International Journal of Networked and Distributed Computing, Vol. 1, No. 2 (April 2013), 89-96

Parallel Implementation of Apriori Algorithm Based on MapReduce

Ning Li*
The Key Laboratory of Intelligent Information Processing,
Institute of Computing Technology, Chinese Academy of Sciences,Beijing,100190,China
Graduate University of Chinese Academy of Sciences, Beijing,100139,China
Key Lab. of Machine Learning and Computational Intelligence, College of Mathematics and Computer Science, Hebei University, Baoding, 071002, Hebei, China

Li Zeng
The Key Laboratory of Intelligent Information Processing,
Institute of Computing Technology, Chinese Academy of Sciences,Beijing,100190,China

Qing He
The Key Laboratory of Intelligent Information Processing,
Institute of Computing Technology, Chinese Academy of Sciences,Beijing,100190,China

Zhongzhi Shi
The Key Laboratory of Intelligent Information Processing,
Institute of Computing Technology, Chinese Academy of Sciences,Beijing,100190,China

E-mail:lin@ics.ict.ac.cn, heq@ics.ict.ac.cn

Received 22 March 2012

Accepted 13 November 2012

Searching frequent patterns in transactional databases is considered as one of the most important data mining problems and Apriori is one of the typical algorithms for this task. Developing fast and efficient algorithms that can handle large volumes of data becomes a challenging task due to the large databases. In this paper, we implement a parallel Apriori algorithm based on MapReduce, which is a framework for processing huge datasets on certain kinds of distributable problems using a large number of computers (nodes). The experimental results demonstrate that the proposed algorithm can scale well and efficiently process large datasets on commodity hardware.

Keywords: Apriori algorithm; Frequent itemsets; MapReduce; Parallel implementation; Large database

1. Introduction

Data Mining has attracted a great deal of attention in the information industry and in society as a whole in recent years. One of the important problems in data mining is discovering association rules from databases of transactions where each transaction consists of a set of items. Many algorithms have been proposed to find frequent item sets from a large database. However, there has not yet been published implementation performing the best under whatever conditions[1]. Apriori is one of the typical algorithms, which is a seminal algorithm proposed by R.Agrawal and R.Srikant in 1994 for mining frequent itemsets for Boolean association rules[2]. It aggressively prunes the set of potential candidates of size *k* by using the following observation: a candidate of size *k* can be frequent only if all of its subsets also meet the minimum threshold of support. Even with the pruning,

* Address :No.6 Kexueyuan South Road Zhongguancun,Haidian District Beijing,China

the task of finding all association rules requires a lot of computation power and memory. Parallel computing offers a potential solution to the computation requirement of this task, if the efficient and scalable parallel algorithms can be designed.

MapReduce is a patented software framework introduced by Google in 2004. It is a programming model and an associated implementation for processing and generating large data sets in a massively parallel manner [3][4]. Some data preprocessing, clustering and classification algorithms have been implemented based on MapReduce[5][6][7].

In this paper, we implemented the parallel Apriori algorithm based on MapReduce, which makes it applicable to mine association rules from large databases of transactions.

The rest of the paper is organized as follows. In Section 2, we introduce the basic Apriori algorithm. Section 3 gives an overview of MapReduce. In Section 4, we present the details of the parallel implementation of Apriori algorithms based on MapReduce. Experimental results and evaluations are showed in Section 5 with respect to speedup, scaleup, and sizeup. Finally, Section 6 concludes the paper.

2. Apriori Algorithm

2.1. *Problem statement*

The problem of mining association rules over market basket analysis was introduced in [8]. It consists of finding associations between items or itemsets in transactional data [9].

As defined in [11], the problem can be formally stated as follows. Let $I=\{i_1,i_2,\ldots,i_m\}$ be a set of literals, called items. Let D be a set of transactions, where each transaction T is a set of items such that $T\subseteq I$. Each transaction has a unique identifier *TID*. A transaction T is said to contain X, a set of items in I, if $X\subseteq T$. An *association rule* is an implication of the form "$X\Rightarrow Y$", where $X\subseteq I$, $Y\subseteq I$ and $X\bigcap Y=\varnothing$.

Each itemset has an associated measure of statistical significance called *support*. For An itemset X, we say its *support* is s if the fraction of transactions in D containing X equals s. The rule $X\Rightarrow Y$ has a *support* s in the transaction set D if *s* of the transactions in D contain $X\bigcup Y$. The problem of discovering all association rules from a set of transactions D consists of generating the rules that have a *support* and *confidence* greater than given thresholds. These rules are called *strong rules*.

This association-mining task can be broken into two steps:

Step1. The large or frequent itemsets which have support above the user specified minimum support are generated.

Step2. Generate confident rules from the frequent itemsets.

2.2. *Apriori algorithm*

The name of the Apriori algorithm is based on the fact that the algorithm uses prior knowledge of frequent itemset property which is that all nonempty subsets of a frequent itemset must also be frequent [2]. The main idea is to find the frequent itemsets.

The process of the algorithm is as follows.

Step1. Set the minimum support and confidence according to the user definition.

Step2. Construct the candidate 1-itemsets. And then generate the frequent 1-itemsets by pruning some candidate 1-itemsets if their support values are lower than the minimum support.

Step3. Join the frequent 1-itemsets with each other to construct the candidate 2-itemsets and prune some infrequent itemsets from the candidate 2-itemsets to create the frequent 2-itemsets.

Step4. Repeat the steps likewise step3 until no more candidate itemsets can be created.

The main steps consist of join and prune actions and the process is followed.

(i) The join step: To find L_k, a set of candidate k-itemsets is generated by joining $(k-1)$-itemsets. This set of candidates is denoted as C_k. Let l_1 and l_2 be itemsets in L_{k-1}. The notation $l_i[j]$ refers to the j th item in l_i. The items within an itemset are sorted in lexicographic order. The join $L_{k-1}\bowtie L_{k-1}$, is performed, where members l_1 and l_2 of L_{k-1} are joinable if their first (k-2) items are in common. The resulting itemsets formed by joining l_1 and l_2 is $l_1[1],l_1[2],\ldots,l_1[k-2],l_1[k-1],l_2[k-1]$.

(ii) The prune step: C_k is a superset of L_k, its member may or may not be frequent. According to the Apriori property, any $(k-1)$-itemsets that is not frequent cannot be a subset of a frequent k-itemsets.

Hence, if any subset with length $(k-1)$ of a candidate k -itemsets is not in L_{k-1}, then the candidate cannot be frequent either and so can be removed from C_k.

3. Introduction to MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. As the framework showed in Figure 1, MapReduce specifies the computation in terms of a *map* and a *reduce* function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks.

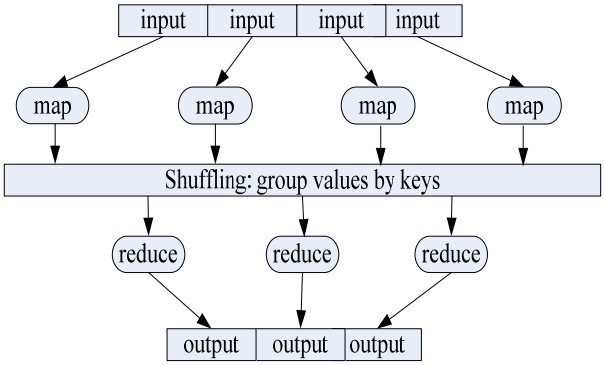


Figure 1. Illustration of the MapReduce framework

Map takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the reduce function [4]. That is, a map function is used to take a single key/value pair and outputs a list of new key/value pairs. It could be formalized as:

$$map :: (key_1, value_1) \Rightarrow list(key_2, value_2)$$

The reduce function, also written by the user, accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values. The intermediate values are supplied to the users reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory. The reduce function is given all associated values for the key and outputs a new list of values. Mathematically, this could be represented as:

$$reduce :: (key_2, list(value_2)) \Rightarrow (key_3, value_3)$$

The MapReduce model provides sufficient high-level parallelization. Since the map function only takes a single record, all map operations are independent of each other and fully parallelizable. Reduce function can be executed in parallel on each set of intermediate pairs with the same key.

4. Parallel Apriori Algorithm based on MapReduce

4.1. *The main idea of the parallel Apriori algorithm*

As described in Section 2, the key step in Apriori algorithm is to find the frequent itemsets. In the k th iteration, it computes the occurrences of potential candidates of size k in each of the transactions. It is obviously that the occurrences counting of candidate itemsets in one transaction is irrelevant with the counting in another transaction in the same iteration. Therefore, the occurrences computation process in one iteration could be parallel executed. After this phase, all the occurrences of candidate itemsets are summed up. Furthermore, the join actions are done on the frequent k-itemsets and prune actions are performed on the candidate $(k+1)$-itemsets. Finally, the frequent $(k+1)$-itemsets are found. According to the frequent itemsets, the rules that have a support and confidence greater than given thresholds are generated.

Figure 2 shows the flow chart of parallel Apriori algorithm, which is denoted as PApriori. The steps are as follows.

Step1. Use MapReduce model to find the frequent 1-itemsets.

Step2. Set $k = 1$.

Step3.If the frequent $(k+1)$-itemsets cannot be generated, then goto Step6.

Step4. According to the frequent k -itemsets, use MapReduce model to generate the frequent $(k+1)$-itemsets.

Step5. If k is less than the max iteration times, then $k++$, goto Step3; Otherwise, continue to the next step.

Step6. According to the frequent itemsets L, generate the strong rules.

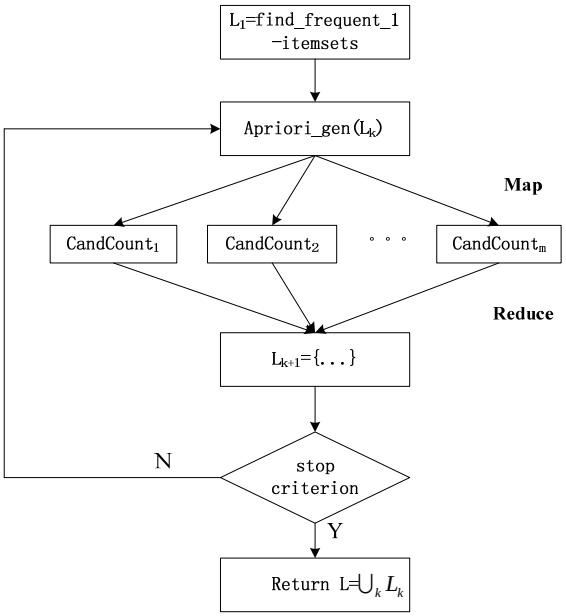


Figure 2. The flow chart of the parallel Apriori algorithm

4.2. *The parallel implementation of the Apriori algorithm based on MapReduce*

As the analysis mentioned above, PApriori algorithm needs one kind of MapReduce job. The map function performs the procedure of counting each occurrence of potential candidates of size k and thus the map stage realizes the occurrences counting for all the potential candidates in a parallel way. Then, the reduce function performs the procedure of summing the occurrences counts. For each round of the iteration, such a job is carried out to implement the occurrences computing for potential candidates of size *k*.

Map-function The input dataset is stored on HDFS[5] as a sequence file of <key, value> pairs, each of which represents a record in the dataset. The key is the offset in bytes of this record to the start point of the data file, and the value is a string of the content of this record. The dataset is splitted and globally broadcasted to all mappers. Consequently, the occurrence computations are parallel executed. For each map task, once the items in the candidate itemsets occur in the transactions, the <key', 1> pair will be outputted, where key' is the candidate itemsets. We use *m_cycles* to represent the maximum cycles of the PApriori. The pseudo-code of map function is shown in Algorithm 1.

Algorithm1. Map(*key, value*)

Input: Global variable *m_cycles*, the offset key, the sample *value*
Output: <*key'*, *value'*> pair, where the *key'* is the candidate itemsets and *value'* is the once occurrence of the *key'*, actually, it equals to 1.

```
1. if (m_cycles>1) /*for the case k>1*/
2.   For each itemset Cki in the
   candidate      k-itemsets
3.      If Cki is a subset of value
4.          Output(Cki , 1);
5.      Endif
6.    End For
7. Else For each itemset Ii in value
   /*k=1*/
8.         If Ii ≠ 0
9.            Output(Ii ,1);
10.        Endif
11.      End For
```

Reduce-function The input of the reduce function is the data obtained from the map function of each host. In reduce step, we sum up all the values with the same key and get the final result. In another word, we can get the total occurrences of potential candidates in the transactions. The pseudo-code for reduce function is shown in Algorithm 2.

Algorithm2. Reduce(*key, Value*)

Input: *key* is the element of the candidate itemsets, *Value* is once occurrence of the key
Output: <*key'*, *value'*> pair, where the *key'* is identical to *key* and value' is total occurrence of the key'.

```
1. sum=0;
2. while( values.hasNext()){
3.     sum+=values.next();
4.     }
5.output (key, sum);
```

5. Experimental Results

In this section, we evaluate the performance of our proposed PApriori algorithm in terms of sizeup, speedup and scaleup to deal with large scale dataset.

5.1. *The datasets*

The transactional data for an AllElectronics branch and T10I4D100K dataset are used in our experiments. As shown in Table1, there are nine transactions in the transactional data. We denote it as dataset1and replicate it

to get 1GB, 2GB, 4GB, and 8GB datasets respectively. They have many short transactions with few frequent itemsets. For the T10I4D100K dataset, we replicate it to 2 times, 4 times, 8 times and get 0.6G, 1.2G, 2.4G datasets, we denote them as T10I4D200K, T10I4D400K and T10I4D800K respectively. They have fewer larger transactions with many frequent itemsets. Performance experiments were run on a cluster of 10 computers, six with four 2.8GHz cores and 4GB memory, the rest four with two 2.8GHz cores and 4GB memory. Hadoop version 0.20.2 and Java 1.5.0_14 are used as the MapReduce system for all the experiments. Experiments were carried on 10 times to obtain stable values for each data point.

Table1. Transactional data for an AllElectronics branch

TID	List of item_IDs
T100	I_1,I_2,I_5
T200	I_2,I_4
T300	I_2,I_3
T400	I_1,I_2,I_4
T500	I_1,I_3
T600	I_2,I_3
T700	I_1,I_3
T800	I_1,I_2,I_3,I_5
T900	I_1,I_2,I_3

5.2. *Optimal number of reducers assigning*

In Hadoop version 0.20.2, the number of mappers is automatically determined by the cluster system while the number of reducers needs to be given by users. So before the parallel performance experiments, we can choose the optimal number of reducers by assigning different ones in the experiments.

The 2GB replication of Dataset1 is used and we choose two nodes in our assigning experiments. The execution times under different number of reducers are shown in Figure 3.

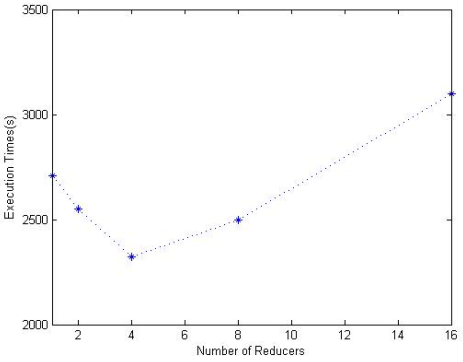


Figure 3. Executing times under different number of reducers

As shown in Figure 3, with the increase of reducers, the degree of parallelism increases, the execution time decreases gradually and reaches its minimum when the number of reducers is 4. After that, the execution time increases with the increase of reducers due to the additional management time and the extra communication time.

As we all known, the total computing time of a Hadoop program is mainly composed of two parts, one part is computing time of each map and reduce phases, the other part is communication and networking between map and reduce phases. The more the number of reduces is, the shorter the computing time of each map and reduce phases trends. However, the longer will be the time of communication and networking between map and reduce phases. Thus there can be some tradeoff between the two parts.

Generally speaking, in a cluster of computers without job fails in the running time, the optimal number of reducers is 0.95 or 1.75 × number of nodes × mapred.tasktracker.tasks.maximum. In case of 0.95, all the reduces can be invoked at the same time when all the maps finish. In case of 1.75, a faster node may execute multiple reduces, thus each node could be loaded in more balance.

According to the configuration of our cluster, each node has 4 cores, so the mapred.tasktracker.tasks.maximum of each node is 2. And there are 2 nodes used in the experiments, therefore, it can get the shortest computing time when the number of reduces is 4 on condition that there are no job fails in the running time.

5.3. *The evaluation measure*

We use scaleup, sizeup and speedup to evaluate the performance of PApriori algorithm.

Scaleup: Scaleup evaluates the ability of the algorithm to grow both the system and the dataset size. Scaleup is defined as the ability of a *m*-times larger system to perform a *m*-times larger job in the same run-time as the original system. The definition is as follows.

$$Scaleup(data, m) = \frac{T_1}{T_{mm}} \quad (1)$$

Where, T_1 is the execution time for processing *data* on 1 core, T_{mm} is the execution time for processing *m**data on *m* cores.

Sizeup: Sizeup analysis holds the number of cores in the system constant, and grows the size of the datasets by the factor *m*. Sizeup measures how much longer it takes on a given system, when the dataset size is *m*-times larger than the original dataset. It is defined by the following formula:

$$Sizeup(data, m) = \frac{T_m}{T_1} \quad (2)$$

Where, T_m is the execution time for processing m**data*, T_1 is the execution time for processing data.

Speedup: Speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm. It is defined by the following formula:

$$Speedup = \frac{T_1}{T_p} \quad (3)$$

Where, p is the number of processors, T_1 is the execution time of the algorithm with one processor, T_p is the execution time of the parallel algorithm with p processors.

Linear speedup or ideal speedup is obtained when $Speedup = p$. When running an algorithm with linear speedup, doubling the number of processors doubles the speed. In practice, linear speedup is difficult to achieve because the communication cost increases with the number of records becomes large.

5.4. *The performance and analysis*

We examine the scaleup, sizeup and speedup characteristics of the PApriori algorithm.

To demonstrate how well the PApriori algorithm handles larger datasets when more cores of computers are available, we have performed scaleup experiments where we have increased the size of the datasets in direct proportion to the number of cores in the system. For dataset1, the datasets size of 1GB, 2GB, 4GB and 8GB are executed on 4, 8, 16 and 32 cores respectively. For dataset T10I4D100K, T10I4D100K, T10I4D4200K, T10I4D400K and T10I4D1800K are executed in the same way.

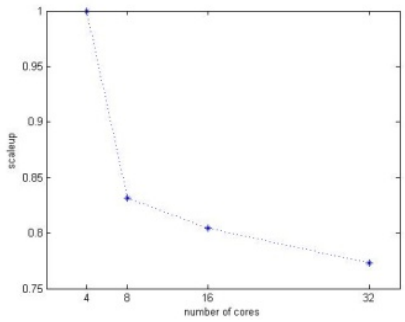
Figure4 shows the scaleup performance of the datasets. Clearly, the PApriori algorithm scales well, the scaleup fall shortly as the database and multiprocessor sizes increase. It always maintains a higher than 78% scalability for dataset1 and 80% for T10I4D100K.

To measure the performance of sizeup, we fix the number of cores to 4, 8, 16 and 32 respectively. Figure 5 shows the sizeup results on different cores. When the number of cores is small such as 4 and 8, the sizeup performances differ little. However, as more cores are available, the sizeup value for 16 or 32 cores decreases significantly compared to that of 4 or 8 cores on the same data sets. The results show sublinear performance for the PApriori algorithm, the program is actually more efficient as the database size is increased. Increasing the size of the dataset simply makes the noncommunication portion of the code take more time due to more I/O and more transaction processing. This has the result of reducing the percentage of the overall time spent in communication. Since I/O and CPU processing scale well with sizeup, we get sublinear performance.

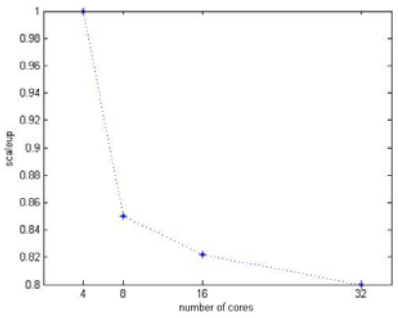
To measure the speedup, we kept the dataset constant and varied the number of cores. The number of cores varies from 4 to 32. We have performed four experiments, the size of the dataset increases from 1GB to 8GB for dataset1, and from 0.3GB to 2.4GB for T10I4D100K.

We have performed the speedup evaluation on datasets with different sizes and systems. Figure6 shows the speedup for different datasets. As the result shows, the speedup performance does however not to be very good in the case of 1GB for dataset1 and 0.3GB for T10I4D100K. This is an artifact of the small amount of data each node processing. In this case, communication cost becomes a significant percentage of the overall response time. This is easily predicted from our sizeup experiments where we notice that the more data a core processes, the less significant becomes the communication cost giving us better performance. Therefore, PApriori algorithm can deal with large

datasets efficiently. Larger datasets would have shown even better speedup characteristics.

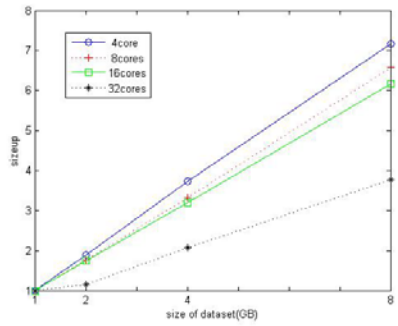


(a) Scaleup for dataset1

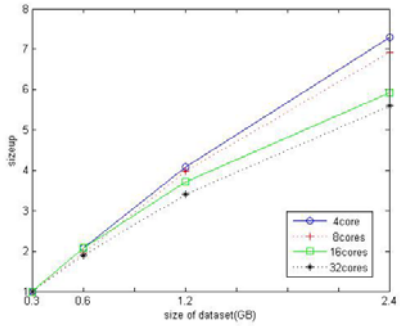


(b) Scaleup for T10I4D100K

Figure4. Scaleup performance evaluation

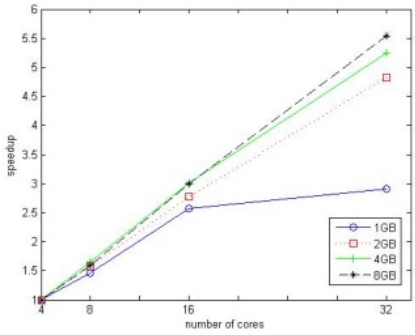


(a) Sizeup for dataset1

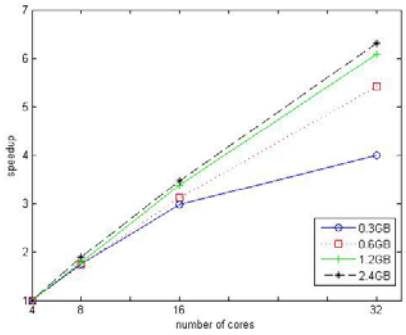


(b) Sizeup for T10I4D100K

Figure5. Sizeup performance evaluation



(a) Speedup for dataset1



(b) Speedup for T10I4D100K

Figure6. Speedup performance evaluation

To sum up, for the datasets either have many short transactions with few frequent itemsets or fewer larger transactions with many frequent itemsets, PApriori algorithm has shown good performance.

5.5. *Application to transaction logs mining*

The PApriori algorithm we proposed has been applied on some transaction logs from a telecommunications company.

The raw data contains 3249144 transaction logs consisting of 34 attributes, such as transaction number, transaction type, transaction source, user ID, product ID and so on. Each log record that one user buy or download one application program by mobile phone or PC. The aim is to find the association rules among the products.

We have done some pre-processing on the raw data. First, we choose the necessary items for association rules mining. In other words, we choose the user ID, product ID as the attributes needed. Since one user may download several products, then we combine the products downloaded by the same user. Finally, we get the data that the PApriori algorithm can deal with, i.e. each line of the dataset record the produces that one user download. For example, 50,60,61,126,1915 indicate that the user download the products whose IDs are 50,60,61,126,1915.

We run PApriori algorithm on the transaction logs and get the association rules within 10 minutes. It indicates that PApriori algorithm can deal with large real datasets showing good performance.

6. Conclusion

Searching for frequent patterns in transactional databases is considered one of the most important data mining problems. The task of finding all association rules requires a lot of computation power and memory. In this paper, we propose a fast parallel Apriori algorithm based on MapReduce, We use sizeup, speedup and scaleup to evaluate the performances of PApriori. The experimental results show that the program is actually more efficient as the database size is increased. Therefore, the proposed algorithm can process large datasets on commodity hardware effectively.

Acknowledgments

The work is supported by the National Natural Science Foundation of China (No.61175052, 61203297, 60933004, 61035003), National High-tech R&D Program of China (863 Program) (No.2013AA01A606, 2012AA011003), National Program on Key Basic Research Project (973 Program) (No.2013CB329502).

References

1. Yanbin Ye, Chia-Chu Chiang, A Parallel Apriori Algorithm for Frequent Itemsets Mining, Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06), pp. 87-93,2006
2. Jiawei Han and Micheline Kamber. Data Mining, Concepts and Techniques. Morgan Kaufmann, 2001
3. Dean J., Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. In: Proc. of Operating Systems Design and Implementation, San Francisco, CA, pp. 137–150 , 2004
4. Lammel, R. Google's MapReduce Programming Model - Revisited. Science of Computer Programming 70, 1–30, 2008
5. Borthakur, D. The Hadoop Distributed File System: Architecture and Design, 2007
6. Q. He, F.Z. Zhuang, J.C. Li, Z.Z. Shi. Parallel implementation of classification algorithms based on MapReduce. RSKT, LNAI 6401, pp. 655-662, 2010
7. W. Z. Zhao, H. F. Ma, Q. He. Parallel k-means clustering based on MapReduce. In CloudCom'09: Proceedings of the 1st International Conference on Cloud Computing, pp. 674-679, Berlin, Heidelberg, 2009
8. R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Database," Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Vol.22, Issue 2, pp. 207-216, 1993
9. Osmar R. Zaiane, Mohammad El-Hajj, Paul Lu. Fast Parallel Association Rule Mining Without Candidacy Generation, Technique Report
10. Ghemawat, S., Gobioff, H., Leung, S. The Google File System. In: Symposium on Operating Systems Principles, pp. 29-43, 2003
11. Hadoop: Open source implementation of MapReduce, Available: http://hadoop.apache.org, June 24, 2010
12. Q. He, Q. Tan, X.D. Ma, Z.Z. Shi. The high-activity parallel implementation of data preprocessing based on MapReduce. RSKT, LNAI 6401, pp. 646-654, 2010
13. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In Proc. 1994 Int. Conf. VeryLarge Data Bases, pages 487-499, Santiago, Chile, September 1994
14. Rakesh Agrawa, John C.shafer. Parallel Mining of Association Rules, IEEE transactions on knowledge and data engineering, Vol. 8, No.6, pp.962-969,1996