Louisiana State University
LSU Digital Commons

LSU Master's Theses | Graduate School

2014

Multi-threaded Implementation of Association Rule Mining with Visualization of the Pattern Tree

Eera Gupta
Louisiana State University and Agricultural and Mechanical College, eerait@gmail.com

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_theses

Part of the Computer Sciences Commons

Recommended Citation
Gupta, Eera, "Multi-threaded Implementation of Association Rule Mining with Visualization of the Pattern Tree" (2014). *LSU Master's Theses*. 3864.
https://digitalcommons.lsu.edu/gradschool_theses/3864

This Thesis is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Master's Theses by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.

MULTI-THREADED IMPLEMENTATION OF ASSOCIATION RULE MINING WITH VISUALIZATION OF THE PATTERN TREE

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Systems Science

in

The Department of Electrical Engineering and Computer Science
The Division of Computer Science and Engineering

by
Eera Gupta
B.Tech.,University Institute of Engineering and Technology, 2009
December 2014

Acknowledgments

There is not a more pleasing exercise of the mind than gratitude. It is accompanied with such an inward satisfaction that the duty is sufficiently rewarded by the performance.
——————————————————————————————Joseph Addison

I would like to express my utmost gratitude to all those who have helped me in my thesis work.

Primarily, I bow down and thank the Almighty for giving me strength and patience to face and solve all the challenges and difficulties that came across.

Next, I express sincere gratitude towards my major professor, Dr. Omer M. Soysal for his immense support and guidance throughout my masters studies. His invaluable suggestions, knowledge and critiques helped me to take my research work in the right direction and increase my learning curve.

I am highly obliged to my co-chair Dr. Jianhua Chen for her considerate advices and teachings. My deepest thanks go to Dr. Jian Zhang for consenting to be in my thesis committee and providing his utmost cooperation. I am also indebted to the assistance from the staff members of the Division of Computer Science and Engineering at LSU.

I also acknowledge the support of my family for having the faith and encouraging me to pursue my dreams. I am very grateful to Shantanu Thatte for his insightful comments and suggestions. Last but not the least, I thank all my friends for always being there for me.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Abstract

Motor Vehicle fatalities per 100,000 population in the United States has been reported to be 10.69% in the year 2012 as per NHTSA (National Highway Traffic Safety Administration). The fatality rate has increased by 0.27% in 2012 compared to the rate in the year 2011. As per the reports, there are many factors involved in increasing the fatality rate drastically such as driving under influence, texting while driving, and various other weather phenomena. Decision makers need to analyze the factors attributing to the increase in an accident rate to take implied measures. Current methods used to perform the data analysis process has to be reformed and optimized to make policies for controlling the high traffic accident rates. This research work is an extension to the data-mining algorithm implementation "Most Associated Sequential Pattern" (MASP). MASP uses association rule mining approach to mine interesting traffic accident data using a modified version of FP-growth algorithm. Owing to the huge amounts of available traffic accident data, MASP algorithm needs to be further modified to make it more efficient with respect to both space and time. Therefore, we present a parallel implementation to the MASP algorithm. In addition to this, pattern tree and apriori-tid algorithm implementation has been done. The application is designed in C# using .NET Framework and C# Task Parallel Library.

Chapter 1
Introduction

As per the report from National Highway Traffic Safety Administration in 2014, the overall cost of motor vehicle crashes in the United States is equivalent of 1.9% of the $14.96 trillion Gross Domestic Product (GDP) in 2010 [8]. This recent report underscores the importance of safety measures that needs to be incorporated into the system. The quality of life and economy in any country are governed by making the roadways safer for the general public. *Highway Safety Research Group* (HSRG) is a division of the Information Systems and Decision Sciences Department (ISDS) within Louisiana State University, Baton Rouge. HSRG aims at storing, maintaining, and analyzing traffic accident related data that is collected from various law enforcement agencies throughout Louisiana. The raw data collected needs to be analyzed to extract interesting patterns from the input data and aid the policymakers in their decision-making.

1.1 Motivation

Department of Transportation and Development (DOTD) requires crash data reports to identify accident prone intersections and roadway segments and advance new projects. For example, if the curve in a road segment is experiencing more incidents due to the speed limit on the segment, then policymakers can reduce the limit for a particular section. There is an immense scope in *data mining* techniques to improve upon the information discovered from the data generated from traffic incidents.

1.2 Problem Statement

The previous research work done to mine crash-related patterns using "Most Associated Sequential Pattern" (MASP) algorithm has few drawbacks [7]. The application cannot handle and mine huge amounts of data efficiently. Moreover, we need to incorporate pattern visualization techniques into the application to display patterns and relationships contained within the data. In order to solve all these problems, we have made some enhancements to the existing application and performed experiments to analyze the performance with respect to CPU usage and memory limits.

1.3 Objective

This thesis aims at several objectives done to enhance the previous research work done in *MASP* (Most Associated Sequential Pattern).

1. Multi-threaded MASP: A multi-threaded implementation is done so that program execution speed can be increased. *C# Task Parallel Library* and *Threads* are used for parallelizing the tasks as well as Linked List structures for storing the huge volumes of data.

2. Sequential and Multi-threaded Apriori-TID: We have implemented both the sequential and multi-threaded versions of Apriori-TID algorithm. `LinkedList` and `HashMap` data structures are used to store huge volumes of frequent itemsets and association rules respectively generated from the datasets.

3. Multi-threaded FP-Growth: The existing sequential execution of the FP-Growth has been enhanced to include the multi-threaded implementation. The multi-threaded implementation improved the execution speed for running the algorithm to a great extent.

4. Pattern Tree Visualization: This implementation enables the visual aspect of the patterns in the form of a tree structure generated from MASP.

1.4 Definitions

1. Most Associated Sequential Pattern (MASP): MASP implies a sequential itemset $\{I_1,I_2,I_3,I_4,...I_i,I_k\}$ wherein each child I_i given its parent, has the highest association threshold and highest frequency among other items at the same level. A MASP pattern is defined as being sequential because each item-pattern depends on the previous pattern [7].

2. MASP Rules: The rules that are obtained after mining the transactional data using MASP is defined as MASP Rules.

3. Block: The transactions generated by the MASP query which comprises of select clause, predicate and data source collection. In the predicate for block, we use the equality of the form "attribute=value".

4. Counter Block: The transactions generated by the MASP query which comprises of select clause, predicate and data source collection. In the predicate for block, we use the equality of the form "attribute$\neq$ value".

5. Itemset: It corresponds to a set or one or more items.

6. k-itemset: Any itemset of length k.

7. Support: It is defined as the fraction of transactions that contain the itemset.

8. Confidence: Conditional probability that a transaction containing X will also contain Y for the transaction $X \Rightarrow Y$.

9. Frequent Itemsets: Given a transaction database, all itemsets that frequently occur i.e. whose support is greater than or equal to the minimum support threshold.

10. Pattern Tree: Binary Tree depicting the association rules/patterns obtained in MASP.

11. Apriori-TID: Algorithm that is a variant of Apriori algorithm such that the whole transactional database is not searched after the first iteration. Instead, we create a subset of the database from the previous iteration to compute the support value for the itemsets.

Chapter 2
Background

2.1 Most Associated Sequential Pattern (MASP)

Most Associated Sequential Pattern (MASP) is a name given to the variant of the association rule mining algorithm [7]. This approach is used to find the most associated sequential patterns and also generate the datasets that contain the transactions. These transactions can be further mined to find the interesting patterns. *Most Associated Sequential Pattern* is different from traditional association rule mining algorithms which deals with generating frequent patterns along with the timestamp. On the other hand, MASP patterns are sequential and they do not have a timestamp associated with it.

MASP construction involves searching for an item I_{max} that satisfies two conditions:

1. Minimum support
2. Association strength threshold

For example, initially the MASP set is empty. We find an item V_1 whose frequency is highest among the group and then check if its frequency is greater than or equal to the minimum association strength threshold. If it passes the above condition, then the item is added to the MASP set. As a result, the parent block is segregated into two parts: *block* which contains V_1 and *counter-block* containing $\overline{V_1}$. The same procedure is now used for the counter-block. This process is repeated until no more data is remaining or no item exists that meets the threshold value. Figure 2.1 shows how the blocks and the counter-blocks are created. This block created at level 1 is formed from the MASP query. The MASP query comprises of select clause, predicate and the data source from which data is selected.

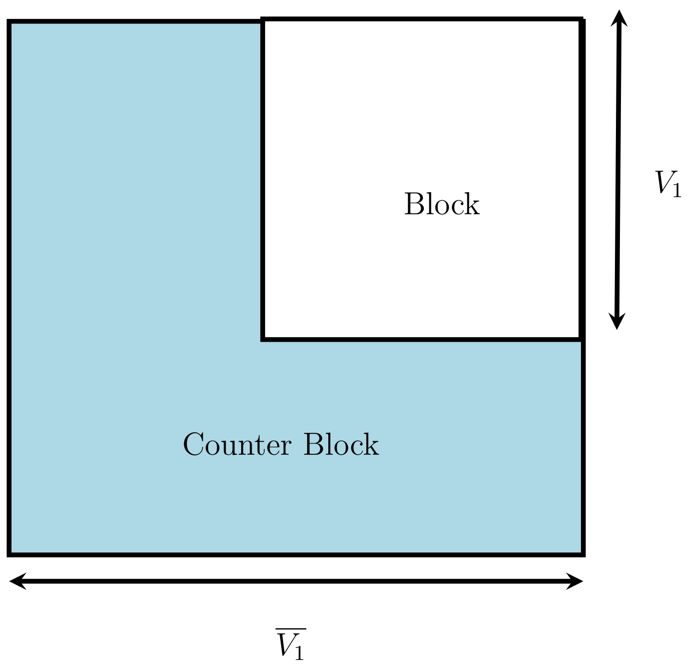


Figure 2.1: MASP Block at level 1, $MASP_1 = \{V_1\}$

2.2 Multi-Threading

Multi-threading is one of the approaches in *Parallel Computing*. Multi-threading is a form of multi-core programming which uses multiple cores on the same processor. A distributed processing environment where execution is performed on separate physical nodes, is beyond the scope of this research. Rather, our application is executed on a single physical machine.

Multi-threading is a mechanism wherein a program has several threads in execution. A thread is defined as an elementary unit of program execution. Several threads execute in parallel to increase the concurrency of the application. These threads share the resources and belong to the same process within which they execute. In a single core system, multi-threading internally uses context switching to switch the processor among several threads. In short, it is similar to multi-tasking. On the contrary, a multi-core system truly provides concurrency in multiple thread execution.

There are several issues involved while designing the algorithm to execute in parallel using the multi-threaded approach. Initially, we need to divide the various steps of our algorithm such that all the activities are divided equally among all threads. Although, we always aim to gain maximum concurrency but sometimes sequential execution is required due to the algorithm design. We need to keep a balance while segregating the tasks among different threads. After task division, we need to split the data appropriately with reference to data dependencies. Multi-threaded implementation requires thorough testing and debugging of the source code. We need to maintain the correctness and accuracy of the designed algorithm while controlling access to shared data.

There are several benefits of a multi-threaded program. It allows for a greater degree of control over the program's execution by separating the control and execution threads. Additionally, it also makes the application scalable by increasing concurrency and better resource utilization.

2.2.1 Conversion of Sequential to Multi-Threaded Code

Figure 2.2 and 2.3 illustrate the CPU performance while running a multi-threaded code over a sequential code execution. Figure 2.3 shows that all of the 8 logical processors are being utilized to their maximum capacity. On the contrary, Figure 2.2 illustrates sub-optimal utilization of the logical processors.

1. Identify the true independent computations: The instructions that are to be executed in parallel need to run independent of one another. There are instances where a sequential code is desirable since dependencies exists in the execution of instructions such as in loop. Consider an illustration of house construction. Different workers like plumber, carpenter etc. are involved in performing different operations. Here, most of the workers are doing independent tasks but the task performed by one affects the task performed by another.

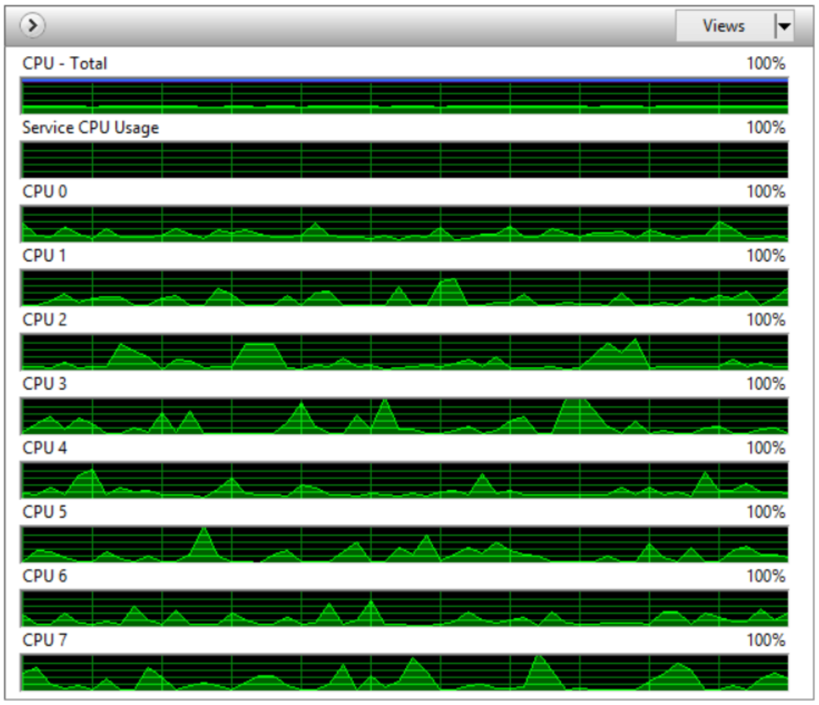


Figure 2.2: CPU Usage for Sequential Execution

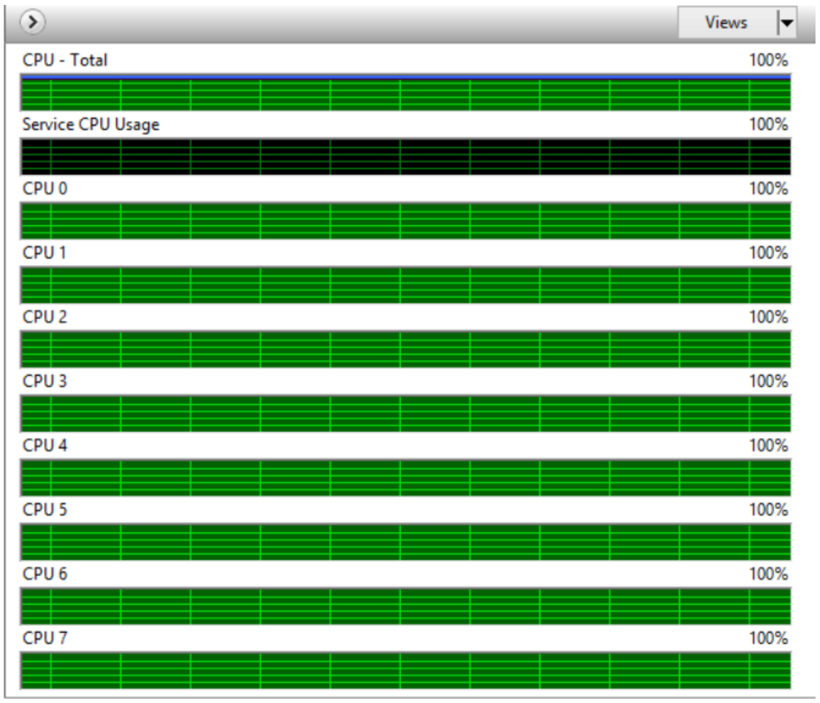


Figure 2.3: CPU Usage for Multi-Threaded Execution

2. Concurrency must be implemented at the highest possible level: We can use either top-down or bottom-up approach to implement threading. We aim to achieve coarse-grained solution where we have lower overheads and better scalability.

3. Do not assume an ordering of execution: In sequential code, it is very trivial to find the ordering in instruction execution. On the contrary, thread order execution is non-deterministic.

4. Choice between Task Parallel Library (TPL) and Threading: Task Parallel Library has the advantage of freeing the user from handling the threads internally. TPL can be used for almost all of the asynchronous needs without worrying too much about managing thread life-cycle. `ThreadPool` is useful when we want to have a greater control in thread creation, management and thread destruction.

   The thread creation involves huge overhead such as context switching etc. *Tasks* can dynamically decide to create different threads of execution. Tasks internally use `ThreadPool` for thread life-cycle management.

   Tasks always guarantee that the application will have maximum performance when run on any given system.

5. Parallel Add-ins: Task Parallel Library contains parallel extensions where the same operation runs concurrently on different elements of the source collection. .NET contains two constructs to achieve data parallelism:

   - Parallel.For
   - Parallel.ForEach

Sequential version

```
foreach (var input in inputCollection)
{
DoWork(input);
}
```

Parallel equivalent

```
Parallel.ForEach(inputCollection, item => DoWork(input));
```

TPL and the parallel extensions enable the users to levarage the maximum hardware capacity.

6. Use Thread-Local variables; lock the data if required: The parallel execution of the code should yield correct output. The amount of synchronization should be kept low else it will increase the overheads in thread execution. The sequential code might contain collections that are populated with the computed data in some loop. We have two choices; either we can apply `lock` to the data variable or else we can use *Concurrent Collections.* Concurrent collections are thread-safe. Details have been described in Section 4.2.3.1.

2.3 Apriori-TID

Association Rule Mining is a method for finding interesting patterns in large databases. As an illustration, the association rule $\{Beer, Bread\} \Rightarrow \{Milk\}$ implies that if a customer purchases beer and bread together, then he/she is more likely to buy milk. Such patterns can be used for decision-making about marketing and promotional pricing etc.

There are two tasks to be performed in mining association rules on a database:

1. Find all Frequent Patterns: *Frequent Itemsets* summarizes datasets. These are the combination of items that frequently occur together. Given a transactional database D and minimum support S_{min}, we find all frequent itemsets as well as the frequency of each set in the collection.

2. Find all Association Rules: We generate high confidence rules from the frequent itemsets.

There are several algorithms for the implementation of association rule mining such as Apriori, FP-Growth, and Apriori-TID etc. Our application already contains the implementation of FP-Growth algorithm. We have implemented Apriori-TID algorithm to mine association rules from the transactional database.

2.3.1 Frequent Itemset Generation

For n items in the transactional database D, there are 2^n possible itemsets that can be generated. Brute force method of generating frequent itemsets requires $O(MT)$, where M is number of candidate itemsets generated and T is the number of transactions in database D. This computation is very expensive since $M = 2^n$; which is exponential. Therefore, we have reduced the number of candidates M generated by using pruning technique as illustrated in Figure 2.4. Apriori-TID reduces the number of transactions T by using a subset of the previous database used at every step of the itemset generation. Additionally, we have used `HashMap` and `LinkedList` data structures for storing the candidates as well as transactions that reduces the number of comparisons at each step.

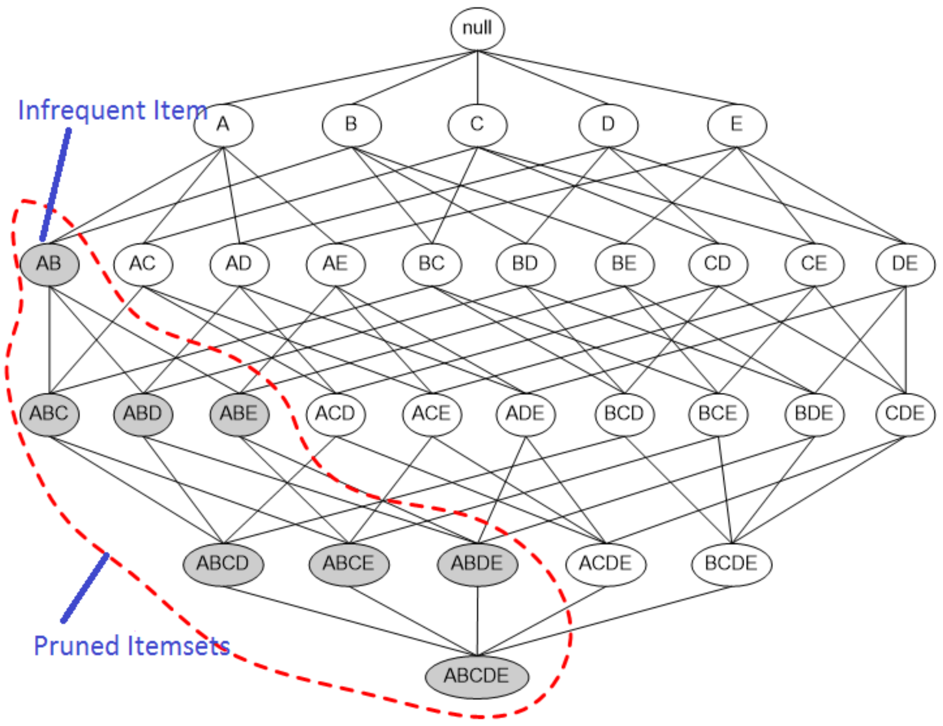


Figure 2.4: Apriori TID itemset pruning [1]

1. Reduce the number of candidates: The anti-monotone property of support mentions that the support of an itemset is always less than the support of its subset. This anti-monotone property is used extensively in Apriori-TID. The property can be illustrated by an example such as Support(Milk) $\geq$ Support(Beer,Milk), where $\{Milk\} \subseteq \{Beer, Milk\}$. We are using `HashSet` to store the transaction Id's where an item occurs and store the items as a `KeyValuePair` of `ItemStruct` consisting of the `Item-id` as `Key` and `Value` as count of transactions and `HashSet` of transactions. The details of the data structures used is elaborated in Section 4.2.3.1.

2. Reduce the number of transactions: The original database is *not* used for computing the support of an item after the first pass. Apriori-TID creates and uses a data structure D' at every step of candidate itemset generation. D' is a `HashSet` of all the transactions that contains the frequent itemset. Therefore, support is calculated just by a single pass over the dataset compared to multiple passes as in Apriori algorithm.

2.3.2 Association Rule Generation

Total number of possible rules for 2^n itemsets using brute-force approach is $3^n - 2^{n-1} + 1$. Confidence of rules generated from the same itemsets possesses anti-monotone property such that $I\{A, B, C, D\} : Confidence(ABC \rightarrow D) \geq Confidence(AB \rightarrow CD)$. Candidate rules are generated by merging the two rules that contain the same prefix in the rule consequent. Figure 2.5 shows how the rules are pruned if the parent rule is infrequent then all of its subsets are pruned.

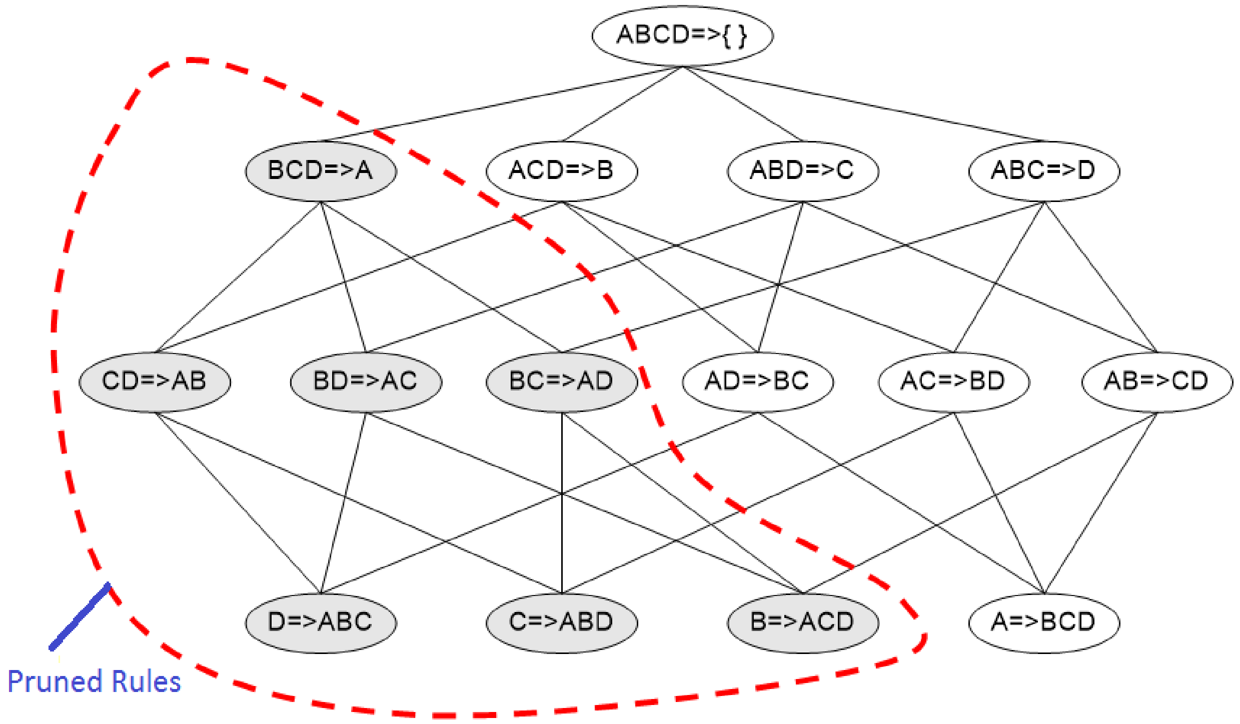


Figure 2.5: Apriori TID Rules pruning [1]

Chapter 3
Related Work

We currently have a large number of algorithms and techniques available that can be used to mine association rules. It is not feasible for a decision maker to analyze all possible patterns generated. Hence, a modular decision support system is needed to enable policymakers for taking timely decisions accurately. There are two measures *Support* and *Confidence* that are widely used in association rule mining [1, 7]. If we want to mine huge volumes of data efficiently, then it might take exponential amount of computing resources (heuristics) [21]. Additionally, it requires lot of computation power along with time and memory space in generating all the association rules. Some of the techniques proposed for such purposes include Intelligent Data Distribution Algorithm, and Hybrid Distribution Algorithm that gives an efficient and scalable solution to the problem [9].

However, before using such parallel approaches, we need to tackle several challenges such as load balancing, data dependencies, data and task segregation, and communication cost [16]. Multi-threading is one of the efficient method that can be implemented to solve these challenges [17]. There can be data dependency requirements to partition data among different processors. The load among different cores needs to be evenly balanced. A classifier to predict the fastest Association Rule Mining algorithm improves the accuracy of the results to 80% with less overhead [11]. All these methods improve the performance by distributing the tasks and data to multiple CPU processors.

Association rule mining algorithms such as Apriori and Apriori-TID are used to discover meaningful relationships/patterns between items in a large transactional database [2]. These algorithms are known to outperform previously used algorithms such as AIS and SETM.

In order to provide scalability to the existing association rule mining algorithms and improve the response time, parallelism is a desired approach. The parallel design considers three main components namely: hardware platform, type of parallelism, and load-balancing technique [22].

There has been relatively less research done to incorporate parallelism in the execution of association rule mining. There are three different types of parallelization approaches for association rule mining algorithms [23]. The first approach is *Count Distribution* algorithm for Apriori rule generation. In this approach, local database partition is used to fetch the partial support for candidate itemsets. After each iteration, global support is calculated from the partial computation of support values. The second approach is *Data Distribution* algorithm wherein candidate itemsets are partitioned into disjoint sets and given to different processors to generate global support values. However, this approach has huge communication overheads. The third method is *Candidate Distribution* where each processor gets partitioned candidated and work independently on their respective replicas of database. The last approach is *Count Distribution* which has better performance compared to above three mentioned approaches. In the *Count Distribution* approach, candidate itemsets are generated in parallel and stored in a hash data structure to speed up performance.

Three types of thread operation models have been proposed [16].The first model *Dispatcher-Worker Thread Model* consists of one dispatcher thread and multiple worker threads. Each worker thread works in parallel with other worker threads and all requests are managed by the dispatcher thread. The second model *Peer to Peer Model* where no dispatcher-worker relationships exist. All are worker threads working on separate requests in parallel. The third kind of model is *Pipeline Model* which represents producer-consumer relationship. In the pipeline model, all threads are organized in the form of a pipeline and the requests are processed in order. The peer-to-peer model has been proved to be a better model compared to rest of the other two models.

Chapter 4
Method

This thesis aims at enhancing the present methodology "Most Associated Sequential Pattern" (MASP) for data mining of interesting traffic accident patterns. There are various objectives that are targeted at in this work:

1. Multi-threaded implementation of MASP
2. Pattern tree visualization
3. Multi-threaded implementation of FP-Growth algorithm
4. Apriori-TID implementation

Each of the above enumerated objectives are implemented after careful consideration of the present system in terms of memory and time, as well as choosing efficient data structures. The initial step is data pre-processing that prepares the input data for further operations.

4.1 Data Pre-Processing

Association Rule Mining involves the extraction of interesting patterns in large databases. However, it uses discrete data as an input. This implies that we need to transform the continuous data to discrete form using discretization or binning method. This initial step is most important since we need to make our input data suitable for pattern mining. Not only this, we must encode the data in a form that can effectively improve the performance of data-mining process as well as help in reducing the memory utilization.

Binning or quantization technique helps in reducing the errors in the observed data. Using this method, we replace the value that falls within a certain interval by the central value of that interval.

The designed application uses the above described discretization method. The user is able to import the raw data and based on the selections made, the application is able to discretize the input data. There are three options provided to set the binning type:

1. Default: No binning is performed on the attribute.
2. Fixed: User is able to provide fixed bin size.
3. Bin Range: User is able to provide range of values for discretizing an attribute's value.

As an illustration, if the user selects the bin range as $\{0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$ and the attribute's values are $\{42.5, 55, 5, 18.6, 22.7, 78.7\}$. Then, after discretization, each of the attribute's value is replaced by the characteristic value of the interval. Hence, we get the discretized value for the attribute as $\{40, 50, 10, 20, 70\}$. Figure 4.1 shows the entire process in more detail.

4.2 Multi-Threaded Implementation of MASP

Of the numerous concurrent programming API's available in C#, we are using the *Task Parallel Library* and *Threads* to execute our application in parallel. We have used the `Threading` namespace which is a part of `System`. We are also using concurrent collections provided by the C# Task Parallel Library.

4.2.1 Multi-Threaded Conversion

The existing application contains the method `Convert` that performs several operations on the raw input data that the user imports into the application. Initially, the raw data is discretized into *Binned Data*. Then *Item Lookup Table* is created and finally data table *Transactional Data* is built.

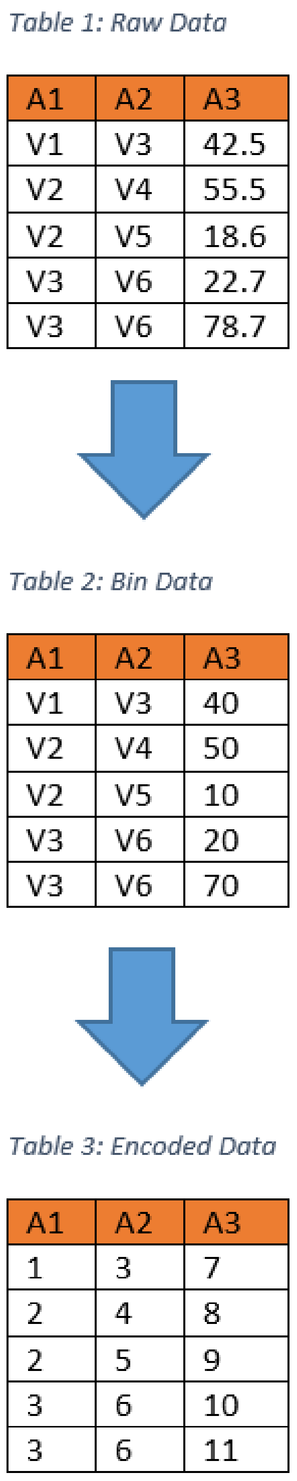


Table 1: Raw Data

A1	A2	A3
V1	V3	42.5
V2	V4	55.5
V2	V5	18.6
V3	V6	22.7
V3	V6	78.7

Table 2: Bin Data

A1	A2	A3
V1	V3	40
V2	V4	50
V2	V5	10
V3	V6	20
V3	V6	70

Table 3: Encoded Data

A1	A2	A3
1	3	7
2	4	8
2	5	9
3	6	10
3	6	11

Figure 4.1: Process of Discretization

The entire conversion method is executed once for any input dataset. Due to the complexity of the conversion algorithm and the number of operations executed on each row, the execution time is substantially high. We aim to reduce the run time by executing the operations in parallel using multi-threading. In order to compare the performance of the sequential and parallel execution of the method, we created a new method `Parallel-Convert` that contains the multi-threaded code and a new button `Parallel-Run`.

We started by parallelizing the section of code that took maximum amount of execution time. In the `Convert` method, the code section that creates the Transactional Database took the longest execution time. The execution time for the remainder of the code sections within the `Convert` method is insignificant and parallelizing it would result in additional overheads. As a result, the code section for creating the Transactional Database is modified by replacing the sequential version with the parallel version. We have used *C# Task Parallel Library* to run the code in parallel across various system's computing cores. The details of the code section and the improved performance are described in Section 4.2.3.1.

4.2.2 Multi-Threaded MASP

Once the data conversion is done, the next stage is to detect all of the *Most Associated Sequential Patterns* in the encoded data. A frequent item-set is considered to be the most associated when it meets the minimum association threshold level $[0, 1]$.

The algorithm execution time is greatly increased by few code sections that takes more execution time to complete. We aimed at parallelizing the code using Thread Class. C# Task Parallel Library cannot be used directly for few of the code sections. This is because there are few limitation in the .NET Framework support with regards to parallel `while` method.

Our overall aim was to make the code run in negligible time without making substantial modifications to it. Therefore, both `Thread` delegate as well as `Parallel.ForEach` method are used to gain maximum performance improvement. Moreover, we have used thread-safe concurrent data-structures whose details are mentioned in Section 4.2.3.1.

An export functionality has been added to export the MASP rules that are generated in this step. The advantage of exporting the rules to a separate excel is to be able to query the database effectively and efficiently. This code section is integrated into the *Export* panel of the application.

4.2.3 Implementation Details

C# Task Parallel Library is based upon the concept of an asynchronous operation (task). At a higher level of abstraction, a *task* refers to a thread or a threadpool work item. Task parallelism is achieved where one or more independent tasks run concurrently. `Tasks` are queued to the `ThreadPool` where we can adjust the number of threads and provides load balancing for maximizing throughput [13].

There are two benefits of using tasks:

1. More efficient and more scalable.
2. More control than thread or work item.

As a result, in .NET Framework, Task Parallel Library is the preferred API for implementing multi-threaded, parallel, and asynchronous codes.

Task Parallel Library (TPL) uses `System.Threading.Tasks.Parallel` class to support data parallelism. *Data Parallelism* is implemented by using `Parallel.For` and `Parallel.Foreach` loops. These methods are parallel implementations of `for` and `foreach` loops. Here, the data is partitioned into sub-parts that execute concurrently. In addition to this, `PLINQ` is also used that is a parallel version of `LINQ` queries.

`Parallel.For` and `Parallel.Foreach` methods provide various functionalities such as controlling the degree of concurrency, monitor thread-state, and maintaining thread-local states etc.

4.2.3.1 Data Structure

`System.Collections.Concurrent` namespace contains many thread-safe collections that are used in place of their corresponding types in `System.Collections` namespace. These collections are accessible by multiple threads concurrently. Concurrent classes include

- `BlockingCollection`$< T >$
- `ConcurrentBag`$< T >$
- `ConcurrentDictionary`$< Tkey, TValue >$
- `ConcurrentQueue`$< T >$

Our multi-threading implementation involves the use of these concurrent collections based on the data requirements.

Table 4.1 shows the overview of the concurrent collections used in our implementation and their descriptions [12].

Table 4.1: Concurrent Collections

Class	Description
`BlockingCollection<T>`	Provides blocking and bounding capabilities for thread-safe collections that implement `IProducerConsumerCollections<T>`
`ConcurrentBag<T>`	Represents a thread-safe, unordered collection of objects.
`ConcurrentDictionary<Tkey,Tvalue>`	Represents a thread-safe collection of key/value pairs that can be accessed by multiple threads concurrently.
`ConcurrentQueue<T>`	Represents a thread-safe first in-first out (FIFO) collection.

4.2.3.2 Details

Below code sections shows the conversion of sequential code into multi-threaded equivalent to improve performance.

1. Sequential and multi-threaded `Conversion` Method

```
//sequential implementation
foreach (DataRow row in BinData.Rows)
{
  string[] itemarray = new string[AttributeBins.Count()];
  int colind = 0;

  foreach (object item in row.ItemArray)
  {
    itemarray[colind] = ItemLookUp.Where(o => o.Value.
       attributeName == BinData.Columns[colind].ColumnName && o.
       Value.valueName == item.ToString()).FirstOrDefault().Key;
    colind++;
  }

  TransactionalData.Rows.Add(itemarray);
}

//parallel implementation
var DataTableBag = new System.Collections.Concurrent.
   ConcurrentBag<List<string[]>>(); //Use a thread-safe
   datastructure in the concurrent collection. The CocurrentBag
   contains unordered data.

Parallel.ForEach<DataRow, List<string[]>>(BinData.AsEnumerable(),
    () => new List<string[]>(),
(row, loop, transactionData) =>
{
  string[] itemarray = new string[AttributeBins.Count()];
  int colind = 0;

  foreach (object item in row.ItemArray)
  {
    itemarray[colind] = ItemLookUp.Where(o => o.Value.
       attributeName == BinData.Columns[colind].ColumnName && o.
       Value.valueName == item.ToString()).FirstOrDefault().Key;
    colind++;
  }

  transactionData.Add(itemarray);
  return transactionData;
},
(finalresult) =>
{
  DataTableBag.Add(finalresult); //adding the data to the bag
      after each thread execution
});
```

2. Sequential and multi-threaded MASP block creation

```
//sequential implementation
while (CandidateBlock.Count != 0)
{
...
...
}

//multi-threaded implementation
int threadCount = Environment.ProcessorCount; //corresponding to
    8 logical cores
int numThreads = 0; //keeps track of the number of threads
bool finished = false;

while (threadCount > 0)
{
  new Thread(delegate() //thread delegate method
  {

    #region Processing
    Interlocked.Increment(ref numThreads); //locks the data
        variable to increment the count
    ...
    ...
    mineBlock.Enqueue(SelWher);
    Interlocked.Decrement(ref numThreads);  //locks the data
        variable to decrement the count
```

```
            Thread.Sleep(2); //to synchronize thread execution
            #endregion Processing

            if (finished) //checks if the thread needs to be destroyed
            { break; }
        }).Start();
        threadCount--;
    }
```

The optimal number of threads that gave maximum performance for the creation of MASP blocks is equal to the number of logical processors in the system. For our system configuration, we have 8 logical cores.

Figure 4.2 shows the performance of MASP block create method with the variation of number of threads spawned in the implementation. As the number of threads increase past the number of logical cores, additional time is consumed by context switching which increases the total execution time.

Table 4.2 shows the speedup obtained by using the multi-threaded version over the sequential method. Amdahl's law is used to obtain the speedup.
According to Amdahl's law [15]:
$Speedup = t_p/t_s$,
where t_p = parallel execution time in seconds
t_s = sequential execution time in seconds

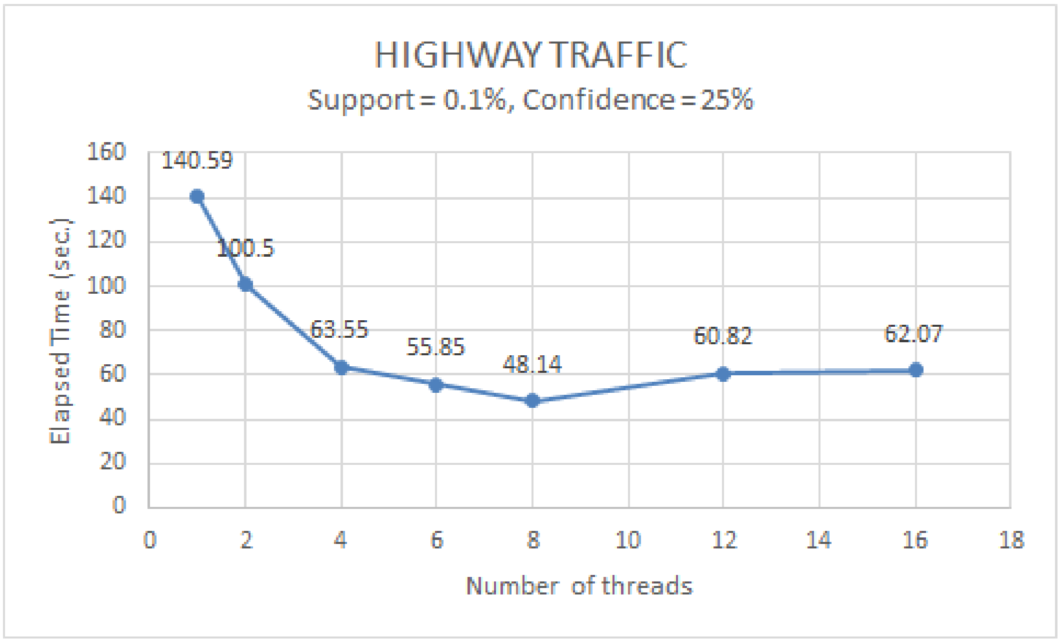


Figure 4.2: MASP Block create method variation with thread count

Table 4.2: Speedup and Execution Time Comparison of Sequential and Multi-Threaded Methods for 8000 records (Highway Traffic Dataset)

Method	Seq Time(sec)	Multi-Threaded Time(sec)	Speedup Obtained
Convert	4.09	1.74	2.351
Create Blocks	6.84	2.6	2.691
FP-Growth	4.09	9.96	2.435

4.3 Multi-Threaded Apriori-TID

Figure 4.3 shows the high-level design of the Apriori-TID algorithm implementation.

4.3.1 Implementation Details

The multi-threaded implementation of Apriori-TID algorithm involves running the code sections in parallel. We have used `Concurrent Collections` available in C# Task Parallel Library that are thread-safe collections. Concurrent collections are used to store *Candidate Itemsets* generation and *Rule Generation* step.

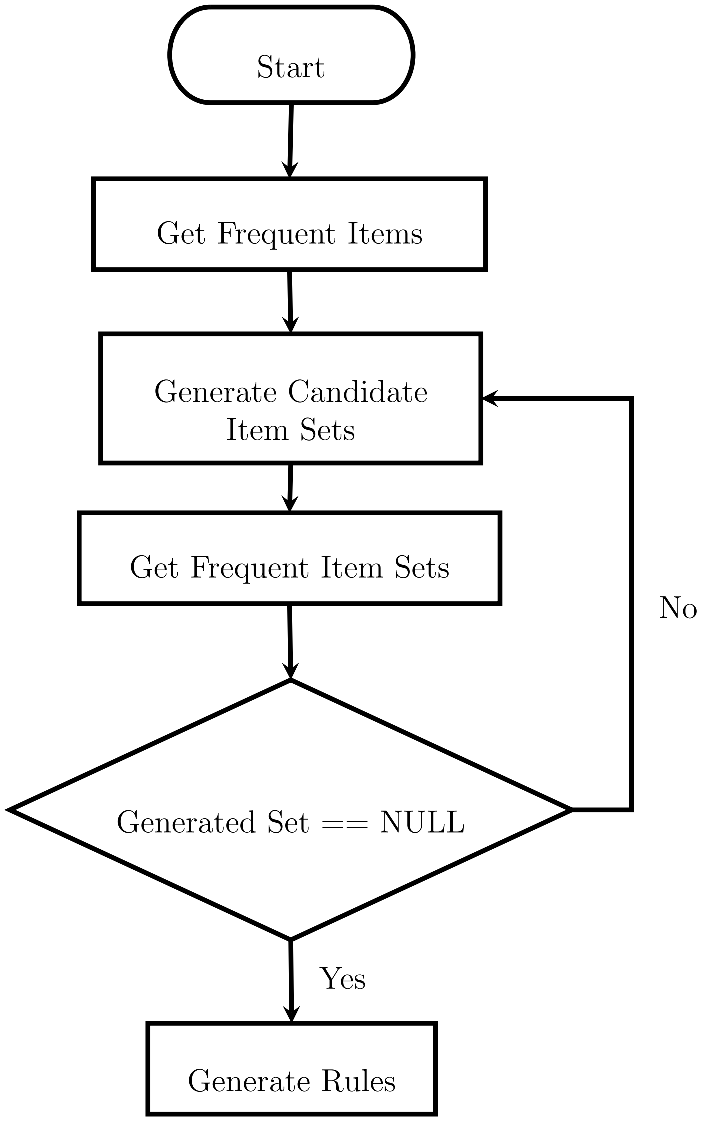


Figure 4.3: Apriori TID High-Level Design

The implementation of Apriori-TID is a slightly different version of regular Apriori-TID algorithm [2]. The algorithm is slightly modified in the construction of the data-structure $\overline{C_k}$ which comprises of $< TID, X_k >$, where X_k corresponds to the k-itemset in the transaction TID. Instead, we assign each candidate itemset with an identifier that uniquely identifies the item and is also used as an indexer. The structure of the itemset comprises of `ItemStruct` class that contains a member variable `Count` which is used for calculating support of each item, the `HashSet` comprise of the transaction identifiers comprising of the itemsets as well as a boolean variable that tracks whether the item is frequent or not. `ItemStruct` eliminates the need of reconstructing the $\overline{C_k}$ in every iteration of the algorithm. We just need to do intersect operation for the Hashset of transactions of k itemsets to create a new HashSet$< T >$ data member of `ItemStruct` class.

```
public void IntersectWith(
IEnumerable<T> otherCollection
)
```

The `IntersectWith` method of `HashSet`$< T >$ is $O(N)$ operation if `otherCollection` has the same equality comparer as the current `HashSet`$< T >$ object. Otherwise, this method is O$(N + M)$ where N is the count of elements in current `HashSet`$< T >$ object and M is number of elements in `otherCollection`.

This reduces the space complexity for creating and storing a new data structure $\overline{C_k}$. This makes our algorithm more efficient with respect to both space and time complexity.

4.3.2 Multi-threaded Apriori-TID

The sequential implementation of Apriori-TID has been multi-threaded to improve the performance. The code section reveals the conversion of sequential to multi-threaded Apriori-TID algorithm. Here, `for` loop has been modified to `Parallel.For` which improves the performance of loop execution considerably.

```
**Data**: Transactional Database $D$
**Result**: Rules generated $RuleSet$
begin
    $ActiveSet \longleftarrow newLinkedList < KeyValuePair < List < int >, ItemStruct >> ();$;
    add all items in transaction dataset $D$ to the $ActiveSet$;
    prune the 1-itemset using minimum support threshold $S_{min}$;
    while $ActiveSet$ is not null do
        use $I_{k-1}$ join $I_{k-1}$ to generate a set of unpruned 1-itemset;
        compute the intersection of Hashset$< T >$ of $I_{k-1}$ itemsets;
        compute the Count $C$ of Hashset$< T >$;
        if $C \geq S_{min}$ then
            add the item to $ActiveSet$ ;
        else
            go back to the beginning of current section;
        end
    end
end
```

Algorithm 1: Apriori-TID implementation

The below mentioned code section is used to generate frequent k-itemsets:

```
//sequential code
for(int i=0;i<wsC;i++) //used for frequent k-itemset generation
{
  LinkedListNode<KeyValuePair<List<int>, ItemStruct>> iNode =
      workingSet.First;
  ...
}


//multi-threaded code
Parallel.For(0, wsC, i => //for k-itemset generation
{
  LinkedListNode<KeyValuePair<List<int>, ItemStruct>> iNode =
      workingSet.First;
  ...
}
```

4.3.2.1 Data Structure

The `ItemStruct` class is used to store the frequent k-itemsets generated from the Apriori-TID algorithm.

`Rule` class is used to store the antecedent, consequent, support, and confidence of each generated rule.

```
public class Rule : IComparable<Rule>
{
    string combination; //contains antecedent
    string remaining; //contains consequent
    double confidence;
    double support;
}

public struct ItemStruct
{
   public double Count { get; set; }  //contains the number of
      transactions in which the item occurs; used for computing
      support
   public HashSet<int> TransactID { get; set; } //Hashset comprising
      of all transaction-id(s)
   public bool Active { get; set; } //tracks whether the item is
      frequent or not
}
```

4.4 Multi-Threaded FP-Growth Tree

This section deals with the generation of association rules from K-Patterns from FP-Growth algorithm. The rules are formed by combining the antecedent and the consequent part of the patterns. This algorithm takes an enormous amount of time to execute. Moreover, there are several dependencies in terms of the data-structure being used in the implementation. Our aim was to take care of the implementation details while trying to find the sections that consume more execution time. We have used `C# Task Parallel Library` for improving the performance of this method. The code sections that are parallelized include:

1. construction of k-pattern tree
2. generation of rules from the frequent patterns

4.4.1 Implementation Details

```
//sequential version
foreach (PatternTreeNodes pk_1 in k_1_patternTree) //this method is
    used for generating the k-patterns in the application
{
  foreach (ChildPositions ck_1 in pk_1.ChildPositions)
  {
  ...
  }
}
//multi-threaded implementation
System.Collections.Concurrent.ConcurrentBag<PatternTreeNodes>
    TreeNodesCollection = new System.Collections.Concurrent.
    ConcurrentBag<PatternTreeNodes>(); //Use the thread-safe concurrent
     collection to store the computational results obtained
Parallel.ForEach(k_1_patternTree, pk_1 =>
{
  foreach (ChildPositions ck_1 in pk_1.ChildPositions)
  {
  ...
  }
}
```

4.5 Visualization of MASP Pattern Tree

The output of the MASP block creation step is a `Mined Database` from which we create a pattern tree. A *Pattern Tree* enables a visual interpretation of the mined data in order to get quantitative and qualitative information from them.

There are several third-party API's available online that can be used to generate the pattern tree, like Google Visualization Charts. However, third party API's have external dependencies and require internet connectivity to communicate and generate graphics. We have used a Binary Tree implementation to store and draw node images for enabling visual perspective from the data.

The mined data has a special property that each node contains at most two children and the generated tree is not balanced. For a given node in a binary tree, the first child is named as the `Left Child` and the second child is referred as the `Right Child`. Additionally, the mined data contains the items either in `Block` or `Counter-Block`. If the item V_1 is present in the `Block`, we add it to the left child. If present in the `Counter-Block`, we add the item to the right child and mark it as the item's complement $\overline{V_1}$.

4.5.1 Implementation Details

Figure 4.5 shows the complete pattern tree generated from the mined dataset and Figure 4.4 represents a subsection of the pattern Tree. Additionally, we can export the MASP Tree into .xlsx format. The table can be later imported to create and display `Pattern Tree` without processing the raw input data. Table 4.3 shows the subset of the Pattern Tree data exported into Excel format. `NODE-ID` is an identifier for each node. `PARENT-ID` denotes the identifier of the parent-node for this node. `ITEMID` is the item contained in the node. `TYPE` specifies if this node is to be added to the left or right of the parent-node. `LEFT CHILD` and `RIGHT CHILD` refers to the left and the right child of the node.

```
Data: Mined DataTable $M$ that contains the set of MASP's obtained from the Section
      4.2.2
Result: Pattern Tree: $Tree$
begin
    $Tree \longleftarrow \{\}$;
    for m in $M$ do
        get the item-id itemId from m;
        perform depth-first search to find the parent-node of itemId;
        if parent-node's child $\neq$ itemId then
            insert the current node $m$ into the $Tree$;
            if m contains "=" then
                add the itemId as the $LeftChild$;
            else
                add the itemId as the $RightChild$;
            end
        else
            go back to the beginning of current section;
        end
    end
end
```

Algorithm 2: Pattern Tree Creation

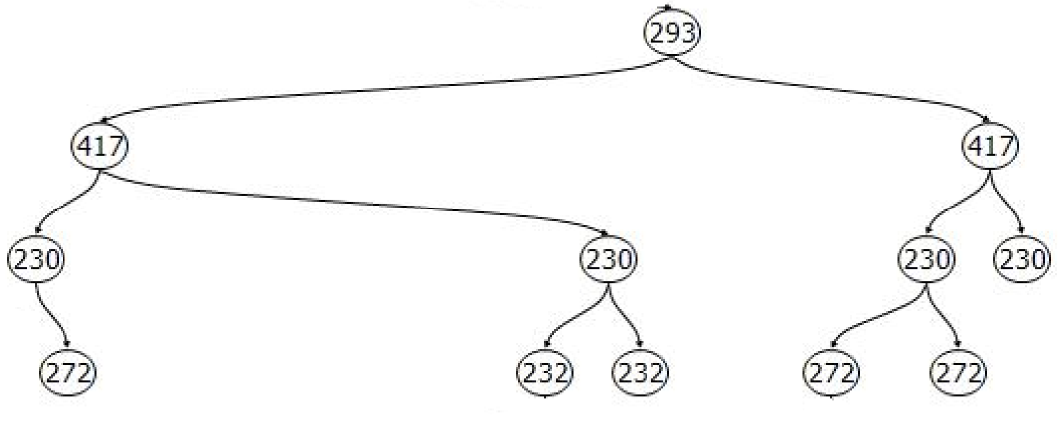


Figure 4.4: Sub-section of the MASP Tree

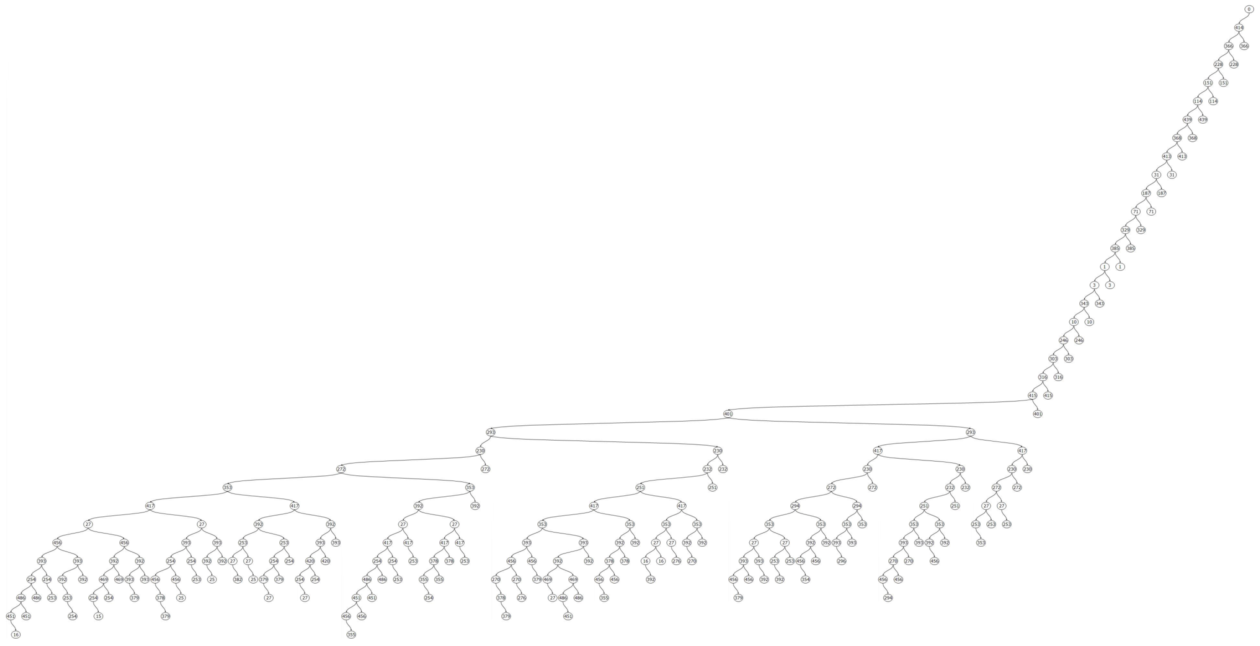


Figure 4.5: MASP Tree

Table 4.3: Pattern Tree in Table Format

NODE-ID	PARENT-ID	ITEMID	TYPE	LEFT CHILD ID	RIGHT CHILD ID
1	0	414	=	2	3
3	1	366	<>	0	0
2	1	366	=	4	5
5	2	228	<>	6	7
7	4	151	<>	0	0
6	4	151	=	8	9
9	6	114	<>	0	0
8	6	114	=	10	11
11	8	439	<>	0	0
10	8	439	=	12	13
13	10	368	<>	0	0
12	10	368	=	14	15
15	12	413	<>	0	0

In addition to generating a visual representation of the MASP tree, the implementation allows for the saving of the Pattern Tree into various image file formats such as *.gif, *.jpg, and *.png.

4.5.1.1 Data Structure

Binary Tree has been implemented for creating the MASP Pattern Tree since we can have at most two children for any node. Also, the tree is unbalanced.

Below class specifies the `BinaryTreeNode` and `BinaryTree` class structures used in the implementation:

```
public class BinaryTreeNode
{
   public int Value { get; set; }
   public int Node { get; set; }
   public int Parent { get; set; }
   public string Block { get; set; }

   // gets or sets the right node connected to this node, if any
   public BinaryTreeNode Right { get; set; }

   // gets or sets the left node connected to this node, if any
   public BinaryTreeNode Left { get; set; }
}

public class BinaryTree
{
   // the root node, it won't be seen on the graph!

   public BinaryTreeNode RootNode { get; private set; }
}
```

4.6 Complexity

1. Binary Tree:

   We have implemented *Depth First Search* (DFS) *Pre-Order* traversal for searching for the parent node of the new node.

   Time complexity for Depth First Search is $O(|N|)$, where $|N|$ is the number of nodes in the tree.
   Space complexity for Depth First Search is $O(|H|)$, where $|H|$ is the depth of the tree. This is because we have done a recursive implementation of the Depth First Search algorithm. DFS stores only the current path in order to get to the solution. Hence, the space complexity is a linear function of the depth.

   Node Insertion takes O(1) time complexity since we got the parent node by DFS search.

2. Cyclomatic Complexity and Class Coupling: Cyclomatic complexity depicts the number of decision logics in the source code [19]. If the number of decisions are more in the code, then the cyclomatic complexity is more.

   Class coupling indicates how many classes a single class/method is referencing [5].

   Table 4.4 shows the cyclomatic complexity and class coupling values obtained for each of the main methods of the application. As depicted in the Table 4.4, the cyclomatic complexity of FP-Growth algorithm is more than any other methods in the table. This is evident through the fact that FP-Growth algorithm takes more execution time than Apriori-TID whose cyclomatic complexity is less. This means that FP-Growth algorithm source code contains large number of decision paths. Binary Tree creation and display has the least cyclomatic complexity.

   The threshold for class coupling is 30 for a method and 80 for a class [6].

Table 4.4: Code Complexity Analysis

Methods	Cyclomatic Complexity	Class Coupling
Sequential Convert	32	34
Multi-threaded Convert	39	44
Sequential Create Blocks	48	26
Mult-threaded Create Blocks	54	36
Sequential FP-Growth	60	29
Multi-threaded FP-Growth	65	38
Sequential Apriori-TID	26	28
Multi-threaded Apriori-TID	30	32
Create Tree	4	10
Display Tree	2	6

Chapter 5
User Interface

The user interface for Pattern Mining is shown in Figure 5.1. The data to be mined is imported into the interface in the desired format. Then the required attributes are selected through “Select Attributes”. Once this is done, then we need to load the attributes and perform discretization if desired. Next, the minimum support and confidence values are specified in the provided textboxes. Once we select the mined type and whether to mine entire data or just the blocks, we can run the MASP algorithm either multi-threaded or sequential. To run the association rules, Apriori-TID or FP-Growth, the required type is selected from the dropdown and then all the patterns can be mined either sequentially or using multi-threaded implementation. We also have the option to create , display, import, and export the pattern tree. In addition to this, a new functionality is added that can export the MASP rules in .xlsx format.

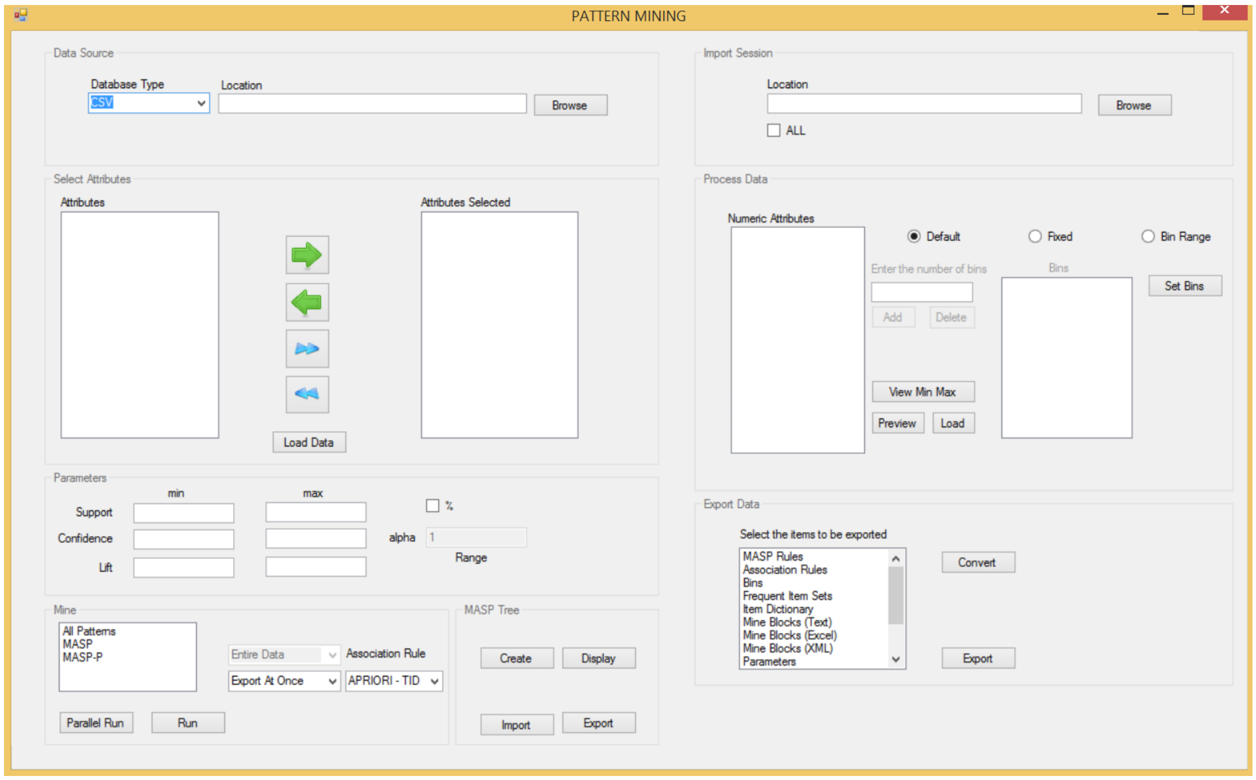


Figure 5.1: User Interface

Chapter 6
Experiment and Results

Following experiments are performed using the five datasets:

1. Compare the performance of sequential and multi-threaded MASP
2. Compare the performance of sequential and multi-threaded Apriori-TID algorithm
3. Compare the performance of sequential and multi-threaded FP-Growth algorithm
4. Compare the performance of sequential Apriori-TID and FP-Growth algorithm
5. Compare the performance of multi-threaded Apriori-TID and FP-Growth algorithm

6.1 System Environment Used

Below are the system, softwares, and dataset specifications used in the experimental runs:

Operating System:

Windows 8.1 Enterprise 64-bit

CPU:

Intel Core i7 4800 MQ

1. No. of Cores: 4
2. No. of Logical Cores: 8
3. Clock Speed: 2.7 GHz
4. Max Turbo Frequency: 3.7 GHz
5. Cache: 6 MB L3 cache

Haswell 22 nm Technology

RAM:

16.0 GB Dual-Channel DDR3 @ 798 MHz (11-11-11-28)

Graphics:

Generic PnP Monitor (1920x1080@60Hz)

Intel HD Graphics 4600

Storage:

232 GB Samsung SSD 840 EVO 250 GB mSATA (SSD)

931 GB Seagate ST1000LM014-1EJ164 (SATA)

Integrated Development Environment:

Visual Studio 2013

Softwares:

.NET 4.5

R 3.1.0 [14]

Rattle (version 3.1.1) [20]

DATASETS:

Highway Traffic [10]

BlogFeedback DataSet [4]

Diabetes Data [18]

Connect IBM [3]

Pumsb IBM [3]

6.2 Data

We have performed our experiments on five datasets. Tables 6.1, 6.2, 6.3, and 6.4 describe the characteristics of the five datasets.

Table 6.1: Highway Traffic Dataset

Highway Traffic	
Characteristics	Values
#Atrributes	38
#Transactiona	174436

Table 6.2: BlogFeedBack Dataset

Characteristics	Values
Data Set Characteristics	Multivariate
Attribute Characteristics	Integer, Real
Number of Instances	60021
Number of Attributes	281
Missing Values	N/A
Area	Social
Date Donated	5/29/2014
Number of Web Hits	4085

Table 6.3: Diabetes 130-US hospitals for years 1999-2008 Dataset

Characteristics	Values
DataSet	Multivariate
Attribute	Integer
Number of Instances	100000
Number of Attributes	55
Missing Values	Yes
Area	Life
Date Donated	5/3/2014
Number of Web Hits	90406

Table 6.4: DataSets Prepared by Roberto Bayardo from the UCI datasets and PUMSB

UCI DataSets and PUMSB			
DataSets	#Items	Avg. Length	#Transactions
pumsb	2113	74	49046
connect	129	43	67557

6.3 Experimental Setup

The results are compared for below mentioned parameters:

1. minimum support(%): 0.1,10
2. minimum confidence(%): 80, 50, 25

R data mining uses apriori implementation of association rule mining. We used R to run the rule mining for the five datasets as a traditional ARM.

6.4 Results and Discussion

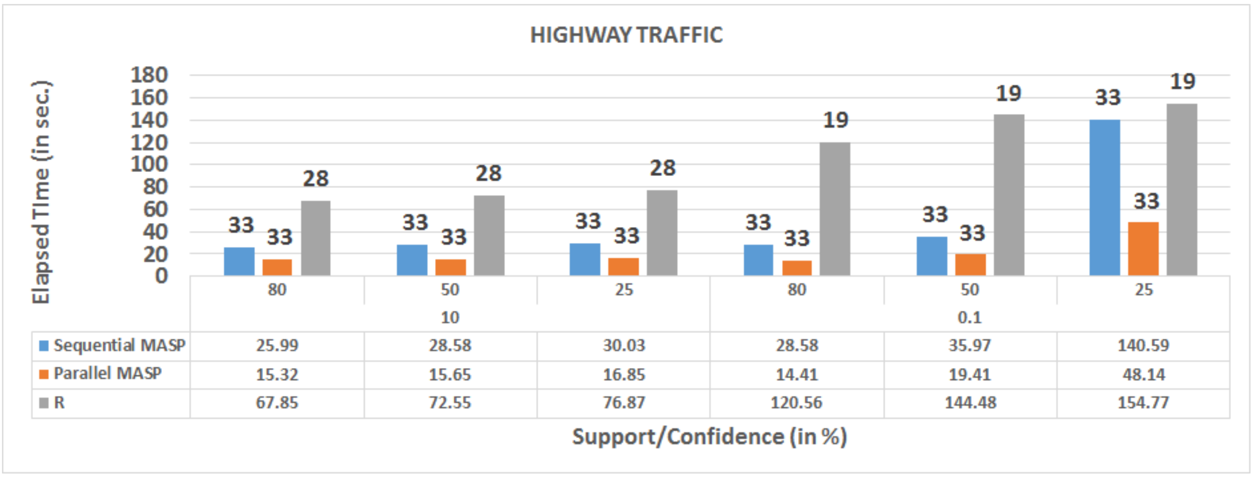
Figures 6.1, 6.2, 6.3, 6.4, 6.5 depict the comparative performance of sequential, parallel implementation of MASP, and R data mining (Traditional Association Rule Mining (T-ARM)).

Figures 6.6, 6.7, 6.8, 6.9, 6.10 represent the comparative performance of sequential, parallel implementation of Apriori-TID and FP-Growth Algorithms.

The Figure 6.1 shows the graph plots between MASP (both sequential and multi-threaded implementation) and R on Highway Traffic Dataset. The horizontal axis gives the support and confidence values (in %) and vertical axis depicts the elapsed time (in sec.). The data-labels represent the number of attributes for which the algorithm was run. Below observations are made based on the Figure 6.1:

1. Multi-threaded implementation has a better performance with respect to time efficiency over sequential implementation of MASP. We obtained an average speedup of 2.013.

2. The sequential MASP takes highest amount of time when running for Support = 0.1(%) and Confidence = 25(%). This is when we obtain the maximum speedup by multi-threading i.e. 2.92.



	80	50	25	80	50	25
	10			0.1		
■ Sequential MASP	25.99	28.58	30.03	28.58	35.97	140.59
■ Parallel MASP	15.32	15.65	16.85	14.41	19.41	48.14
■ R	67.85	72.55	76.87	120.56	144.48	154.77

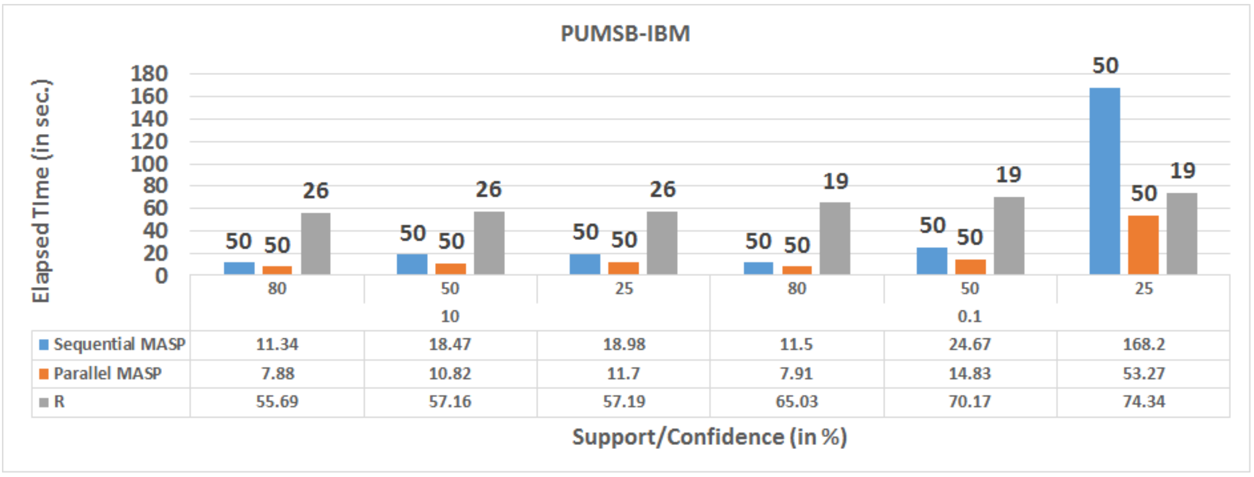
Figure 6.1: MASP vs. R performance (Highway Traffic dataset)

3. R association rule mining takes more time to run even for less number of attributes and lesser number of transactions in comparison to MASP. Minimum number of attributes for MASP is 50 and R is 19.

4. R ARM is able to mine more number of attributes (28 v/s 19) when support is 10% compared to when support is 0.1%.

The Figure 6.2 shows the graph plots between MASP (both sequential and multi-threaded implementation) and R on PUMSB-IBM Dataset. The horizontal axis gives the support and confidence values (in %) and vertical axis depicts the elapsed time (in sec.). The data-labels represent the number of attributes for which the algorithm was run. Below observations are made based on the Figure 6.2:

1. As discussed earlier, multi-threaded implementation has a better performance with respect to time efficiency over sequential implementation of MASP. We obtained an average speedup of 1.84.
2. The sequential MASP takes highest amount of time when running for Support = 0.1(%) and Confidence = 25(%). This is when we obtain the maximum speedup by multi-threading i.e. 3.16.



	10			0.1		
	80	50	25	80	50	25
🟦 Sequential MASP	11.34	18.47	18.98	11.5	24.67	168.2
🟧 Parallel MASP	7.88	10.82	11.7	7.91	14.83	53.27
⬜ R	55.69	57.16	57.19	65.03	70.17	74.34

Figure 6.2: MASP vs. R performance (PUMSB-IBM dataset)

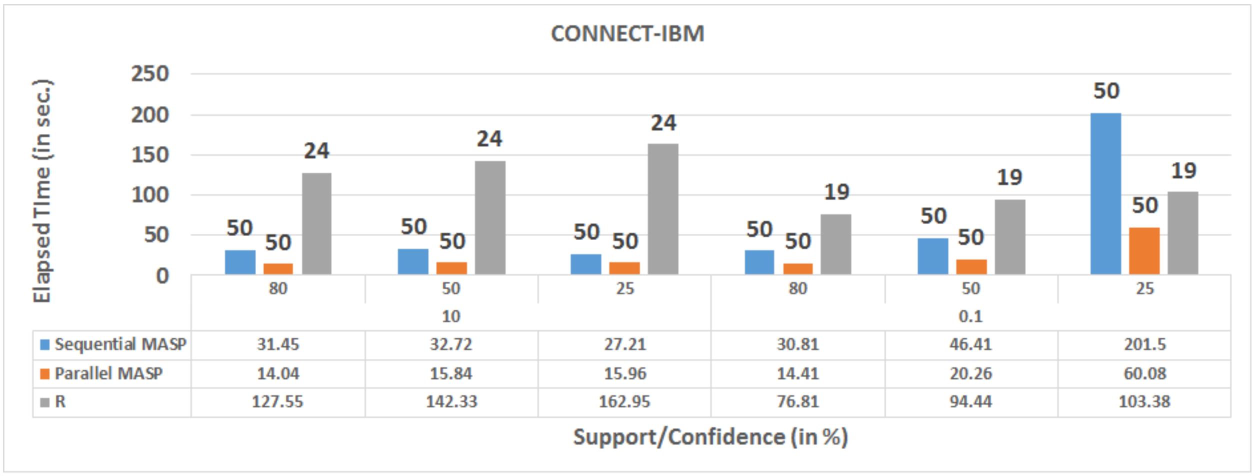
3. R association rule mining takes more time to run even for less number of attributes and lesser number of transactions in comparison to MASP. Minimum number of attributes for MASP is 50 and R is 19.

4. R ARM is able to mine more number of attributes (26 v/s 19) when support is 10% compared to when support is 0.1%.

The Figure 6.3 shows the graph plots between MASP (both sequential and multi-threaded implementation) and R on CONNECT-IBM Dataset. The horizontal axis gives the support and confidence values (in %) and vertical axis depicts the elapsed time (in sec.). The data-labels represent the number of attributes for which the algorithm was run. Below observations are made based on the Figure 6.3:

1. As discussed earlier, multi-threaded implementation has a better performance with respect to time efficiency over sequential implementation of MASP. We obtained an average speedup of 2.3.

2. The sequential MASP takes highest amount of time when running for Support = 0.1(%) and Confidence = 25(%). This is when we obtain the maximum speedup by multi-threading i.e. 3.35.



	80	50	25	80	50	25
		10			0.1	
🟦 Sequential MASP	31.45	32.72	27.21	30.81	46.41	201.5
🟧 Parallel MASP	14.04	15.84	15.96	14.41	20.26	60.08
▨ R	127.55	142.33	162.95	76.81	94.44	103.38

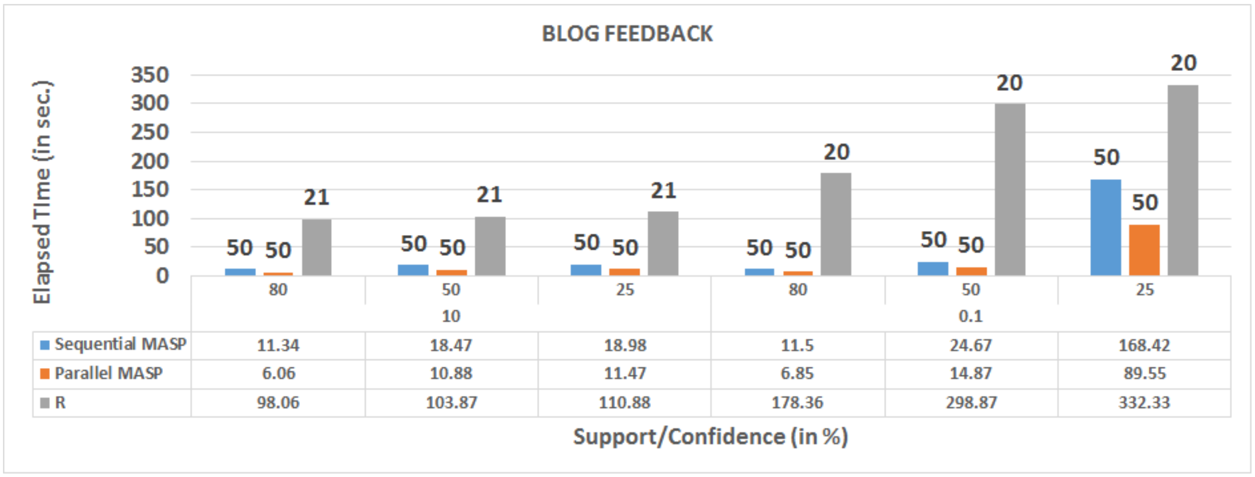
Figure 6.3: MASP vs. R performance (CONNECT-IBM dataset)

3. R association rule mining takes more time to run even for less number of attributes and lesser number of transactions in comparison to MASP. Minimum number of attributes for MASP is 50 and R is 19.

4. R ARM is able to mine more number of attributes (24 v/s 19) when support is 10% compared to when support is 0.1%.

The Figure 6.4 shows the graph plots between MASP (both sequential and multi-threaded implementation) and R on BLOG-FEEDBACK Dataset. The horizontal axis gives the support and confidence values (in %) and vertical axis depicts the elapsed time (in sec.). The data-labels represent the number of attributes for which the algorithm was run. Below observations are made based on the Figure 6.4:

1. As discussed earlier, multi-threaded implementation has a better performance with respect to time efficiency over sequential implementation of MASP. We obtained an average speedup of 1.74.
2. The sequential MASP takes highest amount of time when running for Support = 0.1(%) and Confidence = 25(%). This is when we obtain the maximum speedup by multi-threading i.e. 1.88.



	80	50	25	80	50	25
		10			0.1	
🟦 Sequential MASP	11.34	18.47	18.98	11.5	24.67	168.42
🟧 Parallel MASP	6.06	10.88	11.47	6.85	14.87	89.55
⬜ R	98.06	103.87	110.88	178.36	298.87	332.33

Figure 6.4: MASP vs. R performance (BLOG-FEEDBACK dataset)

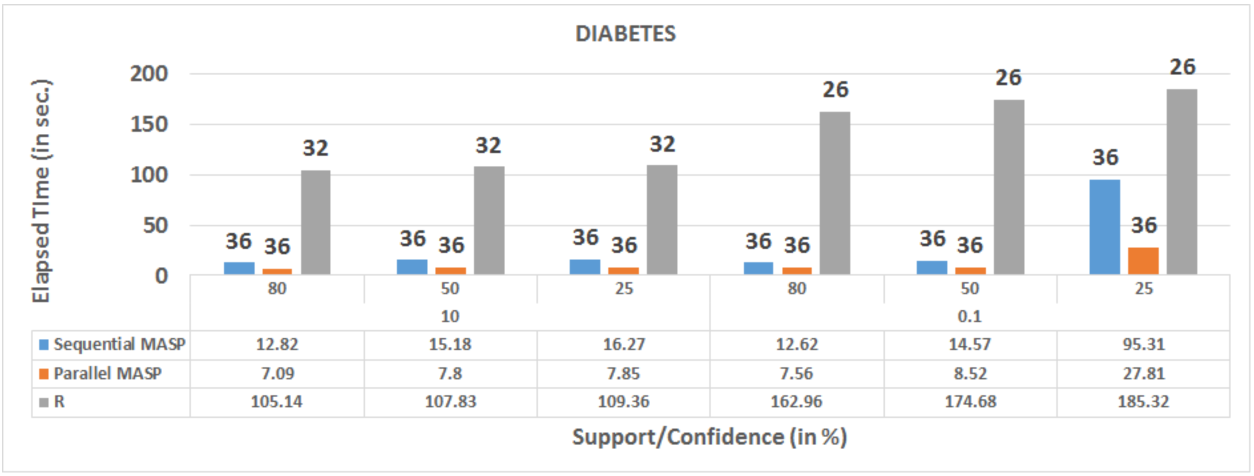
3. R association rule mining takes more time to run even for less number of attributes and lesser number of transactions in comparison to MASP. Minimum number of attributes for MASP is 50 and R is 20.

4. R ARM is able to mine more number of attributes (21 v/s 20) when support is 10% compared to when support is 0.1%.

The Figure 6.5 shows the graph plots between MASP (both sequential and multi-threaded implementation) and R on DIABETES Dataset. The horizontal axis gives the support and confidence values (in %) and vertical axis depicts the elapsed time (in sec.). The data-labels represent the number of attributes for which the algorithm was run. Below observations are made based on the Figure 6.5:

1. As discussed earlier, multi-threaded implementation has a better performance with respect to time efficiency over sequential implementation of MASP. We obtained an average speedup of 2.11.

2. The sequential MASP takes highest amount of time when running for Support = 0.1(%) and Confidence = 25(%). This is when we obtain the maximum speedup by multi-threading i.e. 3.43.



	80	50	25	80	50	25
		10			0.1	
■ Sequential MASP	12.82	15.18	16.27	12.62	14.57	95.31
■ Parallel MASP	7.09	7.8	7.85	7.56	8.52	27.81
■ R	105.14	107.83	109.36	162.96	174.68	185.32

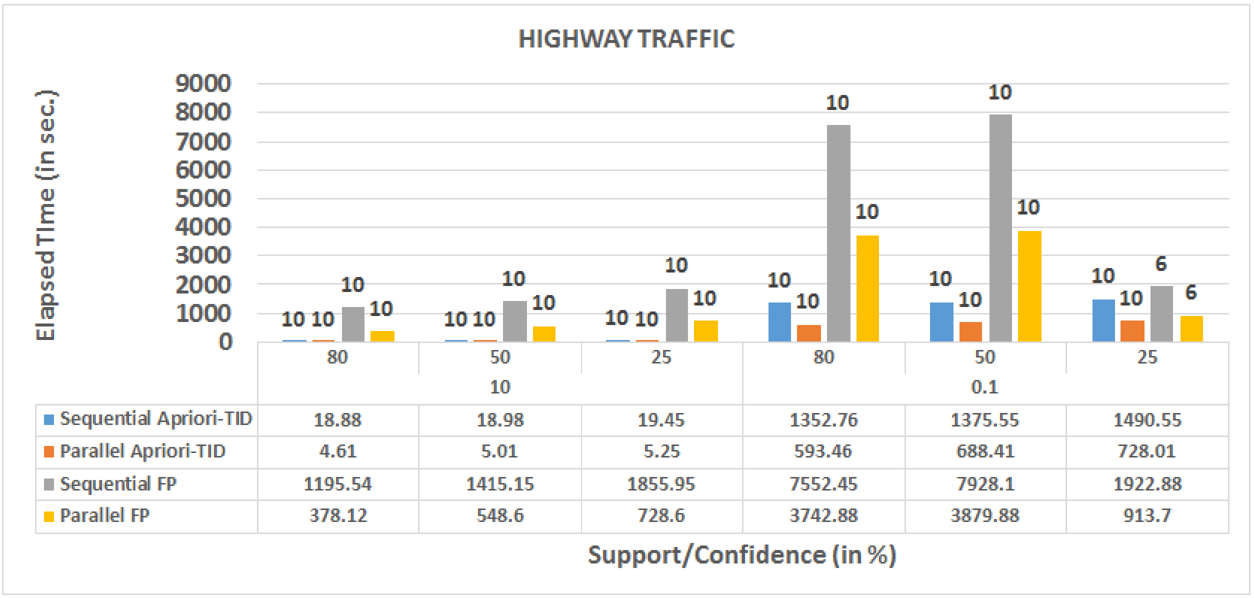
Figure 6.5: MASP vs. R performance (DIABETES dataset)

3. R association rule mining takes more time to run even for less number of attributes and lesser number of transactions in comparison to MASP. Minimum number of attributes for MASP is 36 and R is 26.

4. R ARM is able to mine more number of attributes (32 v/s 26) when support is 10% compared to support = 0.1%.

The Figure 6.6 represent the graph plots between Apriori-TID (both sequential and multi-threaded implementation) and FP-Growth (both sequential and multi-threaded implementation) on Highway Traffic Dataset. The horizontal axis gives the support and confidence values (in %) and vertical axis depicts the elapsed time (in sec.). The data-labels represent the number of attributes for which the algorithm was run. The algorithms are executed for the complete number of records available. Below observations are made based on the Figure 6.6:

1. Multi-threaded implementation of both Apriori-TID and FP-Growth have better performance with respect to elapsed time over sequential implementation of Apriori-TID and FP-Growth respectively. We obtained an average speedup of 2.98 for Apriori-TID and 2.41 for FP-Growth implementation.



	10			0.1		
	80	50	25	80	50	25
■ Sequential Apriori-TID	18.88	18.98	19.45	1352.76	1375.55	1490.55
■ Parallel Apriori-TID	4.61	5.01	5.25	593.46	688.41	728.01
■ Sequential FP	1195.54	1415.15	1855.95	7552.45	7928.1	1922.88
■ Parallel FP	378.12	548.6	728.6	3742.88	3879.88	913.7

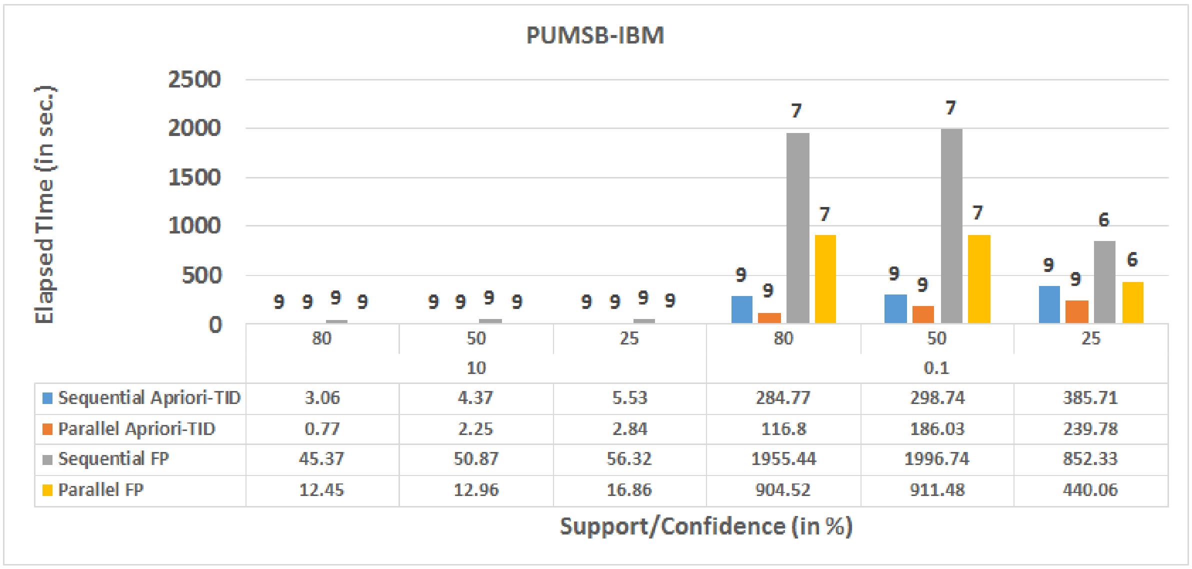
Figure 6.6: Apriori-TID Seq/Multi-Threaded vs. FP-Growth Seq/Multi-Threaded

2. The sequential Apriori-TID and FP-Growth takes highest amount of time when running for Support = 0.1(%) and Confidence = 0.25(%).

3. Apriori-TID is able to incorporate more number of attributes as compared to FP-Growth. Minimum number of attributes for Apriori-TID is 10 and FP-Growth is 6.

The Figure 6.7 represent the graph plots between Apriori-TID (both sequential and multi-threaded implementation) and FP-Growth (both sequential and multi-threaded implementation) on PUMSB-IBM Dataset. The horizontal axis gives the support and confidence values (in %) and vertical axis depicts the elapsed time (in sec.). The data-labels represent the number of attributes for which the algorithm was run. The algorithms are executed for the complete number of records available. Below observations are made based on the Figure 6.7:

1. Multi-threaded implementation of both Apriori-TID and FP-Growth have better performance with respect to elapsed time over sequential implementation of Apriori-TID and FP-Growth respectively. We obtained an average speedup of 2.25 for Apriori-TID and 2.86 for FP-Growth implementation.



	10			0.1		
	80	50	25	80	50	25
🟦 Sequential Apriori-TID	3.06	4.37	5.53	284.77	298.74	385.71
🟧 Parallel Apriori-TID	0.77	2.25	2.84	116.8	186.03	239.78
⬜ Sequential FP	45.37	50.87	56.32	1955.44	1996.74	852.33
🟨 Parallel FP	12.45	12.96	16.86	904.52	911.48	440.06

Figure 6.7: Apriori-TID Seq/Multi-Threaded vs. FP-Growth Seq/Multi-Threaded

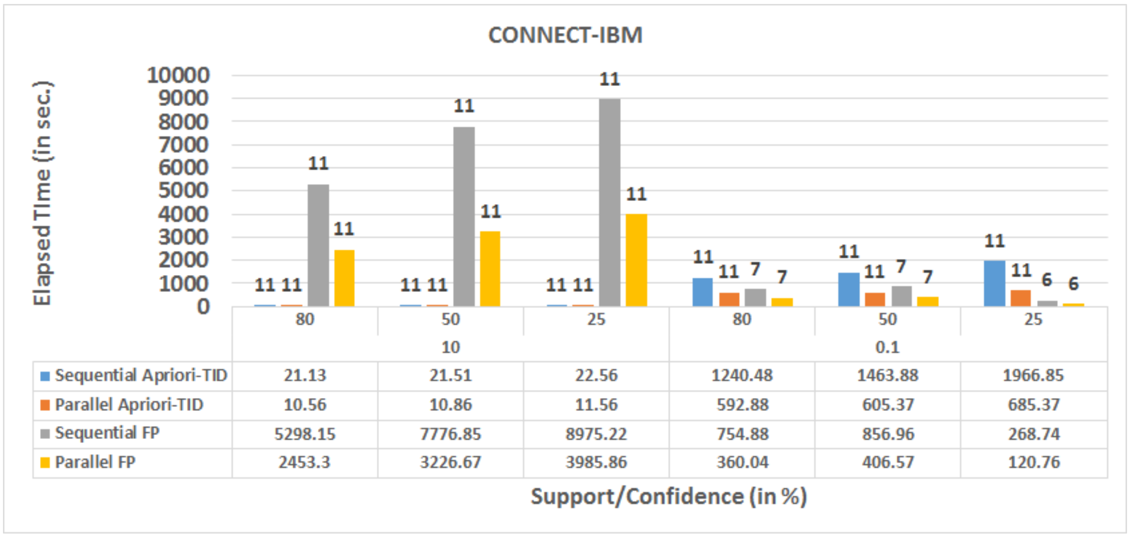
2. As described earlier, the sequential Apriori-TID and FP-Growth takes highest amount of time when running for Support = 0.1(%) and Confidence = 0.25(%).

3. Apriori-TID is able to incorporate more number of attributes as compared to FP-Growth. Minimum number of attributes for Apriori-TID is 9 and FP-Growth is 6.

The Figure 6.8 represent the graph plots between Apriori-TID (both sequential and multi-threaded implementation) and FP-Growth (both sequential and multi-threaded implementation) on CONNECT-IBM Dataset. The horizontal axis gives the support and confidence values (in %) and vertical axis depicts the elapsed time (in sec.). The data-labels represent the number of attributes for which the algorithm was run. The algorithms are executed for the complete number of records available.

Below observations are made based on the Figure 6.8:

1. Multi-threaded implementation of both Apriori-TID and FP-Growth have better performance with respect to elapsed time over sequential implementation of Apriori-TID and FP-Growth respectively. We obtained an average speedup of 2.21 for Apriori-TID and 2.20 for FP-Growth implementation.



	80	50	25	80	50	25
	10			0.1		
■ Sequential Apriori-TID	21.13	21.51	22.56	1240.48	1463.88	1966.85
■ Parallel Apriori-TID	10.56	10.86	11.56	592.88	605.37	685.37
■ Sequential FP	5298.15	7776.85	8975.22	754.88	856.96	268.74
■ Parallel FP	2453.3	3226.67	3985.86	360.04	406.57	120.76

Figure 6.8: Apriori-TID Seq/Multi-Threaded v/s FP-Growth Seq/Multi-Threaded

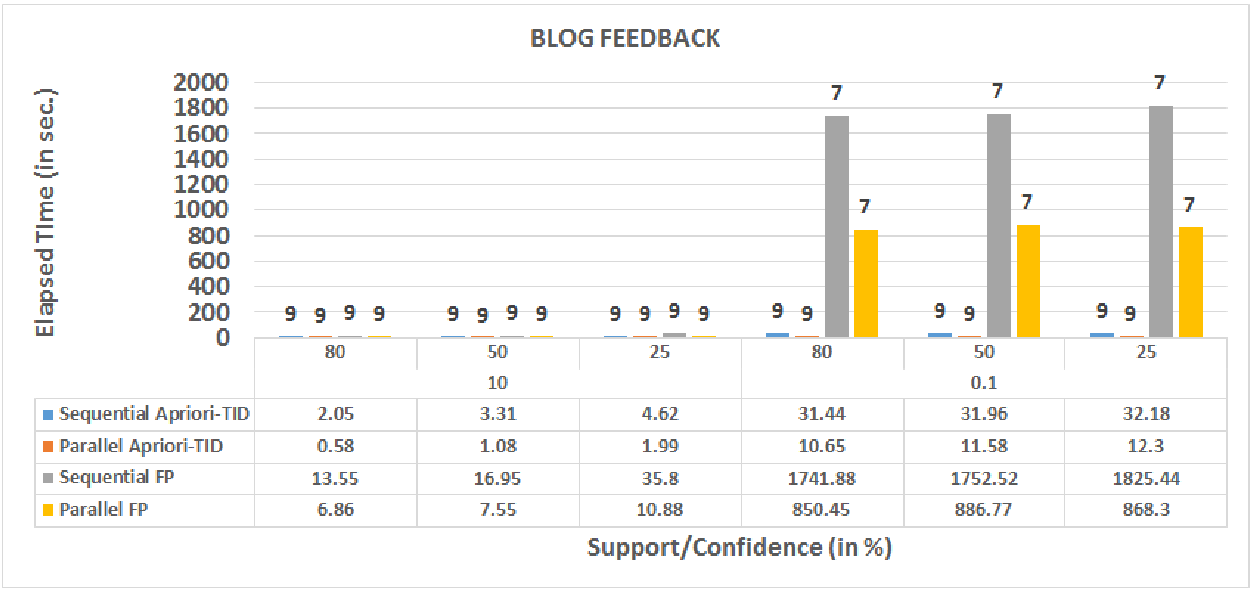
2. As described earlier, the sequential Apriori-TID and FP-Growth takes highest amount of time when running for Support = 0.1(%) and Confidence = 0.25(%).

3. Apriori-TID is able to incorporate more number of attributes as compared to FP-Growth. Minimum number of attributes for Apriori-TID is 11 and FP-Growth is 6.

The Figure 6.9 represent the graph plots between Apriori-TID (both sequential and multi-threaded implementation) and FP-Growth (both sequential and multi-threaded implementation) on BLOG-FEEDBACK Dataset. The horizontal axis gives the support and confidence values (in %) and vertical axis depicts the elapsed time (in sec.). The data-labels represent the number of attributes for which the algorithm was run. The algorithms are executed for the complete number of records available.

Below observations are made based on the Figure 6.9:

1. Multi-threaded implementation of both Apriori-TID and FP-Growth have better performance with respect to elapsed time over sequential implementation of Apriori-TID and FP-Growth respectively. We obtained an average speedup of 2.87 for Apriori-TID and 2.27 for FP-Growth implementation.



	10			0.1		
	80	50	25	80	50	25
🟦 Sequential Apriori-TID	2.05	3.31	4.62	31.44	31.96	32.18
🟧 Parallel Apriori-TID	0.58	1.08	1.99	10.65	11.58	12.3
⬜ Sequential FP	13.55	16.95	35.8	1741.88	1752.52	1825.44
🟨 Parallel FP	6.86	7.55	10.88	850.45	886.77	868.3

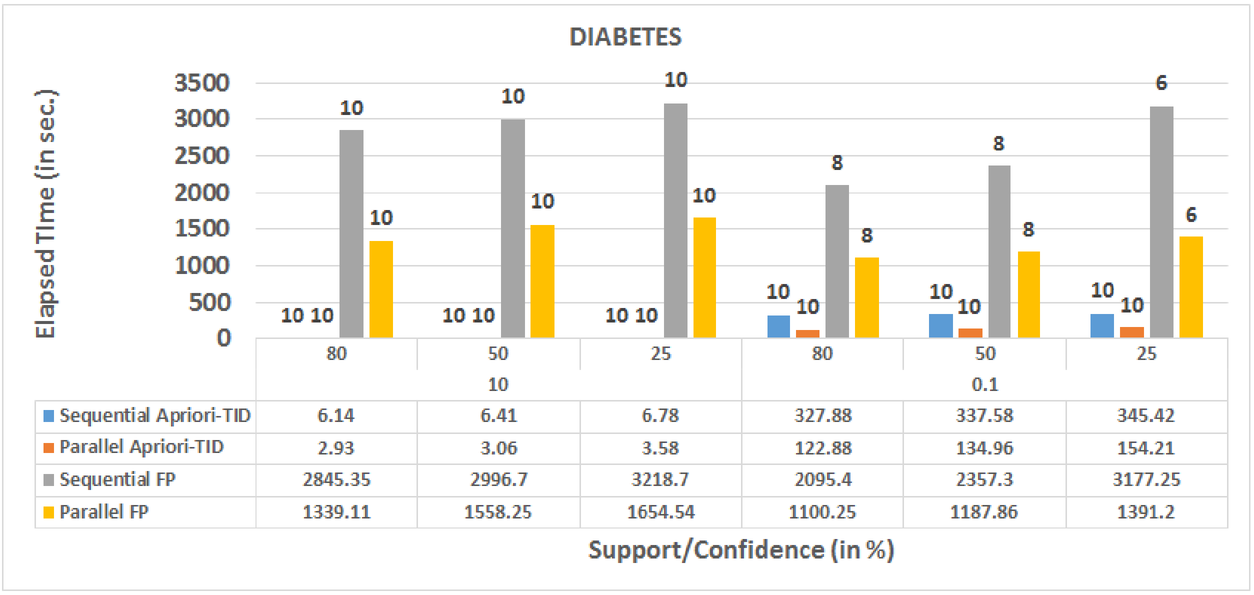
Figure 6.9: Apriori-TID Seq/Multi-Threaded v/s FP-Growth Seq/Multi-Threaded

2. As described earlier, the sequential Apriori-TID and FP-Growth takes highest amount of time when running for Support = 0.1(%) and Confidence = 0.25(%).

3. Apriori-TID is able to incorporate more number of attributes as compared to FP-Growth. Minimum number of attributes for Apriori-TID is 9 and FP-Growth is 7. 4. The longest rule size obtained when Support = 10% is 3 due to the scattered nature of dataset.

The Figure 6.10 represent the graph plots between Apriori-TID (both sequential and multi-threaded implementation) and FP-Growth (both sequential and multi-threaded implementation) on DIABETES Dataset. The horizontal axis gives the support and confidence values (in %) and vertical axis depicts the elapsed time (in sec.). The data-labels represent the number of attributes for which the algorithm was run. The algorithms are executed for the complete number of records available.

Below observations are made based on the Figure 6.10:



	10			0.1		
	80	50	25	80	50	25
🟦 Sequential Apriori-TID	6.14	6.41	6.78	327.88	337.58	345.42
🟧 Parallel Apriori-TID	2.93	3.06	3.58	122.88	134.96	154.21
⬜ Sequential FP	2845.35	2996.7	3218.7	2095.4	2357.3	3177.25
🟨 Parallel FP	1339.11	1558.25	1654.54	1100.25	1187.86	1391.2

Figure 6.10: Apriori-TID Seq/Multi-Threaded v/s FP-Growth Seq/Multi-Threaded

1. Multi-threaded implementation of both Apriori-TID and FP-Growth have better performance with respect to elapsed time over sequential implementation of Apriori-TID and FP-Growth respectively. We obtained an average speedup of 2.24 for Apriori-TID and 2.02 for FP-Growth implementation.

2. As described earlier, the sequential Apriori-TID and FP-Growth takes highest amount of time when running for Support = 0.1(%) and Confidence = 0.25(%).

3. Apriori-TID is able to incorporate more number of attributes as compared to FP-Growth. Minimum number of attributes for Apriori-TID is 10 and FP-Growth is 6.

Table 6.5 shows the experimental results obtained after running sequential and parallel MASP as well as R data mining. The table shows the statistics with respect to the elapsed time of sequential and multi-threaded MASP and R association rule mining. The table also shows the longest rule size obtained. Longest rule size (woN) depict longest of the rule size including negation conditions with respect to attribute values such as A = “10” and B $\neq$ “4”. Longest rule size (wN) shows the longest rule size without any negation condition with respect to attribute values such as A = "4". + symbol in rule size indicates that the rule length can be more than the number provided. The experiments were performed for the maximum number of attributes and number of transactions that can be supported by both MASP and R. Each dataset name contains the total number of transactions in (#Records).

Table 6.6 gives the experimental results obtained from running sequential and parallel Apriori-TID and FP-Growth algorithms. The experiments are performed for the entire datasets based on support (0.1%, 10%) and confidence values (80%, 50%, 25%).

The Table 6.6 depicts the longest rule size obtained for both sequential and parallel Apriori-TID and FP-Growth algorithms. The elapsed time is noted for running the experiments and performance is defined in terms of elapsed time and the number of attributes supported by both the algorithmic implementations. The total number of records for each dataset are mentioned below each dataset.

Table 6.5: MASP Sequential v/s MASP Parallel v/s R (Traditional ARM) on 5 datasets

		Min-support (%)	10			0.1		
		Min-confidence (%)	80	50	25	80	50	25
HIGHWAY TRAFFIC (174437)	MASP (174437)	Max # attributes	33	33	33	33	33	33
		Longest rule size (woN)	18+	19+	19+	18+	27+	28+
		Longest rule size (wN)	18+	19+	19+	18+	27+	30+
		Sequential (Time in sec)	25.99	28.58	30.03	28.58	35.97	140.59
		Parallel (Time in sec)	15.32	15.65	16.85	14.41	19.41	48.14
	R (55000)	Max # attributes	28	28	28	19	19	19
		Longest rule size	19	19	19	16	16	16
		Time in sec	67.85	72.55	76.87	120.56	144.48	154.77
PUMSB (49046)	MASP (49046)	Max # attributes	50	50	50	50	50	50
		Longest rule size (woN)	19+	29+	29+	19+	43+	45+
		Longest rule size (wN)	19+	30+	30+	19+	43+	49+
		Sequential (Time in sec)	11.34	18.47	18.98	11.5	24.67	168.2
		Parallel (Time in sec)	7.88	10.82	11.7	7.91	14.83	53.27
	R (29046)	Max # attributes	26	26	26	19	19	19
		Longest rule size	18	18	18	18	18	18
		Time in sec	55.69	57.16	57.19	65.03	70.17	74.34
CONNECT (67558)	MASP (67558)	Max # attributes	50	50	50	50	50	50
		Longest rule size (woN)	26+	28+	28+	26+	36+	36+
		Longest rule size (wN)	26+	29+	29+	26+	37+	39+
		Time in sec	31.45	32.72	27.21	30.81	46.41	201.5
		Time in sec	14.04	15.84	15.96	14.41	20.26	60.08
	R (15000)	Max # attributes	24	24	24	19	19	19
		Longest rule size	19	19	19	19	19	19
		Time in sec	127.55	142.33	162.95	76.81	94.44	103.38
BLOG (52397)	MASP (52397)	Max # attributes	50	50	50	50	50	50
		Longest rule size (woN)	16+	16+	33+	16+	50+	50+
		Longest rule size (wN)	16+	16+	33+	16+	50+	58+
		Time in sec	11.34	18.47	18.98	11.5	24.67	168.42
		Time in sec	6.06	10.88	11.47	6.85	14.87	89.55
	R (10000)	Max # attributes	21	21	21	20	20	20
		Longest rule size	20	20	20	20	20	20
		Time in sec	98.06	103.87	110.88	178.36	298.87	332.33
DIABETES (37770)	MASP (37770)	Max # attributes	36	36	36	36	36	36
		Longest rule size (woN)	22+	23+	23+	22+	28+	30+
		Longest rule size (wN)	22+	23+	23+	22+	28+	34+
		Time in sec	12.82	15.18	16.27	12.62	14.57	95.31
		Time in sec	7.09	7.8	7.85	7.56	8.52	27.81
	R (9740)	Max # attributes	32	32	32	26	26	26
		Longest rule size	18	18	18	17	17	17
		Time in sec	105.14	107.83	109.36	162.96	174.68	185.32

Table 6.6: APRIORI-TID Sequential/Parallel v/s FP-GROWTH Sequential/Parallel on 5 datasets

		Min-support (%) →	10			0.1		
		Min-confidence (%) →	80	50	25	80	50	25
HIGHWAY TRAFFIC (174437)	APRIORI-TID	Max # attributes	10			10		
		Longest rule size	7	7	7	10	10	10
		Sequential (Time in sec.)	18.88	18.98	19.45	1352.76	1375.55	1490.55
		Parallel (Time in sec.)	4.61	5.01	5.25	593.46	688.41	728.01
	FP-GROWTH	Max # attributes	10			10		6
		Longest rule size	7	7	7	10	10	6
		Sequential (Time in sec.)	1195.54	1415.15	1855.95	7552.45	7928.1	1922.88
		Parallel (Time in sec.)	378.12	548.6	728.6	3742.88	3879.88	913.7
PUMSB-IBM (49046)	APRIORI-TID	Max # attributes	9			9		
		Longest rule size	6	6	6	9	9	9
		Sequential (Time in sec.)	3.06	4.37	5.53	284.77	298.74	385.71
		Parallel (Time in sec.)	0.77	2.25	2.84	116.8	186.03	239.78
	FP-GROWTH	Max # attributes	9			7		6
		Longest rule size	6	6	6	7	7	6
		Sequential (Time in sec.)	45.37	50.87	56.32	1955.44	1996.74	852.33
		Parallel (Time in sec.)	12.45	12.96	16.86	904.52	911.48	440.06
CONNECT-IBM (67558)	APRIORI-TID	Max # attributes	11			11		
		Longest rule size	10	10	10	11	11	11
		Sequential (Time in sec.)	21.13	21.51	22.56	1240.48	1463.88	1966.85
		Parallel (Time in sec.)	10.56	10.86	11.56	592.88	605.37	685.37
	FP-GROWTH	Max # attributes	11			7		6
		Longest rule size	10	10	10	7	7	6
		Sequential (Time in sec.)	5298.15	7776.85	8975.22	754.88	856.96	268.74
		Parallel (Time in sec.)	2453.3	3226.67	3985.86	360.04	406.57	120.76
BLOG (52397)	APRIORI-TID	Max # attributes	9			9		
		Longest rule size	3	3	3	9	9	9
		Sequential (Time in sec.)	2.05	3.31	4.62	31.44	31.96	32.18
		Parallel (Time in sec.)	0.58	1.08	1.99	10.65	11.58	12.3
	FP-GROWTH	Max # attributes	9			7		
		Longest rule size	3	3	3	7	7	7
		Sequential (Time in sec.)	13.55	16.95	35.8	1741.88	1752.52	1825.44
		Parallel (Time in sec.)	6.86	7.55	10.88	850.45	886.77	868.3
DIABETES (37770)	APRIORI-TID	Max # attributes	10			10		
		Longest rule size	9	9	9	10	10	10
		Sequential (Time in sec.)	6.14	6.41	6.78	327.88	337.58	345.42
		Parallel (Time in sec.)	2.93	3.06	3.58	122.88	134.96	154.21
	FP-GROWTH	Max # attributes	10			8		6
		Longest rule size	9	9	9	8	8	6
		Sequential (Time in sec.)	2845.35	2996.7	3218.7	2095.4	2357.3	3177.25
		Parallel (Time in sec.)	1339.11	1558.25	1654.54	1100.25	1187.86	1391.2

Chapter 7
Conclusion

Several experiments have been conducted to test the performance of the multi-threaded approach over the sequential one. We have obtained speedup of around 2.5 on an average. Also, the pattern tree will help in analyzing the mined dataset and provides a visual perspective of the MASP obtained.

MASP implementation is compared with *R* implementation of Apriori algorithm for data mining. MASP is capable of mining longer size of association rule length as compared to R. MASP can handle more number of transactional records and attributes as compared to R implementation. Moreover, the mined data can be exported in table format that can be easily queried and used for analysis purpose.

The experiments conducted to test the performance of both sequential and multi-threaded implementation of Apriori-TID and FP-Growth reveal good results. Apriori-TID is found to be more efficient with respect to space and time as compared to FP-Growth. These results can be attributed to the nature of datasets and algorithmic implementation.

Chapter 8
Future Work

High performance computing (HPC) can be used for association rule mining. The performance tests can be performed on various cloud platforms. This can provide efficient performance measurements with respect to time and space. *MapReduce* services, such as Hadoop DFS (Distributed File System) can be implemented to allow for distributed computation on separate physical nodes.

We can also reduce the memory usage by complementing the execution environment with a low latency storage system such as a RAID (Redundant Array of Independent Disks) of solid state drives to store intermediate data.

Mined data can also be plugged into online visualization services like Google Charts API, to provide an interactive interface for pattern visualization.

Bibliography

[1] R. Agrawal and S. Ramakrishnan. Mining sequential patterns: Generalizations and performance improvements. In *Advances in Database Technology EDBT' 96*, pages 1–17. Springer, 1996.

[2] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[3] Roberto Bayardo. Frequent itemset mining dataset repository. UCI datasets and PUMSB.

[4] Krisztian Buza. Feedback prediction for blogs. In *Data Analysis, Machine Learning and Knowledge Discovery*, pages 145–152. Springer, 2014.

[5] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[6] conorm. *Code Metrics as Check-in Policy.* Microsoft, 2007.

[7] Harisha Donepudi. Detection of interesting traffic accident patterns by association rule mining. Master's thesis, Louisiana State University, 2012.

[8] Blincoe et al. The economic and societal impact of motor vehicle crashes, 2010. (report no. dot hs 812 013). Technical report, National Highway Traffic Safety Administration, Washington, DC, 2014.

[9] Eui-Hong Han, George Karypis, and Vipin Kumar. Scalable parallel data mining for association rules. *Knowledge and Data Engineering, IEEE Transactions on*, 12(3):337–352, 2000.

[10] Highway Safety Research Group, 2014.

[11] Metanat HooshSadat, Hamman W Samuel, Sonal Patel, and Osmar R Zaïane. Fastest association rule mining algorithm predictor (farm-ap). In *Proceedings of The Fourth International C* Conference on Computer Science and Software Engineering*, pages 43–50. ACM, 2011.

[12] Microsoft Developer Network. *System.Collections.Concurrent Namespace.* Microsoft, 2014.

[13] Microsoft Developer Network. *Task Parallelism (Task Parallel Library).* Microsoft, 2014.

[14] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[15] David P Rodgers. Improvements in multiprocessor system design. In *ACM SIGARCH Computer Architecture News*, volume 13, pages 225–231. IEEE Computer Society Press, 1985.

[16] Mrs M Shanthi and A Anthony Irudhayaraj. Multithreading-an efficient technique for enhancing application performance. *International Journal of Recent Trends in Engineering*, 2(4):165–167, 2009.

[17] H. Song-bai, Acad. of Mil. Transp., Tianjin, Wang Ya-jun, Sun Yue-kun, and Gao Wen-Wei. The research of multidimensional association rule mining in traffic accidents. In *4th International Conference on Wireless Communications, Networking and Mobile Computing, WICOM'08*, pages 1–4, 2008.

[18] Beata Strack, Jonathan P DeShazo, Chris Gennings, Juan L Olmo, Sebastian Ventura, Krzysztof J Cios, and John N Clore. Impact of hba1c measurement on hospital readmission rates: Analysis of 70,000 clinical database patient records. *BioMed research international*, 2014, 2014.

[19] Arthur H Watson, Thomas J McCabe, and Dolores R Wallace. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST special Publication*, 500(235):1–114, 1996.

[20] Graham J Williams. Rattle: a data mining gui for r. *The R Journal*, 1(2):45–55, 2009.

[21] E Yilmaz, E Triantaphyllou, J Chen, and TW Liao. A heuristic for mining association rules in polynomial time. *Mathematical and Computer Modelling*, 37(1):219–233, 2003.

[22] Mohammed J Zaki. Parallel and distributed association mining: A survey. *IEEE concurrency*, 7(4):14–25, 1999.

[23] Mohammed J Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery*, 1(4):343–373, 1997.

Vita

Eera Gupta was born in Kanpur, Uttar Pradesh in India. After finishing her high school in Kanpur in 2009, she started her undergraduate studies in Information Technology from University Institute of Engineering and Technology, Kanpur. She completed her summer internship from Indian Institute of Technology, Kanpur in 2008. She worked with Infosys Limited as a Senior Systems Engineer between Nov, 2009 and Dec, 2012. She started her masters studies in Computer Science in Jan, 2013 from Louisiana State University, Baton Rouge and expects to complete her degree by Dec, 2014.