LPAS: High Efficiency Load Balancing Parallel Data Mining Algorithm†

Sheng-Hui Liu¹
¹School of Software
Harbin University of Science and Technology
Heilongjiang, China
172851711@qq.com

Li-Wei Zhou¹
¹School of Software
Harbin University of Science and Technology
Heilongjiang, China
hrbust.lsh@126.com

Kun-Ming Yu²,*
²Department of Computer Science and Information Engineering
Chung Hua University
Hsinchu, Taiwan
yu@chu.edu.tw

Abstract—Association rule discovery plays an important role in knowledge discovery and data mining, and efficiency is especially crucial for an algorithm finding frequent itemsets from a large database. Many methods have been proposed to solve this problem. In addition, parallel computing has been a popular trend, such as on cloud platform, grid system or multi-core platform. In this paper, a high efficiency load balancing parallel data mining method based on Apriori with sorting algorithm so called the Load balancing Parallel mining method based on Apriori with Sorting (LPAS) is proposed. The main goal of the proposed algorithm is to reduce the massive duplicated candidates generated in previous method. Furthermore, this algorithm is performed better than previous methods. The experimental results showed that this method had dramatically reduced computation time with more threads. Moreover, it was observed that the workload was equally dispatched to each computing unit.

Keywords- parallel data mining; apriori; load balancing; association rules

I. INTRODUCTION

With the development of modern society has come about an explosion of digital databases. Identifying important and meaningful information has become much harder than before. Data mining has recently attracted a tremendous amount of attention because of its applicability in many areas. Association rules mining [1] [2] determines relations among itemsets in a database. Applying the results of data mining to the planning of company’s strategy could effectively increase the profit and reduce the risks.

In the digital area, the technology in hardware architecture advanced from single core to; double core to; multi-core or cloud system [6] [7] [10] [12]. These developments provided more computation power. Recently, many algorithms have been focusing on improving the efficiency of the data mining algorithms in multi-core architecture. As a result, a number of efficient algorithms have been proposed [5] [8] [11] [15]. However, most of the algorithms for discovering frequent patterns require multiple passes and placing a huge burden on the I/O subsystem.

For these problems, Yu and Wu present the MATI algorithm [13] to speed up the computation time of data mining by enhancing the efficiency of Apriori on multi-core architecture. The algorithm proposes two strategies, Item set Block and Task Dispatches, by avoiding the duplication of candidate itemsets in the candidate generation stage, better performance has been achieved in multi-core computing system. But some redundant candidate itemsets are still generated, especially in the 2-candidate itemsets.

In this paper, the proposed algorithm LPAS provides a strategy that sorts the 1-frequent itemsets in increasing order according to the occurrence times on the TID table after having constructed the TID table based on the MATI algorithm. This step further prunes the candidate itemsets further. Besides, the experimental results in this study show that the running time of LPAS was significantly faster than that of previous methods.

This paper is organized as follows: Apriori, AprioriTid, and MATI and some other algorithms are described in Section 2. In Section 3, the proposed algorithm LPAS is introduced and Section 4 presents the experimental results. And the final conclusion is given in Section 5.

II. RELATED WORKS

The most well-known and popular data mining technique is the one using Association rules, which indicates relations among itemsets in database. More and more attention has been attracted by the accumulation of association rule mining in large datasets. The Apriori [1] algorithm is one of the oldest and most versatile algorithms of Frequent Pattern Mining (FPM). This algorithm does a complete bottom up search with a horizontal layout and enumerates all frequent itemsets. Because of its significant applicability, many algorithms have been revised since.

The Apriori algorithm employs an iterative approach known as a level-wise search. Large itemsets from previous pass are checked if they are present in the current transaction. First, the set of frequent 1-itemsets is found by scanning the database to assess the count for each item, and then collecting the items that satisfy minimum support, denoted \( L_1 \), \( L_1 \), which is then used to find \( L_2 \), and \( L_2 \), which is used to find \( L_3 \), and so on. Thus new itemsets are formed by

† This work is partially supported by the National Science Council (NSC100-2632-E-216-001-MY3).
* The corresponding author. (yu@chu.edu.tw)
extending existing itemsets. In the Apriori algorithm, if the number of the first element is \( K \), then the database will be scanned \( k \) times at least. However, it generates too many candidate itemsets. It requires too much time making it have low efficiency.

In contrast to the Apriori algorithm, the AprioriTid algorithm [2] [9] uses the database at the first pass. Transaction IDentification (TID) is used to shorten the database scanning process by creating the TID table. In the TID table, the key is ItemID, and the value is TID. The size of this encoding can become much smaller than the database, thus saving much reading time. This leads to a dramatic performance improvement; three times faster than that of the apriori algorithm.

A lot of data mining algorithms [12] [16] have only been designed for single-core computer architecture, making the computation time quite long since the dataset is large. Recently, various studies have tried to find more efficient methods and technology to improve Apriori: [14] proposed a weighted distributed parallel Apriori algorithm (WDPA) based on the TID table. The WDPA algorithm proposes that a database only needs to be scanned once while maintaining load balancing among processors. In this algorithm, by using hash functions to store TID in table structure, the number of itemsets is quickly calculated without the need of rescanning the database. Besides, Chai [4] propose a work-balanced data mining algorithm. This algorithm reduces database scanning time by pruning the candidate item set generation to speed up the mining association rule. The algorithm does not generate redundant candidate itemsets.

Yu and Wu propose the MATI algorithm [13]. This algorithm is to speed up the computation time of the data mining algorithm by enhancing the efficiency of the Apriori algorithm on the multi-core architecture. In the MATI algorithm, two strategies are proposed: Item set Block and Task Dispatches developed to reduce the scan range on the database and the processing time for generating candidate itemsets. In the process of generating candidate itemsets, the algorithm divides frequent itemsets into multiple blocks, all frequent itemsets with the same first item are put into the same block, and the frequent itemsets in the same item set block is generated on the same core in MATI. This reduces unnecessary burdens and avoids same index frequent itemsets of data distributed on different cores in order to be more effective, balancing the load. In the next section, an improved MATI algorithm is given which sort the 1-frequent itemsets with the TID table after constructing it first. This reduces the running time for finding frequent itemsets in specific datasets.

### III. PROPOSED ALGORITHM

The mining of frequent patterns from a transaction database on parallel computing is effective in accelerating the mining process; finding frequent itemsets on multi-core architecture is a popular data mining technique. Therefore, the Load balancing Parallel mining method based on Apriori with Sorting (LPAS) is presented in the paper. The goal of LPAS is to reduce the candidate itemsets in the algorithm.

In the process of generating candidate itemsets with the MATI algorithm, the Item set Block strategy was used to determine whether the generated candidate \((k+1)\)-itemsets were frequent or not. Therefore, the items with the same index were put into the same block. The candidate \((k+1)\)-itemsets were only generated in the same block. This avoided same index frequent itemsets being distributed to a different core. But when the database started getting bigger, the block size and number got larger than before, which would have caused more blocks to merge with each other in \((k+1)\)-itemsets and the computation time of the algorithm would be longer.

With the LPAS algorithm, the numbers of the 1-frequent itemsets occurring in the TID tables were sorted in increasing order. With this strategy, in the process of generating the \((k+1)\)-itemsets, the block numbers with the same index got fewer, since the candidate \((k+1)\)-itemsets had only been generated in the same block, so the candidate \((k+1)\)-itemsets were also reduced improving the efficiency of cutting down the candidate itemsets. The sorting time of the algorithm also was significantly decreased. Meanwhile, with the Item set Block the items with the same index were put into the same block dispatching, the workload equally for the computing units. Therefore, with this strategy in LPAS, the number of candidate items generated in K-itemsets in each core decreased to such an extent that the data size of each core was reduced by 29% compared to the MATI algorithm.

However, when sorting the 1-frequent item in decreasing order using the TID table, the itemsets in the same index were more after 1-frequent itemsets merged with each other. So the item numbers in the same blocks increased, finally, there were more candidate itemsets in the same index block merged with each other. This needed more merging operation in order to find the frequent itemsets. That made the total time for finding frequent itemsets increase significantly.

In Fig. 1, the pseudo-code of LPAS is given.

<table>
<thead>
<tr>
<th><strong>LPAS: main function</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan database and construct TID table</td>
</tr>
<tr>
<td>( T ) = a transaction database over the set of items</td>
</tr>
<tr>
<td>( K ) is a merged round; ( Ck ) represents candidate itemsets in ( K ) round</td>
</tr>
<tr>
<td>( F1 ) represents frequent itemsets in 1 round; ( F1' ) is frequent itemsets that have been sorted; min is minimum itemsets in ( F1 )</td>
</tr>
<tr>
<td>while ( Ck &gt; 1 ) do</td>
</tr>
<tr>
<td>allocate candidate itemsets to each thread</td>
</tr>
<tr>
<td>foreach thread do</td>
</tr>
<tr>
<td>scan TID table</td>
</tr>
<tr>
<td>if candidate itemset count &gt;support</td>
</tr>
<tr>
<td>candidate itemset push ( F1 )</td>
</tr>
<tr>
<td>foreach thread in ( F1 ) do</td>
</tr>
<tr>
<td>for each candidate item set, ((i=1 \text{ to } n)) do</td>
</tr>
<tr>
<td>if candidate item set, ( \leq \text{min} )</td>
</tr>
<tr>
<td>( \text{min} = \text{candidate item set}, )</td>
</tr>
<tr>
<td>push candidate item set, to ( F1' )</td>
</tr>
</tbody>
</table>
end

collect each thread’s frequent itemsets in F1’
allocate frequent itemsets to each thread
foreach thread do
    Merge_function
    collect each thread’s candidate itemsets
    k++
end

Merge_function

if k > 1
    foreach frequent_item set do
        index_item = frequent_item.set.item1
        M_frequent_item set = next frequent_item set
        while index_item == M_frequent_item.set.item1 do
            if frequent_item set.item1 == M_frequent_item set.item1, ..., frequent_item set.item.k-1 == M_frequent_item set.item.k-1 && frequent_item set.item.k < M_frequent_item set.item.k
                frequent_items et merged with M_frequent_item set
            else break;
            M_frequent_item set = next M_frequent_item set
        else
            foreach frequent_item set do
                M_frequent_item set = next frequent_item set
                while M_frequent_item set != NULL do
                    frequent_item set merged with M_frequent_item set
                    M_frequent_item set = next M_frequent_item set
            end
end

Fig. 1. LPAS pseudo-code

Fig. 2 illustrates the procedure of LPAS. In the example, the database DB0 had five transactions from T1 to T5, six items, from A to F, and the support was set to 40%. When LPAS was executed, it scanned the database and built the corresponding TID table. An encoding of the candidate itemsets used in the previous pass was employed for this purpose. When the TID table was created, LPAS rapidly checked the candidate itemsets from the TID table. When the support number was higher than threshold, the itemsets were selected, and C1, denoting the 1-candidate itemsets was generated. The L1 denoted the 1-frequency item set in DB0, and all 1-item set were frequent in the example. Then, LPAS sorted the Items in increasing order, denoted L1’. The 1-itemsets in L1’ merged with each other to generate the 2-candidate itemsets in the table C2. The LPAS also counted the support number and pruned the candidate itemsets, when the support number of candidate itemsets was less than the minimum support, the result was shown as L2. The itemsets with the same first item were put into the same block. By generating L2, three item set blocks were obtained: item set block B = {BC, BD}, item set block A = {AF, AC, AD}, item set block F = {FC, FD}. When merging the itemsets in block B: {BC}, {BD}; with the same index item, a 3-candidate item set {BCD} was produced. Therefore, LPAS combined them into {BCD}, {AFC}, {AFD}, {ACD}, {FCD} in C3. LPAS repeated the above process until non-frequent itemsets were found. L3 was the final result in the example.

IV. EXPERIMENTAL RESULTS

To validate the effect of LPAS, a transactional database was selected. A multi-core architecture was implemented and compared with the performance of WDPA [14], Para_TID [2], and MATI [13]. All the experiments were performed on an Intel Core i5 CPU 650 3.20GHz equipped with 4G memory, running on Microsoft Windows 7. All the programs were written in Microsoft Visual Studio 2010. The experimental data was generated by IBM data generator [3].
In the experiment, para_TID denoted a parallel version of Apriori algorithm with TID; WDPA denoted a Weighted Distributed Parallel Apriori. Execution time was compared with LPAS, MATI, para_TID and WDPA with different size datasets, threshold numbers, transaction numbers, and thread numbers. As is shown, when the threshold was bigger than 0.001, the performance of the LPAS and MATI was nearly the same, so in the experiment, in order to show the sorting effect of LPAS, the threshold was set at 0.0005 to compare the two algorithms.

The computation time shown in Fig. 3 indicates that with the same dataset (T10I4D10KN500K) and threshold (0.005) but different number of threads, the LPAS algorithm took less time than MATI and the other algorithms, and hence resulted in high efficiency. Fig. 3 also shows that with the number of threads increasing, the execution time of the entire algorithm decreased.

Fig. 4 shows the execution time of all the algorithms with different database size. The thread number was 2; the support set to 0.005. With the database size changing that will affect the executing time. Fig. 4 shows that, with the database size increasing, the executing time of the two algorithms increased. However, the execution time of the LPAS algorithm was smaller than that of MATI and the other algorithms.

Fig. 5 shows another experiment in which the transaction numbers for the above four algorithms were changed. In the experiment the thread number is 2; the number of the threshold was 0.005; it is clear that, with the transaction numbers increasing, the database size also increased, and the execution time of the algorithms will increased as well. However, the execution time of the LPAS algorithm was shorter than the others.

Fig. 6 shows the execution time of the MATI and LPAS algorithm with different thresholds, 0.0005 to 0.002. In this experiment, both the Para_TID algorithm and WDPA algorithm took as long as more than 400s when the threshold was lower than 0.001, making for low efficiency. In Fig. 6, the performances of MATI and LPAS are compared. As can be seen, with the threshold increasing, the execution time of both decreased, but the performance of the LPAS algorithm was up to a maximum of 18.11% better than that of the MATI algorithm.

Regarding see the workload balance of the MATI and LPAS algorithms. Fig. 6 indicates the time of each thread cost in the processing of generating 2-itemsets. The dataset was T20I4D20KN500K, threshold was 0.0005, and eight thresholds were given. Since both of the two algorithms have
Using the strategy of adding the sorting of the frequent results, the experimental results show that LPAS reduced the computation time by 29% on the MATI algorithm. Therefore, the performance of the LPAS algorithm improves compared to MATI on multi-core architecture. The number of each thread generated was much less than MATI with more unnecessary candidate itemsets being filtered.

![Figure 7](image-url)  
**Figure 7.** Number of candidate items generated in 2-itemsets.

V. CONCLUSIONS

Association rule discovery plays an important role in knowledge discovery and data mining. However, the process of generating itemsets and confirming them is time consuming. Mining frequent patterns in multi-core architectures is an important problem in data mining research. Therefore, in this paper, a highly efficient load balancing parallel data mining method based on Apriori algorithm with Sorting (LPAS) is proposed. In this study, the 1-frequent itemsets were sorted in increasing order according to the number of occurrences after constructing the TID table, this step pruned the candidate itemsets further, and the performance of the algorithm was better than with the previous methods. The runtime of LPAS, in Fig. 6, was reduced by more than 16s when compared with the MATI algorithm, the performance of the LPAS algorithm was up to a maximum of 18.11% better than that of the MATI algorithm. And the number of candidate items generated in 2-itemsets in Fig. 7, was reduced by nearly 2000 in each core; as a result, the performance of the LPAS algorithm improve 29% on the MATI algorithm. Therefore, the experimental results show that LPAS reduced the computation time by using the strategy of adding the sorting of the frequent itemsets. It actually found frequent patterns reliably and effectively. We also observed that the workload was balanced among each thread in LPAS.

REFERENCES


