Massively Distributed Environments and Closed Itemset Mining: The DCIM Approach

Mehdi Zitouni, Reza Akbarinia, Sadok Ben Yahia, Florent Masseglia

Massively Distributed Environments and Closed Itemset Mining: the DCIM Approach

Mehdi Zitouni[1,2], Reza Akbarinia[1], Sadok Ben Yahia[2], and Florent Masseglia[1]

[1] INRIA & LIRMM Montpellier, FRANCE
[2] Université de Tunis ElManar, Faculté des Sciences de Tunis, LIPAH-LR 11ES14, Tunis, TUNISIA
{Mehdi.Zitouni,Reza.Akbarinia,Florent.Masseglia}@inria.fr
Sadok.Benyahia@fst.rnu.tn

Abstract. Data analytics in general, and data mining primitives in particular, are a major source of bottlenecks in the operation of information systems. This is mainly due to their high complexity and intensive call to IO operations, particularly in massively distributed environments. Moreover, an important application of data analytics is to discover key insights from the running traces of information system in order to improve their engineering. Mining closed frequent itemsets (CFI) is one of these data mining techniques, associated with great challenges. It allows discovering itemsets with better efficiency and result compactness. However, discovering such itemsets in massively distributed data poses a number of issues that are not addressed by traditional methods. One solution for dealing with such characteristics is to take advantage of parallel frameworks like, *e.g.*, MapReduce. We address the problem of distributed CFI mining by introducing a new parallel algorithm, called DCIM, which uses a prime number based approach. A key feature of DCIM is the deep combination of data mining properties with the principles of massive data distribution. We carried out exhaustive experiments over real world datasets to illustrate the efficiency of DCIM for large real world datasets with up to 53 million documents.

Keywords: Distributed Information Systems, Data Analytics, Closed Frequent Itemsets.

1 Introduction

In the past few years, advances in hardware and software technologies have made it possible for the users of information systems to produce large amounts of transactional data. Although data mining has become a fairly well established field now, its applications in massively distributed environment poses a number of thriving challenges which are a well-known source of bottlenecks for the operation of distributed information systems. This is particularly the case of frequent itemset mining (FIM) [1]. FIM allows discovering important correlation for massive sets of data and reveal key insights for numerous applications, ranging from

marketing, to scientific data analytics, and including the optimization of information systems. Actually, discovering the relationship between features in the running traces of a system, for its optimization, is an active research topic [2,3].

Unfortunately, mining only frequent itemsets generates an overwhelming number of itemsets. This makes their interpretation almost impossible and affects the reliability of the expected results.

Several studies were conducted to define and generate condensed representations of frequent itemsets in the past few years. In particular, closed frequent itemsets (CFI in short) [4] have received much attention with very general proposals. Existing algorithms for mining CFI flag out good performance when the input dataset is small or the support threshold is high. However, when the database increases in size or the support threshold turns to be low, both memory usage and communication costs become hard to bear. Some early efforts tried to speed up the mining algorithms by running them in parallel [5], using frameworks such as MapReduce [6] or Spark [7], that allow to make powerful computing and storage units on top of ordinary machines. In [8], Wang et al. propose an approach for mining closed itemsets using MapReduce, but it suffers from the lack of scalability.

In this paper, we propose a new parallel algorithm named *Distributed Closed Itemset Mining* (DCIM) for enumerating CFIs using MapReduce. In DCIM, we develop a new approach based on mathematical techniques. The items from the database are transformed into prime numbers, and CFIs are generated by using only division and multiplication operations. When the scale of datasets gets large, such operations could cause an overwhelming computing and memory utilization. To overcome this issue, we propose insightful optimization techniques that allow extracting CFI from even very large datasets. The main contributions of this paper are as follows :

- We propose a numerical representation of transactional datasets using a new transformation technique. This transformation is embedded in the algorithm for a very low additional cost.
- We design an efficient parallel algorithm for CFI mining by deeply combining MapReduce functionalities with the properties of CFI.
- We exploit the mathematical properties of our numerical representation and provide optimizations both at the architecture level as well as on the computing nodes.
- We carry out exhaustive experiments on real world databases to evaluate the performance of DCIM. The results suggest that our algorithm significantly outperforms the pioneering algorithms in CFI mining over large real world datasets with up to 53 millions articles.

The rest of this paper is organized as follows. In Section 2, we describe some related works. In Section 3, we present some preliminary notions they would be of help for defining the problem. In Section 4, we introduce our DCIM algorithm. The results of our experimental evaluations are reported in Section 5. Finally, in Section 6 we conclude.

2 Related Work

Many research efforts [9, 10] have been introduced to design parallel algorithms capable of working under multiple threads under a shared memory environment. Unfortunately, these approaches do not address the major problem of heavy memory requirement when processing large scale databases. To overcome the latter, MapReduce platform was designed to enable and facilitate the ability to distribute processing of large scale datasets on large computing clusters. In [11], the authors propose a parallel FP-Growth algorithm in MapReduce, which achieves quasi-linear speedups. However, the method presented so far suffers from either excessive amounts of data that need to be transferred and sorted and a high demand for main-memory at cluster nodes.

Moreover, having a large amount of transactional data, finding correlation between them highlights the necessity of discovering a condensed representations of items. Since the introduction of CFI in [4], numerous algorithms for mining it were proposed [12, 13]. In fact, these algorithms tried to reduce the problem of finding frequent itemsets to the problem of mining CFIs by limiting the search space to only CFIs rather than the the whole powerset lattice. Furthermore, they have good performance whenever the size of dataset is small or the support threshold is high. However, as far as the size of the datasets becomes large, both memory use and communication cost are unacceptable. Thus, parallel solutions are of a compelling need. But, research works on parallel mining of CFI are few. In [14] authors proposed a parallel algorithm based on the LCM algorithm [15]. The proposed algorithm uses the concept of Java Tuple Space allowing a dynamic sharing of the work. However, large scale experiments over big datasets are needed to evaluate the scalability of the algorithm. In [8] introduce a new algorithm based on the parallel FP-Growth algorithm PFP [11] that divides an entire mining task into independent parallel subtasks and achieves quasi-linear speedups. The algorithm mines CFI in four MapReduce jobs and introduces a redundancy filtering approach to deal with the problem of generating redundant itemsets. However, we don't find in the literature a work that scales for CFI mining on MapReduce with very large databases, as we tackle in this paper.

3 Preliminaries

Definition 1. *Let $I = \{i_1, ..., i_n\}$ be the set of items. A transaction dataset on I is a set $T = \{t_1, ..., t_m\}$ such that each t_i is included in I. Each t_i is called a transaction. We denote by $||T||$ the sum of sizes of all transactions in T, that is, the size of database T. A set $P \subseteq I$ is called itemset. For an itemset P, a transaction including P is called an occurrence of P, and $T(P)$ is the set of the occurrences of P. $|T(P)|$ is called the frequency of P, and denoted by $frq(P)$. For a given constant θ, called minimum support, the itemset P is frequent if $frq(P) \geq \theta$. For any itemsets P and Q such that $T(P \cup Q) = T(P) \cap T(Q)$, if $P \subseteq Q$ then $T(Q) \subseteq T(P)$. An itemset P is called closed if no other itemset Q satisfies $T(P) = T(Q)$ having $P \subseteq Q$. Given a set $S \subseteq T$ of transactions, let*

$I(S)$ be the set of items common to all transactions in S, i.e., $I(S) = \cap_{(T \in S)} T$. Then, we define $clo(P)$, the closure of itemset P in T, by $I(T(P)) = \cap_{(t \in T(P))} t$.

For every pair of itemsets P and Q, the following properties hold [4]:

1. If $P \subseteq Q$, then $clo(P) \subseteq clo(Q)$.
2. If $T(P) = T(Q)$, then $clo(P) = clo(Q)$.
3. $clo(clo(P)) = clo(P)$.
4. $clo(P)$ is the unique smallest closed itemset including P.
5. An itemset P is a closed itemset if and only if $clo(P) = P$.

MapReduce is one of the most popular solutions for big data processing, in particular owe to its automatic management of parallel execution in clusters of machines. Initially proposed in [6], it has gained increasing popularity, as shown by the tremendous success of Hadoop[3], an open-source implementation.

MapReduce splits the computation in two phases, namely *map* and *reduce*, which in turn are carried out by several tasks that process the data in parallel. The idea behind MapReduce is simple and elegant. Given an input file, and two map and reduce functions, each MapReduce job is executed in two main phases. In the first phase, called *map*, the input data is divided into a set of splits, and each split is processed by a map task in a given worker node. These tasks apply the *map* function on every key-value pair of their splits and generate a set of intermediate pairs. In the second phase, called *reduce*, all the values of each intermediate key are grouped and assigned to a reduce task. Reduce tasks are also assigned to worker machines and apply the *reduce* function on the values of each key to produce the final results.

4 DCIM Algorithm

Manipulating string operations causes multiple problems when handling large scale datasets. In fact, when the support threshold turns to be low, both memory usage and communication costs become unbearable. We overcome this issue by designing a distributed solution to mine CFI using the MapReduce framework. In this section, we propose our algorithm, called DCIM, that distributes the mining process of CFI over a cluster of nodes by using a number of well specified MapReduce jobs adapted to our mining problem.

4.1 Algorithm Overview

The DCIM algorithm uses two MapReduce phases to mine CFIs in three steps which are depicted as follows.

- **Step 1 :** *Splitting* : Splits T into multiple and successive parts and stores the parts on N different computers. Each part is called a *split*.

[3] https://hadoop.apache.org/

- **Step 2 :** *Frequency counting* : Executing a first MapReduce job, this step is dedicated to count the support of each item in T and prune non-frequent ones. The output of this step will be a list of items sorted in descendent order, and each one is linked with a specific prime number.
- **Step 3 :** *CFI Mining* : This is the key step of DCIM that adopts the second MapReduce pass in which Map phase and Reduce phase perform different methods. Here, load balancing is a crucial concern and will call for particular care and a comprehensive approach of distribution principle.

Frequency counting : Using a simple MapReduce count process, in this step, DCIM scans the database and computes the frequency of each item. In fact, the input key-value pair would be like $(key, value = t_i)$, with $t_i \subset T$. For each item, say $i_k \in t_i$, the mapper outputs a key-value pair $(key = i_k, value = 1)$. After all mappers instances are completed, the MapReduce infrastructure feeds the reducers with key-value pairs and the output result is represented as $(key = i_k, value_2 = \Sigma(value))$. Adding the minimum support θ as an input of the job, the set of items is pruned by discarding those who are not frequent and sorted in descending order of their supports in one list, denoted *Frequency-List*. To proceed with DCIM algorithm, each item in *Frequency-List* will receive a specified prime number.

CFI Mining : After generating the *Frequency-List*, sorted in descending order of supports, DCIM starts the second MapReduce job to extract the complete set of CFI. We detail the Map and Reduce phases below. We assume that the mining process of the algorithm is going to be on multiple Sub-Datasets. At this point, we need to deal with our data and well split the dataset, in order to satisfy the correctness and completeness of our results. To do so, a Sub-Dataset definition cited in [12] says :

Definition 2. *For a given dataset T, let i be a frequent item in T. The i-Sub-Dataset is the subset of transactions containing i, while all infrequent items, item i and items following i in the Frequency-List are omitted. And therefore, having j as a frequent item in P-Sub-Dataset, where P is a frequent itemset, the jP-Sub-Dataset is the subset of transactions in the P-Sub-Dataset containing j, while all infrequent items, item j, and items following j in local Frequency-List are omitted.*

Our splitting process is based on item-based partitioning of the dataset. In fact, the idea is based on the creation of one split S_i for every θ-frequent item $i \in$ *Frequency-List*. Thus, we extract, for each item, its appropriate Sub-Dataset. In the Map phase, the algorithm loads the *Frequency-List* of the Dataset. In each split from the inputs, the algorithm treats each transaction t_i from the split S_i. The input pair is like $(key, value = t_i)$. For each t_i, item i is omitted from the transaction t_i and the rest of items are sorted in descending order of supports by checking the *Frequency-List*. Then, DCIM generates a big integer V_{t_i} representing the transaction by multiplying all the primes representing items of

Item	frq(Item)	Prime Nb.
B	4	2
C	4	3
E	4	5
A	3	7
D	1	11

t_i	Original Tr.	Prime Nb.	V_{t_i}
1	A, C, D	7, 3, 11	231
2	B, C, E	2, 3, 5	30
3	A, B, C, E	7, 2, 3, 5	210
4	B, E	2, 5	10
5	A, B, C, E	7, 2, 3, 5	210

Fig. 1: (**Left**) A mapping between items and prime numbers, and (**Right**) a dataset T and its transformation.

the transaction. At the end, the Map phase emits the item i and the appropriates V_{t_i} as follows $(key = i, value = t_i[1], t_i[2], ..., t_i[n])$ where $n \leq ||S_i||$. Figure 1 illustrates the transformation process of our algorithm. Each item is mapped to a prime number (left part of Figure 1), while the dataset (on the right) is transformed by prime number multiplications.

When all mapper instances have completed, reducers read collections corresponding to a group of transactions in form of big integers representing the Sub-Dataset linked to item or itemset in question. Then, the mining process begins literally. Before describing the Reduce phase, some properties and definitions are of use in the remainder. Indeed, for every pair of itemsets P and Q represented respectively as two big integers X and Y, the following properties hold.

1. P is a Closed itemset extracted from a Sub-Dataset. P is discovered by concatenating the items having the same support as P (in the Sub-Dataset)
2. It is not necessary to develop a Sub-Dataset of an itemset Q included in a CFI already discovered P, such that supports of P and Q are equal.
3. $P \subseteq Q$ if the rest of division of Y by X is 0.

In previous works [16], to facilitate the exploration of Sub-Datasets and mine CFI, authors propose a new technique that defines a header table which is associated to each context. This table lists the items contained in the corresponding Sub-Dataset, sorted in descending order of their supports. However, in this current approach, extracting CFI in the reduce phase of DCIM does not need the use of this header table, and thus avoids additional process. To do so, we adopted the notion of greatest common divisor (Gcd). Knowing that the Gcd of two or more integers, when at least one of them is not zero, is the largest positive integer that divides the numbers without a remainder, we deduce our closure operator using the following lemma.

Lemma 1. *: Let P-Sub-Dataset be the subset of transactions containing P. The greatest common divisor in P-Sub-Dataset represents the closure between all transactions.*

Proof. The closure of an itemset P is produced from the intersection between all transactions containing P. Manipulating prime numbers, the Gcd between primes is unique. Thus, having all $V_t i$ from P-Sub-Dataset, extracting the closure from a set of transactions amounts to calculate the Gcd between them. Hence, the Gcd in P-Sub-Dataset is the closure between transactions composing P-Sub-Dataset.

Having the prime number representing the item and its transactions as a set of V_{t_i} as input for reducers, computing the closure from the Sub-Dataset is straightforward by computing the GCD of all transactions of the Sub-Dataset. Doing so, there is no further need to store supports of items contained in the Sub-Dataset. Indeed, if the closure exists, then it will undoubtedly have the same support as that of the item. By concatenating the closure to the candidate item multiplying the prime number of the item and the number representing the closure, the result of our reduce phase will be a CFI that is represented as a number which is added to the set of final results.

Load balancing The principles explained above are a strong basis for high performances when mining CFIs. However, a fully parallel data mining algorithms has to be deeply combined with the intrinsic characteristics of the distributed framework. We know that, in MapReduce, the reducers cannot start applying the reduce function before all mappers finish their work. Thus, when approach-

Algorithm 1 DCIM Algorithm

function MAPPER(i, S_i)
  Load *Frequency-List* ; Load *Primes-List*
  for all $T_i \in S_i$ **do**
    $T_i^{'} \leftarrow ORD(T_i)$ ▷ ORD : sort items from T_i
    $PN(i) = 1$
    if $T_i^{'} \neq \emptyset$ **then**
      for all $j \in T_i^{'}$ **do**
        $PN(i) \leftarrow PN(i) \times Primes\text{-}List(j)$
        ▷ Transforms j and generates V_{t_i}
        Emit($Primes\text{-}List(j+1), PN(i)$)
      end for
    end if
  end for
end function
function COMBINER($i, List\text{-}PN(i)$)
  $List\text{-}Gcd(i) \leftarrow \emptyset$; $k \leftarrow 0$
  for all $PN(i)_k \in List\text{-}PN(i)$, $k < \|List\text{-}PN(i)\|$ **do**
    $Gcd(i) \leftarrow Gcd(PN(i)_k)$ ▷ Computing GCDs
    $k \leftarrow k + 1$
  end for
  $List\text{-}Gcd(i) \leftarrow Gcd(i)$
  Emit($i, List\text{-}Gcd(i)$)
end function
function REDUCER($i, List\text{-}Gcd(i)$)
  $Clos(i) \leftarrow \emptyset$; $CFI \leftarrow \emptyset$
  for all $Gcd(i) \in List\text{-}Gcd(i)$ **do**
    $Clos(i) \leftarrow Gcd(List\text{-}Gcd(i))$ ▷ Results shuffling
  end for
  $CFI \leftarrow i \cup Clos(i)$ ▷ $\cup$: Operation to join items
end function

ing the end of Map phase, there are usually nodes that are idle waiting for the others to finish. It is worth using these nodes for reducing the amount of data that should be transferred from mappers to reducers. The main issue is to find the adequate decomposition of the problem, such that one part of the load may be given to a node that may do some pre-processing and save time to the reducers. This can be done thanks to the nice properties of the GCD, which may be divided into parts of any size. In fact, having a unique GCD for multiple integers, its computation can be done in a successive manner, while maintaining the correctness of the final results. Let us consider that we have n mappers $\{M_1, ..., M_n\}$, and on each mapper i we have $M_{i,k}$ numbers (V_{t_k}) associated to key k. Then, we can compute $\text{GCD}_{i,k}(M_i)$ the local GCD of mapper i for k on the $M_{i,k}$ V_{t_k} it contains. Later, instead of receiving $\sum_{i=1}^{n} M_{i,k}V_{t_k}$ for key k, a reducer will receive a much lower amount of numbers, corresponding to the results of this pre-computing (n, in the ideal case).

Thus, in DCIM, we anticipate the next step of calculating GCDs, avoiding heavy synchronization, and significantly reducing the computing time by performing a reduce-type function, called *combiner*, before starting the reduce phase of the proposed algorithm. Doing so, we limit the volume of data transfer between the map and reduce tasks. This function runs on the output key-value pairs of the map phase which are not immediately written to the output and already available in memory. Instead, they will be collected in lists, one list per each key value. Also, in our new algorithm, we set the combiner class as a shuffling class where all instances of Map's output are handled as a set of transactions, represented as a set of V_{t_i}. In fact, for each map output key, the combiner function is called and tries to compute the global GCD taking V_{t_i}s one by one and applies a series of GCD calculations between them. It is obvious that, besides the technical tricks, passing summarized GCDs to the reduce phase of the algorithm enhances the computation and calculation time.Pseudo-code of Map, Combiner and Reduce phases to enumerate CFIs is sketched in Algorithm 1. An example of DCIM running is presented in Figure 2.

Illustrative example : Figure 1 illustrates how DCIM works on a dataset T. First, having a minimum support $\theta = 2$, the frequency counting pass provides the *Frequency-List* containing items of T with their primes linked sorted in descendent order of frequencies (in case of same frequencies we applied alphabetical order on items). Then, the second MapReduce pass of DCIM is sketched in Figure 2. Starting by the less frequent items from each transaction, DCIM decompose the V_{t_i} to construct Sub-Datasets. In the example, the first mapper took {CA} as a transaction. Having $frq(A) \leq frq(C)$, DCIM starts by dividing V_{t_1} by "7" the prime associated to {A}. The mapper provides {C} as a transaction for A-Sub-Dataset as a first result. Reciprocally, {A} is provided as a transaction for C-Sub-Dataset. The same calculations are applied for the rest of the mappers. Treating {A} as a combine inputs in the second table of the example, A-Sub-Dataset is delivered as a set of V_ts (e.g. {C}=3, {BCE}=30, {BCE}=30). With a GCD= 3 which is the prime associated to {C}, {AC} is a closed frequent itemset. The same calculations are applied to itemset {AB},

Map Inputs (V_{t_i})	**Processing V_{t_i}**	**Map Outputs (Sub-DS)**
{CA} = {21}	$21 = 3 \times 7$	{A} = 7 : {C} = 3
{BCE} = {30}	$30 = 2 \times 3 \times 5$	{E} = 5 : {BC} = 6
	$6 = 2 \times 3$	{C} = 3 : {B} = 2
{BCEA} = {210}	$210 = 2 \times 3 \times 5 \times 7$	{A} = 7 : {BCE} = 30
	$30 = 2 \times 3 \times 5$	{E} = 5 : {BC} = 6
	$6 = 2 \times 3$	{C} = 3 : {B} = 2
{BE} = {10}	$10 = 2 \times 5$	{E} = 5 : {B} = 2
{BCEA} = {210}	$210 = 2 \times 3 \times 5 \times 7$	{A} = 7 : {BCE} = 30
	$30 = 2 \times 3 \times 5$	{E} = 5 : {BC} = 6
	$6 = 2 \times 3$	{C} = 3 : {B} = 2

Combine Inputs (Sub-DS)	**CFI Mining → Reduce Outputs**
{A} = 7 : {3, 30, 30}	GCD(3, 30, 30) = 3 ⇒ 3 × 7 = 21
	21 = {AC} ⇒ {AC} is CFI
{AB} = 14 : {15, 15}	GCD(15, 15) = 15 ⇒ 14 × 15 = 210
	210 = {ABCE} ⇒ {ABCE} is CFI
{AE} ? → {AE} ⊆ {ABCE}	STOP
{E} = 5 : {6, 2, 6, 6}	GCD(6, 2, 6, 6) = 2 ⇒ 2 × 5 = 10
	10 = {BE} ⇒ {BE} is CFI
{EC} = 15 : {2, 2, 2}	GCD(2, 2, 2) = 2 ⇒ 2 × 15 = 30
	30 = {BCE} ⇒ {BCE} is CFI
{C} = 3 : {7, 2, 2, 2}	GCD(7, 2, 2, 2) = 1 ⇒ 1 × 3 = 3
	3 = {C} ⇒ {C} is CFI

Fig. 2: Illustrative example: Map, Combiner and Reduce Phases of DCIM

taking into account its Sub-dataset as inheritance from A-Sub-dataset and so one. The process is stopped in each reducer in two cases. A first case when there is no other item to treat from mappers outputs and a second phase when there is an inclusion relation between closed itemset found and those provided before the latter.

4.2 Optimizing Strategies

The load-balancing technique presented above is a key for high performances. However, massively distributed data mining applied to very large databases calls for thorough optimizations. In this section, we provide insightful optimizing strategies for improving the performance of DCIM in practice.

Document splitting : Collection frequencies of items can be exploited to reduce required work by splitting up every document adopting the item-based partitioning approach. The main idea is to observe the transactional dataset and fit each mapper with a group of dependent transactions. Thus, assuming $i \in$*Frequency-List* a frequent item, we can split the document by searching transactions containing i concatenated to other items having the same supports as i and so on. This allows not only to have fair splits between mappers, but also reduces the time complexity of each mapper by pruning transactions not needed to extract the Sub-Dataset of the item in question.

Multiplying Big Integers : In large datasets, transforming data into numerical forms may generate big integers for which we developed special multiply operator. Before describing this operator, let us recall some definitions about

big integers. A big integer X is handled thanks to its polynomial representation in a given base B as $X = x_0 \times B^0 + x_1 \times B^1 + x_2 \times B^2 + ... + x_n \times B^n$, where B usually depends on the maximal size of the basic data types and the coefficients x_i (also called limbs) are basic number data types (such as long or double in Java) and fulfill $0 < x_i < B$.

Due to the format of our final output, we treat the base B as a power of 10. It significantly reduces the memory usage of the DCIM algorithm. Given two big integers X and Y in their respective canonical forms as follows, $X = \sum_{i=0}^{m}(x_i \times B^i)$ and $Y = \sum_{i=0}^{n}(y_i \times B^i)$, the big integer $Z = X \times Y$ can be obtained thanks to $Z_i = \sum_{k+l=i}(x_k \times y_l)$.

Using these basic definitions, for large integers of size n, all the operations addition, substraction, product and division have a complexity of $O(n)$. This means that the number of basic operations on basic data storage type is proportional to n. Interestingly enough, for the classical product and division operations, the complexity is $O(n^2)$ for multiplying and dividing two integers of size n, when n becomes big, this cost becomes very handicapping. When handling huge integers, it is then of interest to try to obtain a faster algorithm for multiplication and division operations.There are some solutions proposed to overcome the above-mentioned problem, and we tried most of them. One of them is the Karatsuba algorithm [17] proposed for an efficient multiplication of big integers. Karatsuba was the first to observe that multiplication of large integer can be made faster than $O(n^2)$. However, its method is a recursive one. It reduces the number of multiplications from the four products $x_0 \times y_0, x_0 \times y_1, x_1 \times y_0$ and $x_1 \times y_1$ to three by dividing the big integers in two parts. To minimize the complexity caused by Karatsuba, a second algorithm called Toom-Cook algorithm was implemented [18]. In fact, Toom-Cook algorithm takes X and Y as two big integers, and splits them into j lower parts each of length i, and operates on the parts. As j grows, one may mix many of the multiplication sub-processing, thus reducing the overall complexity of the algorithm. The multiplication sub-operations can then be computed recursively using Toom–Cook multiplication again, and so on. Nevertheless, the complexity of Toom-Cook can be further reduced. Indeed, the product of two large integers of size n can be done in $O(n\log(n))$ thanks to Fast Fourier Transform techniques detailed in follow. In fact, two large integers X and Y of size at most $n-1$ can be written in the form of $X = X(B)$ and $Y = Y(B)$, where B is the base (B a power of 10) and X and Y two polynomials as $X(z) = \sum_{i=0}^{n-1}(x_i \times z^i)$ and $Y(z) = \sum_{i=0}^{n-1}(y_i \times z^i)$. Denoting by $R(z)$ the polynomial obtained by the product of $X(z)$ and $Y(z)$, we have $XY = R(B)$ and a final rearrangement on the coefficients of R(z) permits to obtain the product XY. Thus, we are lead to the problem of multiplying two polynomials of degree lower than n. A polynomial of degree lower than n is uniquely defined from its evaluations at n distinct points. Therefore, to obtain the product $R(z) = X(z)Y(z)$, it is sufficient to compute the values $R(w_k)$ at $2 \times n$ distinct points of w_k, that are computing X(w_k) and Y(w_k).

The Fast Fourier Transform idea consists in choosing for w_k the complex roots of unity Ω like $w_k = \exp(\frac{2i\Pi k}{2n}) = \Omega^k$ where $\Omega = \exp(\frac{2i\Pi}{2n})$.

Thus, FFT algorithm proceeds with a transformation technique called the Fourier Transform. For a given sequence $X = (x_0, x_1, ..., x_{2n-1})$ derived from $X(z) = \sum_{i=0}^{n-1}(x_i \times z^i)$, the algorithm computes its Fourier transform F using Ω from below as follows.
$F(X) = (f_0, f_1, ..., f_{2n-1})$; $f_k = \sum_{j=0}^{2n-1}(x_j \Omega^{jk})$
where the conjugate Fourier transform is :
$\overline{F}(X) = (f_0, f_1, ..., f_{2n-1})$ with $f_k = \sum_{j=0}^{2n-1}(x_j \Omega^{-jk})$.
Roughly speaking, to compute the coefficients f_k of $F(X)$, the transformation performs the following steps:

1. Define two sub-sequences of size n:
   X_0=$(x_0, x_2, ..., x_{2n-2})$; X_1=$(x_1, x_3, ..., x_{2n-1})$
2. Compute the Fourrier transform:
   $F(X_0)$=$(a_0, a_1, ..., a_{n-1})$; $F(X_1)$=$(b_0, b_1, ..., b_{n-1})$
3. Deduce the Fourier Transform $F(X)$ with the formulas:
   f_k=$a_k + \Omega^k b_k$; f_{n+k}=$a_k - \Omega^k b_k$; $0 \leq k \leq n$

We now present formally the algorithm to multiply big numbers with FFT algorithm. Let X and Y be two big integers with less than n coefficients. To compute $Z = X \times Y$ in time $O(n \log(n))$, FFT performs the following steps:

1. Compute the Fourier transform X' and Y', of size $2n$ each, of the sequences x_j and y_j : $X' = (x'_0, x'_1, ..., x'_{2n-1})$; $Y' = (y'_0, y'_1, ..., y'_{2n-1})$
2. Compute the product term by term in Z': $Z' = (z'_0, z'_1, ..., z'_{2n-1})$; $z'_i = x'_i \times y'_i$
3. Compute the inverse Fourier transform Z of Z' with the conjugate FFT process: $Z = (z_0, z_1, ..., z_{2n-1}) \equiv \frac{1}{2n}\overline{F}(Z)$

And finally, after rearrangement of the coefficients z_i, the number $Z_i = \sum_0^{2n-1}(z_i B^i)$ is equal to the product of X by Y. The algorithm consists in computing two FFTs of size $2n$ and one reverse FFT of size $2n$. Thus the product of two large integers with n digits has a complexity asymptotically equal to 3 FFTs, let's say $O(n \log(n)^3)$.

Reducing the Size of Prime Numbers : Dealing with large datasets leads us to efficiently manipulate large numbers. Thus, in addition to the efficient multiplication operator, we also tried to reduce the size of generated numbers as much as possible. In fact, when analyzing our execution logs, we observed that items with low-frequency are much more numerous than those having high support values. Thus, for performance enhancements, we tried to attribute the lower primes to items that have higher frequencies. This idea remarkably reduced the running time of our algorithm.

5 Experimental Evaluation

Setup and Implementation : To perform our experiments, we used one of the clusters of Grid5000[4] which is a large-scale and versatile test-bed for experiment-driven research on parallel and distributed computing. Our experiments were

[4] `https://wiki.inria.fr/ClustersSophia/Clusters_Home`

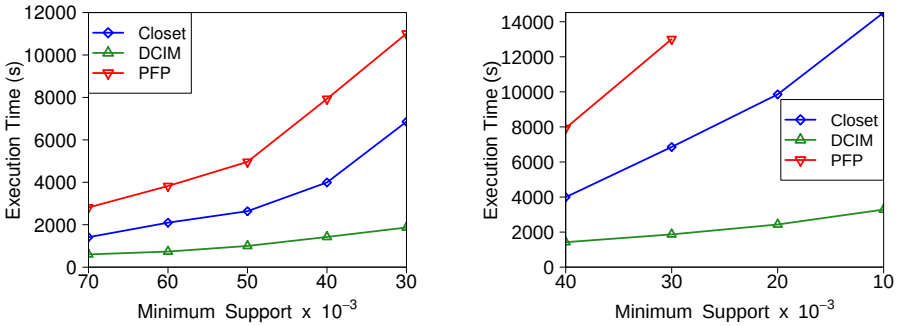
performed on a cluster with 32 nodes (384 cores in total), equipped with Hadoop 2.6.0 version. Each machine is equipped with linux operating system, 96 Gigabytes of main memory, dual-Xeon X5670 with 2.93GHz 12 core CPUs and 320 Gigabytes SATA hard disk.

Due to lack of parallel CFI mining approaches in the literature, we compared our algorithm to our own parallel implementation of CLOSET in MapReduce. We used three Map Reduce jobs. The first job is dedicated to generate the frequency list containing all items in the dataset and for each one we associated its number of occurrences (support) and the final list was sorted in descending order of supports. The second job in CLOSET takes the entire dataset and removes all the infrequent items. Eventually, the third job achieves the CFI mining process. The latter divides the dataset in Map phase into multiple splits using the item-based partitioning approach mentioned earlier in section 4.2. The Map phase finds for each frequent item its Sub-Dataset and the associates header table. The Reduce phase starts by comparing the supports of the items with the supports of the itemsets in the header table of the corresponding Sub-Dataset. Those which have the same supports, their string concatenation produces a CFI which is stored in a hash-table with its corresponding supports.

Finally, we also compared DCIM to the parallel PFP-Growth [11] implementation of the FP-Growth algorithm (PFP in short) for MapReduce. PFP is dedicated to extraction of frequent itemsets only (and the generation of frequent itemsets from closed frequents one can be done in a significant amount of time). However, this is an interesting comparison to a well-known approach of the literature. The default values for PFP in our experiments are: $Q = 30,000$ (the number of groups containing dependent transactions, for the construction of the corresponding FP-Trees from Sub-Datasets to each itemset candidates) and $K = 90$ (the number of top frequent itemsets). For more details see [11].

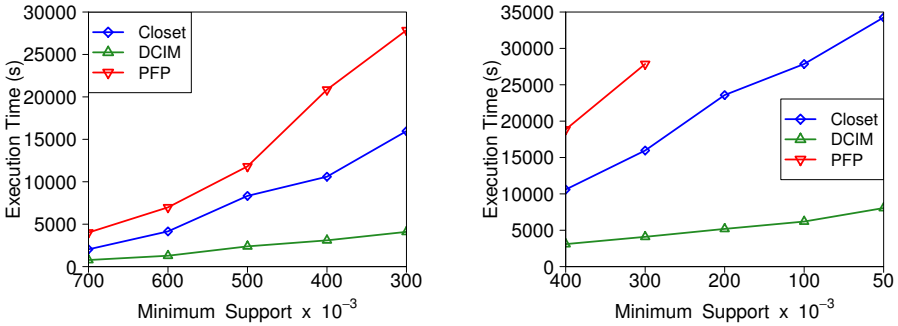
Datasets. We carried out our tests on two real-life datasets. The first one, called "English Wikipedia", represents a transformed set of Wikipedia articles into a transactional dataset, each line mimics an article. It contains 8 millions transactions with 7 millions distinct items, in which the maximal length of a transaction is 150, 000, and the size of the whole database is 4.7 Gigabytes. The second dataset, called "ClueWeb", consists of Web pages that were collected in January and February 2009 and is used by several tracks of the TREC conference. During our experiments, we used a part of this dataset with 53 millions transactions including 11 millions items with a maximal length of a transaction of 700,000. The size of the considered "ClueWeb" dataset is 24.9 Gigabytes.

Runtime. Figures 3 and 4 show the results of our experiments on both English Wikipedia and ClueWeb datasets (respectively). Figure 3a reports the comparative performance of DCIM under different values of minimum support (θ) less then 1% of the overall size of the dataset. We see that DCIM sharply outperform both other algorithms. In fact, Wikipedia dataset contains a most equally number of items and transactions. Thus, as far as θ value is low, PFP and CLOSET generate too many candidates, and a lot of long Sub-datasets for each one. So, the inclusion tests and evaluations under the pruning methods used in



(a) All algorithms (b) Focus on scalable algorithms

Fig. 3: Runtime on the English Wikipedia dataset with a cluster of 16 nodes

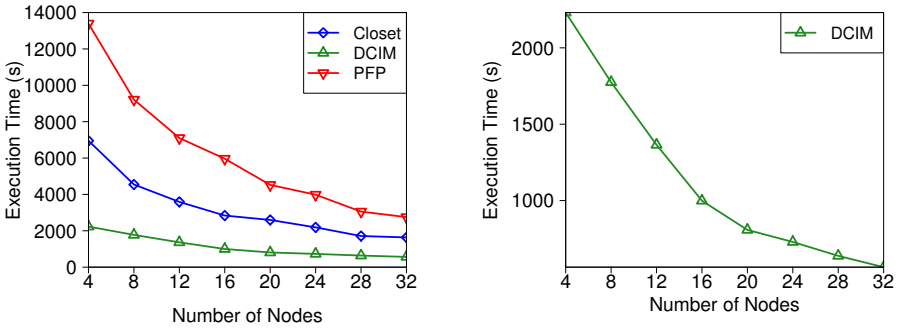


(a) All algorithms (b) Focus on scalable algorithms

Fig. 4: Runtime on the ClueWeb dataset with a cluster of 16 nodes

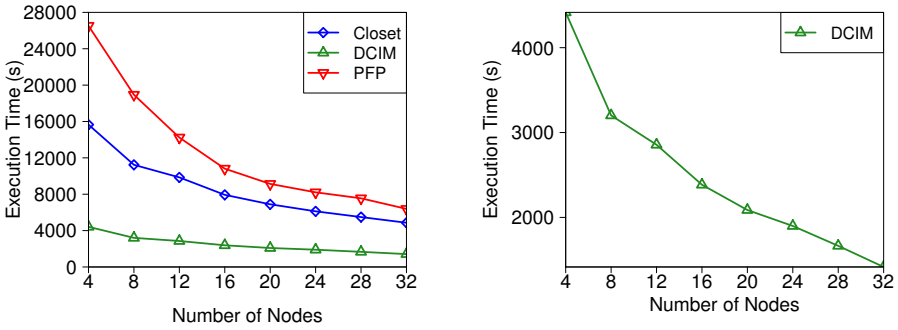
these two algorithms causes lead as expected to poor performances. Therefore, the response time of PFP and CLOSET grows exponentially and gets quickly very high. DCIM overcomes these problems by using prime numbers to generate the Sub-datasets through division operations. Furthermore, the GCD in each Sub-dataset has eliminated the check of supports between the candidate and its deduced closure, leading to much better performances. For instance, on the wikipedia dataset, the difference in response time is 5% with a support of $\theta = 60 \times 10^{-3}$, while it grows up to 43% with a support of $\theta = 10 \times 10^{-3}$.

Figure 3b highlights the difference between the algorithms of Figure 3a that scale. Although Closet continues to scale with $\theta = 40 \times 10^{-3}$, it is outperformed by DCIM, while PFP does not scale for lower threshold values. Also, with $\theta \leq 20 \times 10^{-3}$, we clearly observe a significant difference in the response time between DCIM and all the algorithms from the state of the art, owing to its robust and efficient core mining process. In Figures 4a similar experiments have been conducted on the ClueWeb data set, and we observe very similar behaviors



(a) All algorithms (b) Focus on DCIM

Fig. 5: Speed-up on the English Wikipedia dataset, $\theta = 50 \times 10^{-3}$



(a) All algorithms (b) Focus on DCIM

Fig. 6: Speed-up on the ClueWeb dataset, $\theta = 500 \times 10^{-3}$

(*i.e.*, DCIM outperforms existing approaches, and the same order between all algorithms is kept).

Speedup. In order to assess the speed-up of our approach, we performed experiments where we measured the response times with a varying number of computing nodes. In Figures 5 and 6, we performed multiple evaluations over different number of nodes, whith $\theta = 50 \times 10^{-3}$, on the Wikipedia and ClueWeb datasets (respectively). Figures 5a and 6a show the comparative speed-up results of all algorithms, and confirm the clear advantage of DCIM for all the possible settings in the number of nodes. Figures 5b and 6b focus on the speed-up of DCIM only. This is the same number of nodes and same value of θ (and, of course, the same response times for each number of nodes), with a magnified view on DCIM. We can observe the very good speed-up of DCIM which, by taking into account parallel optmizations in its core design, benefits from an increase in the number of computing nodes.

6 Conclusion

In this paper, we proposed a reliable and efficient parallel algorithm for CFI mining namely DCIM, that shows significantly better performances than approaches from the state of the art. In addition to using prime numbers and processing big integers, we provide DCIM with optimizations designed towards massive distribution and the MapReduce framework. The results illustrate that our method outperforms other alternatives, mainly by reducing the overhead of data exchange between nodes.

Acknowledgments

This work has been partially funded by the European Commission under the CloudDBAppliance project (grant 732051) and performed in the context of the Computational Biology Institute in Montpellier.

References

1. Sandy Moens, Emin Aksehirli, and Bart Goethals. Frequent itemset mining for big data. In *Proceedings of IEEE'13 on Big Data*, Santa Clara, CA, USA, 2013.
2. Ana Gainaru, Franck Cappello, Stefan Trausan-Matu, and Bill Kramer. Event log mining tool for large scale hpc systems. In *Proceedings of Euro-Par'11*, Berlin, Heidelberg, 2011.
3. Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Mining console logs for large-scale system problem detection. In *Proceedings of SysML'08*, Berkeley, CA, USA, 2008.
4. Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of ICDT'99*, Jerusalem, Israel, 1999.
5. Ke Chen, Lijun Zhang, Sansi Li, and Wende Ke. Research on association rules parallel algorithm based on fp-growth. In *Proceedings of the ICICA'11*, Qinhuangdao, China, 2011.
6. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *J. Commun. ACM*, 2008.
7. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of USENIX, HotCloud'10*, Boston, MA, USA, 2010.
8. Su-Qi Wang, Yu-Bin Yang, Yang Gao, Guang-Peng Chen, and Yao Zhang. Mapreduce-based closed frequent itemset mining with efficient redundancy filtering. In *Proceedings of IEEE'12 ICDM*, Brussels, Belgium, 2012.
9. Osmar R. Zaïane, Mohammad El-Hajj, and Paul Lu. Fast parallel association rule mining without candidacy generation. In *Proceedings of IEEE'01 ICDM*, San Jose, California, USA, 2001.
10. Eric Li and Li Liu. Optimization of frequent itemset mining on multiple-core processor. In *Proceedings of VLDB'07*, Vienna, Austria, 2007.
11. Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: parallel fp-growth for query recommendation. In *Proceedings of RecSys'08*, Lausanne, Switzerland, 2008.

12. Jianyong Wang, Jiawei Han, and Jian Pei. CLOSET+: searching for the best strategies for mining frequent closed itemsets. In *Proceedings of SIG-KDD'03*, Washington, DC, USA, 2003.
13. Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. Fast and memory efficient mining of frequent closed itemsets. *J. IEEE'06*, 2006.
14. Benjamin Négrevergne, Alexandre Termier, Jean-François Méhaut, and Takeaki Uno. Discovering closed frequent itemsets on multicore: Parallelizing computations and optimizing memory accesses. In *Proceedings of HPCS'10*, pages 521–528, Caen, France, 2010.
15. Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. LCM: an efficient algorithm for enumerating frequent closed item sets. In *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, Melbourne, Florida, USA, 2003.
16. Shui Wang and Le Wang. An implementation of fp-growth algorithm based on high level data structures of weka-jung framework. *J. JCIT*, 2010.
17. Christophe Nègre. Efficient binary polynomial multiplication based on optimized karatsuba reconstruction. *J. Cryptographic Engineering*, 2014.
18. Alberto Zanoni. Iterative toom-cook methods for very unbalanced long integer multiplication. In *Proceedings ISSAC'10*, Munich, Germany, 2010.

DCIM: Une Approche Massivement Distribuée pour l'Extraction de Motifs Fermés Fréquents

Résumé

L'extraction d'itemsets fermés fréquents (IFF) est l'un des principaux défis de la fouille de données, car elle permet de découvrir des itemsets avec une meilleure efficacité et une plus grande compacité de résultats. Cependant, la découverte de ces itemsets dans des données massivement distribuées pose un certain nombre de problèmes qui ne sont pas abordés par les méthodes traditionnelles. Une solution pour ce problème est de tirer parti des environnements parallèles, comme MapReduce ou Spark, qui permettent de fabriquer de puissantes unités de calculs et de stockage à partir de machines ordinaires. Malheureusement, les implémentations directes d'algorithmes de fouille de données, dans de tels environnements, sont des échecs puisque les principes de distribution ne sont pas pris en considération. Nous abordons le problème de l'extraction des IFF à partir de données massives en introduisant un nouvel algorithme parallèle appelé DCIM. Notre algorithme utilise une approche basée sur les nombres premiers en utilisant uniquement les opérations de multiplication et de division. Une caractéristique clé de DCIM est la combinaison de propriétés de la fouille de données avec les principes de la distribution massive de données. Nous avons effectué des expérimentations complètes sur de grands ensembles de données du monde réel pour évaluer les performances de DCIM. Les résultats obtenus mettent en évidence l'efficacité de DCIM avec des données qui contiennent jusqu'à 53 millions d'articles.