Coverage Pattern Mining Based on MapReduce

Akhil Ralla
ralla.akhil@research.iiit.ac.in
International Institute of Information Technology
Hyderabad, Telangana, India

Shadaab Siddiqie
mashadaab.siddiqie@research.iiit.ac.in
International Institute of Information Technology
Hyderabad, Telangana, India

P. Krishna Reddy
pkreddy@iiit.ac.in
International Institute of Information Technology
Hyderabad, Telangana, India

Anirban Mondal
anirban.mondal@ashoka.edu.in
Ashoka University
Sonipat, Haryana, India

ABSTRACT

Pattern mining is an important task of data mining and involves the extraction of interesting associations from large databases. However, developing fast and efficient parallel algorithms for handling large volumes of data is a challenging task. The MapReduce framework enables the distributed processing of huge amounts of data in large scale distributed environment with robust fault-tolerance. Incidentally, research efforts are being made to develop parallel algorithms for mining frequent patterns, sequential patterns, utility patterns and so on under MapReduce framework. However, none of the existing works addresses parallel mining in the context of coverage patterns, which has important applications in several important and diverse areas such as banner advertising, search engine advertising, visibility mining and graph mining. In this regard, the main contributions of this work are three-fold. First, we introduce the problem of parallel mining in the context of coverage patterns under MapReduce framework. Second, we propose a parallel algorithm for extracting coverage patterns. Third, the results of our performance evaluation with real-world and synthetic datasets demonstrate that it is indeed feasible to extract coverage patterns effectively using our proposed CMineMR algorithm under the MapReduce framework.

KEYWORDS

Data Mining, Knowledge Discovery, Coverage Patterns, MapReduce

1 INTRODUCTION

Pattern mining [1, 16] is an important task of data mining and involves the extraction of interesting associations from large databases. It has significant applications in market basket analysis, recommendation systems, and internet advertising.

In pattern mining based applications, databases are typically huge; this necessitates fast and scalable pattern mining algorithms. This problem can be addressed by the development of parallel algorithms in large-scale distributed environments. However, parallel mining entails issues such as data partitioning, load-balancing, minimization of communication and handling of errors in failure nodes [8]. To overcome such problems, the MapReduce framework [7] has been introduced for enabling the distributed processing of huge amounts of data on a large number of machines in geographically distributed environments with robust fault-tolerance.

Another useful type of pattern is the coverage pattern [16], which has several important and diverse applications in areas such as banner advertising, search engine advertising, visibility mining and graph mining. Given a transactional database and a set of data items, coverage pattern (*CP*) is a set of items covering a certain percentage of transactions by minimizing overlap among the transactions covered by each item of the pattern. It has been demonstrated that *CPs* extracted from transactional log data can be employed to improve the performance of display advertising [18], search engine advertising [4, 5] and *CPs* extracted from spatial databases can be employed to improve the visibility of a given target object [9]. In the literature, a model to extract the knowledge of *CPs* from transactional databases has been proposed in [16]. In this regard, a level-wise *CP* mining algorithm, designated as CMine [16], and a pattern growth approach called CPPG [17] were proposed to extract *CPs* from transactional databases.

Incidentally, MapReduce-based pattern mining approaches have been proposed for extracting frequent patterns [12, 19, 20], periodic frequent patterns [3], utility patterns [15, 24] and sequential patterns [6, 22]. However, the problem of parallel mining for extracting coverage patterns has not been investigated so far. In this paper, we propose a new algorithm, designated as CMineMR, for the parallelization of the CMine coverage pattern mining algorithm under the MapReduce framework. *To our knowledge, this is the first work to study parallel mining for coverage patterns.*

Coverage patterns are characterized by coverage support (*CS*) and overlap ratio (*OR*) measures. Given a transactional database *DB* over a set of items, a set X of items is a coverage pattern, if X satisfies the *maximum OR* and *minimum CS* constraints. In this paper, we distribute *DB* across N machines and propose an efficient approach for extracting the knowledge of *CPs* under the MapReduce framework. The proposed approach extracts *CPs* by checking the values of *CS* and *OR* by accessing the partitions of *DB*. The key issue is to compute the *OR* value of a candidate pattern in an efficient manner under MapReduce. As the *OR* value is a fraction, it cannot be computed in a straightforward manner by simply adding the *OR* values from the partitions of *DB*. To enable the efficient extraction of coverage patterns under the MapReduce framework, we exploit the observation that it is possible to compute the *OR* of a given pattern X by computing the numerator and the denominator of *OR independently* in a distributed manner. In effect, the proposed approach generates candidate patterns iteratively and extracts coverage patterns by computing both *CS* and *OR* values of the patterns in an efficient manner by exploiting the MapReduce framework.

The main contributions of this work are three-fold:

(1) We introduce the problem of parallel mining in the context of coverage patterns under the MapReduce framework.
(2) We propose a parallel algorithm for extracting coverage patterns.
(3) The results of our performance evaluation with real-world and synthetic datasets demonstrate that it is indeed feasible to extract coverage patterns effectively using our proposed CMineMR algorithm under the MapReduce framework.

The remainder of this paper is organized as follows. In Section 2, we discuss related works. In Section 3, we discuss background information concerning coverage patterns and the MapReduce framework. In Section 4, we present the proposed approach. In Section 5, we report the performance evaluation. Finally, we conclude in Section 6 with directions for future work.

2 RELATED WORK

In the literature, MapReduce-based pattern mining was first studied in the context of frequent patterns by means of an iteration-based apriori MapReduce algorithm [1, 21]. The work in [12] proposed three MapReduce-based algorithms by investigating implementations of the apriori algorithm (changing number of databases scans). These algorithms were designated as Single Pass Counting (SPC), Fixed Passes Combined-counting (FPC), and Dynamic Passes Combined-counting (DPC). Moreover, in [20], the MapReduce-based FiDoop algorithm incorporated a new tree structure called ultrametric trees as opposed to FP-trees. The work in [19] proposed the two-phase MapReduce-based MISFP algorithm for extracting frequent patterns with multiple values of minimum support. In [3], a MapReduce-based framework for extracting periodic frequent patterns was proposed along with the notion of partition summary, which reduces the amount of data shuffled for improving the performance.

MapReduce-based pattern mining has also been studied in the context of utility patterns and sequential patterns. The PHUI-Growth algorithm [13] extracts utility patterns using the MapReduce framework. Further improvements have been done by proposing BigHUSP algorithm [24], which uses various pruning strategies for minimizing the search space in large-scale distributed environments. Moreover, the MapReduce-based P-FHM+ algorithm [15] extracts utility patterns of a pre-specified fixed size in a distributed manner.

The work in [11] proposed a cloud-based sequential pattern algorithm, designated as DPSP, using the MapReduce framework. Moreover, in [6], an iterative MapReduce framework has been proposed for pruning candidate patterns when constructing sequence trees. Furthermore, two parallel sequential pattern mining algorithms, namely GSP-S and PrefixSpan-S, have been proposed in [22]. Both of the algorithms use multiple MapReduce operations and the choice of the algorithm depends on the user-defined threshold value of support.

Notably, parallel MapReduce-based mining has been studied in the context of frequent patterns [12], periodic frequent patterns [3], utility patterns [13] and sequential patterns [11]. *However, none of the existing works has investigated parallel mining in the context of coverage patterns under the MapReduce framework.*

3 BACKGROUND INFORMATION

This section discusses background information concerning coverage patterns and the MapReduce framework.

3.1 Model of Coverage Patterns

Let $I = \{i_1, i_2 \ldots.., i_n\}$ be the set of items. Let DB be the transactional database. Each transaction T in DB comprises a set of items i.e., $T \subseteq I$. $|DB|$ represents the total number of transactions in database DB. T^{i_p} represents the set of transactions, which contain the item i_p. $|T^{i_p}|$ represents the number of transactions containing i_p.

The concept of coverage patterns incorporates the following notions: relative frequency, coverage set, coverage support and overlap ratio. We shall now discuss each of these notions.

Definition 1. Relative frequency of item i_p. *The fraction of transactions containing a item i_p is called the Relative Frequency (RF) of i_p and is computed as RF(i_p) = $\frac{|T^{i_p}|}{|DB|}$.*

An item is considered to be frequent if its $RF \geq minRF$, where *minRF* is a user-specified threshold.

Definition 2. Coverage set CSet(X) of a pattern X. *Given a pattern $X = \{i_p, ..., i_q, i_r\}$, $(1 \leq p, q, r \leq n)$, CSet(X) is the set of all transactions containing at least one item of pattern X, i.e., $CSet(X) = T^{i_p} \cup T^{i_q} \cup \ldots T^{i_r}$.*

Definition 3. Coverage support CS(X) of a pattern X. *Given $X = \{i_p, ..., i_q, i_r\}$, $(1 \leq p, q, r \leq n)$, CS(X) is the ratio of the size of $CSet(X)$ to $|DB|$ i.e., $CS(X) = \frac{|CSet(X)|}{|DB|}$*

Definition 4. Overlap ratio OR(X) of a pattern X. *Given $X = \{i_p, ..., i_q, i_r\}$, $(1 \leq p, q, r \leq n)$ and $|T^{i_p}| \geq \ldots \geq |T^{i_q}| \geq |T^{i_r}|$, OR(X) is the ratio of the number of common transactions between $CSet(X - i_r)$ and T^{i_r} to the number of transactions having item i_r, i.e., $OR(X) = \frac{|CSet(X - i_r) \cap T^{i_r}|}{|T^{i_r}|}$*

A pattern is interesting if it has high CS since it covers more number of transactions. Suppose we want to increase the coverage by adding a new item i_k to the pattern X. The addition of item i_k will be more interesting if it adds more number of transactions for the coverage set $CSet(X)$ of pattern X . In essence, adding a new item i_k to pattern X could be interesting if there is a minimal overlap. Thus, a pattern having less OR could be more interesting.

Definition 5. Coverage pattern (CP). *A pattern $X = \{i_p, ..., i_q, i_r\}$, $(1 \leq p, q, r \leq n)$ and $|T^{i_p}| \geq \ldots \geq |T^{i_q}| \geq |T^{i_r}|$ is called a coverage pattern if $OR(X) \leq maxOR$, $CS(X) \geq minCS$ and $RF(i_k) \geq minRF$ $\forall i_k \in X$, where maxOR and minCS are user-specified threshold values maximum overlap ratio and minimum coverage support respectively.*

Given a set I of items, transactional database DB, *minRF*, *minCS* and *maxOR*, the problem of mining *CPs* is to discover the complete set of *CPs*.

About sorted closure property: The overlap ratio satisfies downward closure property if the items are ordered in descending order of their frequencies respective. Such a property is called the sorted closure property [14].

Sorted closure property. Let $X = \{i_p, ..., i_q, i_r\}$, $(1 \leq p, q, r \leq n)$ be a pattern such that $|T^{i_p}| \geq ... \geq |T^{i_q}| \geq |T^{i_r}|$. If OR(X) $\leq$ *maxOR*, all of the non-empty subsets of X containing i_r and having *size* ≥ 2 will also have overlap ratio less than or equal to *maxOR*.

An item a is said to be a non-overlap item w.r.t. a pattern X if $OR(X, a) \leq maxOR$ and $RF(i_k) \geq minRF$ $\forall i_k \in \{X, a\}$. The notion of non-overlap pattern (NOP) is defined as follows.

Definition 6. Non-overlap pattern (NOP)*: A pattern $X = \{i_p, ..., i_q, i_r\}$, $(1 \leq p, q, r \leq n)$ and $|T^{i_p}| \geq ... \geq |T^{i_q}| \geq |T^{i_r}|$ is a called a non-overlap pattern if $OR(X) \leq maxOR$ and $RF(i_k) \geq minRF$ $\forall i_k \in X$.*

3.2 CMine Algorithm

Similar to the apriori algorithm [1], CMine [16] is an iterative multi-pass algorithm for extracting *CPs* from a given transactional database. Apriori algorithm uses frequent patterns of size k to explore size patterns of size k+1 because frequent patterns satisfy the downward closure property. In case of CMine also, *NOPs* of size k are used to explore size k+1. As *NOPs* satisfy sorted closure property, we extract *NOPs*, which satisfy the *maxOR* constraint. Next, *CPs* are extracted by applying the *minCS* constraint.

Let C_k, NOP_k and CP_k denote the candidate, non-overlap and coverage patterns of size k respectively. At the k^{th} iteration, *NOPs* and *CPs* of size k are computed. Given *minCS*, *maxOR*, and *minRF* values, the steps of CMine algorithm for extracting *CPs* from the transactional database *DB* can be summarized as follows:

(1) First iteration: The frequency of each item is computed by scanning *DB*. After scanning, CP_1 and NOP_1 are computed by checking relative frequency. Item is added to NOP_1 if $RF \geq minRF$, and added to CP_1 if $RF \geq minCS$. The items in NOP_1 are sorted in descending order of their frequencies.
(2) Second iteration and beyond: Starting from k=2, the following step is repeated until $C_k = \phi$. C_k is generated by computing $NOP_{k-1} \bowtie NOP_{k-1}$ (self-join). After scanning *DB*, NOP_k and CP_k are computed by checking *OR* and *CS* of candidate patterns in C_k accordingly.

3.3 MapReduce Framework

MapReduce framework [7] has been proposed to enable the processing of large datasets in large-scale distributed environment. In the MapReduce framework, one machine is designated as the master, who schedules tasks for execution among machines, and other machines are the workers. Computations in the MapReduce framework are distributed among worker machines and are described by the *map* and *reduce* functions. The *map* function processes key-value pairs and the *reduce* function merges all the values associated with the same key. Both of the *map* and *reduce* functions in each worker are executed in parallel thereby enabling improvement in the performance.

4 PROPOSED APPROACH

This section presents the proposed approach.

4.1 Basic Idea

The existing CMine algorithm extracts coverage patterns by storing the transactional database *DB* in a single machine. Under MapReduce, we distribute the *DB* across N machines and extract the *CPs* in a distributed manner. Let $X = \{i_1, i_2, ..., i_n\}$ be a pattern, N be the number of machines and let DB_i represent the i^{th} partition of *DB*. The basic idea is to extract the *CPs* by checking the values of *CS* and *OR* by accessing the partitions of *DB*.

Recall that in CMine, the candidate patterns are generated in an iterative manner and for each candidate pattern, if it satisfies the *maxOR* constraint, we check whether it satisfies the *minCS* threshold constraint. If a given candidate pattern does not satisfy the *minCS* threshold, the next-level candidate patterns are generated. If a given candidate pattern fails to satisfy the *maxOR* constraint, as per the sorted closure property, all of its supersets are pruned.

The issue is to compute the *OR* value of a candidate pattern in a distributed manner. Notably, as the $OR(X)$ value is a fraction, i.e., $OR(X) = \frac{|CSet(X-i_n) \cap T^{i_n}|}{|T^{i_n}|}$, it cannot be computed by adding the *OR* values from the partitions of *DB*.

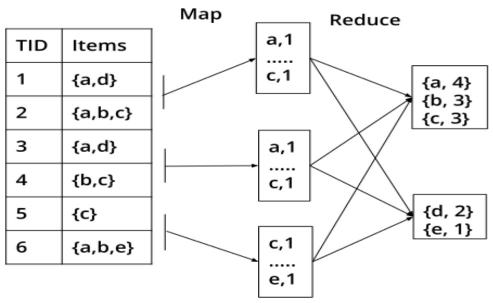
However, $OR(X)$ can be computed efficiently by computing the numerator and the denominator of $OR(X)$ *independently* under the MapReduce framework. It can be observed that the denominator $|T^{i_n}|$ of *OR* is the frequency of a given item. Hence, it is possible to compute the respective frequencies of all of the items in the first phase of MapReduce and store these frequencies in each of the N machines. Moreover, the value $CSet(X)$ of the $CSet$ of pattern X can be computed by aggregating the corresponding coverage sets from the partitions of *DB* stored in each machine in a distributed manner. Once the frequency of the n^{th} item i_n is with every machine, the value of $OR(X) = \frac{|CSet(X-i_n) \cap T^{i_n}|}{|T^{i_n}|}$ can be computed in a distributed manner.

Notably, the value of $CS(X)$ in *DB* can be computed in a distributed environment by adding the coverage support values in each partition of the *DB*, i.e., $CS(X) = \Sigma_{i=1}^{N} CS_i(X) = \Sigma_{i=1}^{N} \frac{|CSet_i(X)|}{|DB|}$, as the numerator $CSet_i(X)$ is disjoint between the partitions of the database and the denominator is a constant i.e., $|DB|$.

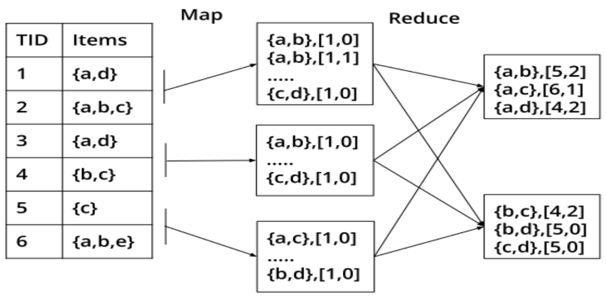
The overview of the proposed approach under MapReduce is as follows. We distribute *DB* into N machines. In the first iteration, we compute relative frequency values of all items using one phase of MapReduce. We broadcast frequencies of all items to all machines. In the second iteration, we compute the *CPs* of size two by using one phase of MapReduce. From the third iteration onwards, we employ two phases of MapReduce; one phase is for generating candidate patterns, while the other phase is for computing *CPs*.

4.2 CMineMR Algorithm

Similar to the CMine algorithm, our proposed MapReduce algorithm is also an iterative algorithm. At the k^{th} iteration of the proposed CMineMR algorithm, we compute *NOPs* and *CPs* of size k . The input to the algorithm consists of a transactional database *DB*, *minRF*, *minCS* and *maxOR*. First, *DB* is segmented into multiple partitions and each partition is loaded onto each machine.



(a) First Iteration-Computing CP_1, NOP_1



(b) Second Iteration-Computing CP_2, NOP_2

Figure 1: First and Second Iterations

(i) **First iteration** Generation of non-overlap and coverage patterns of size one (*MapReduce to compute CP_1 and NOP_1*): Each *mapper* reads each transaction of the data partition and maps each item to 1. *Reducer* groups all the item counts of each item into a list, which we designate as *count-list*. Then item frequencies are computed by adding counts in *count-list* of an each item. Algorithm 1 depicts the procedure to compute frequencies itemsets of size one. The non-overlap and coverage patterns are computed by comparing the frequency with *minRF* and *minCS* respectively. The frequencies of non-overlap itemsets of size one are broadcast among all machines; this is used in the subsequent iterations.

Algorithm 1 First iteration-Computing CP_1, NOP_1 (DB)

```
procedure MAP(key = null,value = $DB_i$)
    for each $t_i$ in $DB_i$ do:
        for each $i_k$ in $t_i$ do:
            output $< i_k, 1 >$
procedure REDUCE(key = $i_k$, value = count-list($i_k$))
    for each count in count-list($i_k$) do:
        $i_k.freq$ += count
```

Example 1. In this example, we explain the first iteration of the proposed CMineMR algorithm by considering the transactional database *DB*, as shown in Table 1. Let the values of *minRF*, *minCS* and *maxOR* used for the extraction of coverage patterns be 0.2, 0.8 and 0.8 respectively. First, the *DB* is distributed across multiple machines. Figure 1(a) shows MapReduce operation to extract frequent itemsets of size one by considering 3 mappers and 2 reducers. Each mapper maps each item in a transaction to 1. Consider item *a* in the first transaction; it is mapped to 1 by the first mapper. Then during reduction, the frequencies of each of the item are computed. After computing the frequencies, non-overlap and coverage patterns are extracted by comparing frequencies with *minRF* and *minCS* respectively. For this example, $NOP_1 = \{a{:}4, b{:}3, c{:}3, d{:}2\}$ and $CP_1 = \phi$. The frequencies of the non-overlap itemsets of size one are broadcast among all machines.

(ii) **Second iteration** In this iteration, we discuss the generation of size two non-overlap and coverage patterns(*MapReduce to compute CP_2 and NOP_2*):

Table 1: Transactional Database

TID	1	2	3	4	5	6
Items	a,d	a,b,c	a,d	b,c	c	a,b,e

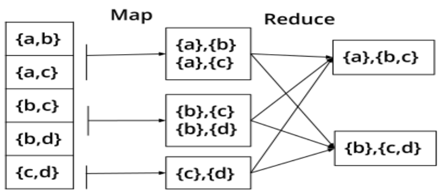
In the second iteration, the candidate patterns are computed by joining the non-overlap patterns of the first iteration. After computing C_2, the overlap ratio and coverage support are computed using one MapReduce phase.

For each transaction and candidate pattern, *mapper* maps the candidate pattern to two integers of the form: <candidate pattern,[x,y]>. The first integer x is equal to 1 if the transaction contains at least one item of the pattern. The second integer y is 1 if transaction has the least frequent item of candidate pattern and at least one item among the remaining items. *Reducer* groups all the item counts of each item into a list, which we designate as *counts-list*. Then the corresponding integers of each candidate pattern are added. Algorithm 2 depicts the procedure to compute the size of coverage set and the numerator of overlap ratio values candidate patterns($k > 1$) of size k. After reduction, the *CS* of a pattern is computed by dividing the first integer with the total number of transactions. The *OR* is computed by dividing the second integer with the frequency of the least frequent item (broadcast in the first iteration).

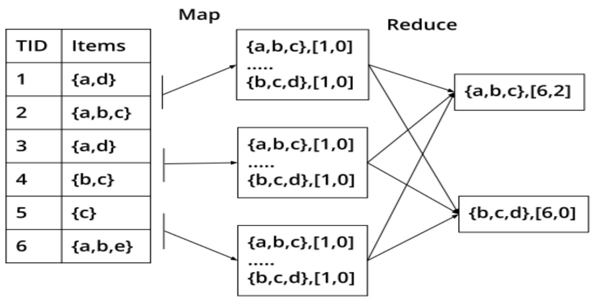
Algorithm 2 k^{th} iteration-Computing CP_k, NOP_k (DB, C_k)

```
procedure MAP(key = null,value = $DB_i$)
    for each $t_i$ in $DB_i$ do:
        for each $X = \{i_1, i_2, ..., i_k\}$ in $C_k$ do:
            if $\exists i_m, m \in [1, k-1] : i_m \in t_i$ and $i_k \in t_i$ then
                output $< X, [1, 1] >$
            else if $\exists i_m, m \in [1, k] : i_m \in t_i$ then
                output $< X, [1, 0] >$
procedure REDUCE(key = $X$, value = counts-list($X$))
    for each count in count-list($X$) do:
        $X.count[0]$ += $count[0]$
        $X.count[1]$ += $count[1]$
```

Example 2. This example is in continuation to Example 1. In this example, we explain the second iteration of the proposed



(a) Computing C_3



(b) Computing CP_3 and NOP_3

Figure 2: Third iteration

CMineMR algorithm. Notably, in the second iteration, the candidate patterns are computed by joining items in NOP_1 i.e., $C_2 = \{\{a,b\},\{a,c\},\{a,d\},\{b,c\},\{b,d\},\{c,d\}\}$. Figure 1(b) shows how MapReduce is used to extract coverage and overlap values of patterns in C_2 by considering 3 mappers and 2 reducers. Consider pattern $\{a,b\}$, which is mapped to [1,0] for the first transaction as the first transaction has item a and not b. Then after reduction, CS and OR are computed. Consider the $\{a,b\}$ with the coverage support 0.83 (5/6) and the overlap ratio 0.67 (2/3). The patter $\{a,b\}$ is both the non-overlap as well as the coverage pattern because CS is greater than $minCS$ (0.83 > 0.8) and OR is less than $maxOR$ (0.67 < 0.8). Similarly, by computing CS and OR of patterns in C_2, $NOP_2 = \{\{a,b\},\{a,c\},\{b,c\},\{b,d\},\{c,d\}\}$ and $CP_2 = \{\{a,b\},\{a,c\},\{b,d\},\{c,d\}\}$ are extracted.

(iii) **Third iteration and beyond:** Now we discuss the generation of non-overlap and coverage patterns of size k, $k > 2$.

From the third iteration, candidate patterns are generated using one MapReduce phase. The overlap ratio and coverage support of these candidate patterns are computed using another MapReduce phase. This procedure of two MapReduce phases for each iteration is repeated until no new candidate patterns can be generated.

(a) MapReduce approach to compute C_k: For each non-overlap pattern of size k-1, *mapper* map the first k-2 items to the least frequent item in the pattern. *Reducer* groups all the least frequent items based on the key into a list, which we designate as *item-list*. For each k-2 pattern (key), candidate patterns of size k are generated by iterating over the *item-list* as shown in Algorithm 3. The candidate patterns are broadcast across all machines. The CS and OR of these candidate patterns are computed by one MapReduce operation. Algorithm 3 depicts the procedure to compute candidate patterns of size k.

(b) MapReduce approach to compute CP_k and NOP_k: After computing C_k, the MapReduce to compute CS and OR is similar to that of the computation of CS and OR of size two candidate patterns as explained for the Second iteration.

Example 3. This example is in continuation to Example 2. After computing NOP_2 and CP_2, C_3 is computed using MapReduce shown in Figure 2(a) and CP_3, NOP_3 are computed using another MapReduce shown in Figure 2(b). Now, in computation of C_3, consider

Algorithm 3 $k^{th}(k > 2)$ iteration-Computing C_k (NOP_{k-1})

```
procedure MAP(key = null,value = $NOP_{k-1}$)
    for each $X = \{i_1, i_2, ..i_{k-1}\}$ in $NOP_{k-1}$ do:
        output $< \{i_1, i_2, .., i_{k-2}\}, i_{k-1} >$
procedure REDUCE(key = $X$, value = $item\text{-}list(X)$)
    for each $i_m$ in $item\text{-}list(X)$ do:
        for each $i_n$ in $item\text{-}list(X)$ do:
            if $Freq(i_m) < Freq(i_n)$ then
                $\{i_1, i_2, .., i_{k-2}, i_m, i_n\}$
```

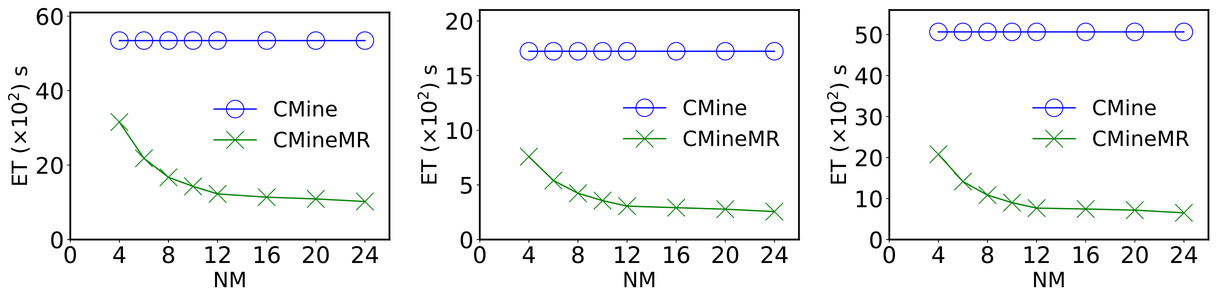
pattern $\{a,b\}$ where $\{a\}$ is mapped to $\{b\}$ by the first mapper. The reduction is done by grouping least frequent items for each key. Then C_3 is computed by iterating over least frequent items for each key (Algorithm 3). After computation of $C_3 = \{\{a,b,c\},\{b,c,d\}\}$, CS and OR are computed using another MapReduce shown in Figure 2(b). Then NOP_3 and CP_3 are computed by comparing CS and OR with $minCS$ and $maxOR$ accordingly, $NOP_3 = \{\{a,b,c\},\{b,c,d\}\}$ and $CP_3 = \{\{a,b,c\},\{b,c,d\}\}$. In this example no new size 4 candidate patterns are generated, hence we terminate the algorithm.

5 PERFORMANCE EVALUATION

We have conducted experiments by implementing our proposed CMineMR algorithm as well as the reference CMine algorithm in Python 2.7. The CMineMR algorithm is written using Apache Spark architecture [23] and it is performed in a cluster of 24 machines, with 2 GB memory each. The experiments on the reference CMine algorithm are performed in one machine of the cluster.

The experiments were conducted on three datasets. The first dataset is BMS-POS [10] dataset, which is a click-stream dataset of an e-commerce company; the dataset has 515,596 transactions and 1656 distinct items. The second dataset is Mushroom dataset [10], a dense dataset having 8,124 transactions and 119 distinct items. The third dataset is the T10I4D100K, which is a synthetic dataset [2] generated by a dataset generator. This synthetic dataset had 100,000 transactions and 870 distinct items. We shall henceforth refer to this dataset as *Synthetic* dataset.

Our experiments were conducted by varying the number of machines (NM), $maxOR$, $minCS$ and $minRF$. Table 2 summarizes the



(a) BMS-POS (b) Mushroom (c) Synthetic

Figure 3: Effect of variations in NM

parameters used in our experiments. We have selected appropriate values of *NM*, *minRF*, *minCS* and *maxOR* (as shown in Table 2) for each of the datasets under consideration based on the results of our preliminary evaluation.

Table 2: Parameters used in our experiments

Dataset	Parameter	Default value	Variations
BMS-POS	N.of Machines (*NM*)	8	[4,6,8,10, 12,16,20,24]
	|DB|	515,596	
	minRF	0.065	[0.065, 0.095] step-size 0.01
	minCS	0.5	[0.1, 0.9] step-size 0.1
	maxOR	0.6	[0.1, 0.9] step-size 0.1
Mushroom	N.of Machines (*NM*)	8	[4,6,8,10, 12,16,20,24]
	|DB|	8,124	
	minRF	0.05	[0.05, 0.2] step-size 0.05
	minCS	0.3	[0.1, 0.9] step-size 0.1
	maxOR	0.3	[0.05, 0.5] step-size 0.05
Synthetic	N.of Machines (*NM*)	8	[4,6,8,10, 12,16,20,24]
	|DB|	100,000	
	minRF	0.045	[0.045, 0.06] step-size 0.0025
	minCS	0.3	[0.1, 0.9] step-size 0.1
	maxOR	0.3	[0.05, 0.5] step-size 0.05

As reference, we used the CMine algorithm [16] discussed in Section 2. Notably, CMine is the closest to our proposed approach because no other work exists for the parallel extraction of coverage patterns. As the performance metric, we use execution time (*ET*), which is the total processing time (in seconds) for extracting *CPs* during the course of a given experiment.

5.1 Effect of variations in NM

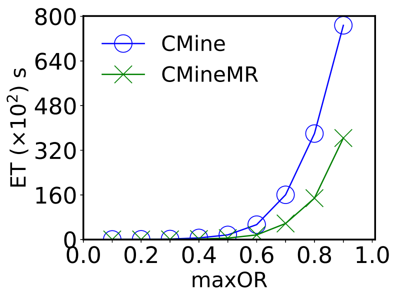
Figure 3 depicts the effect of variations in *NM*. For BMS-POS, the results are shown in Figure 3(a). The *ET* of CMineMR decreased rapidly till *NM*=8 due to a large amount of parallel computation required for extracting coverage patterns. However, the change in *ET* decreases with increase in *NM* and reaches to saturation when 16 machines are used due to increase in the communication cost. The proposed CMineMR algorithm is 3.2 times faster than CMine algorithm when *NM* is 8. Similar trends are observed in Mushroom and Synthetic as shown in Figures 3(b) and 3(c) respectively. When *NM* is 8, the proposed CMineMR algorithm is 4.1 and 4.7 times faster than CMine algorithm for Mushroom and Synthetic datasets respectively.

5.2 Effect of variations in maxOR

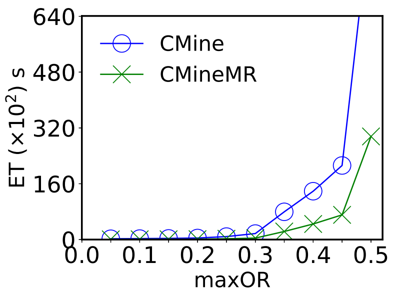
Figure 4 depicts the effect of variations in *maxOR*. For BMS-POS, the results are shown in Figure 4(a). The *ET* of CMine and CMineMR increases with the increase in *maxOR*, as the number of non-overlap patterns generated increases, thereby eventually increasing the runtime of the algorithm. The *ET* of CMineMR is 2.1 times faster than CMine algorithm when *maxOR* is 0.9 due to significant amount of parallel computation required for extracting *CPs*. The results for Mushroom and Synthetic datasets are shown in Figures 4(b) and 4(c) respectively. As Mushroom dataset is dense, the changes in *maxOR* from 0.045 to 0.5 lead to a very high increase in *ET*. The behaviour of *ET* for Mushroom and Synthetic datasets are comparable to that of the BMS-POS dataset. The proposed CMineMR algorithm is 3.2 and 4.1 times faster than CMine algorithm when *maxOR* is 0.5 for Mushroom and Synthetic datasets respectively.

5.3 Effect of variations in minCS

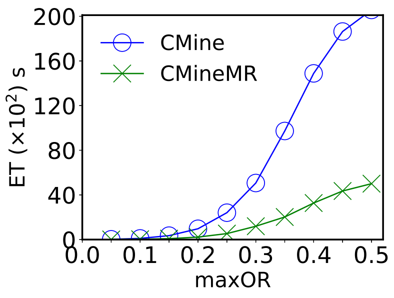
Figure 5 depicts the effect of variations in *minCS*. For BMS-POS, the results are shown in Figure 5(a). Notably, the *CPs* in each iteration are generated by checking *minCS* of *NOPs*, leading to no significant changes in *ET* for CMine and CMineMR due to variations in *minCS*, as shown in Figure 5(a). The results for Mushroom, Synthetic datasets are depicted in Figures 5(b) and 5(c) respectively.



(a) BMS-POS

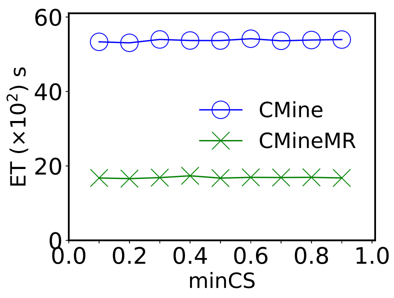


(b) Mushroom

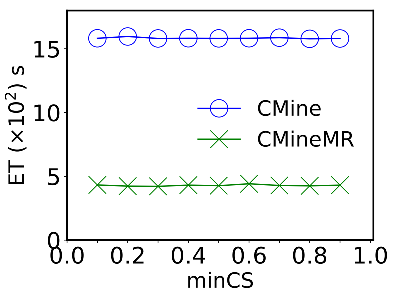


(c) Synthetic

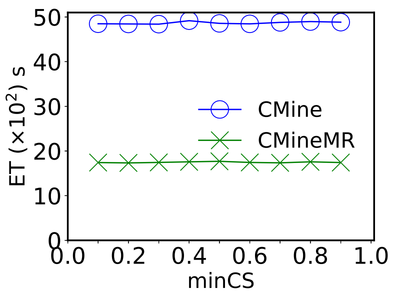
Figure 4: Effect of variations in maxOR



(a) BMS-POS

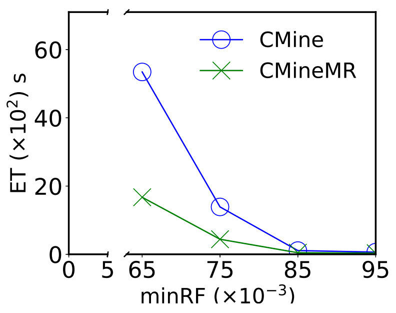


(b) Mushroom

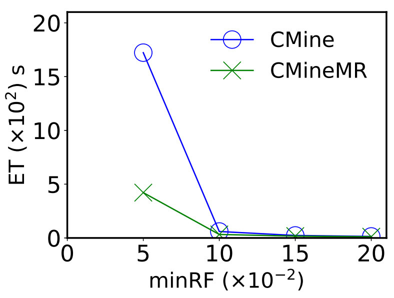


(c) Synthetic

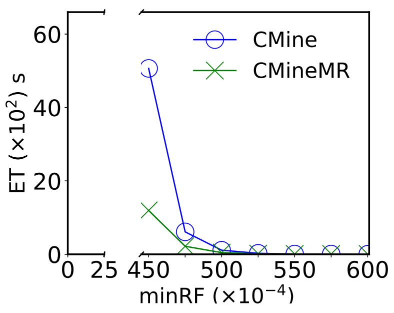
Figure 5: Effect of variations in minCS



(a) BMS-POS



(b) Mushroom



(c) Synthetic

Figure 6: Effect of variations in minRF

The CMineMR algorithm is approximately 3.2, 3.7 and 2.7 faster than the CMine algorithm for BMS-POS, Mushroom and Synthetic datasets respectively.

5.4 Effect of variations in minRF

Figure 6 depicts the effect of variations in *minRF*. For BMS-POS, the results are shown in Figure 6(a). The decrease in *ET* for CMine and CMineMR with increase in *minRF* represents the decrease in the number of size-one frequent itemsets (items satisfying *minRF*). The results for Mushroom, Synthetic datasets are depicted in Figures 6(b) and 6(c) respectively. For BMS-POS, the gradual decrease in *ET* indicates that there are small changes in the number of size-one frequent itemsets with increase in *minRF*. However, for Mushroom and Synthetic datasets, there is a sudden fall in *ET*; this indicates that most of the items are having comparable frequencies. For BMS-POS, CMineMR is 3.2 times faster than CMine when *minRF* is 0.065.

For Mushroom, CMineMR is 4.1 times faster than CMine when *minRF* is 0.05. For Synthetic, CMineMR is 4.2 times faster than CMine when *minRF* is 0.045. The improvement in *ET* is due to significantly parallel computations in the extraction of coverage patterns.

6 CONCLUSION

In pattern mining, developing fast and efficient parallel algorithms handling large volumes of data becomes a challenging task. While parallel mining algorithms have been proposed for extracting frequent patterns, sequential patterns and utility patterns using the MapReduce framework, existing works have so far not investigated parallel mining in the context of coverage patterns. In this paper, we have introduced the problem of parallel mining in the context of coverage patterns and proposed the CMineMR algorithm for efficiently extracting the knowledge of coverage patterns. The results of our performance evaluation with real-world and Synthetic datasets demonstrate that it is indeed feasible to extract coverage patterns effectively using our proposed CMineMR algorithm under the MapReduce framework. As part of future work, we plan to develop parallel algorithms for pattern growth approach towards extracting coverage patterns. Furthermore, we plan to investigate the parallel coverage pattern extraction by considering issues such as skew in transactional databases and load-balancing.

REFERENCES

[1] Rakesh Agarwal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proc. VLDB*. 487–499.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD*, Vol. 22. 207–216.

[3] Alampally Anirudh, R Uday Kiran, P Krishna Reddy, Masashi Toyoda, and Masaru Kitsuregawa. 2017. An efficient map-reduce framework to mine periodic frequent patterns. In *Proc. DaWaK*. 120–129.

[4] Amar Budhiraja, Akhil Ralla, and P Krishna Reddy. 2018. Coverage pattern based framework to improve search engine advertising. *International Journal of Data Science and Analytics* (2018), 1–13.

[5] Amar Budhiraja and P Krishna Reddy. 2017. An improved approach for long tail advertising in sponsored search. In *Proc. DASFAA*. 169–184.

[6] Chun-Chieh Chen, Chi-Yao Tseng, and Ming-Syan Chen. 2013. Highly scalable sequential pattern mining based on MapReduce model on the cloud. In *BigData Congress*. IEEE, 310–317.

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[8] Wensheng Gan, Jerry Chun-Wei Lin, Han-Chieh Chao, and Justin Zhan. 2017. Data mining in distributed environment: a survey. *WIREs Data Mining and Knowledge Discovery* 7, 6 (2017), 1–19.

[9] Lakshmi Gangumalla, P Krishna Reddy, and Anirban Mondal. 2019. Multi-location visibility query processing using portion-based transactional modeling and pattern mining. *Data Min Knowl Disc* 33, 5 (2019), 1393–1416.

[10] Bart Goethals and MJ Zaki. 2012. Frequent itemset mining implementations repository. *URL: http://fimi.cs.helsinki.fi* (2012).

[11] Jen-Wei Huang, Su-Chen Lin, and Ming-Syan Chen. 2010. DPSP: Distributed progressive sequential pattern mining on the cloud. In *Proc. PAKDD*. Springer, 27–34.

[12] Ming-Yen Lin, Pei-Yu Lee, and Sue-Chen Hsueh. 2012. Apriori-based frequent itemset mining algorithms on MapReduce. In *Proc. ICUIMC*. 76.

[13] Ying Chun Lin, Cheng-Wei Wu, and Vincent S Tseng. 2015. Mining high utility itemsets in big data. In *Proc. PAKDD*. 649–661.

[14] Bing Liu, Wynne Hsu, and Yiming Ma. 1999. Mining association rules with multiple minimum supports. In *Proc. ACM SIGKDD*. 337–341.

[15] Krishan Kumar Sethi, Dharavath Ramesh, and Damodar Reddy Edla. 2018. P-FHM+: Parallel high utility itemset mining algorithm for big data processing. *Procedia Comput. Sci* 132 (2018), 918–927.

[16] P Gowtham Srinivas, P Krishna Reddy, S Bhargav, R Uday Kiran, and D Satheesh Kumar. 2012. Discovering coverage patterns for banner advertisement placement. In *Proc. PAKDD*. 133–144.

[17] P Gowtham Srinivas, P Krishna Reddy, and AV Trinath. 2013. CPPG: efficient mining of coverage patterns using projected pattern growth technique. In *Proc. PAKDD*. 319–329.

[18] AV Trinath, P Gowtham Srinivas, and P Krishna Reddy. 2014. Content specific coverage patterns for banner advertisement placement. In *Proc. DSAA*. 263–269.

[19] Chen-Shu Wang and Jui-Yen Chang. 2019. MISFP-Growth: Hadoop-Based Frequent Pattern Mining with Multiple Item Support. *Applied Sciences* 9, 10 (2019).

[20] Yaling Xun, Jifu Zhang, and Xiao Qin. 2015. Fidoop: Parallel mining of frequent itemsets using mapreduce. *IEEE Trans. Syst. Man Cybern. Syst.* 46, 3 (2015), 313–325.

[21] Xin Yue Yang, Zhen Liu, and Yan Fu. 2010. MapReduce as a programming model for association rules algorithm on Hadoop. In *The International Conference on Information Sciences and Interaction Sciences*. 99–102.

[22] Xiao Yu, Qing Li, and Jin Liu. 2019. Scalable and parallel sequential pattern mining using spark. *WWW* 22, 1 (2019), 295–324.

[23] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[24] Morteza Zihayat, Zane Zhenhua Hut, Aijun An, and Yonggang Hut. 2016. Distributed and parallel high utility sequential pattern mining. In *International Conference on Big Data (Big Data)*. 853–862.