# A fast Parallel Association Rule Mining Algorithm Based on the Probability of Frequent Itemsets

**Marghny H. Mohamed**[†] **and Hosam E. Refaat**[††]
margami@gmail.com hosam.refaat@yahoo.com
Dept. of Computer Science, Faculty of Computers and Information, Asyut University, Egypt

**Summary**
Frequent itemset finding is the most costly processing step in analyzing large transactional databases. At each stage in discovering frequent itemset a huge number of candidate itemsets are produced. Then, if we predict which candidate itemset will be frequent and which will not, we can reduce wastage of time in the processing unfrequent itemsets. In this paper we propose a new parallel algorithm for frequent itemset mining, called *probability of frequent itemset* (PFI) mining algorithm. The PFI algorithm can predict frequency of the candidate based on the probability of its subset and makes priority between candidate itemsets base on it's probability. Moreover, the PFI algorithm passes the database only one time by dividing the database horizontally and distributes it over the system nodes. Also, while finding the k-itemsets, the algorithm can start a new stage (finding k+1 itemsets) with the discovered frequent k-itemsets while some other itemsets in the same stage have not been finished yet. Moreover, we introduce a method for reducing the number of transactions. We present the result on the performance of our algorithm on various datasets, and compare it against well known algorithms.

***Key words:***
*Parallel Systems, Distributed shared memory, data mining, Association rule, Linda system, Tuple-space, Jini, JavaSpace.*

## 1. Introduction

Association rule mining(ARM), one of the most important techniques of data mining, finds interesting associations and/or correlation relationships among large set of data items. Discovering this association rules in data can guide the decision making. A typical and widely-used example of association rule mining is Market Basket Analysis. An interesting algorithm, Apriori [3], has been proposed for computing large itemsets. Because databases are increasing in terms of both dimensions (number of attributes) and size (number of records), parallel computation is a crucial component for successful large-scale data mining applications. To construct a parallel system, there are three models, namely; distributed memory systems(DMS), shared memory systems (SMS), and distributed shared memory systems (DSMS). The DSMS is the newest parallel technique [20]. In DMS each node has its private memory. If any node needs data from another node, it will send a request message to it. Hence, this system is also called "*message passing*". In this system the direct connection between nodes reduces system portability [12]. The SMS is based on the existence of a global memory shared among all nodes in the system. The SMS has various advantages, such as; it eases the data sharing and it eases the implementation of the parallel application [5]. The DSMS takes the advantages of the previous models. The DSMS systems have some countable advantages over the DMS based ones, e.g.; the application level ease of use, the DSMS is portable, and it is easy to share data and processes [5]. This is done by constructing a virtual shared memory using the available distributed memories system. Moreover, the DSMS has standard operations that make parallel programming portable and more comfortable [17]. The Jini system is an extension of the Java environment. The DSMS had been implemented as a service in Jini system. A JavaSpace is a service in Jini system that implements the DSMS model. A JavaSpace inherits the advantages of Jini and the Java platforms [16].

To have efficient ARM algorithm, the number of database scanning (I/O operation) must be minimized. The new algorithm in this paper does one parallel scan for the database. This scan is done by dividing the database horizontally over the distributed nodes. The algorithm after finding 1-frequent itemset can start a new stage while the current stage is not finished yet. For example, suppose that, at the second stage ( $L_2$ ) the itemsets $AB, AC, BC$ are finished and frequent and the others itemsets like $AD, AE, BD$ are not finished yet. Then the algorithm can start creating a new task (in $L_3$ ) for counting the itemset "$ABC$", that can be processed by any system clients. This allows creating new tasks for different stages. So, no node will be idle, because there are lots of new independent tasks in the distributed shared memory that can be taken to process. The performance of our algorithm has been done by making a comparison with three algorithms, the first one called "*Hori-Vertical*" which was proposed and implemented by us [25]. Also, Hori-Vertical uses the same parallel model. The second one called "*Eclat*" and it is the most important consideration when one want to solve such problems[23]. The third one is the one that introduced by Limine et al.

Manuscript received November 1, 2010
Manuscript revised November

that called "*Workload Management Distributed Frequent itemsets mining*" (*WMDF*) is used in performance comparison [4]. The WMDF algorithm is based on the horizontal database partitioning and it makes load balancing between system nodes.

The rest of the paper is organized as follows. Next section discusses the preliminary concepts of association rules. Section(3) introduces a JavaSpace service as a DSMS implementation over the Jini system. Section(4) introduces our new algorithm (PFI). Section(5) shows the results and performance discussion.

## 2. Preliminary Concepts

The problem of mining association rules can be formally stated as follows: Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items. Let $DB$ be a database transactions, where each transaction consists of a set of items such that $T \subseteq I$. The support of an itemset $X$, denoted $\sigma(X)$, is the percentage of transactions in $DB$ which it occurs as a subset. Given an itemset $X \subseteq I$, a transaction $T$ contains $X$ iff $X \subseteq T$. A $X$ *frequent* or *large* if its support is more than a user-specified *minimum support* (*min_sup*) value($S$). An itemset is maximal if it is not a subset of any other itemset [1].

An association rule is an implication of the form $X \Rightarrow Y$ has support $p$ in the $DB$ if the probability of the transaction in $DB$ contains both $X$ and $Y$ is $\sigma(X \cup Y) = p$. Where, $X, Y \subseteq I$ and $X \cap Y = \phi$. The *confidence* of the rule is the conditional probability that a transaction contains $Y$, given that it contains $X$, and is given as $\sigma(X \cup Y) / \sigma(Y)$. A rule is *frequent* if its support is greater than min_sup, and it is *strong* if its confidence is more than a user-specified *minimum confidence* (*min_conf*). The task of mining association rules is to find all the association rules whose support is larger than a minimum support threshold and whose confidence is larger than a minimum confidence threshold. The data mining task for association rules can be broken into two steps. The first step consists of finding all large itemsets, i.e., itemsets that occur in the database with a certain user-specified frequency, called minimum support. The second step consists of forming implication rules among the large itemsets [10]. In this paper, we only deal with the first step.

The way itemsets are represented is decisive to compute their supports. Conceptually, a database is a two-dimensional matrix where the rows represent the transactions and the columns represent the items. This matrix can be implemented in the following four different formats [24]:

- Horizontal item-list (HIL): The database is represented as a set of transactions, storing each transaction as a list of item identifiers (item-list).
- Horizontal item-vector (HIV): The database is represented as a set of transactions, but each transaction is stored as a bit-vector (item-vector) of 1's and 0's to express the presence or absence of the items in the transaction.
- Vertical tid-list (VTL): The database is organized as a set of columns with each column storing an ordered list (tid-list) of only the transaction identifiers (TID) of the transactions in which the item exists.
- Vertical tid-vector (VTV): This is similar to VTL, except that each column is stored as a bit-vector (tid-vector) of 1's and 0's to express the presence or absence of the items in the transactions.

| TID | Item-lists |
|---|---|
| 1 | 1 2 3 5 |
| 2 | 2 3 4 5 |
| 3 | 3 4 5 |
| 4 | 1 2 3 4 5 |

HIL

| Item-vectors |
|---|
| 1 1 1 0 1 |
| 0 1 1 1 1 |
| 0 0 1 1 1 |
| 1 1 1 1 1 |

HIV

| Tid-lists | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 1 4 | 1 2 4 | 1 2 3 4 | 2 3 4 | 1 2 3 4 |

VTL

| Tid-vectors | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

VTV

Fig. 1. Database layout

To scan the dataset in parallel way, it must be divided vertically or horizontally. The vertical division is based on the items and the horizontal division is based on the number of transaction per part. Our algorithm is based on the horizontal partitioning in the first stage and it handle the dataset vertically in the other stages. In the horizontally division, the database is divided into $N$ parts. So, the database $DB$ will be divided into ($DB_1$, $DB_2$, .... , $DB_N$). Then, $DB = \cup_{i=1}^{N} DB_i$. As the number of database parts increases, the size of each part decreases. If the size of database partition is very small, all nodes will waste their time in taking and retrieving the database partitions. The best choice of $N$ will depend on the number of nodes and its resources [4]. For a given minimum support threshold $S$, an itemset $x$ is globally frequent if it is frequent in $DB$; its support x.sup is

greater than $S \times DB$, and is locally frequent in a node $N_i$ if it is frequent in $DB_i$; its support $x.\sup_i$ is greater than $S \times DB_i$.

- **Property 1:** A globally frequent itemset must be locally frequent in at least one node [7].
- **Property 2:** All subsets of a globally frequent itemset are globally frequent [7].

## 3. Jini-JavaSpace System

Jini system extends the Java environment from a single virtual machine to a network of virtual machines. Jini system is a distributed system based on the idea of federating group of users and the resources required by these users to have a large monolithic system [16], [15]. The power of the Jini comes from the services, since services can be anything joined to the network. A JavaSpace is a service in Jini system that implements the DSMS model.

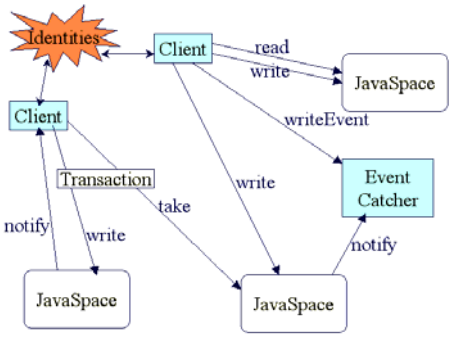


Fig. 2. JavaSpace model

JavaSpace is a distributed shared memory service that is implemented over Jini System [14]. The object that can be written in JavaSpace service is called "*entry*". The entry can contain data or/and processes. Sometimes the entry is called *tuple*. JavaSpace contains the following operations: take, takeIfExists, read, readIfExists, write, notify, snapshot. The write operation is to write an entry in JavaSpace. To read an entry from the JavaSpace, the read() or readIfExists() operation is used. The consecutive reading operation of the same template may return different entries even if JavaSpace contents are not changed. The difference between these two versions of reading is that; readIfExist-s() is not blocked if the tuple is not found in the space, it returns a null tuple if there is no matching tuple. Take() or takeIfExists() are two operations that extract entries from JavaSpace. In other words, these operations are similar to read and readIfExists() operations except that; taking operations remove the entry from the space. The snapshot operation is to take a copy of existing entry, but this copy is not updated in spite of the changes that may occur in the original entry. The notify operation is used to define an event that triggers when a specific entry is written [14]. See figure 2.

## 4. The proposed Algorithm

The first issue to have high performance ARM algorithm is reducing number of database scanning. But it is impossible to reduce number of database scanning to be less than one. The second issue is reducing number of comparison and searching area. PFI algorithm does only one scan over the database. Finding the first frequent itemset is done by dividing the dataset horizontally and the other stages is done using VTV layout. Then, reducing number of comparison is done in both of dataset dimensions.

So, now we discuss the ways of reducing number of comparisons. The frequent itemset of size $k$ ($k$-itemsets) is denoted by $L_k$. The set $L_k$ is founded scanning its superset, that called $C_k$, to determine the support for each candidate in $C_k$. The set $C_k$ is generated form $L_{k-1}$ from $C_k = \{c \mid join(c, L_{k-1}) \wedge prune(c, L_{k-1})\}$. This is means $L_k \subset C_k$. Practically $C_k$ is so much big than frequent itemsets $L_k$. So, the new idea for reducing the number of comparison is to reduce searching of unfrequent itemsets in $C_k$. This is done by predicting the probability of each element in $C_k$ from the support of its subsets in $L_{k-1}$. The support of any itemset is the percentage of transactions in database which it occurs as a subset. This is also meaning the probability of appearing this itemset. So, let $P_1, P_2, ..., P_n$ be the independent probability of the items $A_1, A_2, ..., A_n$ respectively. Let the probability for any two items $A_f, A_g$ is $P_f, P_g$ such that $P_f < P_g$. Then the probability of both itemset $A_f, A_g$ appearing in one transaction is $P_{fg}$. If $A_f$ and $A_g$ are total non-correlation, $P_{fg} = P_f.P_g$. Also if $A_f$ and $A_g$ are total correlation, $P_{fg}$ is the minimum of $P_f$ and $P_g$ (i.e. $P_{fg} = P_f$). Then $P_f.P_g \le P_{fg} \le P_f$ [26]. This is means that, the upper pound of the probability of any two itemsets ($A_f$ and $A_g$) to be exist in the same transactions is minimum probability of these itemsets($P_f$), and the lower bound of probability of these itemsets to be exist in the same transactions is $p_f p_g$. So,

for any two frequent itemsets $A_f, A_g$, if the $P_f.P_g < S.$ This means that; if these two itemsets are non-correlation they will not be frequent. Also, if the upper bound (the items are correlation) is not big enough than minimum support $S$, the new itemset $A_{fg}$ has a weak probability to be frequent.

Let $\delta$ is be the ratio of $S$ that decided if any itemset has big enough probability to be frequent. Where $0 \le \delta \le 1.$ Then, we suppose that the upper bound of the probability of the new itemset $P_f$ must be:

$$P_f \ge S(1+\delta) \qquad (1)$$

The previous equation means that, the maximum probability of the new itemset $A_{fg}$ must exceed the minimum support with $\delta$. For example, let $\delta = 0.1$, then the estimated maximum support of $A_{fg}$ must exceed the minimum support with ten percent. This is because, the intersection between two tid lists (transaction id lists for $A_f$ and $A_g$) will not be exceed the shortest one( $A_f$ list). Also, if the length of shortest list is much closed to the minimum support, the intersection between these two lists will be less than the minimum support. In the practical section (section 5) different values of $\delta$ will be tested to show how much the dataset vertical dimension will be shrunk in all stages. Also, we will show the effect of $\delta$ in the algorithm performance. So, before creating a task for the new itemset from the frequent itemsets that we have, the new itemsets can be divided in to three parts:

a) Strong probability frequent itemsets: that has lower probability bound greater than or equal the minimum support ( $p_f p_g \ge S$ ). These itemsets will take high priority in processing by the algorithm.
b) Itemsets that pass the condition in equation(1). In another words, the new itemsets that it's maximum estimated probability big enough than $S$. These itemset will take the second priority in processing.
c) Very weak frequent probability itemsets: that does not pass the condition in equation(1). These itemsets can take the lower priority in processing or it can be omitted. In our algorithm and the practical test we had omit these itemsets. Also, in some sensitive application these itemsets can take the third priority.

What we discussed so far is the method of reducing the searching range vertically. Now we turn to discuss the way of reducing the search range horizontally (number of transactions). Each client sorts its database partition depends on number of items in transaction into segments. In other words, the HIV for each node is divided into segments. These segments defined by range of ratio of items in each transaction $(r)$. For example, the first segment contains the very rich transaction (transaction contains more than equal 70% of the database items) $70 \le r \le 100$. The second segment contains more than or equal 50% and less than 70% of the database items, and so on. Each node after finishing the creation of the sorted HIV, it will encapsulates HIV with meta-data about the start and the end of each segment into result entry. Then the node writes result entry into JavaSpace. So database will be transformed into HIV that is sorted into segments ( $g_1,....,g_q$ ). The first segment $g_1$ contains the very rich transactions. Also, the last segment $g_q$ contains the very poor transactions. The definition of all segments is depend on the application. So, the probability of any frequent item to be locally frequent in the first segment is very high. Also, the probability of any frequent item to be locally frequent in the last segment is very low. This is because, let the definition of the first segment is that contains transactions that have more than $t$ of the database items. So, the total probability of a transaction in this segment contains an item:

$$P = \sum_{i=1}^{t} p_i \qquad (2)$$

Where $p_i$ represents the probability of item $i$. This means that the probability of purchasing any item in rich basket (transaction) is very high. Also, if $t$ is small, this means that transaction belongs to poor segment and it has low probability to exist in this transaction.

**- Property 3:** for all globally frequent items $a, b \in I$ such that; $a$ is locally frequent in the segment $r$ but $b$ is not. $a$ will be globally frequent in the itemset $DB - r$, but $b$ may not .

*Proof:* Let the segment $r$ contains two items $a, b \in I$, such that both of $a, b$ are globally frequent. Also, let $a$ be locally frequent in $r$ but $b$ is not. So, in the dataset $DB - r$ the item $a$ may be not globally frequent. Because the item $a$ may be only locally frequent in segment $r$ (Property 1). But the item $b$ will still globally frequent, because the item $b$ in the neglected segment $r$ is not locally frequent and it is frequent in one or more segments in $DB - r$. □

**- Property 4:** If a segment $g$ its transactions contains only one item( $g = \{T : |T| = 1\}$, this segment can be neglected without effecting k-frequent itemsets(where k>1).

*Proof:* Let all transaction in a segment $g$ has one items and suppose that two items $i, j$ are globally frequent and locally frequent in $g$. Nothing transactions in this

segment contains both of $i$ and $j$. So, if we neglecting this segment in finding the highest frequent itemsets (k>1), it will not effect the count of the itemset $ij$.

Also, let the segment $g$ is the only locally frequent segment for the item $i$. Then after neglecting this segment the item $i$ will not be frequent. But $g$ is omit at k>1. This is means that, $i$ has been detected as frequent item. □

Then we can use these two properties to shrink the dataset as follows. If the segment $r$ is very poor segment and $\forall i \in I$ is not locally frequent in $r$, the segment $r$ can be omitted in the searching. Also, if all transaction in a segment contains only one item, this segment will be omitted. This will reduce searching in database size dimension.

Our algorithm only scans the database in the initial stage ( $k = 1$ ) (see algorithm 1). In the other stages ( $k > 1$ ) the algorithm depends on the distributed shared memory, because the size of the database will be shrunk. In the initial stage of our algorithm the database is divided horizontally. So, the first stage of our algorithm is to create a task for each database partition. That is done by calling the Initial_task_creation procedure (see algorithm 2). Initial_task_creation procedure writes the tasks in the distributed shared memory to be executed by the system nodes. The algorithm will collect the results by calling InitialResultCollector procedure to collect the results from system nodes. Then, the master process merges the HIV partitions, that come from the nodes, into single HIV with keeping the segment structure. After that the algorithm omit the poor segments that do not have any local frequent itemsets. Also, the segments that its transaction has only one item will be omitted. These tasks in the form of JavaSpace entry are called "*taskEntry*". Each client will take one taskEntry after anther to execute it. Each client will scan its partition and convert it to the HIV (Horizontal Item Vector).

**Algorithm 1:** The master Process in the algorithm

$V_{C_k}$ */*Vector of stages for Candidate itemsets */*
Call Initial_task_creation() procedure
Call Initial_Result_Collector() procedure
Omit the neglectable segments
Convert the HIV to VTV format in $V_{C_k}$.
Call taskCreator() thread.
Call ResultCollector() thread
**While** true **do**
  **If** all tasks finished and $C_k$.size < 1 then
    Kill ResultCollector() thread
    Kill taskCreator() thread
    Break the loop
  **End**
**End**

**Algorithm 2:** Initial_task_creation procedure

**For** i = 1; i ≤ N; i ++ **do**
  Create taskEntry( $DB_i$ )
  Write taskEntry( $DB_i$ ) in the JavaSpace
**End**

**Algorithm 3:** The taskEntry Algorithm

**If** (k=l) **then**
*/*The 1-itemsets (need database scan)*/*
**For** all transactions $t \in DB_i$ **do**
**If** item $i \in t$ **then**
Increase the item $i$ counter
Convert the transaction into binary form $HIV_i$.
Put the transaction ( $t$ ) into its segments.
**End**
**End**
Encapsulate the $HIV_i$ into a resultEntry.
**Else**
*/*At the stages k>1. We have two itemsets X and Y must be joined into new itemset XY and count it's frequent*/*
resultList = X.TIDList $\cap$ Y.TIDList
**If** the itemset XY is non-frequent **then**
resultList= $\phi$
**End**
Encapsulate the resultList into a resultEntry.
**End**
Return a resultEntry that to the DSMS (JavaSpace)

**Algorithm4:** The Initial_Result_Collector procedure

**While** there are unfinished initial task entry **do**
**If** new initialResultEntry written **then**
Take resultEntry;
$HIV = \bigcup_{i=1}^{N} HIV_i$ */*Merge the $HIV_i$ depends on the segments */*
**End**
**End**

**Algorithm 5:** The ResultCollector thread algorithm

**While** true **do**
**If** new resultEntry written **then**
Take resultEntry;
Update $V_{C_k}$;
**End**
**End**

**Algorithm 6:** The taskCreator thread algorithm

$L_1$ ={large 1–itemset}
$C_2 = L_1 \times L_1$
**For** $\forall c \in C_2$
**If** lower bound of $P_c \geq S$ **Then** /* The probability of the candidate $c$ is strong */

Create a taskEntry for $c$ with priority =1;
Write the new taskEntry into JavaSpace;
update $V_{C_k}$;
**Else if** $P_c \geq S(1+\delta)$ **then**
Create a taskEntry for $c$ with priority =2;
Write the new taskEntry into JavaSpace;
update $V_{C_k}$;
**Else**
Neglect creating task for $c$
update $V_{C_k}$;
**End**
**while** true **do**
**If** new itemset finished **then**
**If** the itemset is large **then**
Join the itemset with the other finished itemsets in the same stage;
Prune the new candidate $c$;
**If** lower bound of $P_c \geq S$ **then**
Create a taskEntry for $c$ with priority =1;
Write the new taskEntry into JavaSpace;
Update $V_{C_k}$;
**Else if** $P_c \geq S(1+\delta)$ **then**
Create a taskEntry for $c$ with priority =2;
Write the new taskEntry into JavaSpace;
Update $V_{C_k}$;
**Else**
Neglect creating task for $c$
update $V_{C_k}$;
**End**
**Else**
update $V_{C_k}$;
Delete related information this itemset from the $V_{C_k}$;
**End**
**End**
**End**

Finding the first frequent itemset (k=1) is done by scanning the database. Finding the other itemsets (k>1) is done on the VTV(Vertical tid–Vector ). The "taskEntry" is a JavaSpace entry that contains algorithm for counting the frequency of the itemset $i$. The taskEntry algorithm contains two cases, see algorithm(3). The first case, when $k=1$, the client must create the $HIV_i$ by scanning its database partition to count the frequency of all database items and sort this partition transaction depending on segments. The second case is at $k>1$. At this stage the taskEntry contains two TID lists of two itemsets ( $X$ and $Y$ ). Also at this case, the client must create a resultList that contains the intersection between the two itemset( $X, Y$ ) TIDLists. If the new itemset is not frequent, the result list will be empty. Then, the client will create an entry called "*resultEntry*" that encapsulates the $HIV_i$ or resultList. The resultEntry must be written to the JavaSpace by the client.

The master process calls the Initial_Result_Collector procedure to collect all result entries. This procedure merges the HIV presentation for all database partitions that exists in all result entries into single *HIV* format and keeping the segment sort(see algorithm 4). Also, the master process omits poor segments that do not contain any frequent items. Then, the algorithm starts building $V_{C_k}$ vector, that is a vector to register candidates, frequent itemsets and the related information for each stage. The first stage in $V_{C_k}$ vector is $C_1$, that contains the first frequent itemset that has been taken form the HIV structure. So, the algorithm generates new tasks by calling the taskCreator thread. The taskCreator thread checks if there is any finished itemset. If the finished itemset is not large, that itemset must be registered as unfrequent and its related information like tid list must be deleted from the vector. If the itemset is large, it will join this itemset to the other finished large itemset in the same stage. After joining, the algorithm must make prune to the new candidates. Then, it creates a new taskEntry for the new candidates and puts it in the JavaSpace. Algorithm(6) contains the pseudo code of taskCreator thread. The function of ResultCollec- tor thread is to collect any result entry in the JavaSpace and inform the taskCreator thread(seen algorithm 5). The algorithm will finish if all tasks in the $V_{C_k}$ are finished and the size of the last stage of this vector is less than 1. This means that, all taskEntries created by the algorithm are finished and the last stage of the algorithm does not have any candidates. At this point, the algorithm must kill all the threads it has and the large frequent itemsets is exists in $V_{C_k}$ vector.

## 5. Experimental Results

The experimental test of PFI was performed in three objectives. The first objective is to list effect of $\delta$ on the set of candidate $C_k$ (section 5.1). The second objective is to list different values of $\delta$ to determine the best (section 5.2). The third objective is to test the performance of the new algorithm by comparing it with the other algorithms (section 5.3). All experiments ware performed on five PCs. These PCs can be heterogeneous, but in performance test we would like to unify the resource of the system nodes. This was to highlight the effect of other parameter like, database size, minimum support, $\delta$, and number of nodes in the system. These PCs had a CPU of type Intel(R)

Core(TM) 2Duo 1.6 G.H and 2GB RAM. The intercommunication between the machines was done by 100 Mbps Ethernet. The software environment was as follows; Windows XP professional, Java JDK 1.4.2 04 [13], Jini(TM) Technology Starter Kit v2.0.2 [16] and a free visual platform for Jini 2.0 was called Inca X(TM) [9]. To measure the performance of the algorithms we use three synthetic datasets: D1=T10I4D10K, D2=T10I4D100K, D3=T10I10D1000K. The dataset T10I4D10K meant an average transaction size of 10, an average size of the maximum potentially frequent itemsets of 4, and 10000 generated transactions. These datasets generated by the algorithm for generating synthetic databases that described by Agrawal and Srikant [3]. Each experiment was repeated 4 times. The average of the four runs was taken and used for analysis. We chose four values of $\delta$ (these are {0.01, 0.1, 0.2, 0.3}) for PFI test.

## 5.1 The Reduction of Candidate Test:

In this section the effect of $\delta$ on the number of candidates was tested. Figures(4,5,6) show the effect of $\delta$ at minimum support 1% on the candidates in D1, D2, D3 respectively. The same test was done with minimum support 10% (figures 7, 8, 9). Moreover, this test was also repeated using high ratio of minimum support(35%) in figures(10, 11, 12). Form these figures we noticed that, increasing the value of $\delta$ the number of candidate decreased. Decreasing the number of candidate will increase the performance but it will decrease the accuracy. At $\delta = 1\%$ the number of candidates is closed to that yield by Apriori algorithm. But at big database or at small value of minimum support the difference between the number of candidates that yields from Apriori and PFI(at $\delta = 1\%$ ) was thousands of candidates. This makes predication that the performance will be enhanced by PFI. But first we must check the effect of $\delta$ on the accuracy in the next section.

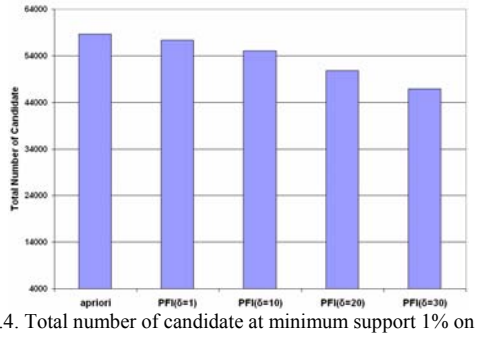

Fig.4. Total number of candidate at minimum support 1% on D1

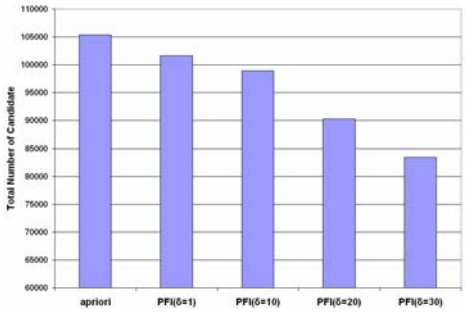

Fig.5. Total number of candidate at minimum support 1% on D2

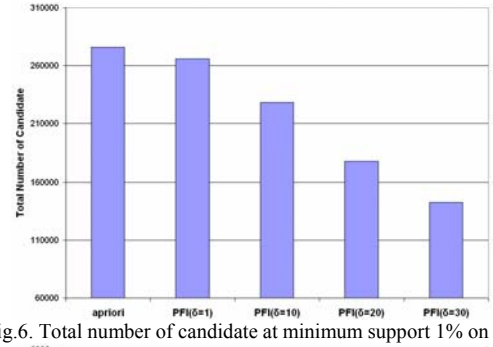

Fig.6. Total number of candidate at minimum support 1% on D3

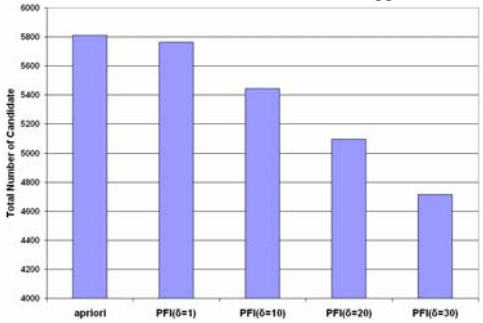

Fig.7. Total number of candidate at minimum support 10% on D1

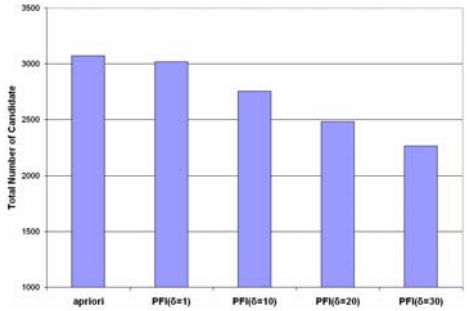

Fig.8. Total number of candidate at minimum support 10% on D2

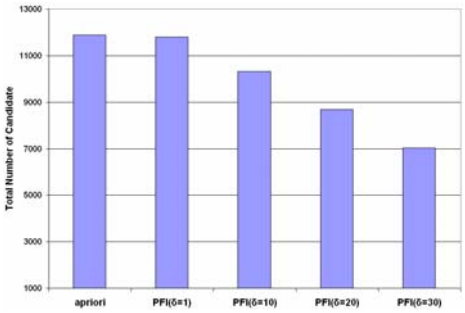

Fig.9. Total number of candidate at minimum support 10% on D3

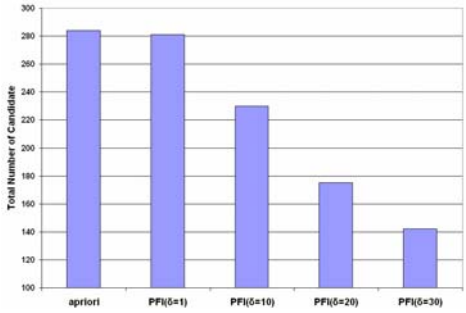

Fig.10. Total number of candidate at minimum support 35% on D1

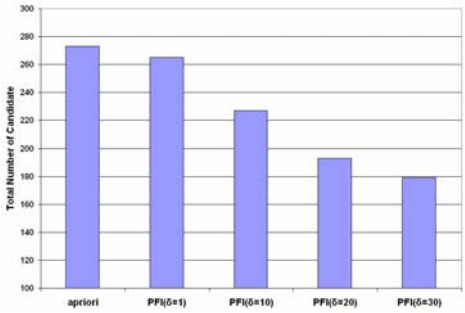

Fig.11. Total number of candidate at minimum support 35% on D2

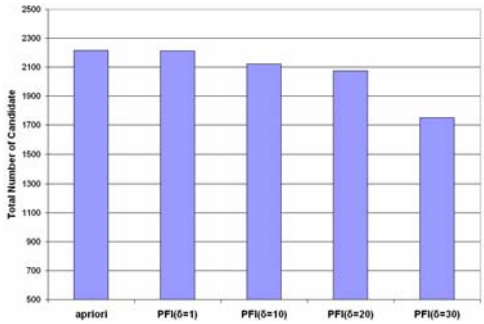

Fig.12. Total number of candidate at minimum support 35% on D3

## 5.2 The Accuracy Test:

In this section we will test the effect of $\delta$ on the number of frequent itemset in each stage ( $k$ - itemsets). We compare PFI with Apriori algorithm that represents the complete result that should be appearing. Also, PFI algorithm classifies the candidate into three groups. The first group, called strong probability frequent itemsets, contains the itemsets that have lower probability bound greater than or equal the minimum support ( $p_f p_g \geq S$ ). We test the performance of the new algorithm with this hard condition and we called it "PFIHard". We had to show the effect of this condition on filtering the itemset. Also, we will show whether the hard condition is sufficient or it can be use to accelerate the algorithm response time.

Figures from 13 to 21 show frequent itemsets comparison between the frequent itemsets that yield from PFI and the Apriori algorithm at different minimum supports ( $s = \{1\%,\ 10\%,\ 35\}$ ). We can notice that, the curves of PFI and Apriori are closed at the beginning and at the end. This means that, the smallest and maximal frequent itemset that will yield from PFI and Apriori will be approximately identical. Also, the PFI at $\delta = 0.01$ and Apriori curve are completely identical.

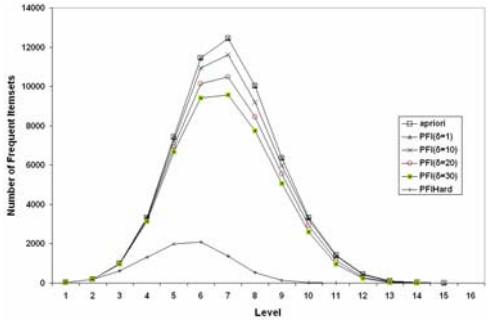

Fig.13. number of frequent itemsets at minimum support 1% on D1

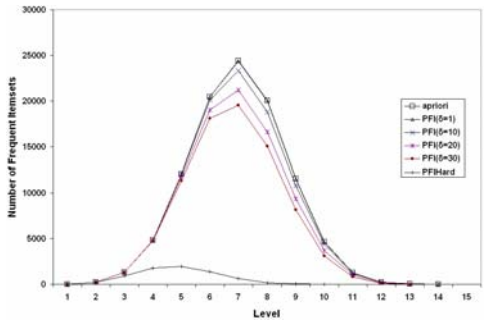

Fig.14. number of frequent itemsets at minimum support 1% on D2

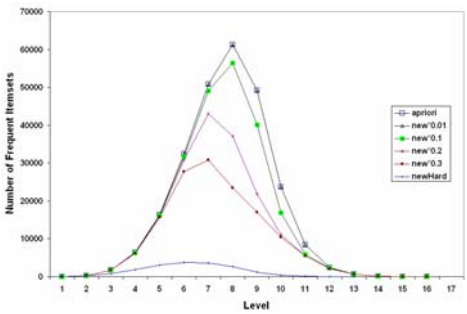

Fig.15. number of frequent itemsets at minimum support 1% on D3

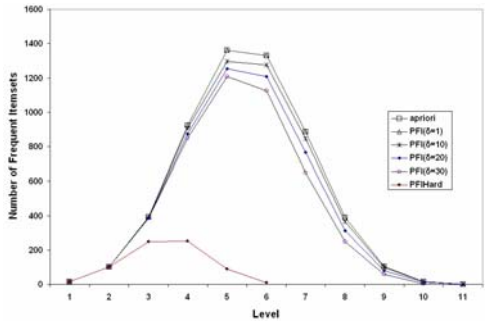


Fig.16. number of frequent itemsets at minimum support 10% on D1

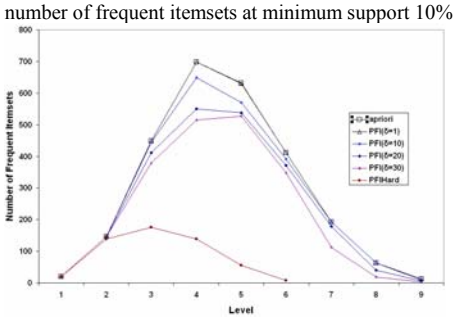


Fig.17. number of frequent itemsets at minimum support 10% on D2

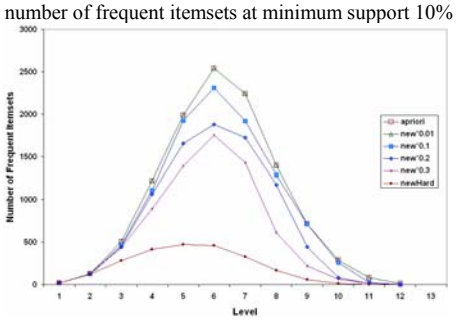


Fig.18. number of frequent itemsets at minimum support 10% on D3

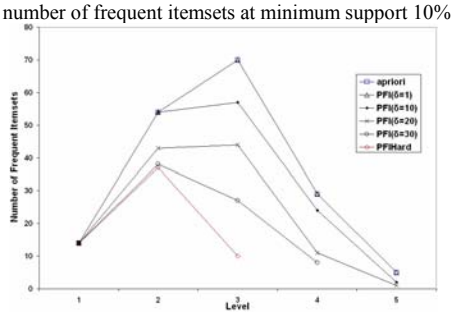


Fig.19. number of frequent itemsets at minimum support 35% on D1

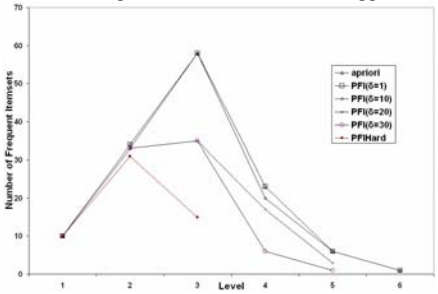


Fig.20. number of frequent itemsets at minimum support 35% on D2

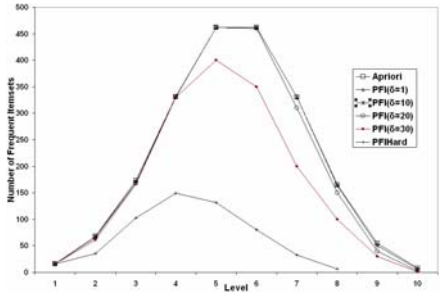


Fig.21. number of frequent itemsets at minimum support 35% on D3

## 5.1 The Performance Test:

Now, we compare the performance of PFI algorithm with the Hori-Vertical, Eclat, WMDF and Apriori algorithms. Figure (22) shows the performance comparison on dataset D1. From this figure we can notice that PFI has performance better than Hori-Vertical algorithm. By increasing the minimum support the PFI at $\delta = 0.01$ curve is coming closed to Hori-Vertical. Also, by increasing the value of $\delta$ the performance of the PFI is increased.

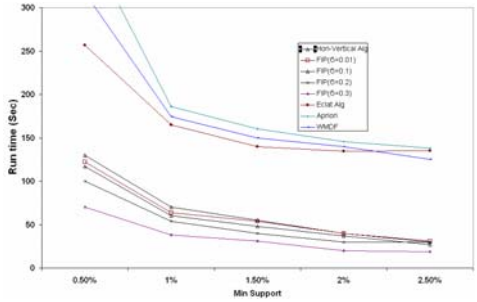


Fig.22. performance comparison on database D1

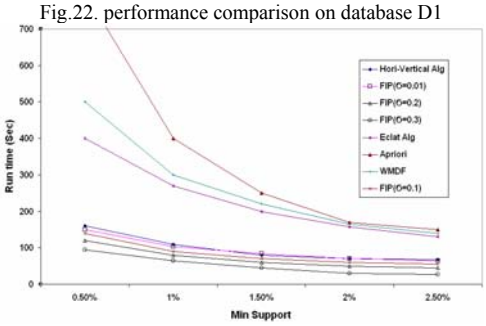


Fig.23. performance comparison on database D2

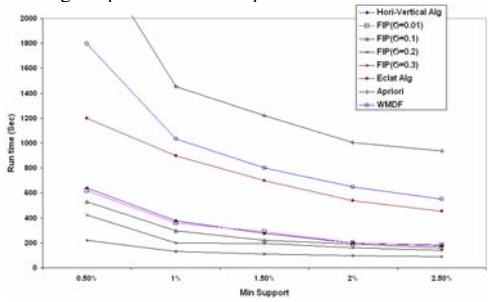


Fig.24. performance comparison on database D3

Figure (23) shows the same test using D2 as datasets. We can notice that, at the big minimum support the performance of the Eclat, WMDF and Apriori algorithms are closed. The performance comparison using big datasets (D3) is shown in figure (24). From all of the previous figures, we notice that, the PFI algorithm has the best performance. Also, at $\delta = 0.01$ the PFI and HoriVertical are closed at the big minimum support. The Apriori algorithm has the worst performance because this algorithm is a sequential and runs on one machine. The Eclat algorithm have performance better than the WMDF algorithm. This is because, the Eclat algorithm scans the database three times and the WMDF algorithm scans the database a lot of times. But at the big minimum support, the performance of the WMDF is going to be better than Eclat.

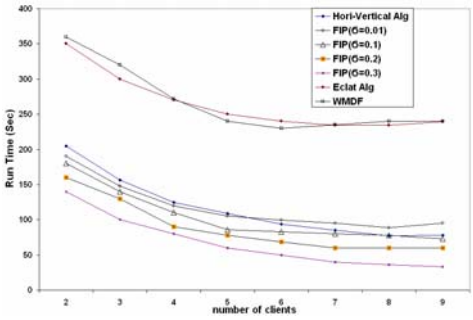

Fig.25. performance comparison on database D3

Now we will test the effect of increasing number of nodes on the performance of PFI algorithm. Also we will compare it with the previous parallel association rules algorithms. The Apriori will not be in this test because its sequential algorithm. So, we measure the performance of the parallel algorithms (Ecalt, WMDF, HoriVertical) using different number of clients in the system, as seen in figure 28. This test is done using the biggest dataset we have (D3) and in case of minimum support equals 0.5%. From this figure we can notice that, the WMDF curve is not smooth because the redistribution of the database blocks can raise the communications. The Hori-Vertical algorithm is going to be better than the PFI specially as increasing number of clients. Also, PFI algorithm has the performance better than PFI algorithm at small number of nodes.

From the entire previous test we can conclude the following. The PFI produce very accurate output at $\delta = 0.01$. Also, as increasing the value of $\delta$ the number of candidate will be decreased but the accuracy will also decreased. The performance of PFI is better than the performance of HoriVertical algorithm at the small value of minimum support and on big database, this is because $\delta$ will be more effective. As increasing number of nodes the performance HoriVertical algorithm is being better than PFI.

## 6. Conclusion

Through this paper, we have presented PFI algorithm. The PFI uses a DSMS which has various advantages over the other parallel models. The PFI algorithm based on reducing dataset searching area vertically (the number of candidates) and horizontally(number of transaction). Also, PFI has powerful features, such as; scanning the database only one time and processing different stages of large itemsets at the same time. Moreover, a comparison of PFI algorithm with our previous algorithm "HoriVertical", a well-known algorithm Eclat, Apriori algorithm and a new load balanced algorithm called WMDF was made. In general the PFI algorithm has the best performance. The PFI has reasonable performance and produce accurate output at $\delta = 0.01$. Also, increasing the value of $\delta$ will give a good performance enhancement but will reduce the accuracy. So, $\delta$ will be depending on the type and the size of the dataset. As increasing the number of system nodes the performance of the HoriVertical was better than the performance of the PFI. Future studies are required to fix this problem.

## References


[1] R. Agrawal, T. Imielinski, and A. Swami, Mining association rules between sets of items in large databases., In Proc. of the ACM SIGMOD Conference on Management of Data, pages 207-216, Washington (1993).

[2] R. Agrawal and J. C. Shafer, Parallel mining of association rules, IEEE Transactions On Knowledge And Data Engineering, Volume 8 pages:962-969 (1996).

[3] R. Agrawal and R. Srikant, Fast algorithms for mining association rules in large databases, Proceedings of the 20th International Conference on Very Large Data Bases, pages 487–499 (1994).

[4] L. M. Aouad, N Le-Khac, and T. M. Kechadi, Distributed frequent itemsets mining in heterogeneous platforms, Engineering, Computer and Architecture, Volume 1 (2007).

[5] U. Badawi., A single system image supporting distributed objects, Ph.D. thesis, Dept. of Mathematics, Faculty of Science,Cairo University, nov. 2000.

[6] D. W. Cheung and Y. Xiao, Effect of data skewness in parallel mining of association rules, Lecture Notes in Computer Science, Volume 1394 (1998).

[7] T. Vincent W. Ada D. Cheung, H. Jiawei and Y. Yongjian, A fast distributed algorithm for mining association rules., 4th Intl. Conf. Parallel and Distributed Info. Systems, (1996.).

[8] M. Hahsler, G. Bettina, K. Hornik, and C. Buchta, Introduction to arules a computational environment for mining association rules and frequent item sets, 2010.

[9] inca X, Inca x(tm) community edition, available from Incax WWW Site (http://www.incax.com/download.com), 2007.

[10] H. Jiawei and M. Kamber, Data mining: Concepts and techniques, second edition (the morgan kaufmann series in data management systems), 2 ed., vol. 2, Morgan Kaufmann; 2 edition, November 2005.

[11] S. Kotsiantis and D. Kanellopoulos, Association rules mining: A recent overview, GESTS International Transactions on Computer Science and Engineering, Volume 32 Pages:71–82 (2006).
[12] T. G. Mattson, Programming environments for parallel and distributed computing: A comparison of p4, pvm, linda and tcgms-g., ftp Server, ftp.cs.yale.edu (1995).
[13] Sun Microsystems., Java development kit, vol. 1.4.2 04, available from Sun Microsystems WWW Site (http://www.sun.com/products/jdk), 2004.
[14] Sun Microsystems, Javaspaces specification, vol. 2.0.2, available from Sun Microsystems WWW Site (http://java.sun.com/products/javaspaces), jun 2008.
[15] Sun Microsystems, Jini architecture specification, vol. v2.0.2, available from Sun Microsystems WWW Site (http://www.sun.com/jini/), jun 2008.
[16] Sun Microsystems, Jini technology core platform specification., vol. v2.0.2, available from Sun Microsystems WWW Site (http://www.sun.com/jini/), jun 2008.
[17] H. E. Refaat, New mechanism to integrate fault tolerance in a distributed shared memory based system, Computer science, Cairo Uni, 2007.
[18] A. Schuster and R. Wolff, Communication-efficient distributed mining of association rules, ACM SIGMOD Int'l. Conference on Management of Data, Santa Barbara, California, pp. 473-484. (2001).
[19] P. Tang and M. Turkia, Parallelizing frequent item-set mining with fp-trees., Technical Report titus.compsci.ualr.edu/ ptang/papers/par-fi.pdf, Department of Computer Science,University of Arkansas at Little Rock (2005).
[20] M. Tomasevic, J. Protic, and V. Milutinovic., Distributed shared memory: Concepts and systems., IEEE Parallel and Distributed technology, 4(2):63-79 (1996).
[21] D. YaJun and L. HaiMing, Strategy for mining association rules for web pages based on formal concept analysis, Appl. Soft Com-put. volume 10 pages:772–783 (2010).
[22] M. J. Zaki, Parallel and distributed association mining: A survey, IEEE Concurrency 7 (1999), 14–25.
[23] M. J. Zaki, S. Parthasarathy, and L. Wei, A localized algorithm for parallel association mining, In Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures, 1997, pp. 321–330.
[24] P. Shenoy, J. R. Haritsa, S. Sundarshan, G. Bhalotia , M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In Proceedings of 2000 ACM SIGMOD International Conference on Management of Data, pages 22–33, 2000
[25] H. Marghny, and H. Refaat. Hori-Vertical Distributed Frequent Itemsets Mining Algorithm on Heterogeneous Distributed Shared Memory System. IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.11 , pages 56–62 , November 2010.
[26] LI Pingxiang, CHEN Jiangping and BIAN Fuling. A Developed Algorithm of Apriori Based on Association Analysis. Geo-Spatial Information Science. VOL 7. Issue 2. June 2004.

**Marghny H. Mohamed**, Dept. of Computer Science, Faculty of Computers and Information Science, Asyut University, Asyut, Egypt. He received the PhD degree in computer science from the University of Kyushu, Japan, in 2001, and the MS from Asyut university in computer science, in 1993 and BS degree in Mathematics from Asyut University, Egypt, in 1988. He is an associate professor in the Department of Computer Science, University of Asyut. He has many publications which in the fields of Data Mining, Text Mining, Information Retrieval, Web Mining, Machine Learning, Pattern Recognition, Neural Networks, Evolutionary Computation, Fuzzy Systems. Dr. Marghny is a member of the Egyptian mathematical society and Egyptian syndicate of scientific professions., he is a member of some research projects in Asyut university, Egypt. He is a Manager of the project entitled "Medical Diagnostic System for Endemic Diseases in Egypt Using Self Organizing Data Mining".

**Hosam E Refaat**: has been graduated from the Faculty of Science, Asyut university, Egypt, in 1998. In October 2006, he finished his Master degree in the field of distributed systems from the faculty of Science, Cairo University, Egypt. Currently, he is a lecturer of Computer Science in King Khalid University– Kingdom of Saudi Arabia.