PIMiner: Parallel Interval Mining of Temporal Patterns using MapReduce

Prakhar Dhama
Department of Computer Science and Engneering
Indian Institute of Technology Roorkee
Roorkee, India
Email: prakhardhama@gmail.com

***Abstract*—The data generated, in recent times, from myriad of networked sensors and embedded systems is increasing exponentially. Many such real-world data streams comprise time interval-based data. Finding interesting temporal patterns on such interval-based events that have non-zero duration is an emerging research problem. Temporal pattern mining is useful in various domains, such as stock analysis in financial markets, tracking user activity patterns on website, meteorological traceability, examining interval activities in medical domain and sequencing in DNA. Serial pattern mining algorithms are infeasible in handling large scale data. To the best of our knowledge, all the current pattern mining algorithms on interval data are serial in nature. Therefore there is an increasing need to develop a methodology which allows parallel interval mining. In this paper, we propose a framework, called *PIMiner*, that can find frequent temporal patterns in distributed fashion. Our work presents a way to achieve both task parallelism and data parallelism to mine temporal patterns on multiple computing nodes using threading and MapReduce. Experimental studies indicate that the task-level and data-level parallelisms alone are quite effective compared to serial interval pattern mining.**

***Keywords*-Data mining; interval-based events; sequential patterns; temporal patterns; Hadoop cluster; task-level parallelism; threading; data-level parallelism; MapReduce; PIMiner;**

I. Introduction

Temporal pattern mining is critical to data mining in time series data which discovers relevant relations among interval events. It is useful in various domains like stock analysis in financial markets, tracking user activity patterns on website, meteorological traceability, examining interval activities in medical domain and sequencing in DNA.

Much of the existing work focuses on discovering frequent patterns among point-based time events and not with interval events. Mining temporal patterns is much more demanding task compared to simple events because of the complex relation among two interval events [**?**]. This results in huge search space and exponential increase in possible temporal patterns as the support reduces. For Apriori based algorithms the complex relation among intervals may lead to large number of candidate patterns followed by tedious work of support calculation. On the other hand for the pattern growth algorithm complexity within the long pattern increases complexity.

Advances is computing and sensing technologies has resulted in exponential growth in data volume. It has now become infeasible to analyse such large scale data on single system. The scalable distributed algorithms can solve these problems dealing with big data. Parallel mining can be easily achieved once the ample storage and numerous computing resources is made available in distributed fashion. To the best of our knowledge, neither the task parallelism, nor the data parallelism has been addressed for interval pattern mining. In this paper, we propose a new framework PIMiner - **P**arallel **I**nterval Miner which incorporates both task and data level parallelism.

The rest of the paper is organized as follows: Section II presents the related work carried on interval pattern mining and parallel work on point-based pattern mining. Section III presents the proposed model. Section IV briefly demonstrates the experimental setup used for performance study. Section V exhibits the experimental results with the performance as well as large scale tests of the proposed framework used for interval mining. Finally, we conclude the paper in Section VI.

II. Background and Related Work

Mining Algorithms, of late, have focused only on finding frequent patterns from immediate events i.e. events with zero duration. In real world, various events, that instead of being immediate, persist for some time period. The sequential algorithms for time point cannot be applied to these sequence of events having start and finish time. The immediate events can only find sequence of pattern like fever followed by indigestion followed by vertigo. However, this sequence of pattern is not appropriate to evince the complex relations in medical field or financial market where duration of events plays a crucial role. Similarly power meter in house that logs interval based data i.e. when each appliance is on or off.

A. Interval Events

A temporal database can handle data with time. It stores all the interval sequences in which each interval denoting an interval-based event can have starting and finish times. Multiple sequences are present in the database each with

Table I: Database with Interval Sequences

SID	event symbol	start time	finish time	interval duration
1	A	1	3	2
	B	2	6	4
	C	6	10	4
	D	6	10	4
2	A	2	6	4
	B	4	10	6
	C	12	15	3
	D	12	15	3
	E	6	7	1
3	C	5	7	2
	D	5	7	2
4	A	1	5	4
	B	3	8	5
	C	10	14	4
	D	10	14	4
	E	4	6	2

such interval-based events which can be present in any fashion including overlapping each other.

A temporal database can handle data with time. It stores all the interval sequences in which each interval denoting an interval-based event can have starting and finish times. Multiple sequences are present in the database each with such interval-based events which can be present in any fashion including overlapping each other. An interval sequence is a collection of several intervals. Table I shows the example database with four interval sequences 1, 2 3 and 4 as sequence id containing subset of five interval events A, B, C, D and E. Then, the support of temporal pattern *(C equals D)* is 4, support of pattern *(A overlaps B)* is 3 and support of pattern *(E during B)* is 2.

All of the work related to temporal pattern mining is based on the 13 relations among temporal events given in original James F. Allens work [**?**]. These various relations between two interval events say X and Y is depicted in Table II. The support of a temporal pattern, which is a composite of two events with a relation, is the number of interval sequences in database in which it occurs. If the support is greater than minimum threshold it is a frequent temporal pattern. The length of temporal pattern is the number of events in the pattern. The Allen 13 relations can be reduced to 7 as expect equal relation 6 relations are just inverse of other 6, so identifying one set implicitly expresses the rest. For example, the relation A before B implicitly implies the inverse relation B after A.

B. Representation Methods

An appropriate representation of the temporal pattern is crucial for further mining techniques. One of the earliest representation to express temporal pattern was hierarchical as proposed by Fu and Kam [**?**]. The temporal pattern was formed by combining frequent temporal pattern to original long pattern. However this relationship was ambiguous. Since same relation can be mapped into different temporal pattern based on the candidate selection. The ambiguity

Table II: Relations among Temporal Events

Relation	Symbol	Symbol for Inverse	Pictorial Example
X before Y	<	>	XXX YYY
X equal Y	=	=	XXX YYY
X meets Y	m	mi	XXXYYY
X overlaps Y	o	oi	XXX YYY
X during Y	d	di	XXX YYYYY
X starts Y	s	si	XXX YYYYY
X finishes Y	f	fi	XXX YYYYY

was addressed by Hoppner [**?**] by using a matrix to list exhaustively all the relations and using graph connectivity to mention a temporal pattern only once if it is connected. The ambiguity of same pattern inferring possible different relations was addressed by Patel [**?**] by including the count of each relation. Wu and Chen proposed endpoint representation [**?**] to unambiguously depict the pattern where starting and finishing points of each interval event.

In this paper we used endtime representation as proposed in [**?**] which is an extension to the endpoint representation for expressing temporal patterns. Endtime representation also includes the time of occurence of the endpoint in the representation.

C. Interval Pattern Mining Algorithms

Interval pattern mining is based on identifying the aforementioned Allens relation in the given temporal database. Only recently the work is carried out on interval pattern mining. To the best of our knowledge all the interval pattern mining algorithms are serial in order and no prior work is done on addressing the parallel techniques for interval pattern mining because of the sheer complexity involved in the same.

In early days, most of the algorithms for discovering temporal patterns were Apriori-like. Villifane [**?**] proposed a mining method by converting interval sequences into containment graphs. Kam and Fu [**?**] used Apriori like approach on hierarchical representation. Cooper et al. proposed Recent Temporal Pattern (RTP) [**?**] mining that mines frequent time-interval patterns backward in time starting from patterns related to the most recent observations, which can be typically most important for prediction in certain situations. HDFS [**?**] converts the temporal database into a list of sequence ids for each event. Then merging the sequence ids using enumeration tree. The enumeration tree is generated one by one for each frequent sequence. IEMiner [**?**] is one other Apriori-based algorithm. First, it scans the entire database to obtain all the frequent single events. Then generates the higher level candidate set until the candidate set is empty. The candidate generation follows that a (m+1)-pattern is a candidate for frequent itemset given it is formed

using a frequent m-pattern and 2-pattern which is present in at least m-1 of the frequent m-patterns.

Projection-based interval mining uses divide and conquer strategy to find frequent temporal patterns. If α is a sequential pattern, then let $DB_{|\alpha}$ denotes α projected database, which is a collection of sequences suffixes with regard to prefix α. Then α+1 pattern is generated using the projected database and extending the α sequential pattern. *TPrefixSpan* [**?**] uses projection based method along with the endpoint representation of interval sequence to extend the patterns. It recursively scans the projected database to find longer sequential patterns. Sadasivam [**?**] refined TPrefixSpan so that number of database scans is reduced. HTPM [**?**] was developed to mine hybrid patterns i.e. both point-based and temporal patterns from the sequence database. The most recent *TPMiner* [**?**] extends the concept of aforementioned database projection. Events with same sequence ID are clubbed in single interval sequence. The procedure for *TPMiner* follows the steps mentioned below. Julia et al. developed *Inc_TPMiner* (Incremental Temporal Pattern Miner) [**?**] to incrementally discover temporal patterns from interval-based data. The *TPMiner* [**?**], as the author claims, performs better than previous serial interval mining algorithms. In this work we have extended the algorithm to parallel paradigm using MapReduce.

D. Parallel Frequent Itemset Mining

Although parallel itemset mining is not addressed for finding temporal patterns due to complexity of the various interval pattern mining algorithms. But there is some work being carried out to find frequent patterns in sequential pattern mining in parallel fashion which in turn can provide the insight to manifesting the similar techniques for interval pattern mining. Most of the previously developed parallel algorithms [**?**][**?**] were Apriori based which suffer from high I/O overhead and synchronization.

A small number of FP-growth-like implementation [**?**][**?**][**?**] address parallel FIM problem. PFP [**?**] is a popular parallel frequent itemset mining algorithm that parallelize the classic FPGrowth algorithm. Recent work in parallel Apriori-like algorithm, such as sequence-growth [**?**] that uses lexicographical tree, has to scan database multiple times and exchange large number of candidate itemset. PFP uses the MapReduce approach which address the problem of dataset not fitting in memory. PFP uses three MapReduce phases to parallelize FP-growth by dividing the data and applying FP-growth in parallel. FiDoop [**?**] is the most recent technique that incorporated FIU-tree [**?**] to mine frequent itemset mining. FIU-tree has certain advantages over traditional FP-tree. FIU-tree provides reduced I/O overhead, offers an integral way to partition the dataset, compressed storage, and avoids the recursive traversal of tree.

III. Proposed Methodology

Temporal Pattern Mining is much computationally expensive compared to point-based events because of the complex number of relations between two interval events. This can be illustrated in terms of search space for both types of mining.

A. Complexity Analysis

Point-based Events. For a point based database with two frequent events A and B, the top three levels of sequential mining search space is illustrated in Equation 1. Note that for a combination other event can either coincide with previous or just occurs after the previous event. So if the number of frequent events is e and the longest pattern is of length is l, the order of the search space would be,

$$e + e * (2 * e) + e * (2 * e)^2 + ... + e * (2 * e)^{l-1} = \frac{e((2e)^l - 1)}{(2e) - 1} = O((2e)^l) \quad (1)$$

Interval-based Events. Although for a temporal database containing two frequent interval events A and B, the search space is illustrated in Equation 2. The search space for temporal database is highly explosive. Note that the Allens 13 relations is reduced in 7 relations without loss of generality. So any new interval event to be appended to *l*-pattern can have 7^l possible arrangement with previous l events. So again, if the number of frequent interval events is e and the longest temporal pattern is of length is *l*, the order of the search space would be,

$$e + e(7 * e) + e(7 * e)(7^2 * e) + ... + e(7 * e)(7^2 * e)..(7^{l-1} * e) = O(7^{l^2} e^l) \quad (2)$$

From the search space analysis, it is shown that complex relationships of temporal events is critical to designing an efficient algorithm. In Apriori-like algorithms, for each iteration this complex relationships leads to generation of huge number of candidate sequence which ultimately leads to the arduous task of finding support of it, although most them might not be present in the temporal database itself. Furthermore, in Pattern-growth algorithms, the complexity of temporal relation within the pattern itself leads to expensive computation. The low level tasks is expensive in terms of space and time complexity and are potential candidate for parallel techniques. So the motivation is to devise a complete, efficient, and scalable patterngrowth algorithm.

B. Representation Scheme

We used *endtime representation* in implementation which in contrast to *endpoint representation* also incorporates the actual time of each endpoint that is used in some quantitative analysis. For example, including the time of endpoints can help predicting the duration and time of the next occurrence of the event. The endtime representation of the temporal database presented in Table I is depicted in Table III.

Table III: Sample Endtime Representation

SID	Event symbol	Start time	Finish time	Endtime representation
1	A	1	3	$\binom{A+}{1}\binom{B+}{2}\binom{A-}{3}\binom{B-}{6}\binom{C+\ D+}{6\ \ 6}\binom{C-\ D-}{10\ \ 10}$
	B	2	6	
	C	6	10	
	D	6	10	
2	A	2	6	$\binom{A+}{2}\binom{B+}{4}\binom{A-\ E+}{6\ \ 6}\binom{E-}{7}\binom{B-}{10}\binom{C+\ D+}{12\ \ 12}\binom{C-\ D-}{15\ \ 15}$
	B	4	10	
	C	12	15	
	D	12	15	
	E	6	7	
3	C	5	7	$\binom{C+\ D+}{5\ \ 5}\binom{C-\ D-}{7\ \ 7}$
	D	5	7	
4	A	1	5	$\binom{A+}{1}\binom{B+}{3}\binom{E+}{4}\binom{A-}{5}\binom{E-}{6}\binom{B-}{8}\binom{C+\ D+}{10\ \ 10}\binom{C-\ D-}{14\ \ 14}$
	B	3	8	
	C	10	14	
	D	10	14	
	E	4	6	

For a given interval sequence Q = $< (e_1, s_1, f_1), (e_2, s_2, f_2), ..., (e_n, s_n, f_n) >$ and the corresponding endpoint sequence $ES_Q =< p_1, p_2, ..., p_{2n} >$ the time sequence would be $TS_Q =< t_1, t_2, ..., t_{2n} >$ where t_i is the occurrence time of endpoint p_i. Then the endtime representation would be a tuple of the corresponding items of endpoint sequence ES_Q and temporal sequence TS_Q i.e.

$$(ES_Q, TS_Q) =< (p_1, t_1), (p_2, t_2), ..., (p_{2n}, t_{2n}) > \quad (3)$$

Temporal Pattern. Suppose two endpoint sequences $x =< p_1, p_2, ..., p_n >$ and $y =< q_1, q_2, .., q_m >$, here the p_i and q_j are pointsets such that $n \leq m$. Then x is the subsequence of y denoted by $x \subseteq y$ if there exist n integers $1 \leq g_1 \leq g_2 ... \leq g_n \leq m$ such that $p_1 \subseteq q_{g_1}, p_2 \subseteq q_{g_2}, ..., p_n \subseteq q_{g_n}$. Also, the y is termed as the supersequence x. For a temporal database sequence $< ID, Q >$ where id is sequence id and Q is endpoint sequence, Q is said to contain sequence x if x is subsequence of Q. Then the support of x would be number of sequences containing x i.e.

$$s(x) = |\{< id, Q > | \ (< id, Q >\subseteq DB) \wedge (x \subseteq Q)\}| \quad (4)$$

Therefore, the temporal pattern is a frequent endpoint sequence, having support greater than specified minimum support threshold, where every starting endpoint as a corresponding finishing endpoint and vice versa in the sequence that is endpoints exist in pairs.

Consider the database in Table III and minimum support threshold as 3, then sequence $< (C + C-)(D + D-) >$ and $< A + B + A - B- >$ are frequent with support 4 and 3 respectively, while sequence $< B + E + E - B- >$ is infreqeunt with support 2.

C. Proposed PIMiner Algorithm

In this section, proposed Parallel Interval Miner (PIMiner) algorithm is depicted based on the endtime representation described in previous section. It illustrates how to incorporate data and task parallelism in the proposed interval pattern mining algorithm. The pseudocode of PIMiner is shown in Figure 1. PIMiner manifests divide and conquer technique using projection database.

Algorithm: PIMiner (DB, min_support, [max_len, num_threads, blocksz])

INPUT – DB: a given temporal database
min_support: minimum support threshold
max_len: optional maximum length of pattern, default: 0 i.e. all patterns
num_threads: optional number of threads, default: 1
blocksz: optional block size for distributed storage, default: 0 i.e. for HDFS size is 128 MB
OUTPUT – the set of all frequent temporal patterns

1. TP ← Φ
2. *Map* DB *block wise* into endtime representation
3. *MapReduce block wise* all frequent endpoints
4. *Filter block wise* infrequent endpoints
5. FE ← all the frequent starting endpoints
6. QU ← synchronized thread safe queue
7. for each X ∈ FE do
8. *Map* Project $DB_{|X}$
9. Enqueue tuple (x, $DB_{|X}$) to QU
10. *Concurrently* start num_threads with procedure **PISpan** (QU, min_support, max_len, blocksz, TP)
11. Join all threads
12. Output TP

Figure 1: PIMiner Algorithm Pseudocode

Projected Database. For a given endpoint sequence α in the database DB represented in endpoint representation. Then, the α-projected database, can be denoted by $DB_{|\alpha}$, is the collection of all the suffixes with respect to prefix α in all the sequences in DB.

1) PIMiner Framework: In the main framework of algorithm PIMiner in Figure 1, the input is the temporal database which contains the sequences of temporal events with starting and ending time. If the database is given in tuple format with sequence id and temporal event, then extra pre-processing step takes care of it by merging all the temporal events of same sequence in one list. The first step is to transform the database into endtime representation, detailed implementation would be followed in next section. The endpoints in the endtime representation is sorted by time of each point-event. The frequency of all the endpoints is calculated concurrently (Line 3, Figure 1).

The infrequent endpoints are removed partition wise in parallel (Line 4, Figure 1). Then all the frequent starting endpoints is collected as a starting point of temporal pattern. A thread safe queue is maintained so that different thread can access the queue concurrently. For each starting endpoint the projected database is generated, which is ultimately enqueued as a tuple of starting endpoint and the projected database (Line 9, Figure 1).

Then task parallelism is achieved as the mentioned number of threads are started which concurrently executes the PISpan procedure for extending the frequent sequence. All the temporal patterns are listed at last after the queue is empty and all threads finishes there execution.

2) PISpan Procedure: The *PISpan* procedure, as shown in Figure 2 extends the frequent sequence to find longer frequent sequence. The endtime sequence and the respective

Procedure: PISpan (QU, min_support, max_len, blocksz, TP)

INPUT – QU: synchronized queue containing tuple of sequence and projected db
min_support: minimum support threshold
max_len: maximum length of pattern, default: 0
blocksz: block size for distributed storage, default: 0
TP: thread safe queue for frequent temporal patterns
OUTPUT – append found temporal patterns to TP

1. $(\alpha, DB_{|\alpha})$ ← Dequeue QU
2. FE ← **PostEndpoints** $(\alpha, DB_{|\alpha}$, min_support)
3. for each X ∈ FE do
4. Append X to α to generate sequence β
5. if β is a temporal pattern:
6. TP ← TP ∪ β
7. $DB_{|\beta}$ ← **ProjectDB** $(\beta, DB_{|\alpha})$
8. if $DB_{|\beta} \neq \Phi$ and (max_len = 0 or β.len < 2*max_len) then
9. Enqueue tuple $(\beta, DB_{|\beta})$ to QU

Figure 2: PISpan Procedure Pseudocode

projected database is dequeued (Line 1, Figure 2). The pruning is done by borrowing idea from PrefixSpan [**?**].The possible extension to the sequence is generated by procedure PostEndpoints collected as a list of frequent endpoints (Line 2, Figure 2). For each of the frequent endpoint, new sequence is formed by appending the endpoint (Line 4, Figure 2). If the new sequence is a valid temporal pattern, where all the starting endpoints have a respective finishing endpoints and vice versa, then the sequence is appended to output queue of temporal patterns (Line 6, Figure 2).

The projected database is generated with respect to the new sequence. The tuple of this projected database and new sequence is enqueued if the projected database is not empty and either max_len is not specified, which is the case for finding all the frequent temporal patterns, or the current length of the pattern is less than the max_len (Line 9, Figure 2).

Procedure: PostEndpoints (α, $DB_{|\alpha}$, min_support)

INPUT – α: frequent endpoint sequence
$DB_{|\alpha}$: projected database w.r.t. α
min_support: minimum support threshold
OUTPUT – FE: frequent endpoints that can be appended α

1. FE ← Φ
2. *Map block wise* for each ETS ∈ $DB_{|\alpha}$ do
3. stop = position of *leftmost finishing endpoint* which has a corresponding starting endpoint in α ***or*** end of ETS
4. for each X ∈ ETS until the stop
5. if X is *starting endpoint* or is at stop then
6. FE ← FE ∪ X
7. *Reduce* infrequent endpoints from FE
8. Return FE

Figure 3: PostEndpoints Procedure Pseudocode

3) PostEndpoints Procedure: The *PostEndpoints* procedure as depicted in Figure 3 finds the frequent endpoints that can be appended to the α sequence and the resulting sequence is also frequent. We need to collect the endpoints only till the leftmost endpoint which has a corresponding starting endpoint in α, because beyond it collecting the endpoints would not lead to valid temporal pattern on concatenation.

If no such finishing endpoint is found all starting endpoints until the end of the sequence is collected (Line 3, Figure 3). The frequency of each endpoint is calculated concurrently (Line 7, Figure 3). Finally, the list of frequent endpoints is returned.

Procedure: ProjectDB (β, $DB_{|\alpha}$)

INPUT – β: new appended sequence
$DB_{|\alpha}$: projection prior appendation
OUTPUT – $DB_{|\beta}$: database projection w.r.t new sequence β

1. $DB_{|\beta}$ ← Φ
2. *Map block wise* for each EPS ∈ $DB_{|\alpha}$ do
3. SQ ← list of postfix endpoints of EPS w.r.t β
4. *Reduce finishing endpoints* which neither has corresponding starting endpoint in β nor in the SQ prior to it
5. $DB_{|\beta}$ ← $DB_{|\beta}$ ∪ SQ
6. Return $DB_{|\beta}$

Figure 4: ProjectDB Procedure Pseudocode

4) ProjectDB Procedure: The *projectDB* procedure is used of generating the projected database with respect to the endtime sequence β. The framework of the procedure with pseudocode is shown in Figure 4. The list of suffix sequence SQ is collected with respect to the prefix β. Note that we can prune finishing endpoints *ev* which has no corresponding starting endpoints in β nor in the prefix of *ev* in SQ (Line 4, Figure 4). The projected database is returned in the last step.

D. MapReduce Framework

The framework of PIMiner is shown in MapReduce flowchart in Figure 5.

The input database is first converted to endtime representation (ETS). Each sequence in ETS is a list of tuple of endpoint and the time of occurrence. The *MapReduce* job is used to find the frequency of such endpoints in block partition way. This way, in PIMiner the frequency of all the endpoints is reduced which is even faster than single serial scan of the database and maintaining a dictionary as in TPMiner. All the endpoints in Map job is converted to tuple of endpoint and count one in a single sequence.

The three phases involved in finding frequent endpoints and pruing infrequent ones while achieving block parallelism are:

1) Map Phase
2) Reduce Phase
3) Filter Phase

All the three phases are involved in achieving the task parallelism at block level. The number of partitions in the actual implementation can be determined using the optional input parameter *blocksz*. The number of partitions are determined at run time based on the variable size of the projected database. In *reduce* step all the values are

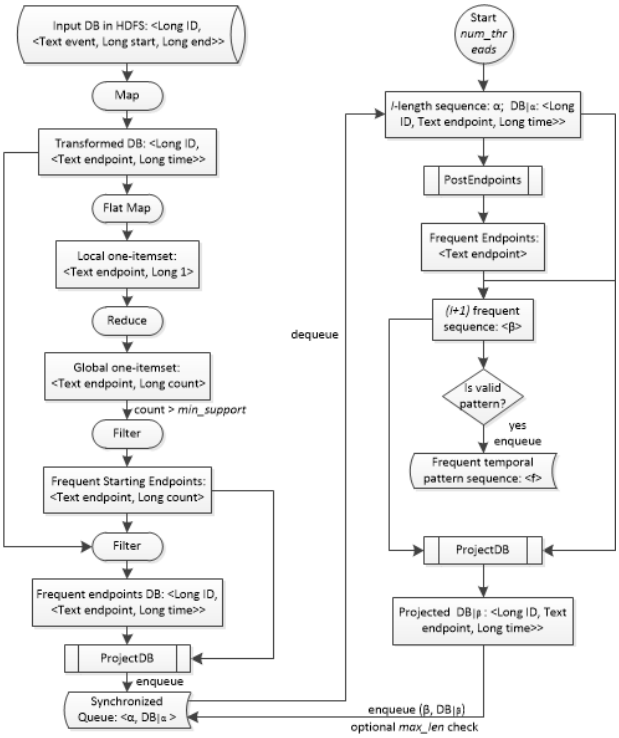

Figure 5: PIMiner MapReduce Job Flowchart

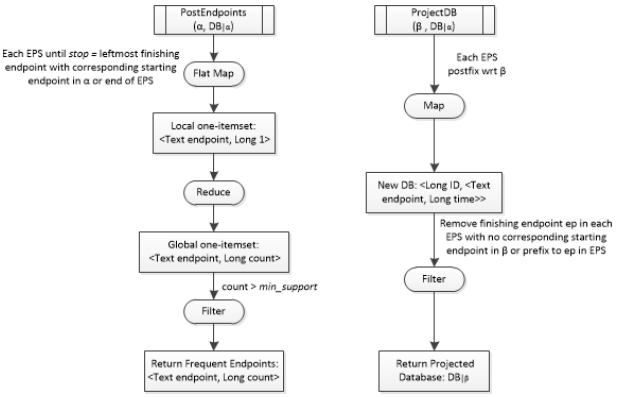

Figure 6: Subprocesses MapReduce Job Flowchart

merged by specified addition operator based on the key value. Finally, the frequency count can be determined of all the endpoints involved in the whole projected database. All the endpoints without occurrence consideration are obtained with their count. In *filter* step the endpoints with frequency less than the specified minimum threshold value are filtered as they cannot generate frequent endpoint sequence on concatenation with previous sequence.

IV. Experimental Setup

Data nowadays is so large that traditional data processing techniques are infeasible, then big data technologies are adopted. Datasets are growing rapidly as generated by numerous mobile devices, software logs, radio-frequency identification (RFID) chips and wireless sensors. Relation database cannot handle size of such magnitude. One of the most famous approach to handling big data was MapReduce model [**?**] given by Google in 2004. Apache open source framework Hadoop adopted the same MapReduce programming model. We used Apache Spark framework which is a lightning-fast cluster computing framework designed for fast computation. It was built on top of MapReduce and extends MapReduce model to other types of computations like stream processing and interactive queries. The distributed file used by Spark is HDFS which is nothing but Hadoop Distributed File System only. Spark is faster than Hadoop because the computation is carried out in-memory. Sparks uses Resilient Distributed Data storage (RDDs) which have fault tolerant implicitly. We have used python API for spark.

Table IV: Dell Rack Server

Name	Local IP	Roles	Cores	Memory	Disk
blade1.cloud.org	10.0.0.1	HDFS Balancer, HDFS NameNode, YARN Resource Manager, YARN JobHistory Server, Spark (standalone) Master	24	128 GiB	2 TiB
blade2.cloud.org	10.0.0.2	HDFS DataNode, YARN NodeManager, Spark (standalone) Worker	12	32 GiB	1 TiB
blade3.cloud.org	10.0.0.3	HDFS DataNode, YARN NodeManager, Spark (standalone) Worker	12	32 GiB	1 TiB
blade4.cloud.org	10.0.0.4	HDFS DataNode, YARN NodeManager, Spark (standalone) Worker	12	32 GiB	1 TiB
blade5.cloud.org	10.0.0.5	HDFS DataNode, YARN NodeManager, Spark (standalone) Worker	12	32 GiB	1 TiB
blade6.cloud.org	10.0.0.6	HDFS DataNode, YARN NodeManager, Spark (standalone) Worker	12	32 GiB	1 TiB
blade7.cloud.org	10.0.0.7	HDFS DataNode, YARN NodeManager, Spark (standalone) Worker	12	32 GiB	1 TiB

We used CDH (Cloudera Distribution of Hadoop) which is an open source Hadoop distribution provided by Cloudera Inc, an American enterprise software company. It is the most complete, robust and widely deployed Apache Hadoop distribution. More enterprises have used CHD than all the other distributions combined together. In CDH, Spark is deployed standalone where it sits above the HDFS and space for HDFS is though allocated explicitly. In this way both MapReduce and Spark and MapReduce run concurrently to complete all the Spark jobs in the cluster.

The details of the CDH used is mentioned as following. The CDH is installed over the Dell rack server. **Version:** Cloudera Express 5.6.0 **Java VM Name:** Java HotSpot(TM) 64-Bit Server VM **Java VM Vendor:** Oracle Corporation **Java Version:** 1.7.0_67

The Cloudera Manager has the following specifications. The Cloudera Manager system is used to send commands over the server over which the CDH is installed. **OS:** Ubuntu 12.04 LTS 64-bit **Memory:** 11.6 GiB **Processor:** Intel® Xeon® CPU E5-16500 @3.20GHz X12 **Disk:** 357.4 GB

The details of rack server installed in Department of Computer Science and Engineering, IIT Roorkee to run the job is illustrated in Table IV. The data is distributed stored across HDFS DataNode and there are 6 standalone Spark Workers for carrying out the computation.

Table V: Synthetic Data Generation Parameters

Parameters	Description
S	Number of temporal sequences
C	Average length of sequence
E	Total number of interval events
N_f	Average length of frequent pattern
N_e	List of possible duration of an interval

V. Performance Study

We conduct several tests including execution time, distribution of patterns, number of sparks, effects of threads, and scalability tests among others. When size of the dataset is not huge in kilo bytes the task parallelism is evident and only when size of dataset is in several hundred mega bytes, then the effects of data parallelism is evident.

A. Synthetic Data Generation

The synthetic datasets for experiments are generated as proposed by Srikant and Agarwal [**?**]. This proposed program was meant for time point-based events and hence needed to be modified accordingly. The several parameters that were used for synthetic temporal data generation are shown in Table V. The sequences generated are maximal as no same temporal event meets or overlaps within the same sequence. The total number of sequences generated is S. The event symbol is randomly selected while generating the sequence from the list of integers from 0 to E-1 where E is the total number of interval events across the temporal database.

Since the interval events have a duration, the program requires additional tuning for interval data generation. We used the method as proposed by Wu. The duration of the events are chosen from three categories viz. short, medium and long events. The duration of short, medium and long events are 4, 8 and 12 units respectively. Hence, the list N_e contains three elements i.e. $N_e = [4, 8, 12]$ in the program. The duration of the interval event is randomly selected from the list.

Finally, the temporal relations among the consecutive interval events is randomly selected to generate maximal sequence. The temporal relation is chosen from the set {*before, equal, meets, overlaps, starts, finishes*}. We generate S such sequences. The size of the sequence is determined prior to generation with from the Poisson distribution with mean equal to C. Additionally, a list of frequent sequences are generated that is distributed for the support P = 0.5 in the sequences randomly selected. If the length of the sequence is smaller than the randomly selected length than extra events are added with the randomly selected relations, or if the length of sequence is longer then the events are pruned at the end. The length of the generated frequent sequence is determined from the Poisson distribution with mean equal to N_f.

Table VI: S10kC20E1k Dataset Aspects

Parameters	Values
Number of sequences (S)	10000
Average length of sequence (C)	20
Number of events (E)	1000
Average length of frequent pattern (N_f)	7
Duration of the event (N_e)	[4 8 12]
Average duration of interval (D)	7
Support for pattern generation (P)	0.5
Total number of intervals (I)	200326

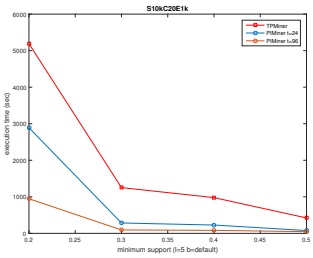

Figure 7: Execution Time Analysis for S10kC20E1k

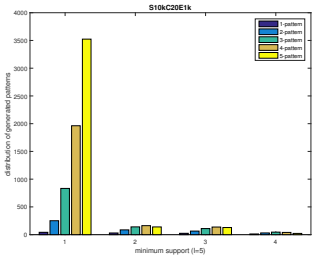

Figure 8: Distribution of Generated Patterns for S10kC20E1k

B. Performance on Synthetic Datasets

In the dataset generation several parameters are fixed. For the dataset S10kC20E1k the number of sequences is S = 10000, the total number of events is E = 1000 and the average length of the sequence is C = 20 as shown in Table VI.

In Figure 7 the execution time of our algorithm is compared to the serial TPMiner algorithm for the dataset S10kC20E1k. Maximum length of the patterns to be mined is set to L = 5. Since the size of the dataset is only 2.2MB the block size is set to default and there is no need for data parallelism as the completer dataset would fit in a single block. Here the task parallelism is achieved by increasing the number of threads i.e. in Figure 7 the number of threads is set to 24 and 96 and execution time is determined for different support values at 0.2, 0.3, 0.4, and 0.5. The execution time graph in clearly shows that PIMiner takes less time than TPMiner as the support is reduced. This is due to the increased task parallelism by increasing number of threads.

The distribution of the patterns for the dataset S10kC20E1k as 1-length, 2-length, 3-length, 4-length and 5-length is shown in Figure 8 for different values for support. It is evident that the number of patterns of shorter length hardly increases compared to the patterns of longer length. This is because the dataset is generated for frequent patterns with average size of N_f = 7.

We can observe the effects of changing the maximum length allowed for mining frequent pattern for different support values in Figure 9. The different values of maximum length of frequent pattern is used as L=2, 5 and 9. The support values used for comparison are 0.3, 0.35, 0.4, 0.45, and 0.5. As the maximum length allowed is increased

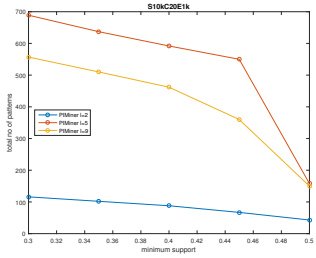

Figure 9: Number of Patterns on Maximum Length of Patterns for S10kC20E1k

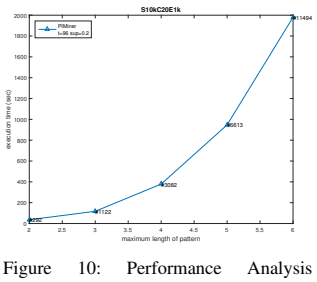

Figure 10: Performance Analysis on Maximum Length of Pattern for S10kC20E1k

Table VII: ASL Dataset Aspects

Parameters	Values
Number of sequences (S)	730
Average length of sequence (C)	31
Number of events (E)	234
Average duration of interval (D)	13
Total number of intervals (I)	22727

more number of patterns is generated as expected. Again for low value of L = 2 the number of patterns tends to increase linearly but as the length is increase and support is decreased, longer patterns are more frequently mined as observed in distribution of found patterns in Figure 8. Hence, the total number of patterns is more steeper for the high value of the maximal length of frequent value allowed. We can also observe the effects of changing the maximum length allowed on execution time. In Figure 10 the execution is plotted against different values for maximum allowed pattern length. The support value is fixed to sup = 0.2 and number of threads to t = 96 to so that program takes reasonably less time.

It can be illustrated that as the maximum length is increased the execution time increases exponentially. The total number of generated patterns is also shown in the figure prefixed with #. The total number of new x-length pattern can be determined by subtracting the total patterns at (x-1)-length by x-length. For example the total number of 4-length patterns generated for the support value 0.2 is 1960 (3082 1122) as the total number of frequent patterns up to length 4 are 3082 and total number of frequent patterns up to length 3 are 1122 respectively.

C. Real World Dataset Analysis

In addition to the experiments on synthetic dataset, we have conducted the performance study on a real dataset to show the applicability of the algorithm and efficiency compared to traditional serial algorithms. The dataset we have used is of American Sign Language (ASL) dataset.

In ASL dataset we can determine the relationship between gesture field used and the grammatical structure. The dataset contains total of 730 utterances. Each of the utterance contains the gestural as well as the grammatical field. The complete specification of ASL dataset is shown in Table VII.

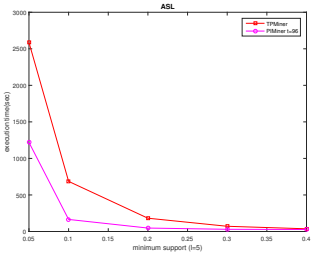

Figure 11: Execution Time Analysis for ASL Dataset

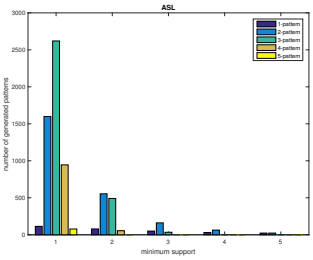

Figure 12: Distribution of Generated Patterns for ASL Dataset

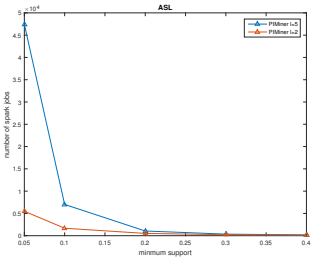

Figure 13: Spark Jobs Analysis for ASL Dataset

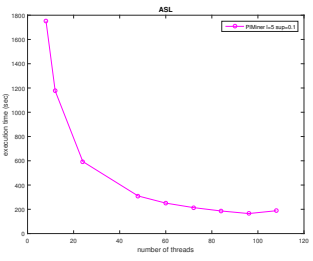

Figure 14: Threads Influence on ASL Dataset

The size of the dataset is 213.6KB. So task parallelism is enough to observe the performance of the PIMiner compared to serial TPMiner. The block size is again default and the whole dataset is in the single block. The execution time for the both the algorithms is shown in the Figure 11. The number of threads is set to 96 and different support values used are 0.4, 0.3, 0.2, 0.2, 0.1 and 0.05. It can be observed that the execution is exponentially increasing as the support decreases. The distribution of generated patterns is shown in Figure 12. It can be observe that medium length patterns are among most frequent that is of 3-pattern is one of most frequent which holds truth as verified against Papapetrou et al.

The number of spark jobs are increased drastically as the support is decreased as evident in Figure 13. The number of spark jobs includes mapping, reducing and collection among other RDD jobs. The number of spark jobs is plotted against support values 0.4, 0.3, 0.2, 0.1 and 0.05. The two different values of the maximum length of the frequent temporal pattern allowed i.e. L = 2 and L = 5. When the maximum length of the frequent temporal patter is increased the number of spark jobs are increased by large factors for small support which is more dependent on the dataset. The number of spark jobs is proportional to the total frequent patterns mined. As evident from Figure 12 as the number of frequent patterns is increased exponentially, so is the rate of the number of spark jobs increased.

The effect of the number of threads to execution time is shown in Figure 14. The permissible maximum value of the length of the frequent pattern is L = 5. The execution time is evaluated for the support value of sup = 0.1. When

Table VIII: S100kC40E1k Dataset Aspects

Parameters	Values
Number of sequences (S)	730
Average length of sequence (C)	31
Number of events (E)	234
Average duration of interval (D)	13
Total number of intervals (I)	22727

the number of threads are increased the execution time is reduced exponentially. Also observe that when number of threads are increased beyond threshold value the execution time again starts increasing. This is because as the number of threads grow the considerable amount of time is spent managing and scheduling all the threads. This threshold value is determined based on the system configuration. This value approximately converges to the total number of cores available across the whole distributed system.

D. Large Scale Tests

The real scalability is observed when the size of the dataset is considerably larger. Then the data parallelism is more evident. The first synthetic dataset used for this purpose is S100kC40E1k. This dataset contains 100000 sequences, the average length of the sequence is 40 and number of events is 1000. The size of dataset S100kC40E1k is 45.4MB. The detailed specifications of the dataset is shown in the Table VIII.

The execution time of our PIMiner algorithm is stacked against the serial algorithm TPMiner in Figure 15. Here to observe how data parallelism is achieved we even use a single thread for comparison. Here the parameter *b* is the block size in MBs in which the database is divided. As the block size value is decreased the number of blocks is increased and more parallelism is evident. The different parameters for comparison are:

1) TPMiner
2) PIMiner with number of threads (t) = 1 and block size (b) = 4MB
3) PIMiner with number of threads (t) = 1 and block size (b) = 2MB
4) PIMiner with number of threads (t) = 2 and block size (b) = 2MB

Analogous to the task parallelism using number of threads in Section 5.2, as the block size decreases the time of execution is decreases, but beyond certain threshold time starts to increase. This threshold value can be obtained by hit and trial, and is dependent on the dataset. Beyond this threshold the cost of maintaining the number of blocks per computing node would be more than the data parallelism achieve by partitioning the database in first place. The different support values used for different configurations of our algorithm are 0.3, 0.35, 0.4, 0.45 and 0.5. The maximum length of the frequent pattern L is set to L = 1. Hence this eventually finds the number of frequent temporal events in

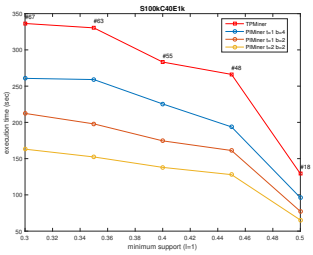

Figure 15: Execution Time Analysis for S100kC40E1k

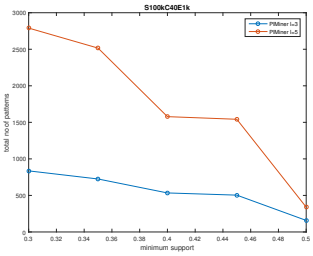

Figure 16: Patterns Generated on Maximum Length of Pattern on S100kC40E1k

the database. As the value of L is increased the difference between our algorithm and serial algorithm would be such drastic that it would be possible to plot the values in same graph.

It can be deduced from Figure 15 that when the value of block size is 4MB, PIMiner takes order of hundreds of seconds less time when support value is small. Also when the block size is further reduced to 2MB the execution time graph is further reduced. Finally, when the number of threads is increased by just 1 for the same block size of 2MB, the execution time is further reduced. The number of frequent interval events is show with prefix # before the value in the graph for reference.

Therefore, from this observation it can be safely inferred that to achieve the maximum possible performance we need to tweak both the parameters the number of threads to control the task parallelism as well as the block size to control the data parallelism. This sweet spot can be obtained by some hit and trial and is more dependent on the dataset as well as the distributed system on which the algorithm is executed. The total number of patterns generated for different support values for dataset S100kC40E1k is shown in Figure 16. We have used two different values of maximum length permissible for frequent temporal pattern viz. L = 3 and L = 5.

The values of support for which the total number of patterns are shown is 0.3, 0.35, 0.4, 0.45 and 0.5. As expected with decreasing support the total number of patterns are increasing. Similarly when maximum length of the allowed frequent pattern is increased from to 5, there is again a surge in number of temporal patterns as expected, as new 4-length and 5-length patterns are reported.

The large scale dataset used for the final evaluation are:

1) S100000C40E1k
2) S200000C40E1k
3) S300000C40E1k
4) S400000C40E1k
5) S500000C40E1k

These five datasets vary in number of total sequences. The average length of sequence is same for all the datasets C = 40 and number of total temporal events is also same for all datasets E = 1000. The number of sequences are

Table IX: C40E1k Dataset Aspects

Parameters	S100k	S200k	S300k	S400k	S500k
Number of sequences (S)	100000	200000	300000	400000	500000
Average length of sequence (C)	40	40	40	40	40
Number of events (E)	1000	1000	1000	1000	1000
Total number of intervals (I)	3997520	7999442	12004982	16002474	20002495

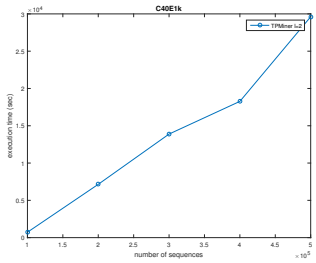

Figure 17: TPMiner Performance Analysis on C40E1k

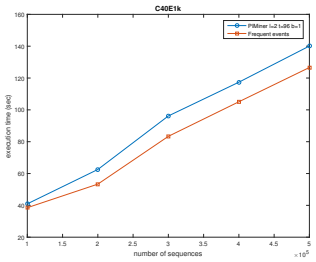

Figure 18: PIMiner Performance Analysis on C40E1k

100000, 200000, 300000, 400000, and 500000. The size of the databases are 45.4MB, 91.0MB, 136.9MB, 185.5MB and 232.4MB respectively. The detailed specification of the datasets is shown in Table IX mainly illustrating the differences among the datasets. The execution time for the TPMiner algorithm is shown in Figure 17 with maximum allowed length of temporal pattern is L = 2 for the number of sequences for the five datasets.

The execution time of PIMiner for same maximum length of frequent pattern I = 2 is plotted in Figure 18. Note that here we have tried to achieve maximum performance by tweaking both the parameters several number of threads and block size to t = 96 and b = 1MB respectively. Notice the considerable execution time improvement in two cases where TPMiner is taking time in order of several thousand seconds and PIMiner is taking only in few hundred seconds. For example, when tuned with optimal parameters PIMiner is 211 times faster than TPMiner for dataset S500kC40E1k. This is because the size of dataset is considerably large that data parallelism is quite influential. The only limiting factor is now the underlying distributed system.

VI. Conclusion

ining of frequent temporal patterns on interval based data is a crucial subfield of data mining. The complex relations among intervals make it inherently arduous to design efficient distributed temporal pattern mining algorithm. All the current pattern mining algorithms on interval-based events are sequential in nature. They cannot scale to large data set which cannot be stored in single memory. The various parallel techniques proposed in mining frequent patterns are carried out on instantaneous events, i.e. point-based events and not on interval events. In this work, we have developed a new algorithm *PIMiner* which runs in distributed fashion, which is a first such attempt on interval pattern mining. *PIMiner* employs both task and data parallelism techniques and is therefore highly scalable. The performance study of *PIMiner* indicate that the algorithm is much efficient compared to serial temporal pattern mining algorithms as it is evident when stacked against the state-of-the-art TPMiner algorithm for mining temporal patterns.

For future work no techniques is developed to utilize GPU computations in finding the frequent patterns in interval data. Although GPU based techniques for instantaneous events are presented which can help in tackling this issue. In order to achieve this an entirely new techniques need to be devised so that there is some portion of algorithm which can run in parallel on GPU cores. [**?**]

Acknowledgment

We would like to thank Dr. Dhaval Patel for his invaluable insight in the area of interval pattern mining.