IMPROVING PERFORMANCE OF SEQUENTIAL RULE MINING WITH PARALLEL COMPUTING

Nguyen Thon Da[*] and Tan Hanh[+]
[*] Khoa Hệ thống thông tin, Trường Đại học Kinh tế - Luật, ĐHQG TP. HCM
[+] Học Viện Công Nghệ Bưu Chính Viễn Thông

Abstract: Aiming to improve the performance of sequential rules mining algorithm for the large-scale data sets, this paper presents parallel algorithms for mining sequential rules which directly using MPJ Express for passing message base on multicore configuration and cluster configuration (master-slave structural model). Results analysis showed that the mining time of the parallel algorithms (both multicore and cluster model) which proposed in this paper have better performances compared with the sequential state-of-art algorithm.

Keywords MPI, MPJ Express, Sequential Rule, Association Rule, Parallel Computing, High Performance.

I. INTRODUCTION

Sequential pattern mining has many real-life applications since data is encoded as sequences in many fields such as bioinformatics, e-learning, market basket analysis, text analysis, and webpage click-stream analysis. This is a very active research topic, where hundreds of papers present new algorithms and applications each year, including numerous extensions of sequential pattern mining for specific needs. The task of sequential pattern mining has many applications. A first important limitation of the traditional problem of sequential pattern mining is that a huge number of patterns may be found by the algorithms, depending on a database's characteristics and how the minsup threshold is set by users. Finding too many patterns is an issue because users typically do not have much time to analyze a large amount of patterns.

A good solution for this is sequential rule mining. Sequential rule mining is a variation of the sequential pattern mining problem where sequential rules of the form $X \rightarrow Y$ are discovered, indicating that if some items X appear in a sequence it will be followed by some other items Y with a given confidence.

The concept of a sequential rule is similar to that of association rules excepts that it is required that X must appear before Y according to the sequential ordering, and that sequential rules are mined in sequences rather than transactions. Sequential rules address an important limitation of sequential pattern mining, which is that although some sequential patterns may appear frequently in a sequence database, the patterns may have a very low confidence and thus be worthless for decision-making or prediction.

In this paper, in order to improve the performance of sequential rule mining algorithms, we chose ERMiner to investigate because recently it has become a state-of-art sequential rule mining algorithm comparing to other ones. In next section, we will discuss clearer about this. We propose two models to improve performance of ERMiner algorithm in terms of time execution by using MPJ Express [1] : (1) M-ERMiner (Multicore model for ERMiner algorithm) and (2) C-ERMiner (Cluster model for ERMiner algorithm).

II. RELATED WORKS

The authors of the paper [2] proposed an algorithm based on a distributed application data framework and does not need to create an overall FP-tree. This can avoid the problem that the overall. FP-tree may become too large to be created in RAM. The algorithm uses parallel processing in all its principal steps. It can greatly improve the efficiency and processing ability of the association-rule mining algorithm. It is suitable for association-rule mining on massive data sets which the traditional FP-growth algorithm cannot handle. Their experiments have shown that this algorithm is faster than the FP-growth algorithm for association-rule mining on problems at the same data scale.

The work [3] presented three parallel algorithms for this task based on the Apriori approach. They consist of the Count distribution algorithm, the Data distribution algorithm and the Candidate algorithm. The authors studied the above trade-offs and evaluated the relative performance of the three algorithms by implementing them on 32-node SP2 parallel machine. The Count distribution emerged as the algorithm of choice. It exhibited linear scaleup and excellent speedup and sizeup behavior. When using N processors, the overhead was less than 7.5% compared to the response time of the serial algorithm executing over 1/N amount of data.

The authors of [4] proposed parallel algorithms for the discovery of association rules. The algorithms use novel itemset clustering techniques to approximate the set of potentially maximal frequent itemsets. Using the above techniques they introduced four new algorithms. The Par-Eclat (equivalence class, bottom-up search) and Par-Clique (maximal clique, bottom-up search) algorithms, discover all frequent itemsets, while the Par-MaxEclat (equivalence class, hybrid search) and Par-MaxClique (maximal clique, hybrid search) discover the maximal frequent itemsets. They implemented the algorithms on a 32 processor DEC cluster interconnected with the DEC Memory Channel network, and compared it against a well-known parallel algorithm Count Distribution [3]. Their experimental results indicate that a substantial performance improvement is obtained using their techniques.

The authors of [5] proposed the parallel algorithm called MLFPT, for mining frequent patterns without candidate generation. Their experiments showed that with I/O adjusted, the MLFPT algorithm could achieve an encouraging many-fold speedup improvement. The implementation of their algorithm and the experiments conducted were on a shared memory and shared hard drive architecture.

The work [6] presented parallel Data Mining architecture for large volume of data which eventually scanning billions of rows of data per record. The authors of this paper compare the different parallel algorithms for Association Rule Mining and discuss the advantages and disadvantages of each method. They also compare the computational time of serial and parallel algorithms for Association Rule Mining.

However, models based on Association Rules have many backwards. Costly, for example, especially when there exist a large number of patterns and/or long patterns. Moreover, they was built prediction lossy models from training sequences. Thus, they do not use all the information available in training sequences for making predictions. Besides, if applied on data with time or sequential ordering information, this information will be ignored.

In the next section, we will present the approach of sequential rules mining then we also introduce a parallel method for it.

III. THE METHOD OF SEQUENTIAL RULES MINING

There are many algorithms proposed for mining sequential rules:

CMDeo [7]: A main drawback of CMDeo is that it can generate a huge amount of candidates. A better algorithm, the CMRules algorithm was proposed [7]. It was shown to be much faster than CMDeo for sparse datasets. Moreover, the RuleGrowth [8], an algorithm relying on a pattern-growth approach to avoid candidate generation was proposed. It was shown to be more than an order of magnitude faster than CMDeo and CMRules. However, for datasets containing dense or long sequences, the performance of RuleGrowth rapidly deterioates because it has to repeatedly perform costly database projection operations.

Authors of proposed the ERMiner (Equivalence class based sequential Rule Miner) algorithm. It relies on a vertical representation of the database to avoid performing database projection and the novel idea of explorating the search space of rules using equivalence classes of rules having the same antecedent or consequent. Besides, it consists of a data structure named SCM (Sparse Count Matrix) to prune the search space.

Fig.1 depicts the core pseudocode of ERMiner. ERMiner takes as input a sequence database SDB, and the minsup and minconf thresholds. It first scans the database once to build all equivalence classes of rules of size 1 ∗ 1. Then, to discover larger rules, left merges are performed with all left equivalence classes by calling the leftSearch procedure. Similarly, right merges are performed for all right equivalence classes by calling the rightSearch procedure. In this case, the rightSearch procedure may generate some new left-equivalence classes because left merges are allowed after right merges. These equivalence classes are stored in the leftStore structure. To process these equivalence classes, an extra loop is performed. Finally, the algorithm returns the set of rules found rules.

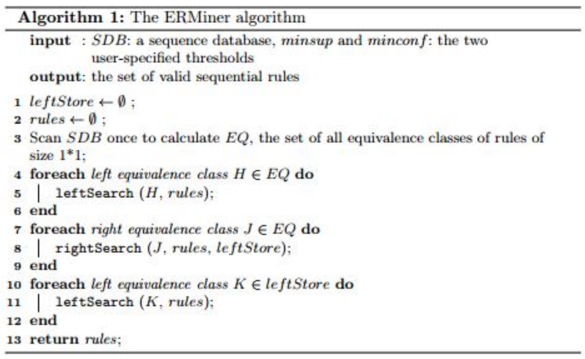

```
Algorithm 1: The ERMiner algorithm
input  : $SDB$: a sequence database, $minsup$ and $minconf$: the two
         user-specified thresholds
output: the set of valid sequential rules
$leftStore \leftarrow \emptyset$ ;
$rules \leftarrow \emptyset$ ;
Scan $SDB$ once to calculate $EQ$, the set of all equivalence classes of rules of
size 1*1;
foreach left equivalence class $H \in EQ$ do
    leftSearch ($H$, $rules$);
end
foreach right equivalence class $J \in EQ$ do
    rightSearch ($J$, $rules$, $leftStore$);
end
foreach left equivalence class $K \in leftStore$ do
    leftSearch ($K$, $rules$);
end
return $rules$;
```

Fig. 1. The ERMiner algorithm [9]

Fig.2 depicts the pseudocode of the *leftSearch* procedure. It takes as parameter an equivalence class LE. Then, for each rule r of that equivalence class, a left merge is performed with every other rules to generate a new equivalence class. Only frequent rules are kept. Moreover, it is output if a rule is valid. Then, *leftSearch* is recursively called to explore each new equivalence class generated that way. Similarly, we have the *rightSearch* (see Fig. 3). The important difference is that new left equivalences are stored in the left store structure because their exploration is postponed, as previously explained in the main procedure of ERMiner.

```
Algorithm 2: The leftSearch procedure
input : LE: a left equivalence class, rules: the set of valid rules found until
        now, minsup and minconf: the two user-specified thresholds
foreach rule $r \in LE$ do
    $LE' \leftarrow \emptyset$ ;
    foreach rule $s \in LE$ such that $r \neq s$ and the pair $r, s$ have not been
    processed do
        Let $c, d$ be the items respectively in $r, s$ that do not appear in $s, r$ ;
        if $countPruning(c, d) = false$ then
            $t \leftarrow leftMerge(r, s)$ ;
            $calculateSupport(t, r, s)$;
            if $sup(t) \geq minsup$ then
                $calculateConfidence(t, r, s)$;
                if $conf(t) \geq minconf$ then
                    $rules \leftarrow rules \cup \{t\}$;
                end
                $LE' \leftarrow LE' \cup \{t\}$;
            end
        end
    end
    leftSearch ($LE'$, $rules$);
end
```

Fig. 2: The leftSearch procedure [9]

Besides, an optimization is to use the Sparse Count Matrix structure (SCM). This structure is built during the first database scan and record in how many sequences each item appears with each other items. For example, Fig. 3 depicts the structure built for the database of Fig. 1 (left), represented as a triangular matrix. Consider the second row. It indicates that item b appear with items b, c, d, e, f, g and h respectively in 2, 1, 3, 4, 2 and 1 sequences. The SCM structure is used for pruning the search space as follows (implemented as the countPruning function in Fig. 3 and 2). Let be a pair of rules r, s that is considered for a left or right merge and c, d be the items of r and s that respectively do not appear in s and r. If the count of c, d is less than minsup in the SCM, then the merge does not need to be performed and the support of the rule is not calculated. Another important optimization is how to implement the left store structure for efficiently storing left equivalence classes of rules that are generated by right merges. In our implementation, the authors of [9] use a hashmap of hashmaps, where the first hash function is applied to the size of a rule and the second hash function is applied to the left itemset of the rule. This allows to quickly find to which equivalence class belongs a rule generated by a right merge.

```
Algorithm 3: The rightSearch procedure
input : RE: a right equivalence class, rules: the set of valid rules found until
        now, minsup and minconf: the two user-specified thresholds,
        leftStore: the structure to store left-equivalence classes of rules
        generated by right-merges
foreach rule $r \in RE$ do
    $RE' \leftarrow \emptyset$ ;
    foreach rule $s \in RE$ such that $r \neq s$ and the pair $r, s$ have not been
    processed do
        Let $c, d$ be the items respectively in $r, s$ that do not appear in $s, r$ ;
        if $countPruning(c, d) = false$ then
            $t \leftarrow rightMerge(r, s)$ ;
            $calculateSupport(t, r, s)$;
            if $sup(t) \geq minsup$ then
                $calculateConfidence(t, r, s)$;
                if $conf(t) \geq minconf$ then
                    $rules \leftarrow rules \cup \{t\}$;
                end
                $RE' \leftarrow RE' \cup \{t\}$;
                $addToLeftStore(t)$
            end
        end
    end
    rightSearch ($RE'$, $rules$);
end
```

Fig. 3. The rightSearch procedure [9]

Item	a	b	c	d	e	f
b	3					
c	2	2				
d	1	1	1			
e	3	3	2	1		
f	3	4	2	1	3	
g	1	2	1	0	1	2
h	0	1	0	0	0	1

Fig. 4. The Spare Count Matrix [9]

For the time complexity, the brief idea is the following: We have a database containing *n* transactions and some thresholds set by the user. The algorithm first scan the database, which takes *O(n)* time. Then the algorithm processes several equivalence classes using either *leftSearch* or *rightSearch*. In the worst case, the algorithm will process all possible equivalence classes that could exist in the database. However, generally, the minsup threshold will be useful to reduce the search space and the algorithm will not need to process all the equivalence classes. The *leftSearch* procedure is applied to an equivalence class containing *r* rules. The *leftSearch* procedure will compare each pair of rules from that equivalence classes using two *for* loops. Thus, it will approximately do *O(r^2)* comparison. For each pair or rules *R1* and *R2*, if the pruning conditions are passed, the support and confidence will be calculated. Calculating the support and confidence is done by comparing the list of occurrences of *R1* and *R2* as done in RuleGrowth [8]. The list of occurrences are implemented as hashmaps. Thus, the cost of this comparison is *O(k)*, where *k* is the longest list of occurrences between those of *R1* and *R2*. Thus globally, we can say that the complexity is roughly exponential for processing each equivalence class *(O(r^2))*. But in practice the equivalence classes are not always very large. For *rightSearch*, it is similar to *leftSearch*. For the overal complexity, if there are *w* equivalence classes that are processed by the algorithm, then the time complexity would be *O(w*y^2)*, where *y* is the average number of rules per equivalence class.

IV. THE METHOD OF SEQUENTIAL RULES MINING

In this section, we will introduction to MPI, especially MPJExpress, in Section A, an implementation of a parallel sequential rule mining model based on multicore configuration, called M-ERMiner in Section B, another model based on cluster configuration, called C-ERMiner in Section C.

A. Introduction to MPJ Express

MPI is a communication protocol for programming parallel computers. Both point-to-point and collective communication are supported. MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today [10].

MPI model have been developed in various languages such as C/C++, Python, .NET, Java… According to the authors of [11]: Most popular and adopted implementations are written in C/C++ as they are suited for a wide range of scientific and research communities for enabling parallel applications. However it lacks the support for heterogeneous operating system in an integrated environment. Though there are few MPI implementations in Python but all of them are being utilized in specific projects and have communication performance issues. For future implementations Java remains an obvious choice for developing parallel computing applications for multi core hardware mainly because of its diversity and features. MPI.Net is the only implementation other than A-JUMP that provides interoperability between different programming languages within the Microsoft .Net framework. The study of different grid implementations clearly shows that MPI over Internet is a challenge because of its volume and complexity. Among approaches using Java, MPJ Express is a good choice.

MPJ Express is a message passing library that can be used by programmers to run their parallel Java applications on clusters or network of computers. Compute clusters is common parallel platform, that is extensively used by the High Performance Computing (HPC) community for computing large data. MPJ Express is necessarily a middleware that supports communication between individual processors of cluster. The programming model of MPJ Express is Single Program Multiple Data (SPMD).

In the paper [1], the authors have benchmarked our system against various other messaging libraries and shown that MPJ Express is able to achieve comparable performance to other systems. There is an overhead associated with MPJ Express pure Java devices that can potentially be resolved by extending the MPJ API to allow communicating data to and from ByteBuffers. The very important contribution of the works related to parallel Apriori algorithm based on MPI is the development of a Java-based thread-safe messaging system. This messaging system coupled with Java or JOMP threads can help with more efficiently programming parallel applications on the emerging multi-core HPC systems. This is the first effort to address efficient programming of multicore HPC systems by using nested parallelism with a Java messaging system. Moreover, a very good feature of MPJ Express is that it provides thread-safe communication devices that allow multiple threads in an application to communicate safely. The paper [12] presented two new communication devices for MPJ Express to improve scalability of parallel Java applications on modern HPC systems. In particular they developed *hybdev* for clusters with shared memory and multicore processors *native* for using native MPI libraries from within MPJ Express programs. With the addition of these new device, MPJ Express users have the option to either opt for portability - by using pure Java device - or performance - by using the *native* device. The other device, *hybdev*, is developed to allow efficient and transparent execution of parallel Java applications on clusters of shared memory or multicore processors.

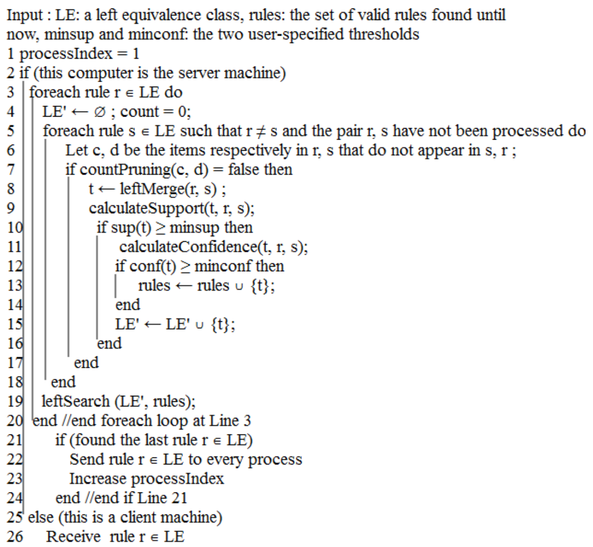
B. M-ERMiner Model (Multicore Configuration)

We modified two procedures of original ERMinner: Algorithm 2' and Algorithm 3'.

Algorithm 2' is the variant of the *leftSearch* procedure. It was parallelized by changes compare to the original *leftSearch* procedure. Explanation for the algorithm 2':

Line 1: Initialize with the first process.
Line 2: If the operation running at server machine
Line 3 - Line 20: The loop find valid rules from left equivalence classes
Line 21 - 24: Share works to processes
Line 25 - 26: Clients receive passing message from the server machine

Thus, if we called the number of jobs be J and N be the number of processes, we have J = (K mod N). It means that if there are 10 lines and N = 4, it will share groups 3, 3, 3, 1 lines for every process.



```
Input : LE: a left equivalence class, rules: the set of valid rules found until
now, minsup and minconf: the two user-specified thresholds
1 processIndex = 1
2 if (this computer is the server machine)
3    foreach rule r ∈ LE do
4       LE' ← ∅ ; count = 0;
5       foreach rule s ∈ LE such that r ≠ s and the pair r, s have not been processed do
6          Let c, d be the items respectively in r, s that do not appear in s, r ;
7          if countPruning(c, d) = false then
8             t ← leftMerge(r, s) ;
9             calculateSupport(t, r, s);
10             if sup(t) ≥ minsup then
11                calculateConfidence(t, r, s);
12                if conf(t) ≥ minconf then
13                   rules ← rules ∪ {t};
14                end
15                LE' ← LE' ∪ {t};
16             end
17          end
18       end
19       leftSearch (LE', rules);
20    end //end foreach loop at Line 3
21       if (found the last rule r ∈ LE)
22          Send rule r ∈ LE to every process
23          Increase processIndex
24       end //end if Line 21
25 else (this is a client machine)
26    Receive  rule r ∈ LE
```

Fig.5. Algorithm 2': leftSearch procedure (Parallel)

Algorithm 3' is the variant of the *rightSearch* procedure. It was parallelized by changes compare to the original *rightSearch* procedure. Explanation for the Algorithm 3':

Line 1: Initialize with the first process
Line 2: If the operation running at server machine
Line 3 - Line 22: The loop find valid rules from equivalence classes.
Line 22 - 26: Share works to processes.
Line 27 - 28: Clients receive passing message from the server machine.

Thus, if we called the number of jobs be J and N be the number of processes, we have *J = (K mod N)*. It means that if there are 14 lines and *N = 4*, it will share groups 4, 4, 4, 2 lines for every process.

```
Input : RE: a right equivalence class, rules: the set of valid rules found until
        now, minsup and minconf: the two user-specified thresholds, leftStore:
        the structure to store left-equivalence classes of rules
        generated by right-merges
1 processIndex = 1
2 if (this computer is the server machine)
3  | foreach rule r ∊ LE do
4  |   LE' ← ∅ ; count = 0;
5  |  foreach rule s ∊ RE such that r # s and the pair r, s have not been processed
6  |   | Let c, d be the items respectively in r, s that do not appear in s, r ;
7  |   | if countPruning(c, d) = false then
8  |   |  | t ← rightMerge(r, s) ;
9  |   |  | calculateSupport(t, r, s);
10 |   |  |  if sup(t) ≥ minsup then
11 |   |  |   | calculateConf idence(t, r, s);
12 |   |  |   | if conf(t) ≥ minconf then
13 |   |  |   |  | rules ← rules ∪ {t};
14 |   |  |   | end // end if at line 12
15 |   |  |   | RE' ← RE' ∪ {t};
16 |   |  |   | addToLef tStore(t)
17 |   |  |  end // end if at Line 10
18 |   |  end // end if at Line 7
19 |   end // end for loop at Line 5
20 |   rightSearch (RE, rules);
21 | end //end foreach loop at Line 3
22 |     if (found the last rule r ∊ LE)
23 |        Send rule r ∊ LE to every process
25 |        Increase processIndex
26 |     end //end if Line 22
27 else (this is a client machine)
```

Fig. 6. Algorithm 3': rightSearch procedure (Parallel)

For the time complexity in parallel cases, we set *p* is number of cores in the computer we are considering. For *LeftSearch* procedure and *RightSearch* procedure, if there are *w* equivalence classes that are processed by the algorithm, the time complexity would be *O((w*y^2)/p)*, where *y* is the average number of rules per equivalence class.

C. C-ERMiner Model (Cluster Configuration)

In this model, we execute M-ERMiner with computing parallel in a network non-shared system. We mainly investigate two kinds of cluster configuration including niodev and hybdev using MPJ Express in the Cluster Configuration.

(1) niodev: This a one of four communication devices in the cluster configuration: niodev, mxdev, hybdev and native. The Java NIO device driver (called niodev) can be used to execute MPJ Express programs on clusters or network of computers. Its driver utilizes Ethernet-based interconnect to pass message.

(2) hybdev: The hybrid device allows users plan to execute their parallel Java application on such a cluster of multicore computers. Hybrid device transparently utilizes both multicore configuration and network of computers configuration for intra-node communication and cluster configuration (just for NIO device) for inter-node communication, respectively.

We utilized the M-ERMiner for parallel computing in C-ER Model. Figure 7 shows the network diagram of Cluster Configuration:

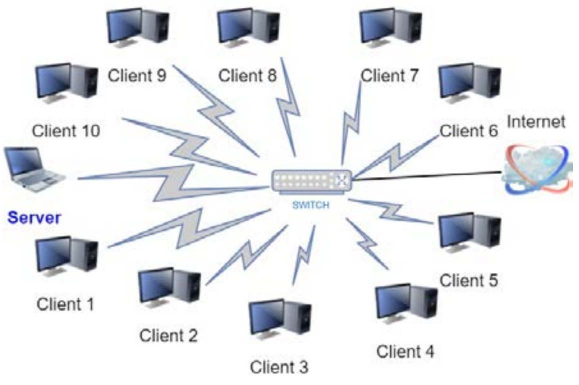


Fig. 7. The network diagram of Cluster Configuration

IV. EXPERIMENTAL RESULTS

A. Experimental Environment

(1) For M-ERMiner model:

The hardware platform uses a laptop with the configuration: 32GB RAM, Intel 8-core processor-i7-4800M, CPU@2.70 GHz, 256 GB hard drive (SSD 256 MB);

(2) For C-ERMiner model:

The hardware platform uses a PC plays a role as master machine with the configuration like that of M-ERMiner model and 10 slave PCs.

Every slave PC has the configuration: 4 GB RAM, Intel 4-core processor-i3-4130, CPU@3.4GHz, 200 GB hard drive.

The software environment for two above model uses the following configuration: the operation system is Ubuntu 14.04 LTS 64 bit, the parallel and distributed environment is the MPJ Express v0_44, Java development platform is the JDK 8u131; Network environment is 1000M- LAN.

Considering the fairness of comparison, the configuration of MPI parallel development platform is based on open resource project Eclipse Neon.3 in Linux.

B. Data

We investigate on real-life datasets such as *SIGN*, *LEVIATHAN* and *FIFA*, *MSNBC* (www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php).

SIGN: This is a dataset of sign language utterance containing approximately 800 sequences. The original dataset file in another format can be obtained here with more details on this dataset.

LEVIATHAN: This dataset is a conversion of the novel Leviathan by Thomas Hobbes (1651) as a sequence database (each word is an item). It contains 5834 sequences and 9025 distinct items. The average number of items per sequence is: 33.8. The average number of distinct item per sequence is 26.34.

FIFA: a dataset of 20,450 sequences of click stream data from the website of FIFA World Cup 98. It has 2,990 distinct items (webpages). The average sequence length is 34.74 items with a standard deviation of 24.08 items.

MSNBC: a dataset of click-stream data. The original dataset contains 989,818 sequences obtained from the UCI repository.

All these real-life datasets are in SPMF format [http://www.philippe-fournier-viger.com/spmf/]

The SPMF format is defined as follows. It is a text file where each line represents a sequence from a sequence database. Each item from a sequence is a positive integer and items from the same itemset within a sequence are separated by single spaces. Note that it is assumed that items within a same itemset are sorted according to a total order and that no item can appear twice in the same itemset. The value "-1" indicates the end of an itemset. The value "-2" indicates the end of a sequence (it appears at the end of each line). For example, the sample input file as follows contains the following four lines (4 sequences).

```
1 -1 1 2 3 -1 1 3 -1 4 -1 3 6 -1 -2
1 4 -1 3 -1 2 3 -1 1 5 -1 -2
5 6 -1 1 2 -1 4 6 -1 3 -1 2 -1 -2
5 -1 7 -1 1 6 - 1 3 -1 2 -1 3 -1
-2
```

The first line represents a sequence where the itemset {1} is followed by the itemset {1, 2, 3}, followed by the itemset {1, 3}, followed by the itemset {4}, followed by the itemset {3, 6}. The next lines follow the same format.

C. Evaluation

In the first experiment, we compare the performance of sequential ERMiner [9] with that of M-ERMiner (multicore-ERMiner). We have performed an experiment on four datasets and measured the execution time. In conclusion, we can see that M-ERMiner is up to from 0.4 to 0.8 times faster than sequential ERMiner for above datasets.

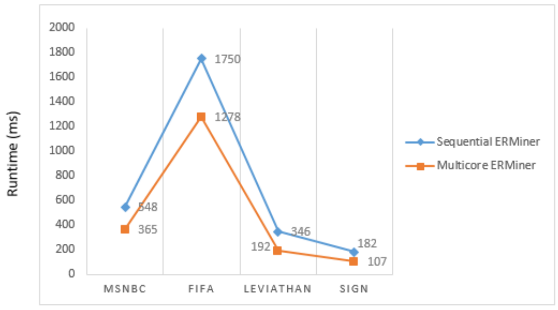


Fig. 8. Comparison of execution time of Sequential ERMiner and Multicore ERMiner

In the second experiment, we compare the performance of the Cluster Configuration (niodev) with that of the Cluster Configuration (hybdev). We have performed an experiment on four datasets and measured the execution time. In conclusion, we realize that Cluster Configuration (hybdev) is up to from 2 to 5 times faster than Cluster Configuration (niodev) for above datasets.

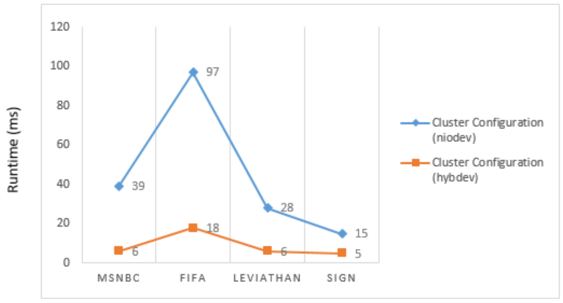


Fig.9. Comparison of execution time of Cluster Configuration (niodev) and Cluster Configuration (hybdev)

V. CONCLUSION

We present a sequential rule mining parallel computing approach consisting of 3 main models: (1) ERMiner in Multicore configuration, (2) ERMiner in Cluster Configuration (niodev), (3) ERMiner in Cluster Configuration (hybdev). The experimental results indicate that The ERMiner in Multicore configuration model is much better than the original (sequential) ERMiner, ERMiner in Cluster Configuration (hybdev) is much better than ERMiner Cluster Configuration (niodev).

I. ADKNOWLEGMENTS

The authors thank the Center of Business Intelligence, Faculty of Information System, University of Economics and Law for its network of computers environment support. Besides, we are also be grateful to Full Professor Philippe Fournier Viger (Director of Center of Innovative Industrial Design, Harbin Institute of Technology (Shenzhen)) for his helps in order that we could finish this paper.

II. REFERENCES

[1] M. Baker, B. Carpenter, and A. Shafi, "MPJ Express: towards thread safe Java HPC," in *Cluster Computing, 2006 IEEE International Conference on*, 2006, pp. 1-10: IEEE.
[2] Z.-g. Wang and C.-s. Wang, "A parallel association-rule mining algorithm," in *International Conference on Web Information Systems and Mining*, 2012, pp. 125-129: Springer.
[3] R. Agrawal and J. C. Shafer, "Parallel mining of association rules," *IEEE Transactions on knowledge and Data Engineering*, vol. 8, no. 6, pp. 962-969, 1996.
[4] M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, P. Stolorz, and R. Musick, "Parallel algorithms for discovery of association rules," in *Scalable High Performance Computing for Knowledge Discovery and Data Mining*: Springer, 1997, pp. 5-35.
[5] O. R. Zaïane, M. El-Hajj, and P. Lu, "Fast parallel association rule mining without candidacy generation," in *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, 2001, pp. 665-668: IEEE.
[6] S. Einakian and M. Ghanbari, "Parallel implementation of association rule in data mining," in *System Theory, 2006.*

SSST'06. Proceeding of the Thirty-Eighth Southeastern Symposium on, 2006, pp. 21-26: IEEE.

[7] P. Fournier-Viger, U. Faghihi, R. Nkambou, and E. M. Nguifo, "CMRules: Mining sequential rules common to several sequences," *Knowledge-Based Systems,* vol. 25, no. 1, pp. 63-76, 2012.

[8] P. Fournier-Viger, R. Nkambou, and V. S.-M. Tseng, "RuleGrowth: mining sequential rules common to several sequences by pattern-growth," in *Proceedings of the 2011 ACM symposium on applied computing*, 2011, pp. 956-961: ACM.

[9] P. Fournier-Viger, T. Gueniche, S. Zida, and V. S. Tseng, "ERMiner: sequential rule mining using equivalence classes," in *International Symposium on Intelligent Data Analysis*, 2014, pp. 108-119: Springer.

[10] A. Shafi, B. Carpenter, and M. Baker, "Nested parallelism for multi-core HPC systems using Java," *Journal of Parallel and Distributed Computing,* vol. 69, no. 6, pp. 532-545, 2009.

[11] M. Hafeez, S. Asghar, U. A. Malik, A. ur Rehman, and N. Riaz, "Survey of MPI implementations," in *International Conference on Digital Information and Communication Technology and Its Applications*, 2011, pp. 206-220: Springer.

[12] A. Javed, B. Qamar, M. Jameel, A. Shafi, and B. Carpenter, "Towards Scalable Java HPC with Hybrid and Native Communication Devices in MPJ Express," *International Journal of Parallel Programming,* vol. 44, no. 6, pp. 1142-1172, 2016.

Thon Da Nguyen received Master degree in Computer Science from the University of Technology, VNU-HCM in 2013. In November 2016, he was accepted as a Ph.D. Student in Information Systems at Posts and Telecommunications Institute of Technology, Vietnam. He is now working as researcher and an assistant teacher at Faculty of Information Systems, University of Economics and Law, VNU-HCM. His research interests include data mining, pattern mining, sequence analysis and prediction.

Hanh Tan received the PhD degree from Grenoble Institute of Technology, France. Currently, he is vice president of Posts and Telecommunications Institute of Technology. His research interests are machine learning, information retrieval, and data mining.