). doi:10.1007/3-540-44801-2_8
10. Park, J.S., Chen, M.S., Yu, P.S.: Using a hash-based method with transaction trimming and database scan reduction for mining association rules. IEEE Trans. Knowl. Data Eng. **9**(5), 813–825 (1997)
11. Zaki, M.J.: Scalable algorithms for association mining. IEEE Trans. Knowl. Data Eng. **12**(3), 372–390 (2000)
12. Grahne, G., Zhu, J.: Fast algorithms for frequent itemset mining using FP-trees. Trans. Knowl. Data Eng. **17**(10), 1347–1362 (2005)
13. Liu, G., Lu, H., Lou, W., Xu, Y., Yu, J.X.: Efficient mining of frequent itemsets using ascending frequency ordered prefix-tree. DMKD J. **9**(3), 249–274 (2004)
14. Shenoy, P., Haritsa, J.R., Sudarshan, S.: Turbo-charging vertical mining of large databases. In: SIGMOD 2000, pp. 22–33 (2000)
15. Zaki, M.J., Gouda, K.: Fast vertical mining using diffsets. In: 9th SIGKDD, pp. 326–335 (2003)

16. Liu, J., Wu, Y., Zhou, Q., Fung, B.C.M., Chen, F., Yu, B.: Parallel eclat for opportunistic mining of frequent itemsets. In: Chen, Q., Hameurlain, A., Toumani, F., Wagner, R., Decker, H. (eds.) DEXA 2015. LNCS, vol. 9261, pp. 401–415. Springer, Cham (2015). doi:10.1007/978-3-319-22849-5_27
17. Yun, U., Lee, G.: Incremental mining of weighted maximal frequent itemsets from dynamic databases. Expert Syst. Appl. **54**, 304–327 (2016)
18. Huynh, V.Q.P., Küng, J., Dang, T.K.: Incremental frequent itemsets mining with IPPC tree. In: Benslimane, D., Damiani, E., Grosky, W.I., Hameurlain, A., Sheth, A., Wagner, R.R. (eds.) DEXA 2017. LNCS, vol. 10438, pp. 463–477. Springer, Cham (2017). doi:10.1007/978-3-319-64468-4_35