Manila Journal of Science 9 (2016), pp. 115-135

Survey on the Current Status of Serial and Parallel Algorithms of Frequent Itemset Mining

Roger Luis Uy[1]*, Merlin Teodosia C. Suarez[2]

1Computer Technology Department, De La Salle University, 2401 Taft Ave., Manila 0922, Philippines
2Software Technology Department, De La Salle University, 2401 Taft Ave., Manila 0922, Philippines

(*Corresponding Author) Email: roger.uy@dlsu.edu.ph

ABSTRACT

Frequent itemset mining is one of the fundamental but time-demanding tasks in data mining. It is used to find frequent patterns and generate association rules for these patterns. With the availability of inexpensive storage and progress in data capture technology, the availability of data has reached exa-scale already. But improvements in processor and network technology open up opportunity for parallel and distributed computing to be applied in frequent itemset mining to improve its performance in the light of the challenge of "big data". Thus, there are challenges in frequency itemset mining to fully harness the parallel processing capability of the computer hardware technologies. This paper reviews the development of current serial and parallel approaches to frequent itemset mining and discusses future research directions in this field.

Keywords: Frequent itemset mining, Data mining, GPU computing, Multi-core computing, Distributed computing

Copyright © 2016 by De La Salle University Publishing House

INTRODUCTION

Frequent itemset mining (FIM) is a fundamental task in the field of data mining such as association rule discovery (Agrawal, Imielinksi, & Swami, 1993), clustering (Kosters, Marchiori, & Oerlemans, 1999), classification (Hu, Lu, Zhou, & Shi, 1999), web mining (Woon, Ng, & Lim, 2002), and others. It aims to extract hidden patterns in large volumes of data by discovering frequently occurring groups of items in the database. Once the hidden patterns are extracted and strong associations are found, useful information can be derived from these patterns. By going through volumes of data, valuable information can be extracted in commerce, bioinformatics, electronic commerce (Sarwar, Karypis, Konstan, & Riedl, 2000), network intrusion detection, and other real-world applications.

In recent years, vast amounts of data are generated in many fields (Lynch, 2008) (Szalay & Gray, 2006). These range from commerce to scientific research data. To put this number in perspective, Wal-Mart generates more than 200 million transactions per day in all its stores worldwide and the volume of business data worldwide is estimated to double every 1.2 years (Manyika, et al., 2012). The data being generated by scientific research is also enormous. The large hadron collider (LHC), a particle accelerator, generates 60 terabytes of data per day while 32 petabytes of climate data are stored in the NASA discovery supercomputer cluster (Brumfiel, 2011).

Generating frequent itemsets is a time-demanding task. Its operation includes generating combinations of set given n items. This operation leads to an exponential time complexity (i.e., $\sum_{i=1}^{n}\binom{n}{i} = 2^n - 1$). With the trends toward increase volume of data, n is potentially a very large number.

Computing hardware technologies have also improved tremendously during the past few decades. Processor speed has improved to the point that it has hit the power and thermal wall constraints. This led to the redesigning of processor architecture which espouses lower speed but with increased number of processing cores (Uy, 2014). With the widespread availability and affordability of these multi-core processor systems, using parallel computing is now a viable solution rather than an expensive option.

Another improvement in processor technology is the redesigning of graphics processing unit (GPU) as a programmable processor. This development transforms the GPU from a traditional graphics coprocessor to a general-purpose programming processor. This paradigm shift is known as GPU computing (NVIDIA Corporation, 2015). A GPU is composed of many cores and uses single-instruction multiple-thread (SIMT) model. The introduction of Compute Unified Device Architecture (CUDA) provides a platform to program GPU using various high level programming languages (Glaskowsky, 2009). This makes a GPU device, which are affordable and widely available, suitable for parallel computing as well (Barlas, 2015).

Improvement in network technology leads to the rise of cloud computing. This technology promises a reliable software, hardware and infrastructure-as-a-service over the internet (Armburst, et al., 2010). With cloud computing, off-the-shelf grid and cluster computing is now available as a service. This makes distributed computing, which is a form of parallel computing, a common and affordable option as well.

Parallel implementations of FIM algorithms using multi-core, GPU, as well as distributed computing have started to emerge (Zhang, Zhang, & Bakos, 2013), (Lu & Alaghband, 2014), (Lin & Chung, 2015). Though, development of serial algorithms remain very much active (Deng & Wang, 2010) (Wang, Jiang, & Deng, 2012) (Lv & Deng, 2014). This paper surveys the current development of serial and parallel

approach to frequent itemset mining and discusses future research directions in this field.

DEFINITIONS

This section discusses key terms and definitions of FIM (Goethals, 2003). A more in-depth introduction can be found in (Gorunescu, 2011).

Frequent Itemset Mining

Formally, the task of FIM can be described as follows. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of all items. A subset $X = \{i_1, i_2, \ldots, i_k\} \subseteq I$ is called an itemset or k-itemset if it contains k items.

A transaction over I is a pair $T = (tid, J)$, where tid is the transaction identifier and J is an itemset.

A transaction database $D = \{T_1, T_2, \ldots, T_m\}$ contains a set of transactions over I.

The cover of an itemset X in D consists of the set of tid of transaction in D that supports X:

$$cover(X,D) := \{tid \mid (tid,J) \in D, X \subseteq I\}$$

The support of an itemset X is the number of transactions in the cover of X in D (i.e., number of transactions that contain the itemset X):

$$support(X,D) := |cover(X,D)|$$

An itemset is frequent if its support is no less than a given minimal support threshold σ. The threshold can either be absolute σ_{abs}, with $0 \leq \sigma_{abs} \leq |D|$ or relative σ_{rel}, with $0 \leq \sigma_{rel} \leq 1$. In this paper, the relative threshold is used and omits the subscript *rel* unless stated otherwise.

The collection of frequent itemsets in D with respect to σ is defined as:

$$F(D, \sigma) := \{X \subseteq I \mid support(X,D) \geq \sigma\}$$

The task of frequent itemset mining is to find the set of itemsets F.

Associative Rule Mining

Association rule is the form of $X \rightarrow Y$ where X is the rule antecedent a while Y is the rule consequent. Thus, {diaper}→{beer} means that customer who buys diapers tends to buy beer as well.

Generating association rules involves two steps: (1) generate the frequent k-itemsets (i.e., frequent itemset mining), and (2) for each frequent k-itemset, generate all rules using the items in the k-itemset that meet the minimum specified confidence (i.e., associative rule generation). The confidence of the association rule $A \rightarrow B$ is defined as the ratio of the number of transactions that include all items in the antecedent and consequent to the number of transactions that include all items in the antecedent only:

$$Confidence(A \rightarrow B, D) = \frac{support(A \cup B, D)}{support(A, D)}$$

The rule is confident if its confidence exceeds a given minimal confidence threshold γ, with $0 \leq \gamma \leq 1$.

An example is given to illustrate these definitions. Table 1 shows an example of a transaction database. Assume that set of items I = {a,b,c,d}

Table 1. An Example of a Transaction Database

tid	***J***
100	{a, b, d}
200	{a, b}
300	{c, d}
400	{b, c}

Table 2 shows all frequent itemsets in D with σ_{abs} of at least 1 or, equivalently, σ_{rel} of at least

Table 2. Itemsets and Their Support

Itemset	Cover	Support (*σabs*)	Frequency (*σrel*)
{}	{100,200,300,400}	4	100%
{a}	{100,200}	2	50%
{b}	{100,200,400}	3	75%
{c}	{300,400}	2	50%
{d}	{100,300}	2	50%
{a,b}	{100,200}	2	50%
{a,d}	{100}	1	25%
{b,c}	{400}	1	25%
{b,d}	{100}	1	25%
{c,d}	{300}	1	25%
{a,b,d}	{100}	1	25%

Table 3. Association Rules and Their Support and Confidence

Rule	Support (σabs)	Frequency (σrel)	Confidence
a→b	2	50%	100%
a→d	1	25%	50%
b→a	2	50%	66%
c→b	1	25%	50%
c→d	1	25%	50%
d→a	1	25%	50%
d→b	1	25%	50%
d→c	1	25%	50%
{a,b} →d	1	25%	50%
{a,d} →b	1	25%	100%
{b,d} →a	1	25%	100%
a→{b,d}	1	25%	50%
d→{a,b}	1	25%	50%

25%. Table 3 shows all frequent and confident association rules with σ_{abs} of at least 1 and γ of at least 50%.

Database Layout

The transactional database, which is stored in the secondary storage, can be arranged either as horizontal layout or vertical layout. For the horizontal layout, each row is composed of a transaction identifier and its corresponding items. Thus, each row is equivalent to one transaction. Table 1 illustrates an example of a horizontal layout. For the vertical layout, each row is composed of an item and its corresponding cover. Thus, each row contains complete transaction information of an item. Table 4 illustrates an example of a vertical layout.

Table 4. Example of a Vertical Data Layout

J	*Cover*
a	100, 400
b	100, 200, 400
c	300, 400
d	100, 300

CLASSICAL FIM ALGORITHMS

This section discusses the three classical FIM algorithms on which majority of the existing algorithms are based. These are Apriori (Agrawal & Srikant, 1994), ECLAT algorithm (Zaki, Parthasarathy, Ogihara, & Li, 1997), and FP-growth algorithm (Han, Pei, & Yin, 2000).

Apriori Algorithm

The concept of frequent itemset mining and association rule mining was first proposed by Agrawal, Imielinski and Swami (1993). In this landmark paper, the authors analyzed the items purchased by the customers in a supermarket (i.e., market basket analysis) in order to correlate the buying behavior of the customers. The algorithm, known as AIS, was improved by Agrawal and Srikant (1994) and later renamed as Apriori. The algorithm introduces the concept of downward closure property known as *Apriori*. This anti-monotonicity property states that a k-itemset is frequent if and only if all of its subsets (i.e., k-1) are frequent. The same technique was separately proposed by Mannila, Toivonen, and Verkamo (1994). The pseudocode of Apriori algorithm (Goethals, 2003) is listed in figure 1.

```
Input: D, σ
Output: F(D, σ)
Method: Apriori(D, σ)
1: C1 := { {i} | i ∈ I }
2: k := 1
3: while Ck ≠ { } do
4:   // Compute the supports of all candidate itemsets
5:   for all transactions (tid , I) ∈ D do
6:     for all candidate itemsets X ∈ Ck do
7:       if X ⊆ I then
8:         X.support++
9:       end if
10:    end for
11: end for
12: // Extract all frequent itemsets
13: Fk := { X | X.support ≥ σ }
14: // Generate new candidate itemsets
15: for all X, Y ∈ Fk, X[i] = Y [i] for 1 ≤ i ≤ k − 1, and X[k] < Y [k] do
16:   I = X ∪ { Y[k] }
17:   if ∀J ⊂ I, |J| = k : J ∈ Fk then
18:     Ck+1 := Ck+1 ∪ I
19:   end if
20: end for
21: k++
22: end while
```

Figure 1. Pseudocode of Apriori Algorithm

Initially, all items in the database serve as initial candidate k-itemsets (line 1). The database, which is stored in horizontal layout, is scanned one transaction at a time and the support counts of the candidate itemsets are incremented (lines 6-10). Based from the input σ, the frequent itemsets are then extracted (line 13). From these k-itemsets, the candidate (k+1)-itemsets are then generated (line 15-18) and D is rescanned again to tally the support counts. The process is repeated until there are no k-itemsets that can be generated anymore (line 3). Thus, the number of candidate k-itemsets generated is reduced since not all combinations of k-itemsets are considered.

Because of the breadth-first search technique, the database needs to be constantly rescanned to tally the support counts. This negates the advantage of having a reduced search space. Also, if σ is set too low, the number of frequent itemsets generated can be very large. There have been many variants to improve the algorithm either by further reducing the candidate itemsets or by minimizing database scan. In dynamic hashing and pruning (DHP) (Park, Chen, & Yu, 1995), information regarding (k+1)-itemsets is gathered while performing support count of k-itemsets. This information, which is stored in a hash table, contains counters to represent how many itemsets have been hashed so far. If the counter of the candidate (k+1)-itemset is below σ, then it is not generated. Thus, fewer candidates are generated. But this comes with a significant overhead of creating and maintaining a separate hash table.

In dynamic itemset counting (DIC) (Brin, Motwani, Ullman, & Tsur, 1997), the database is divided into intervals of specific size to minimize the number of database scan. Candidate itemsets are generated and counted at every interval. But the performance is heavily dependent on the heterogeneity of the data.

Another improvement known as sampling technique (Toivonen, 1996) performs at most two scans to the database. During the first scan, random samples are obtained and frequent itemsets are generated and verified with the rest of the database. If the samples failed to generate all frequent itemsets, a second pass to the database is performed to generate the rest of the frequent itemsets. To minimize the sampling failure, the value of σ can be decreased gradually. But a slight decrease in σ can cause large amount of candidate itemsets to be generated.

In the partitioning technique (Savasere, Omiecinski, & Navathe, 1995), the database is divided into multiple non-overlapping partitions and the frequent itemsets mining is done in two phases. In the first phase, each partition scanned the database and generates its own local frequent itemsets. Since the local frequent itemsets may not be frequent in relation to the entire database, they are merged together in preparation for the next stage. In the second stage, the merged itemsets now serve as candidate itemsets. Their support counts are tallied against the entire database and the frequent itemsets are then generated. A unique feature of this algorithm is the use of vertical layout for the database. The data of k-itemsets are read from the secondary storage and stored in the main memory. The support counts of candidate itemsets are computed by intersecting the covers of the corresponding k-itemsets. This operation performs faster than the scan-and-tally operation of horizontal layout. There is a possibility though that during the generation of local frequent itemsets, the covers of all local candidate k-itemsets cannot fit the main memory. Also, it is highly dependent on the distribution of data in the database and may cause too many frequent itemsets to be generated in one partition.

ECLAT Algorithm

The equivalence class clustering and bottom-up lattice traversal (*ECLAT*) algorithm (Zaki, Parthasarathy, Ogihara, & Li, 1997) uses depth-first search and vertical database layout to generate frequent itemsets. Using depth-first strategy minimizes the number of candidate *k*-itemsets generated as compared to the partitioning technique. The pseudocode of ECLAT algorithm (Goethals, 2003) is listed in figure 2.

The candidate *k*-itemsets are generated by intersecting two (*k*-1) frequent itemsets (lines 6-7) and the support counts are calculated and determine if it is frequent (line 8). The vertical database layout is then updated to reflect the updated frequent itemsets (line 9). The process is repeated using depth-first recursion (lines 13-14) until all the itemsets are covered. Since this algorithm does not use the anti-monotonicity property, the candidate itemsets generated are larger than that of Apriori.

As compared to Apriori algorithm, the database needs only to be scanned once since the vertical layout carries the complete information required for support counting. The computation of support count is faster since it is done by intersecting the covers of the two *k*-itemsets. But, this is offset by the need for more memory space since it needs to maintain a separate vertical database layout. Also, depth-first search prevents the implementation of algorithm in parallel.

FP-Growth Algorithm

Both the Apriori and ECLAT algorithms uses the methodology known as candidate generation and support counting. Another algorithm proposed by Han, Pei and Yin (2000) generates frequent itemsets without the need for candidate generation. The frequent-pattern (FP)-growth algorithm instead generates a FP-growth tree, which stores the compressed version of the database. The frequent itemsets are then generated based from the FP-growth tree using a method known as FP-growth pattern. The pseudocode of FP-growth algorithm (Goethals, 2003) is listed in figure 3.

```
Input: D, σ, I
Output: F[I](D, σ)
Method: ECLAT(D, σ, I)
1: F[I] := {}
2: for all i occurring in D do
3:   F[I] := F[I] ∪ {I ∪ {i}}
4:   // Create D^i
5:   D^i := {}
6:   for all j ∈ I occurring in D such that j > i do
7:     C := cover({i}) ∩cover({j})
8:     if |C| ≥ σ then
9:       D^i := D^i ∪ {(j,C)}
10:    end if
11: end for
12:  // Depth-first recursion
13:  Compute F[I ∪ {i}](D^i, σ)
14:  F[I] := F[I] ∪ F[I ∪ {i}]
15: end for
```

Figure 2. Pseudocode of ECLAT Algorithm

```
Input: D, σ, I
Output: F[I](D, σ)
Method: FPgrowth(D, σ, I)
1: F[I] := {}
2: for all i occurring in D do
3:   F[I] := F[I] ∪ {I ∪ {i}}
4:   // Create D^i
5:   D^i := {}
6:   H := {}
7:   for all j ∈ I occurring in D such that j > i do
8:     if support(I ∪ {i,j}) ≥ σ then
9:       H := H ∪ {j}
10:    end if
11: end for
12: for all (tid, X) with i do
13:   D^i := D^i ∪ {(tid, X ∩ H)}
14: end for
15:   // Depth-first recursion
16:   Compute F[I ∪ {i}](D^i, σ)
17:   F[I] := F[I] ∪ F[I ∪ {i}]
18: end for
```

Figure 3. Pseudocode of FP-growth Algorithm

This algorithm only needs to scan the database twice to generate the FP-growth tree. During the first scan, the database is scanned to obtain the 1-itemsets. These are then sorted in frequency-descending order (line 3). During the second scan, the items in each transaction are processed based on the sorted 1-itemset order and the FP-growth tree is then built based on the 1-itemset and the transactions of the database (lines 7-9). A separate header structure is also created (line 5) which contains support counts and links to the initial 1-itemset. Once the tree is created, the frequent itemsets are then generated by recursively building a sub-tree starting from the initial 1-itemset with the lowest frequency (lines 16-17). The sub-tree consists of set of prefix paths co-occurring with the suffix pattern. Thus this algorithm is composed of two steps. These are FP-tree generation and frequent itemset generation through FP-growth method.

The advantage of FP-growth algorithm over Apriori and ECLAT is that there is no need for candidate generation and support count. Also, the database is converted to a more compact data structure, which contains the complete information, thus eliminating repeated database scans. But the compact data structure is also a complex data structure composed of linked lists with each node consisting of a counter, a pointer to the next node, a pointer to its branches, and a point to the parent node. Adding to the complexity is that using pointers requires a lot of dereferencing. Whereas for ECLAT, the vertical data layout can only be implemented using a simpler array data structure.

Several variations to this algorithm include using depth-first method (Agarwal, Aggarwal, & Prasad, 2000) and using an array-based instead of linked list implementation of the FP tree (Grahne & Zhu, 2003).

CURRENT SERIAL FIM ALGORITHMS

Several algorithms have been proposed in recent years to further improve the frequent itemset mining. These range from improving the data structure to limiting the search space. The following subsections provide an overview of the recent work of these algorithms.

Improvement of Data Structure

Most of the improvements in frequent itemset mining are focused on FP-growth algorithm and vertical data layout. This is due to the advantage of not having to generate candidate itemsets as well as using intersection of covers instead of scan-and-tally method, which results in faster execution. One algorithm combining these strong points was proposed by Deng and Wang (2010). A new data structure called *Node-list* was proposed to improve on the FP-growth tree. It is based on the PPC-tree data structure, which is a prefix-based tree to store the compressed version of the database. The PPC-tree added preorder and postorder traversal sequence information of each node. This eliminates the need to maintain a separate header table structure. Each node N consists of five fields: *N.name*, *N.count*, *N.child*, *N.preorder* and *N.postorder*. Compare this with a node in the FP-tree that contains more than five fields. The pseudocode in creating the PPC-tree is listed in figure 4 (Deng & Wang, 2010).

Before the PPC tree is generated, all 1-itemsets are first generated. Those itemsets that are frequent are retained and sorted in support descending order (lines 1-3). The PPC-tree is then generated by first creating a root node that is labelled as "null" (line 5). The leaf nodes are then created based on the support order of the frequent itemsets (lines 6-9, 13-21). Once the PPC tree is generated,

Input: A transaction database *D* and a minimum support σ
Output: A PPC-tree and *F*1 (the set of frequent 1-itemsets).
Method: Construct-PPC-tree(*D*, σ)
1: //Frequent 1-itemsets generation
2: According to σ , scan *DB* once to find *F*1, the set of frequent 1-itemsets and their supports.
3: Sort *F*1 in support descending order as *L*1, which is the list of ordered frequent items.
4: //PPC-tree construction
5: Create the root of a PPC-tree, *Tr* , and label it as "null".
6: for each transaction *Trans* in DB do
7: Select the frequent items in *Trans* and sort out them according to the order of *F*1. Let the sorted frequent-item list in *Trans* be [*p*|*P*], where *p* is the first element and *P* is the remaining list.
8: Call *insert tree*([*p*|*P*],*Tr*).
9: end for
10: //Pre-Post code generation
11: Scan PPC-tree to generate the *pre-order* and the *post-order* of each node.
12: //Function *insert tree*([*p*|*P*],*Tr*)
13: if *Tr* has a child *N* such that *N.item-name* = *p.item-name* then
14: increase *N*'s *count* by 1;
15: else
16: create a new node *N*, with its count initialized to 1, and add it to T_r*'s* children-list;
17: if P is nonempty then
18: call *insert tree*(*P*,*N*) recursively.
19: end if
20: end if

Figure 4. Pseudocode of *PPC*-Tree Construction

the tree is traverse twice to generate the preorder and postorder traversal information (lines 10-11). The *PP* code of each node *N* in the PPC tree is then created. The structure of the *PP* code is in in the form of <(*N.preorder*, *N.postorder*): *N.count*>. The *Node-list* is then created based from the *PP* code. A *Node-list* contains the sequence of all *PP* codes of a particular 1-itemset node in the PPC tree. Thus, a *Node-list* is denoted by {<($N.preorder_1$, $N.postorder_1$): $N.count_1$>, <($N.preorder_2$, $N.postorder_2$): $N.count_2$>,..., <($N.preorder_n$, $N.postorder_n$): $N.count_n$>}. These 1-itemsets are sorted in support ascending rank, while the *PP* codes within the *Node-list* are sorted in preorder ascending rank. Once the *Node-lists* of all 1-itemsets are generated, the PPC tree can then be deleted to free up memory space. The PPV algorithm is then used for frequent itemset mining. The pseudocode of PPV algorithm (Deng & Wang, 2010) is listed in Figure 5.

The *Node-list* of a candidate *k*-itemset is generated by finding the ancestor-descendant relationship of the *Node-lists* of two (*k*-1)-itemsets. The *Node-list* of the candidate *k*-itemset is the descendant *Node-list* of the two (*k*-1)-itemsets. The support count of the candidate *k*-itemset is computed by adding the support count of each *PP*-code in the *Node-list* and those support count below σ are removed. This process is repeated until all frequent itemsets are found.

The algorithm benefitted from the PPC-tree data structure as well as the intersection-based approach of vertical data layout. But, it still inherits the weakness of candidate generation-and-test of Apriori-like methods. The PrePost algorithm proposed by Wang, Jiang and Deng (2012) eliminates the need for candidate generation. A new data structure called *N-list* is proposed in this algorithm. The generation of *N-list* is similar to that of the *Node-list* except that the *N-lists* of the candidate *k*-itemsets are based from the ancestor *N-list* instead. Thus, the length of the *N-list* is shorter than *Node-list*. This makes *N-list* more compact than *Node-list*. Also, *N-list* has a property known as single-path property, which allows the generation of frequent itemsets without generating the equivalent candidate itemset. The PrePost algorithm is used to generate frequent itemsets using the *N-list* data structure. The pseudocode of PrePost algorithm (Wang, Jiang, & Deng, 2012) is listed in Figure 6.

Both *Node-list* and *N-list* store preorder and postorder information, which result in higher memory consumption. Another approach proposed by Lv and Deng (2014) is to store either preorder or postorder traversal information only. The new data structure is known as *Node-set*. The number of fields in each node is now reduced to four, which translates to lesser memory requirements. The *Node-set* is generated based from POC tree. The POC tree is similar to PPC tree except only the preorder traversal information is stored. The pseudocode in creating the POC tree (Lv & Deng, 2014) is listed in Figure 7.

Once the POC tree is generated, the tree is traverse once to generate either the preorder or postorder information. The *N-info* of each node *N* in the POC tree is then created. The structure of *N-info* is in in the form of (*N.preorder*, *N.count*). The *Node-set* is then created based from the *N-info*. A *Node-set* contains the sequence of all *N-infos* of a particular 1-itemset in the POC tree. Thus, a *Node-set* is denoted by {($N.preorder_1$, $N.count_1$), ($N.preorder_2$, $N.count_2$),..., ($N.preorder_n$, $N.count_n$) }. These 1-itemsets are sorted in support- ascending rank, while the *N-infos* within the *Node-set* are sorted in preorder ascending order. Once the *Node-sets* of all 1-itemsets are generated, the POC tree can then be deleted to free up memory usage. The FIN algorithm is then used to generate the frequent itemsets. The pseudocode of FIN algorithm (Lv & Deng, 2014) is listed in Figure 8.

```
Input: the threshold σ, the frequent 1-patterns L_1 and their Node-lists NL_1
Output: The complete set of frequent patterns.
Method: PPV (σ, L_1, NL_1)
1: L_1 = {frequent 1-patterns};
2: NL_1 = {the Node-lists of L_1};
3: For ( k = 2; L_{k-1} ≠ Ø; k++) do begin {
4:     For all p∈L_{k-1} and q∈L_{k-1}, where p.i_1 = q.i_1,..., p.i_{k-2} = q.i_{k-2}, p.i_{k-1} q.if_{k-1} do begin {
5:             l = p.i_1, p.i_2,...,p.i_{k-1},q.i_{k-1}; // Candidate k-pattern
6:             If all k-1 subsets l of are in L_{k-1} {
7:                l.Node-list = code-intersection(p.Node-list, q.Node-list);
8:                If (l.count ≥|DB|×ξ) { // Use Property 5 to get l.count from l.Node-list
9:                       L_k = L_k ∪{l};
10:                      NL_k = NL_k ∪ {l.Node-list}; }
11:            }
12:     end For; }
13:     Delete NL_{k-1};
14: end For; }
15: Answer = ∪_k L_k;
```

Figure 5. Pseudocode of PPV Algorithm

```
Input: the minimum support σ, the frequent 1-itemsets L1 and their N-lists NL1
Output: The frequent itemset F
Method: PrePost(L1, NL1)
1: for i ← L_k.size() − 1 to 1 do
2:    L_k^i+1 ← Ø;
3:    NL_k^i+1 ← Ø;
4:    for j ← i − 1 to 0 do
5:          Assume L_k[i] = x_1x_2 … x_k and L_k[j] = yx_2 … x_k(y > x_1 > x_2> · · · >x_k)
6:          l ← yx_1x_2 … xk;
7:          l.N − list ← NL_intersection(NL_k [i],NL_k[j]);
8:          if l.count ≥ |DB| × σ then
9:             L_k^i+1 ← L_k^i+1 ∪ {l};
10:            F ← F ∪ {l};
11:            NL_k^i+1 ← NL_k^i+1∪ {l.N − list};
12:         end if
13:   end for
14:   if L_k^i+1 ≠ 0 then
15:         if NL_k[i].length() = 1 then
16:            Assume L_k^i+1= {P_1, . . . , P_n} where P_i = y_ix_1x_2 … x_k
17:            for any p = y_v1y_v2… y_vux_1x_2 … x_k(1 ≤ v_1 < v_2 < … < v_u ≤ n) do
18:               p.count ← NL_k[i].count;
19:               F ← F ∪ {p};
20:            end for
21:         else
22:            Call PrePost(L_{k+1}^i,NL_{k+1}^i);
23:         end if
24:   end if
25: end for
```

Figure 6. Pseudocode of PrePost Algorithm

Input: A transaction database *D* and a minimum support σ
Output: A POC-tree and *F*1 (the set of frequent 1-itemsets)
Method: Construct-POC-tree(*D*, σ)
1: [Frequent 1-itemsets Generation]
According to σ, scan *D* once to find *F*1, the set of frequent 1-itemsets, and their supports. Sort *F*1 in support descending order as *L*1, which is the list of ordered frequent items.
2: [POC-tree Construction]
Create the root of a POC-tree, *Tr*, and label it as ''null''. For each transaction Trans in *D* do the following:

- Select the frequent items in Trans and sort them according to the order of *F*1. Let the sorted frequent-item list in Trans be [*p* | *P*], where *p* is the first element and *P* is the remaining list. Call insert_tree ([*p* | *P*], *Tr*).
- The function insert_tree([*p* | *P*], *Tr*) is performed as follows:
    - If *Tr* has a child *N* such that *N*.item-name = p.item-name, then increase *N*'s count by 1;
    - else create a new node *N*, with its count initialized to 1, and add it to *Tr*'s children-list. If *P* is nonempty, call insert_tree(P, N) recursively.

3: [Pre-code Generation]
Scan the *POC*-tree to generate the pre (or post) -order of each node by the pre (or post)-order traversal.

Figure 7. Pseudocode of POC-Tree Construction

Input: A transaction database *D* and a minimum support σ
Output: *F*, the set of all frequent itemsets
Method: FIN(*D,* σ)
1: $F \leftarrow \emptyset$;
2: Construct the POC-tree and find *F*1, the set of all frequent 1-itemset;
3: $F_2 \leftarrow \emptyset$;
4: Scan the POC-tree by the pre-order traversal do
5: $N \leftarrow$ currently visiting Node;
6: $i_y \leftarrow$ the item registered in *N*;
7: For each ancestor of *N*, N_a, do
8: $i_x \leftarrow$ the item registered in *N*a;
9: If $i_x i_y \in F_2$, then
10: $i_x i_y$.support $\leftarrow i_x i_y$.support + *N*.account;
11: Else
12: $i_x i_y$.support $\leftarrow$ *N*.account;
13: $F_2 \leftarrow F_2 \cup \{i_x i_y\}$;
14: Endif
15: Endfor
16: For each itemset, *P*, in F_2 do
17: If *P*.support < σ x |*D*|, then
18: $F_2 \leftarrow F_2$ - {*P*};
19: Else
20: *P*.Nodeset $\leftarrow \emptyset$;
21: Endif
22: Endfor
23: Scan the POC-tree by the pre-order traversal do
24: *Nd* $\leftarrow$ currently visiting Node;
25: $i_y \leftarrow$ the item registered in *Nd*;
26: For each ancestor of *Nd*, Nd_a, do
27: $i_x \leftarrow$ the item registered in Nd_a;
28: If $i_x i_y \in F_2$, then
29: $i_x i_y$.Nodeset $\leftarrow i_x i_y$.*Nodeset* $\cup$ *Nd*.*N_info*;
30: Endif
31: Endfor
32: $F \leftarrow F \cup F_1$;
33: For each frequent itemset, $i_s i_t$, in F_2 do
34: Create the root of a tree, R_{st}, and label it by $i_s i_t$;
35: Constructing_Pattern_Tree(R_{st}, $\{i \mid i \in F_1, i > i_t\}$, $\emptyset$);
36: Endfor
37: Return *F*;

Figure 8. Pseudocode of *FIN* Algorithm

The *Node-set* of a candidate *k*-itemset is generated by finding the ancestor-descendant relationship of the *Node-sets* of two (*k*-1)-itemsets. The *Node-set* of the candidate *k*-itemset is the descendant *Node-set* of the two (*k*-1)-itemsets. The support count of the *k*-itemset is computed by adding the support count of the *Node-set*. The process is repeated until all frequent itemsets are found.

To limit the search space of the data, the set-enumeration search tree (Rymon, 1992) is used. This tree generated is based from the *Node-sets* created.

Limiting Search Space

Another approach to improve the frequent itemset mining is to limit the search space by placing additional constraints. Some of the methods are closed frequent itemset (CFI), maximally frequent itemset (MFI) and top-rank-*k* frequent itemset (TRI).

A *k*-itemset β is closed frequent if β is frequent in *D* and there exists no proper superset α such that α has the same support count as β in *D*. On the other hand, a *k*-itemset β is maximal frequent if β is frequent in *D* and no proper superset α is frequent in *D*. Thus, the relationship is maximal frequent itemset ⊆ closed frequent itemset ⊆ frequent itemset.

Top-rank-*k* frequent itemset is defined as follows. The rank R_x of a frequent itemset *X* is defined as $R_x = |\{ \sigma_y \mid Y \subseteq I \text{ and } \sigma_y \geq \sigma_x\}|$, where $|Y|$ is the number of items in *Y*. Given the database *D* and a threshold of *k*, the top-rank-*k* frequent itemsets are the set of frequent itemsets whose ranks are no greater than *k* (i.e., $R_x \leq k$). This means that if a frequent itemset has a higher support count, its rank is higher.

A-Close (Pasquer, Bastide, Taouil, & Lakhal, 1999), which is based on the Apriori algorithm, is one of the earliest algorithms to limit the search space by using CFI. Other related works include CLOSET (Pei, Han, & Mao, 2000) and CLOSET+ (Wang, Pei, & Han, 2003). The FPClose (Grahne & Zhu, 2003) algorithm is based on the FP-growth approach. There are some algorithms such as CHARM (Zaki & Hsiao, 2002) and AFOPT (Liu, Lu, Lou, & Yu, 2003) that use a hybrid approach. The NAFCP algorithm (Le & Vo, 2015) is a recent work on CFI, which is based on the *N-list* data structure. The algorithm is similar to the PrePost algorithm but uses CFI instead.

MAFIA (Burdick, Calimlim, Flannick, Gehrke, & Yiu, 2005) is a pioneer algorithm that is based on MFI. It uses vertical bitset representation to compute for support count. An additional constraint to maximal frequent itemset was introduced by the MWFIM algorithm (Yun, Shin, Ryu, & Yoon, 2012). The algorithm added a weight constraint to each item. The weight of an item is a non-negative number that is assigned to reflect the importance of each item in the database. This algorithm employs the same vertical bitmap representation as MAFIA to compute for support count.

The *FAE* ("Filtering and Extending") algorithm is one of the first algorithms to use top-rank-*k* frequent itemset (Deng & Fan, 2007) and is based on the Apriori algorithm. The *NTK* algorithm (Deng Z., 2014) combines *Node-list* data structure and PPV algorithm with top-rank-*k* frequent itemsets. On the other hand, the *iNTK* algorithm (Huynh-Thi-Le, Le, Vo, & Le, 2015) uses *N-list* and subsume index (Song, Yang, & Xu, 2008) instead to mine top-rank-*k* frequent itemsets. The algorithm is thus similar to PrePost algorithm, except for the constraint search space.

Putting the constraint search space in perspective, the improvement factor is due to fewer candidate itemsets generated, which translates to lesser execution time. But this is offset by the concern that there might be some important information that may be left out.

Any existing serial FIM algorithms can be modified to use with constraint search space.

CURRENT PARALLEL IMPLEMENTATIONS OF FIM ALGORITHMS

In recent years, there have been many developments in parallel implementations of FIM algorithm to take advantage of the improvement of processor and network technology. The availability of *OpenMP* library for multi-core processor and cloud computing and Nvidia *CUDA* and ATI *OpenCL* library for GPU have make it easier to implement algorithms in parallel paradigm. The following subsections provide an overview of the recent works in parallel implementation of FIM algorithms.

GPU Computing

One the earliest FIM implementation to utilized GPU computing was proposed by Fang, Lu, Xiao, He, and Luo (2009) and is based on the Apriori algorithm. It uses vertical data layout instead and is implemented as bitset representation. Since the Apriori algorithm uses breadth-the first search method, each candidate generation and support count can be viewed as one thread and can be processed in parallel.

The PBI algorithm is a pure GPU implementation for performing both candidate generation and support counting in the GPU. The pseudocode of PBI algorithm (Fang, Lu, Xiao, He, & Luo, 2009) is listed in Figure 9.

Since the vertical bitset representation is viewed as an array of bits, a candidate k-itemset is generated by performing a bitwise OR operation of two (k-1) itemsets (lines 4-6) while the corresponding transaction is obtained by performing a bitwise AND operation. The support count is done by performing a binary search on a lookup table. The lookup table contains the mapping of an integer and the number of 1s in its binary equivalent.

Pure GPU implementation eliminates the need to transfer data from the main memory to GPU memory. But if the number of items is large, then it will generate a significant overhead in accessing the lookup table. As an alternative, the TBI algorithm implemented using CPU-GPU combination. This algorithm uses trie data structure to represent the data set. It uses CPU to generate candidate itemsets and GPU to perform parallel support count. The pseudocode of TBI algorithm (Fang, Lu, Xiao, He, & Luo, 2009) is listed in Figure 10.

Another *Apriori*-based GPU algorithm was proposed by Zhang, Zhang and Bakos (2011). The GPApriori preprocesses the dataset and stores it as vertical bitset representation. Candidate generation is done by the CPU while the support count is performed in the GPU by intersecting the bitset and the results are copied back to the main memory. The process is almost identical to the PBI algorithm except for the process involving support count.

ECLAT-based GPU implementation of GPU was proposed by Zhang, Zhang, and Bakos (2013). The Frontier Expansion algorithm preprocesses the dataset stores as vertical bitset representation using stack. The process of candidate itemsets generation is done by the CPU while the support count is done in parallel by the GPU. The pseudocode of the Frontier Expansion algorithm (Zhang, Zhang, & Bakos, 2013) is listed in Figure 11.

Most of the current implementations of GPU computing are based on the Apriori or ECLAT algorithm. This is due to the fact that candidate generation and support count are two independent tasks that can be executed in parallel. Also, data are preprocessed and store as vertical bitset structure instead of the traditional horizontal structure. Vertical bitset structure can take advantage of the

Input: L_x represents the x-th (K-1)-itemset (i.e, the x-th row vector in the bitmap for (K-1)-itemsets)
Output: the set of all frequent itemsets
Method: PBI(L_x)
1: for each L_i in parallel do
2: for each L_j where $j = i + 1$ to m do
3: if L_i and Lj are joinable then
4: //Join
5: Union on L_i and L_j to obtain a candidate k-itemset by performing a bitwise OR operation
6: //Pruning
7: (K - 1)-subset test on the candidate k-itemset by a binary search in the (K-1)-itemset bitmap.
8: else
9: break
10: end if
11: end for
12: end for

Figure 9. Pseudocode of PBI Algorithm

Input: u represents a node at depth K-1 in the trie
Output: the set of all frequent itemsets
Method: TBI(u)
1: for each u at depth K - 1 do
2: for each w that is a right sibling of u do
3: //Join
4: Union on the two (k - 1)-itemsets represented by u and w to obtain a candidate k-itemset
5: //Pruning
6: (k - 1)-subset test on the candidate K-itemset by following the path of the trie with the same prefix
7: end for
8: end for

Figure 10. Pseudocode of TBI Algorithm

Input: frontier_stack, ε, σ
Output: the set of all frequent itemsets
Method: Frontier(frontier_stack, ε, σ)
1: if *frontier_stack* is empty
2: return false
3: *expansion_size* = 0
4: *frequent_itemset* = $\emptyset$
5: while *expansion_size* < ε
6: pop equivalent class *s_eqv* from stack
7: *t_eqv* = expand(*s_eqv*)
8: *expansion_size* = *expansion_size* + size(*t_eqv*)
9: support_count(*t_eqv*)
10: remove infrequent nodes from *t_eqv*
11: add *t_eqv* to *frequent_itemset*
12: push *t_eqv* to *frontier_stack*
13: return true

Figure 11. Pseudocode of Frontier Expansion Algorithm

built-in *popcount* and *intersect* operation of the GPU hardware to process support count.

Multi-Core Processor

One of the earliest works involving multi-core processor was proposed by Liu, Li, Zhang, and Tang (2007) and is based on the FP-growth algorithm. Since, the strength of multi-core micro-architecture is in the cache structure, the algorithm uses array data structure to make it cache conscious. Also the data are rearrange in such a way the data are temporal locality aware in order to maximize cache access.

The SHAFEM algorithm (Lu & Alaghband, 2014) uses a different approach. It dynamically chose between two algorithms to handle sparse and dense database. Both algorithms are based on FP-growth algorithm. The MineFPTree algorithm uses *XFP* tree data structure to handle sparse data set. *XFP* tree are composed of multiple local FP-growth trees that are merged together. On the other hand, MineBitVector converts the XFP tree data structure to vertical bitset data structure to handle dense data sets. The pseudocode of the MineFPTree algorithm is listed in Figure 12 while the pseudocode of MineBitVector algorithm is listed in Figure 13.

In multi-core implementation, the focus is to make the data more cache conscious. The idea is to limit the communication between tasks by having all related data fetched in the same cache line.

Distributed Computing

One the earliest FIM algorithms to utilized distributed computing was proposed by Yu and Zhou (2008), which is based on the FP-growth algorithm. The algorithm known as tidset-based parallel FP-tree algorithm divides the database into a number of partitions depending on the available nodes in the cluster. Each local computer node creates a vertical bitset representation of its partition as well as the corresponding 1-itemsets and returns the results back to the main node. The main node then creates a global header table and distributes the mining set information equally among the nodes. Each node then creates its local FP-growth tree and mines the frequent pattern. The main node then collects all result from all the nodes and integrates the frequent itemsets.

```
Input: FP-tree T, suffix, minsup
Output: the set of all frequent itemsets
Method: MineFPTree(FP-tree T, suffix, minsup)
1: If T contains a single path P then
2:       For each combination x of the items in P
3:          Output β = x U suffix
4:          Compute and update threshold K_i
5: Else For each item α in the header table for FP-tree T
6:          Output β = K_i U suffix
7:          Size = the size of α's conditional pattern base
8:          Compute and update threshold K_i
9:          If Size > K_i then
10:             Construct α's conditional FP-tree T'
11:             Call MineFPTree(T', β, minsup)
12:         else
13:             Construct α's private bit vectors V and w
14:             Call MineBitVector(V,w, β, minsup)
15:         Endif
16: Endif
```

Figure 12. Pseudocode of MineFPTree Algorithm

```
Input: vectors V, vec.w, suffix, minsup
Output: the set of all frequent itemsets
Method: MineBitVector(vectors V, vec.w, suffix, minsup)
1: Sort V in support-descending order of their items
2:      For each vector v_k in V
3:          Output β = item of v_k ∪ suffix
4:          For each vector v_j in V with j < k
5:              u_j = v_k AND v_j
6:              sup_j = support of u_j computed using w
7:              If sup_j ≥ minsup then add u_j into U
8:          If all u_j in U are identical to v_k
9:          Then For each combination x of the items in U
10:             Output β' = x ∪ β
11:         Else if U is not empty
12:             Call MineBitVector(U, w, β ,minsup)
```

Figure 13. Pseudocode of MineBitVector Algorithm

An improved version of the algorithm known as balanced tidset-based parallel FP-tree algorithm was proposed by the same proponents (Zhou and Yu, 2008). Instead of dividing the mining set equally among the nodes, a performance index is collected among the nodes, which act as a load- balancing metric. This metric is then use to determine the amount of mining set workload to be sent to each node.

The algorithm proposed by Lin and Luo (2009) focused on data privacy in the distributed cloud computing environment. The FD-mine algorithm assigns a kernel or trusted node within the intranet to have full access to the database. Within each cloud, a connection node is assigned to communicate with the kernel node. If another node within the cloud needs data, it will coordinate with the connection node of its cloud. The algorithm is also based on FP-growth algorithm.

The FLR-mining algorithm proposed by Lin and Chung (2015) improved the FD-mine algorithm by adding load balancing to the algorithm. It iteratively estimates the workload based on the number of header items. The pseudocode of the FLR-mining algorithm is listed in figure 14.

Distributed computing is concerned with mainly with load balancing and data privacy. As the nodes are heterogeneous, the workload of each node is different. Also, once the data leaves the host computer and transfer to an other node, there is no guarantee on the security and privacy of that particular data.

FUTURE RESEARCH DIRECTIONS

With the volume of data reaching exa-scale level, there is a need to take advantage of all available computing resources. Improvement in computer hardware technology and its widespread availability has made parallel computing a viable option in implementing FIM algorithm.

Implementations of existing FIM algorithms have to take advantage of the parallel capabilities of the computing hardware or it will be useless. It requires a novel approach as it is not a mere one-to-one mapping approach. It requires programmers to have a good working knowledge of multi-core computing, GPU computing, cloud computing and other parallel computing paradigm in order to take full advantage of it.

```
Input: D, σ, CN
Output: FP the set of all frequent itemsets
Method: FLR_mine(D, σ, CN)
1: HT = getHT(D,S);
2: tree = buildFPtree (D,S);
3: t_t = getTreeTransmissionTime();
4: FP = Ø;
5: t_cn = 0;
6: t_n = 0;
7: t_avg = 0;
8: While (isCompleted(HT) == false)
9:    n = getAvailableNode(CN);
10:   transmitTree(tree,n);
11:   x = calculateNumOfHeaderItems(t_t, t_avg);
12:   hi = selectIIS(x,HT);
13:   HT = HT – hi;
14:   n.BeginMining(hi);
15:  End While
16: Return FP
```

Figure 14. Pseudocode of FLR-Mining Algorithm

Some of the challenges in parallel implementations include finding ways to look for independent tasks, communication issues between tasks load balancing, and defining tasks as multi-thread. Finding independent tasks in FIM is a challenge as both data and control dependencies are fundamental in FIM algorithms.

Most of the attempts to implement FIM algorithm in parallel are based either on Apriori or ECLAT algorithm. Both of these algorithms use generate-and-count methodology, which involves two independent tasks. Furthermore, vertical representation of dataset using bitset is preferred due to the availability of parallel join operation as primitive which allows support count to be computed in parallel.

But these types of algorithms need to generate candidate itemsets before coming up with frequent itemsets. This is opposed to tree-based projection algorithms such as FP growth and Pre-post, which generates frequent itemsets without the need for candidate itemsets. That is why most of the current algorithms are geared towards improving this type of algorithm. But tree-based projection algorithm is inherently difficult to implement in parallel as it uses pointer and recursion. Novel solutions are needed to solve the problems. These include changing data structure from pointer to array and tiling the program structure in such a way to make it cache-conscious.

Thus, the challenge of future research in FIM is come up with algorithms and implementations that will harness the full capability of current computer hardware technology in parallel processing. In multi-core processor, the focus should be on its cache structure. In GPU processor, its strength lies in multi-thread execution. While in distributed computing, its strength is in the availability of multiple computer nodes for processing. Other feature such as limiting the search space can be incorporated to further improve FIM operation performances.

REFERENCES

Agarwal, R. C., Aggarwal, C. C., & Prasad, V. (2000). Depth first generation of long patterns. *Proceedings of the sixth ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 108-118). New York.

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. *Proceedings of the 1994 international conference on very large data bases (VLDB'94)* (pp. 487-499). Santiago.

Agrawal, R., Imielinksi, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. *Proceedings of the 1993 ACM-SIGMOD international conference on management of data (SIGMOD '93)* (pp. 207-216). Washington D.C.

Armburst, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., . . . Zaharia, M. (2010). A view of cloud computing. *Communications of the ACM*, 50-58.

Barlas, G. (2015). *Multicore and GPU programming: An integrated approach*. Waltham, MA, USA: Morgan Kaufmann.

Brin, S., Motwani, R., Ullman, J. D., & Tsur, S. (1997). Dynamic itemset counting and implication rules for market basket data. *Proceedings of the 1997 ACM-SIGMOD international conference on management of data (SIGMOD '97)* (pp. 255-264). Tucson.

Brumfiel, G. (2011). High-energy physics: down the petabyte highway. *Nature*, 282-283.

Burdick, D., Calimlim, M., Flannick, J., Gehrke, J., & Yiu, T. (2005, November). MAFIA: A maximal frequent itemset algorithm. *IEEE transactions on knowledge and data engineering, 17*(11), 1490-1504.

Deng, Z. (2014). Fast mining Top-Rank-k frequent patterns using Node-lists. *Expert Systems with Applications, 41*, 1763-1768.

Deng, Z., & Fan, G.-D. (2007). Mining top-rank-k frequent patterns. *Proceedings of the Sixth International Conference on Machine Learning Cybernetics*, (pp. 851-856). Hong Kong.

Deng, Z., & Wang, Z. (2010, December). A new fast vertical method for mining frequent patterns. *International Journal of Computational Intelligence Systems, 3*(6), 733-744.

Fang, W., Lu, M., Xiao, X., He, B., & Luo, Q. (2009). Frequent itemset mining on graphics processors. *Proceedings of the 5th international workshop on data management on new hardware (DaMon'09)* (pp. 34-42). Providence.

Glaskowsky, P. N. (2009). *NVIDIA's Fermi: The first complete GPU computing architecture*. White paper, NVIDIA Corporation.

Goethals, B. (2003). *Survey on frequent pattern mining*. Helsinki.

Gorunescu, F. (2011). *Data mining: concepts, models and techniques, ISRL 12*. Berlin, Germany: Springer-Verlag.

Grahne, G., & Zhu, J. (2003). Efficiently using prefix-trees in mining frequent itemsets. *Proceeding of the ICDM'03 international workshop on frequent itemset mining implementations (FIMI'03)* (pp. 123-132). Melbourne, Florida.

Han, J., Pei, J., & Yin, Y. (2000). Mining frequent patterns without candidate generation. *Proceedings of the 2000 ACM-SIGMOD international conference on management of data (SIGMOD '00)* (pp. 1-12). Dallas.

Hu, K., Lu, Y., Zhou, L., & Shi, C. (1999). Integrating classification and association rule mining: A concept lattice framework. *Proceedings of the 7th international workshop on new directions in rough Sets, data mining, and granular-soft computing* (pp. 443-447). London: Springer-Verlag.

Huynh-Thi-Le, Q., Le, T., Vo, B., & Le, B. (2015, January). An efficient and effective algorithm for mining top-rank-k frequent patterns. *Expert Systems With Applications, 42*(1), 156-164.

Kosters, W. A., Marchiori, E., & Oerlemans, A. A. (1999). Mining clusters with association rules. *Proceedings of the 3rd international symposium on advances in intelligent data analysis* (pp. 39-50). London: Springer-Verlag.

Le, T., & Vo, B. (2015, May). An N-list-based algorithm for mining frequent closed patterns. *Expert Systems with Applications, 42*, 6648-6857.

Lin, K. W., & Luo, Y.-C. (2009). A fast parallel algorithm for discovering frequent patterns. *IEEE international conference on granular computing* (pp. 398-403). Nanchang.

Lin, K., & Chung, S.-H. (2015, May). A fast and resource efficient mining algorithm for

discovering frequent patterns in distributed computing environments. *Future Generation Computer Systems, 52*, 49-58.

Liu, G., Lu, H., Lou, W., & Yu, J. X. (2003). On computing, storing and querying frequent patterns. *Proceeding of the 2003 ACM SIGKDD international conference on knowledge discovery and data mining (KDD'03)* (pp. 607-612). Washington, DC.

Liu, L., Li, E., Zhang, Y., & Tang, Z. (2007). Optimization of frequent itemset mining on multiple-core processor. *Proceedings 33th international conference on very large data bases*, (pp. 1275-1285). Vienna.

Lu, V., & Alaghband, G. (2014, December). Novel parallel method for association rule mining on multi-core shared memory systems. *Parallel Computing, 40*(10), 768-785.

Lv, S.-L., & Deng, Z.-H. (2014). Fast mining frequent itemsets using Nodesets. *Expert Systems with Applications, 41*, 4505-4512.

Lynch, C. (2008). Big Data: How do your data grow? *Nature*, 28-29.

Mannila, H., Toivonen, H., & Verkamo, A. (1994). Efficient algorithms for discovering association rules. *Proceedings of the AAAI'94 workshop knowledge discovery in databases (KDD'94)* (pp. 181-192). Seattle.

Manyika, J., Chui, M., Brown, B., Baughin, J., Dobbs, R., Roxburgh, C., & Byers, A. (2012). *Big data: The next frontier for innovation, competition and productivity*. McKinsey Global Institute.

NVIDIA Corporation. (2015). *CUDA C programming guide*. Programming Guide, NVIDIA Corporation.

Park, J., Chen, M.-S., & Yu, P. S. (1995). An effective hash-based algorithm for mining association rules. *Proceedings of the 1995 ACM-SIGMOD international conference on management of data (SIGMOD '95)*, (pp. 175-186). San Jose.

Pasquer, N., Bastide, Y., Taouil, R., & Lakhal, L. (1999). Discovering frequent closed itemsets for association rules. *Proceedings of the 7th international conference on database theory (ICDT'99)* (pp. 398-416). Jerusalem.

Pei, J., Han, J., & Mao, R. (2000). CLOSET: an efficient algorithm for mining frequent closed itemsets. *Proceeding of the 2000 ACM-SIGMOD international workshop data mining and knowledge discovery (DMKD'00)* (pp. 11-20). Dallas.

Rymon, R. (1992). Search through systematic set enumeration. *International conference on priciples of knowledge representation and reasoning* (pp. 539-550).

Sarwar, B., Karypis, G., Konstan, J., & Riedl, J. (2000). Analysis of recommendation algorithms for e-commerce.*Proceedings of the 2nd ACM conference on electronic commerce* (pp. 158-167). Ney York: ACM Press.

Savasere, A., Omiecinski, E., & Navathe, S. (1995). An efficient algorithm for mining association rules in large databases. *Proceedings of the 1995 international conference on very large data bases (VLDB'95)* (pp. 432-444). Zurich.

Song, W., Yang, B., & Xu, Z. (2008, August). Index-BitTableFI: An improved algorithm for mining frequent itemsets. *Knowledge-Based Systems, 21*(6), 507-513.

Szalay, A., & Gray, J. (2006). Science in an exponential world. *Nature*, 23-24.

Toivonen, H. (1996). Sampling large databases for association rules. *Proceedings of the 1996 international conference on very large data bases (VLDB'96)* (pp. 134-145). Bombay.

Uy, R. (2014). Beyond multi-core: A survey of architectural innovations on microprocessor. *2014 international conference on humanoid, nanotechnology, information technology, communication and control, environment and management (HNICEM 2014)* (pp. 1-6). Palawan.

Wang, J., Pei, J., & Han, J. (2003). CLOSET+: Searching for the best strategies for mining frequent closed itemsets. *Proceeding of the 2003 ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 236-245). Washington D.C.

Wang, Z.-H., Jiang, J., & Deng, Z.-H. (2012). A new algorithm for fast mining frequent itemsets using N-lists. *Science China Information Services, 55*(9), 2008-2030.

Woon, Y., Ng, W., & Lim, E.-P. (2002). Online and incremental mining of separately-grouped web access logs. *Proceedings of the 3rd*

international conference on web information systems engineering (pp. 53-62). Washington D.C.: IEEE Computer Society.

Yu, K.-M., & Zhou, J. (2008). Tidset-based parallel FP-tree for the frequent pattern mining problem on PC cluster. In S. Wu, L. T. Yang, & T. L. Xu, *Lecture notes in computer science 5036* (pp. 18-28). Berlin: Springer-Verlag.

Yun, U., Shin, H., Ryu, K., & Yoon, E. (2012, February). An efficient mining algorithm for maximal weighted frequent patterns in transactional databases. *Knowledge-Based Systems, 33*, 53-64.

Zaki, M. J., & Hsiao, C.-J. (2002). CHARM: an efficient algorithm for closed itemset mining. *Proceeding of the 2002 SIAM international conference on data mining (SDM'02)* (pp. 457-473). Arlington.

Zaki, M., Parthasarathy, S., Ogihara, M., & Li, W. (1997). New algorithms for fast discovery of association rules. *Proceedings of 3rd international conference on knowledge discovery and data mining* (pp. 283-286). Newport Beach.

Zhang, F., Zhang, Y., & Bakos, J. (2011). GPApriori: GPU-accelerated frequent itemset mining. *IEEE international conference on cluster computing* (pp. 590-594). Austin.

Zhang, F., Zhang, Y., & Bakos, J. (2013, October). Accelerating frequent itemset mining on graphics processing units. *Journal of Supercomputing, 66*(1), 94-117.

Zhou, J., & Yu, K.-M. (2008). Balanced tidset-based parallel FP-tree algorithm for the frequent pattern mining on grid system. *Fourth international conference on semantics, knowledge and grid* (pp. 103-108). Beijing.