A fast parallel algorithm for frequent itemsets mining

Dora Souliou, Aris Pagourtzis, and Panayiotis Tsanakas *

School of Electrical and Computer Engineering
National Technical University of Athens
Heroon Politechniou 9, 15780 Zografou, Greece
{dsouliou, panag}@cslab.ece.ntua.gr, pagour@cs.ntua.gr

Abstract. Mining frequent itemsets from large databases is an important computational task with a lot of applications. The most known among them is the market-basket problem which assumes that we have a large number of items and we want to know which items are bought together. A recent application is that of web pages (baskets) and linked pages (items). Pages with many common references may be about the same topic. In this paper we present a parallel algorithm for mining frequent itemsets. We provide experimental evidence that our algorithm scales quite well and we discuss the merits of parallelization for this problem.

Keywords: *parallel data mining, association rules, frequent itemsets, partial support tree, set-enumeration tree.*

1 Introduction

The market-basket problem is described as follows: a database D of transactions is given, each of which consists of several distinct items. The goal is to determine association rules of the form $A \Rightarrow B$, where A and B are sets of items (itemsets).

In order to determine validity of a rule $A \Rightarrow B$, two quantities are taken into account: the *confidence* of the rule, which is the fraction $freq(A \cup B)/freq(A)$, where $freq(I)$ is the number of transactions that contain itemset I, and the *support* of the rule, which is equal to $freq(A \cup B)$. The *support* of the rule is usually called frequency. One is usually interested in rules the frequency of which is above a threshold t and the confidence of which is above another threshold c. Once we have found which itemsets have frequency larger than t it is only a matter of simple calculations to find rules whose confidence exceeds c. Therefore, a fundamental ingredient of discovering association rules is the generation of all itemsets the frequency of which exceeds threshold t; from now on, we will call such itemsets *t-frequent* or simply *frequent*.

Several sequential methods for computing frequent itemsets have been proposed in the literature [3], [8], [11]. Sequential algorithms however, can not cope with very large databases. In that case parallelization techniques seem to be necessary [4], [14], [13], [9]. In this paper we implement a new parallel algorithm named PMP (Parallel Multiple Pointer) which achieves a satisfactory speedup and is particularly adequate for certain types of data sets.

The paper is organized as follows. In section 2 we give a brief description of the sequential algorithm on which our parallel algorithm is based. In section 3 we present the parallel algorithm. In section 4 we give experimental results that demonstrate the efficiency on certain datasets. Conclusions and directions for future work are discussed in the last section.

* This research is supported by the PENED 2003 Project (EPAN), co-funded by the European Social Fund (75%) and National Resources (25%).

2 The Sequential Algorithm

The main difficulties in computing frequent itemsets are related to two factors: the number of transactions and the number of items. When the number of transactions increases each scan of the database becomes time consuming. A large number of items on the other hand, implies many possible itemsets and this means delays on computations and increased requirements on memory space. Some known algorithms [11], [8] in the area gave an answer to the first problem by using structures in which the database is stored in a more compact form. For the second problem techniques have been developed for reducing the number of itemsets that need to be examined e.g the well known A-priori Algorithm [3].
The sequential algorithm that we use as a basis for our parallel algorithm makes use of a structure called *Partial Support Tree* (or P-tree for short), introduced in [8], in which the whole database is stored in one scan. It is a set enumeration tree [12] that contains itemsets in its nodes. These itemsets represent either whole transactions of database D or common prefixes of transactions of D. An integer is stored in each tree node which represents the partial support of the corresponding itemset I, that is, the number of transactions that contain I as a prefix. The construction of the P-tree was described in [8]; a more detailed description can be found in [7].

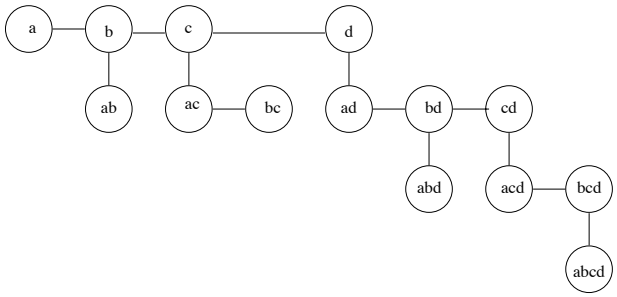

Fig. 1. An example of T-tree.

The sequential algorithm that we use for parallelization uses a second tree structure, namely the T-tree on the nodes of which candidate frequent itemsets are stored. This structure was introduced in [6] and algorithms that use it are presented in [10]. Here we propose a modification of this structure as follows: each node of the tree in the initial structure had two pointers. The first one points to the right sibling and the second one to the lexicographically smaller child. In the modified T-tree we use more than one pointers to the children. Figure 1 shows a T-tree for items a,b,c and d. For example, the node with label "d" would have one pointer to the node "ad" in the initial structure. In the proposed structure it may have up to two pointers more: one that points to the node "bd" and one that points to the node "cd". The number of pointers depends on the desired tradeoff between memory space and running time.

We give here some details of how the algorithm calculates the itemset frequencies. The algorithm traverses the P-tree and for each node visited, all subsets of this node with k items are generated (k is the current level of the T-tree). For each such itemset the nodes of the T-tree are visited in order to find the itemset. If it is found, the frequency of the node that contains the itemset is increased by the partial support of the current node of the P-tree. Suppose for example that we visit the node "abef" of the P-tree with partial support 12 and that the current level of the T-tree is three. The subsets with 3 items of "abef" are "abe", "abf", "aef", and

"bef". In order to find "abe" we visit the following nodes of the T-tree "e", "be", "abe". If "abe" is found, we increase its frequency by 12.

3 The Parallel Algorithm (PMP)

Algorithm PMP (Parallel Multiple Pointer *)*
distribute the database D to the processors in a round-robin manner;
let d_j denote the part of D assigned to processor p_j;
in each processor p_j **in parallel do**
build local P-tree from local database d_j;
(* *1st level construction* *)
for each node I of the local P-tree
compute all subsets of I with *1 item*;
for each $I \in L_1$ frequency($\{I\}$)++; (*L_1 is the first level of T-tree *)

(* *Global synchronized computation* *)
for $i := 1$ **to** *nitems* **do** (* *nitems* = number of items *)

total_frequency($\{i\}$):= parallel_sum$_{j=1}^{nprocs}$ local_frequency$_j(\{i\})$; (* *nprocs* = number of processors *)

(* *Local computation continues* *)
for $i := 1$ **to** *nitems* **do**
if total_frequency($\{i\}$) $\geq t$ **then** append $\{i\}$ to L_1; (* *all processors obtain the same list* L_1 *)

k:=2;
while L_{k-1} not empty and $k \leq$ *nitems* **do**
(* *k-th level construction* *)
set L_k to be the empty list;
for each itemset $I \in L_{k-1}$ **do**
for each item $x_i >$ last item of I **do** $I' := I \cup x_i$;
insert I' into C_k;
update pointers for I';
for each node J of the local P-tree
compute all subsets s of J with k *items*
for each s go down to the level k of T-tree and
if s is found then frequency($\{s\}$)++;
for all itemsets $I_k \in C_k$ **do**
get local_frequency$_j(\{I_k\})$ from local P-tree;
(* *Global synchronized computation* *)
for all itemsets $I_k \in C_k$ **do**
total_frequency(I_k):= parallel_sum$_{j=1}^{nprocs}$(local_frequency$_j(I_k)$);
(* *Local computation continues* *)
for all itemsets $I_k \in C_k$ **do**
if total_frequency($\{I_k\}$) $\leq t$
then delete I_k from L_k and update pointers;
$k := k + 1$;
(* *end of while-loop* *)

Fig. 2. The Parallel Algorithm

In this section we give a detailed description of the new parallel algorithm. Our approach follows the ideas of a parallel version of A-priori, called Count Distribution, which was described by Agrawal and Shafer [4]; the difference is, of course, that

PMP makes use of two tree structures P-tree for storing the database and T-tree for storing the frequent itemsets.

In the beginning, the root process distributes the database transactions to the processors in a round-robin fashion; then, each of the processors creates its own local P-tree based on the transactions that it has received. Next, each processor traverses each local P-tree and for each node visited produces all subsets with 1 item. For each such item the algorithm traverses the T-tree in order to find it and increase its frequency. When all the nodes of the P-tree have been visited the local frequencies have been calculated. The calculation of total frequencies of singletons takes place as follows. Each processor sends the local frequencies and an appropriate parallel procedure (MPI All_reduce function) sums total frequencies. This function is used to make calculations in an efficient way. The result is distributed to all processors so that each one ends up with the same list L_1 of singletons. Following each processor visits the nodes of L_1 and removes all infrequent singletons. In this step each processor has the same list of frequent singletons. During k-level computation (for each $k \geq 2$), all processors first generate the same list of candidate itemsets C_k from the common list L_{k-1}. Then the same procedure as that followed for the first level allows each processor to obtain the total frequencies for all itemsets of C_k, and finally to derive list L_k by removing infrequent itemsets of C_k.

A detailed description of the algorithm is given in Figure 2.

4 Numerical Results

Our experimental platform is an 8-node Pentium III dual-SMP cluster interconnected with 100 Mbps FastEthernet. Each node has two Pentium III CPUs at 800 MHz, 256 MB of RAM, 16 KB of L1 I Cache, 16 KB L1 D Cache, 256 KB of L2 cache, and runs Linux with 2.4.26 kernel. We use MPI implementation MPICH v.1.2.6, compiled with the Intel C++ compiler v.8.1. The experiments presented below study the behavior of the algorithm in terms of running time and parallel efficiency.

We have implemented our parallel algorithm (PMP) using synthetic datasets generated by the IBM Quest generator, described in [3]. We have generated our datasets using the following parameters:

- D: the number of transactions
- N: the number of items
- T: the average transaction length

We have used four synthetic datasets in our experiments, with number of transactions varying from 100K to 500K and number of items between 100 and 200; For the datasets with 100 items the average transaction length that we used was 10 ($T = 10$ 10 items per transaction) while for the itemsets with 200 items the average transaction length was 20. We have chosen relatively small minimum frequency threshold values varying from 0.1% to 1%.

In Figures 3 and 4 we observe the time performance of the algorithm for number of transactions 100, 200K, 300K and 500, number of items 100 and threshold values 0.1%, 0.5% and 1%.

We observe that in most cases PMP algorithm is a time efficient algorithm. For the dataset D100KN100T10 PMP gives satisfactory results for every number of processors and every threshold value. For the dataset D200KN100T10 the results show that the algorithm achieves a good time performance with the exception of one processor and threshold values 200 and 1000. For the datasets with 300K or 500K transactions and 100 items the algorithm scales well enough with the same

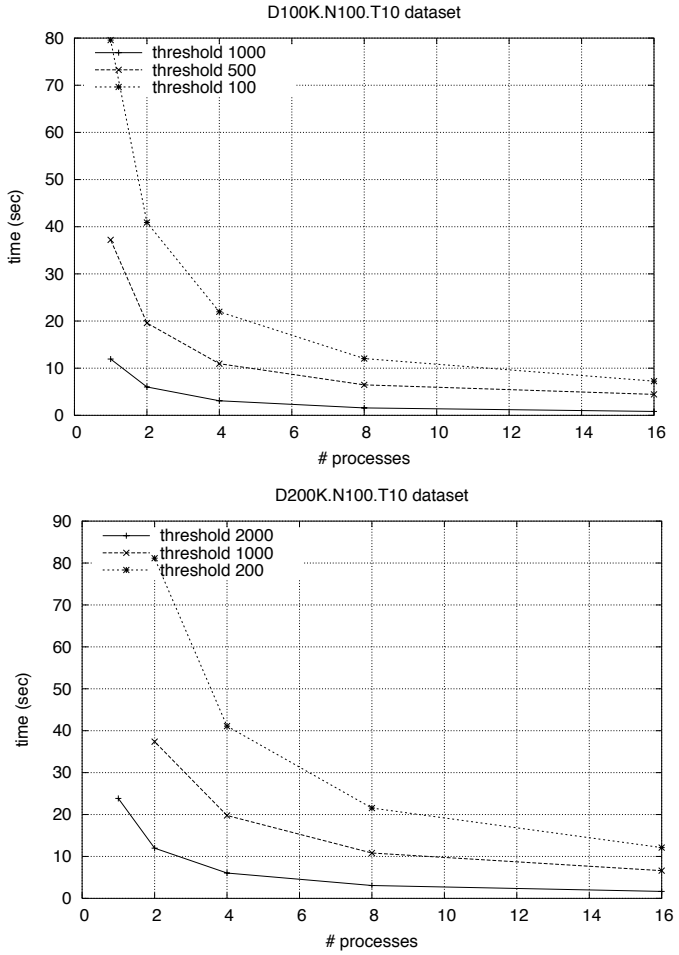

Fig. 3. Time performance for datasets D100K.N100.T10 and D200K.N100.T10

exception which now expands to two processors. This drawback is due to the inefficient memory space which slows down the computations. The same remarks could be made in Figure 5. The two tree structures are space consuming and that is the main reason for not getting satisfactory results in cases of one or two processors. On the other hand the PMP algorithm behaves equally well when the number of processors increases and that renders the parallelization meaningful. This is more obvious in Figure 6, and in Figure 7 where we can see speedups even above 90%. Our technique is efficient when the P-tree fits into the local memory of each processor. If it is not possible due to the limited number of processors our technique would require special customization in order to reduce disk accesses.

5 Conclusions

In this work we have developed and implemented the Parallel Multiple Pointer algorithm and investigated the efficiency of this parallelization technique. The use of P-tree and T-tree has facilitated the process of computing frequencies. This process is further accelerated by the use of parallelization. In particular, each processor handles a part of the database and creates a small local P-tree that can be kept in memory, thus providing a practicable solution when dealing with extremely large datasets.

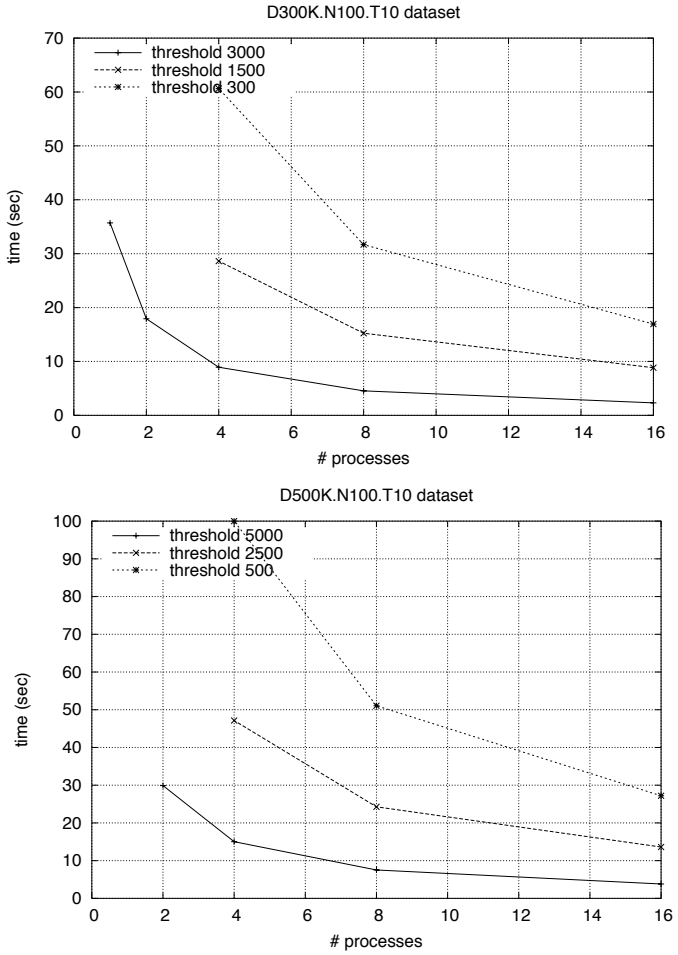

Fig. 4. Time performance for datasets D300K.N100.T10 and D500K.N100.T10

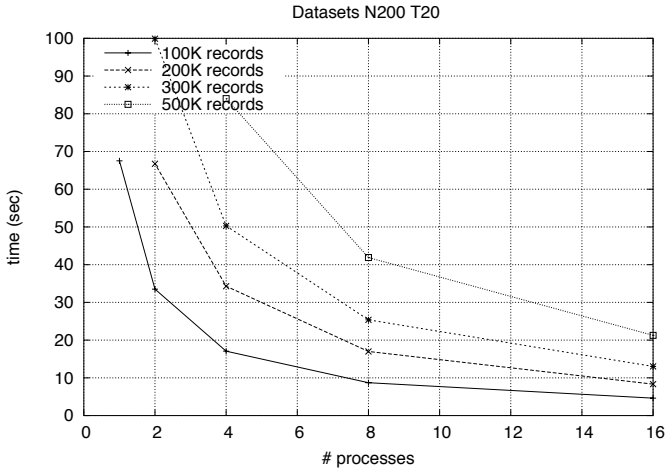

Fig. 5. Time performance for datasets with 200 items and average transaction length 20

We have implemented the algorithm using message passing, with the help of the Message Passing Interface (MPI). Experiments show that the above described parallel strategy is generally competitive. However the time efficiency sometimes

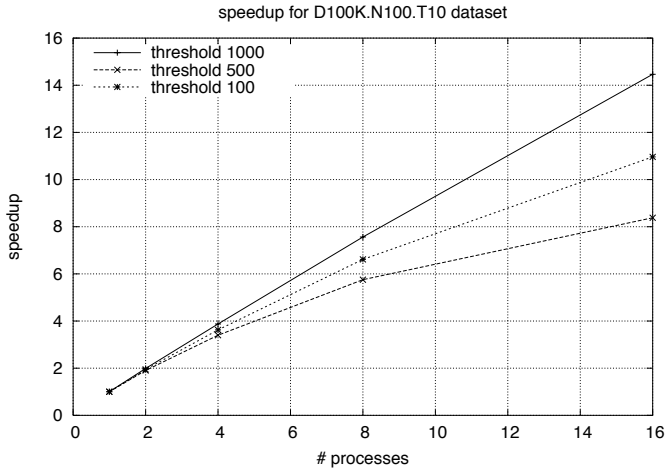

Fig. 6. Speedup obtained by PMP for dataset D100K N100 T10.

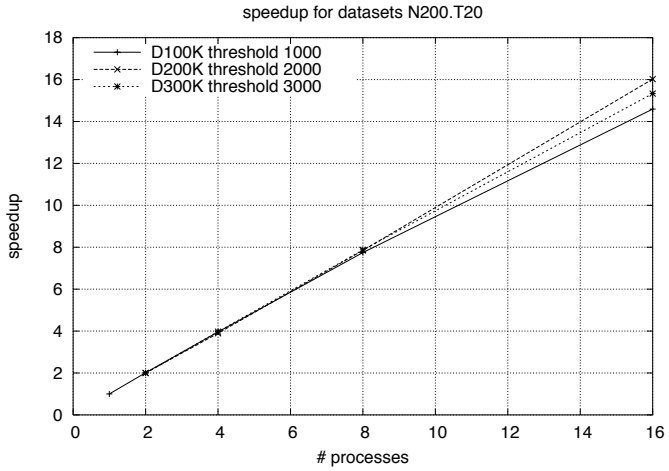

Fig. 7. Speedup obtained by PMP for datasets with 100 items.

causes problems in memory space. An answer to this problem could be the reduction on the number of pointers used to construct the T-tree. This could cause delays on frequency computations but it is necessary when the datasets used are extremely large and the needs in memory space are increased. An interesting research direction is therefore to fine tune the number of pointers used in the T-tree structure in order to balance the needs in space and time.

References

1. S. Ahmed, F. Coenen, and P.H. Leng: A Tree Partitioning Method for Memory Management in Association Rule Mining. In Proc. of Data Warehousing and Knowledge Discovery, 6th International Conference (DaWaK 2004), Lecture Notes in Computer Science 3181, pp. 331–340, Springer-Verlag 2004.
2. R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In Proc. of the 1993 ACM SIGMOD Conference on Management of Data, Washington DC, pp. 207–216 1993.
3. R. Agrawal and R. Srikant. Fast Algorithms for mining association rules. In Proc. VLDB'94, pp. 487–499 1994.

4. R. Agrawal and J.C. Shafer. Parallel Mining of Association Rules. *IEEE Transactions on Knowledge and Data Engineering* 8(6), pp. 962–969, 1996.
5. R. J. Bayardo, Jr. and R. Agrawal. Mining the Most Interesting Rules. In Proc. of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 145–154, 1999.
6. F. Coenen, G. Goulbourne, and P. Leng. Computing Association Rules using Partial Totals. In L. De Raedt and A. Siebes eds, *Principles of Data Mining and Knowledge Discovery* (Proceedings of the 5th European Conference, PKDD 2001, Freiburg), Lecture Notes in AI 2168, Springer-Verlag, Berlin, Heidelberg: pp. 54–66 2001.
7. F. Coenen, G. Goulbourne, and P. Leng. Tree Structures for Mining Association Rules. *Data Mining and Knowledge Discovery*, pp. 25–51, 8 2004
8. G. Goulbourne, F. Coenen, and P. Leng. Algorithms for Computing Association Rules using a Partial-Support Tree. *Journal of Knowledge-Based Systems* pp. 141–149, 13 2000.
9. F. Coenen, P. Leng, and S. Ahmed. T-Trees, Vertical Partitioning and Distributed Association Rule Mining. In Proc. of the 3rd IEEE International Conference on Data Mining pp. 513–516, ICDM 2003.
10. F. Coenen, P. Leng, A. Pagourtzis, W. Rytter, D. Souliou. Improved Methods for Extracting Frequent Itemsets from Interim-Support Trees. In Proc. of AI 2005.
11. J. Han, J. Pei, Y.Yin, and R. Mao. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. *Data Mining and Knowledge Discovery*, pp. 53–87, 8 2004.
12. R. Raymon. Search Through Systematic Search Enumeration. In Proc. of the 3rd Internaltional Conference on Principles of Knowledge Representation and Reasoning, pp. 539–550 1992.
13. D. Souliou, A. Pagourtzis, N. Drosinos, P. Tsanakas. Computing Frequent Itemsets in Parallel Using Partial Support Trees. in Proceedings of 12th European PVM/MPI Conference (Euro PVM/MPI 2005), Sorrento (Naples), Italy, Lecture Notes in Computer Science 3666, pp. 28-37, Springer-Verlag 2005
14. Osmar R. Zaine Mohammad El-Hajj Paul Lu. Fast Parallel Association Rule Mining without Candidate Generation. In Proc. of ICDM 2001.