Distributed and Shared Memory Algorithm for Parallel Mining of Association Rules

J. Hernández Palancar, O. Fraxedas Tormo, J. Festón Cárdenas, and R. Hernández León

Advanced Technologies Application Center (CENATAV), 7a ♯ 21812 e/ 218 y 222, Rpto. Siboney, Playa, C.P. 12200, La Habana, Cuba
{jpalancar,ofraxedas,jfeston,rhernandez}@cenatav.co.cu

Abstract. The search for frequent patterns in transactional databases is considered one of the most important data mining problems. Several parallel and sequential algorithms have been proposed in the literature to solve this problem. Almost all of these algorithms make repeated passes over the dataset to determine the set of frequent itemsets, thus implying high I/O overhead. In the parallel case, most algorithms perform a sum-reduction at the end of each pass to construct the global counts, also implying high synchronization cost. We present a novel algorithm that exploits efficiently the trade-offs between computation, communication, memory usage and synchronization. The algorithm was implemented over a cluster of SMP nodes combining distributed and shared memory paradigms. This paper presents the results of our algorithm on different data sizes experimented on different numbers of processors, and studies the effect of these variations on the overall performance.

1 Introduction

The discovery of Association Rules is one of the most productive fields in the development of sequential algorithms as well as parallel algorithms for Data Mining. Simultaneously, with the evolution of these algorithms the possible applications of Association Rule Mining (ARM) has also been extended together with a corresponding increase in the volume of the databases to be mined. As a consequence of the latter, even using the most efficient sequential ARM algorithms, it is not possible to reduce the support threshold to the desired level without causing a combinatorial explosion in the number of identified frequent itemsets coupled with a corresponding computational overhead.

The situation described above confirms the relevance of the application of Parallel Computing for Association Rule Mining, which is a very active global research area. The main challenges of Parallel Computing are: load balancing, minimization of the inter-process communication overhead, the reduction of synchronization requirements and effective use of the memory available to each processor.

These issues must be taken into account in the development of efficient parallel algorithms for Association Rule Mining; basic references to consider are [8,9,10].

P. Perner (Ed.): MLDM 2007, LNAI 4571, pp. 349–363, 2007.

The prototypical ARM application is the analysis of sales or *basket data* [1]. The task can be broken into two steps. The first step consists of finding the set of all frequent sets of items that can be the transaction database. The second step consists of forming implication rules among the sets of items found; the latter can be done in a straightforward manner so we will focus on the first step.

In previous papers [11,12], we proposed a new algorithm called CBMine (Compressed Binary Mine) for mining association rules and frequent itemsets. Its efficiency is based on a compressed vertical binary representation of the database. CBMine has been compared with other efficient ARM algorithms to obtain frequent itemsets, including: Fp-growth (implementation of Bodon), MAFIA and Patricia Trie . The experimental results obtained showed that CBMine gives the best performance in most cases, especially on big and sparse databases.

In this paper we propose a new parallel algorithm based in CBMine named ParCBMine(Parallel Compressed Binary Mine). ParCBMine exploits efficiently the trade-offs between computation, communication, memory usage and synchronization. The algorithm was implemented over a cluster of SMP nodes combining distributed and shared memory paradigms. Section 5 of this paper shows the experimental results of our algorithm on different data sizes, evaluated on different numbers of processors, and studies about the effect of these variations on the overall performance.

The paper is organized as follows: the next section is dedicated to related work; in section 3 we give a formal definition of association rules; section 4 contains a description of ParCBMine algorithm; experimental results are discussed in section 5; and some conclusions are presented in section 6.

2 Related Work

Until now the great majority of the parallel algorithms for Association Rule Mining are based on the sequential Apriori algorithm. An excellent survey made by Zaki in 1999 [18] classifies different algorithms up to that date, according to the load balance strategy, the architecture and the type of parallelism used in the algorithm. Other important references are [2,15,16,19,20,22].

Apriori algorithm has been the most significant of all sequential algorithms proposed in the literature. Yet, directly adapting an Apriori-like algorithm will not significantly improve performance over frequent itemsets generation. To perform better than Apriori-like algorithms, we must focus on the disadvantages associated with this approach. The main challenges include synchronization, communication minimization, work-load balancing, finding good data layout and data decomposition, and disk I/O minimization.

Recently interesting parallel ARM has increased as a result of this early work. We can identify a number of early ARM algorithms: *Count Distribution*, *Data Distribution* and *Candidate Distribution*. These algorithms were first presented in [2] and offer a fairly simple parallelization of Apriori using different paradigms of parallelization; namely data-parallelism and control-parallelism, or a combination of both. In Count Distribution the dataset is partitioned equally among

the nodes of the parallel system. Each of these nodes computes the local support for every candidate k-itemset in the iteration k. At the end of each iteration by exchanging the local supports the global support is generated and the frequent itemsets determined. The nodes must be synchronized to receive the candidate itemsets and the coordinator node must wait for all local counts to generate the global support. The former factors affects communication cost and load-balancing; however the Count Distribution algorithm represents a good first step and can be the core of subsequent implementations that address these issues. In Data Distribution the set of candidate itemset is partitioned into disjoint sets and these are sent to different nodes. The problem in this parallel version of Apriori is the magnitude of the huge communications required at the end of each iteration. In Candidate Distribution load-balancing is thus the main target, selectively replicating the dataset so that each processor proceeds independently. The algorithm requires redistribution of the dataset at level l, this is identified using a heuristic approach.

There are other parallel versions of well-known sequential algorithms like PDM (parallelizing DHP) [3]. But this was not a successful attempt due to its poor performance with respect to the above algorithms. Other algorithms that address the size of candidacy and better pruning techniques are DMA and FDM presented in [4,5]. In [7] the Optimized Distributed Association Mining (ODAM) algorithm is proposed based on Count Distribution which reduce both the size of the average transaction and the number of message exchanges among nodes in order to achieve better performance.

The Eclat(Equivalence CLass Transformation) algorithm [17] uses an itemset grouping scheme based on equivalence classes and partitions them into disjoint subsets among the processors. At the same time Eclat makes use of a kind of vertical representation of the dataset and then selectively replicates it so that each processor has the portion of the dataset it needs for calculations. After the initial phase the algorithm eliminates the need for later communication or synchronization. The algorithm scans the local partition of the dataset three times, therefore diminishing the I/O overhead. Unlike other earlier algorithms, Eclat uses simple intersection operations to compute frequent itemsets and does not use complex hash tables structures. The main deficiency of this algorithm lies in the need for a proper heuristic to achieve a suitable load balance among the processors as of the L_2 partitioning, because the equivalence classes do not have the same cardinality.

In [15] a collection of algorithms with different partitioning and candidate itemsets count schemes are described. Like Eclat, all of them assume a vertical representation of the dataset (tidlits per item), which facilitates the intersection operation of tids of items that make up an itemset. The dataset is duplicated in a selective fashion to reduce synchronization. Two of these algorithms (Par-Eclat and Par-MaxEclat) are based on the classes of equivalence formed by the candidates first item, whereas the other two algorithms (Par-Clique and Par-MaxClique) use the maximum closed hypergraph to partition the candidates.

In [21], a parallel algorithm is proposed for Association Rule Mining that uses a classification hierarchy named HPGM (Hierarchical Hash Partitioned Generalized Association Rule Mining). In this algorithm, the available memory space is completely used identifying the frequent occurrence of candidates itemsets and replicating them to all processors, considering that frequent itemsets can be locally processed without communication. This way the load asymmetry among processors can be effectively reduced.

3 Problem Definition

In this section we define some necessary terminology to facilitate understanding of the following sections. In this context it should be noted that we are only focused on the problem of identifying frequent itemsets on large databases.

A *dataset* is a set of *transactions* and each of these is composed by a transaction identifier (TID) and a set of *items*. The items in a transaction may represent a shopping list in a supermarket by a customer (known as basket data) or words in a document or stocks movements. A set of items, called *itemset* is *frequent* if it is contained in a number of transactions above a user-specified threshold (minimum support-*minsup*).

An itemset with k items will be referred to as *k-itemset* and its support will be denoted as $X.sup$, where X is the k-itemset in question; support is represented as a percentage rather than an absolute number of transactions.

More formally: $I = \{i_1, i_2, ..., i_n\}$ be a set of n distinct items. Each transaction T in the dataset D contains a set of items, such that $T \subseteq I$. An itemset is said to have a support s if $s\%$ of the transaction in D contains the itemset.

4 ParCBMine

In this section we describe the parallel version of the CBMine algorithm which we have named ParCBMine (Parallel CBMine).

ParCBMine takes advantage of the vertical representation of the dataset as in CBMine and combines suitably the parallel programming models of shared and distributed memory using the libraries *pthreads* (for multithreads programming) and MPI (for message passing programming) respectively.

The mixture of multithreads programming and message passing in ParCBMine was done based on the fact that the algorithm was implemented over a cluster with SMP (Symmetric Multi-Processing) nodes for the parallel processing managed by a GNU/Linux operating system; each node is composed by a dual processor. All processors are Intel Xeon with *hyperthreading* technology, which provides up to four threads on each node.

Although the algorithm is not tied to the number of real threads (processors) that could be deployed on each node, this is an important element on the scalability of ParCBMine, because it allows the use of global information in the shared memory of each node in a better way, i.e., in candidate generation and support counting. Many authors refer to this as “intra-node parallelism”, and in

certain way we have incorporated some aspects of the Candidate Distribution algorithm, in this case by means of multithread programming using the Pthreads library.

4.1 CBMine Algorithm

CBMine is a breadth-first search algorithm with a VTV organization, that uses compressed integer-lists for itemset representation.

Let T be the binary representation of a database, with n filtered items and m transactions. Taking from T the columns associated with frequent items, each item j can be represented as a list I_j of integers (integer-list) of word size w, as follows:

$$I_j = \{W_{1,j}, \ldots, W_{q.j}\}, q = \lceil m/w \rceil, \tag{1}$$

where each integer of the list can be defined as:

$$W_{s,j} = \sum_{r=1}^{min(w, m-(s-1)*w)} 2^{(w-r)} * t_{((s-1)*w+r),j}. \tag{2}$$

The upper expression $min(w, m-(s-1)*w)$ is included to consider the case in which the transaction number $(s-1)*w+r$ does not exist due to the fact that it is greater than m. The value $t_{i,j}$ is the bit value of term j in the transaction i.

This algorithm iteratively generates a prefix list PL_k. The elements of this list have the format: $\langle Prefix_{k-1}, CA_{Prefix_{k-1}}, Suffixes_{Prefix_{k-1}} \rangle$, where $Prefix_{k-1}$ is a $(k-1)$-itemset, $CA_{Prefix_{k-1}}$ is the corresponding compressed integer-list, and $Suffixes_{Prefix_{k-1}}$ is the set of all suffix items j of k-itemsets extended with the same $Prefix_{k-1}$, where j is lexicographically greater than every item in the prefix and the extended k-itemsets are frequent. This representation not only reduces the required memory space to store the integer-lists but also eliminates the Join step described in Apriori algorithm.

The Prune step of Apriori algorithm is optimized by generating PL_k as a sorted list according to the prefix field and, for each element, by the suffix field.

In order to determine the support of an itemset with a compressed integer-list CA, the following expression is considered:

$$Support(CA) = \sum_{\langle s, B_s \rangle \in CA} BitCount(B_s), \tag{3}$$

where $BitCount(B_s)$ represents a function that calculates the Hamming Weight of each B_s.

Although this algorithm uses compressed integer-lists of non null integers (CA) for itemset representation, in order to improve the efficiency, we maintain the initial integer-lists (including the null integers) $I_j = \{W_{1,j}, \ldots, W_{q,j}\}$ associated with each large 1-itemset j. This consideration allows direct accessing for any I_j the integer position defined in CA.

The above allows us to define the following formula (notice that this function represents a significant difference and improvement with respect to other methods):

$$CompAnd(CA, I_j) = \{\langle s, B'_s\rangle | \langle s, B_s\rangle \in CA, B'_s = (B_s and W_{s,j}), B'_s \neq 0\}. \quad (4)$$

Note that the cardinality of CA is reduced as the size of the itemsets increases due to the downward closure property; thus the application of identities 3 and 4 becomes more efficient.

The complete CBMine algorithm is presented in Table 1.

Table 1. CBMine algorithm

Algorithm 1: CBMine

```
1  L_1 = {large 1-itemsets} ;                          // Scanning the database
2  PL_2 = {⟨Prefix_1, CA_{Prefix_1}, Suffixes_{Prefix_1}⟩};
3  for k = 3; PL_{k-1} ≠ ∅; k++ do
4      forall ⟨Prefix, CA, Suffixes⟩ ∈ PL_{k-1} do
5          forall item j ∈ Suffixes do
6              Prefix' = Prefix ∪ {j};
7              CA' = CompactAnd(CA, I_j);
8              forall (j' ∈ Suffixes) and (j' > j) do
9                  if Prune(Prefix' ∪ {j'}, PL_{k-1}) and Support(CompactAnd(CA',
                   I_{j'})) ≥ minsup then
10                     Suffixes' = Suffixes' ∪ {j'};
11                 end
12                 if Suffixes' ≠ ∅ then
13                     PL_k = PL_k ∪ {⟨Prefix', CA', Suffixes'⟩};
14                 end
15             end
16         end
17     end
18 end
19 Answer=⋃_k L_k ;                                     // L_k is obtained from PL_k
```

Note that this algorithm only scans the dataset once in the first step.

4.2 Intelligent Block Partitioning

Given PL_{k-1} we need to partition it among the threads in the most efficient manner. In the literature we can identified several partitioning techniques, such as Bitonic Partitioning from Zaki [19]. Nevertheless, given the features of the sequential algorithm, we achieve the best results making a block partitioning, so the information to be processed by each thread was not fragmented.

For this purpose we develop Intelligent Block Partitioning (IBP), dynamically recomputing the load balance for each thread in a straightforward manner. We

use equation 5 to compute the work load generated by a Prefix based on the size of its Suffixes. The pseudo-code of IBP is given in Table 2.

$$G(x) = \frac{x(x-1)}{2} \tag{5}$$

Table 2. Intelligent Block Partitioning algorithm

Algorithm 2: IBP

```
1  Total = Σ_{j=0}^{|PL_{k-1}|} G(|Suffixes_{Prefix_j}|);          /* Total Work Load */
2  Ideal = Total / MaxThreads;                                    /* Ideal Work Load for each thread */
3  i = 1;
4  load = starts[0] = 0;
5  forall ⟨Prefix, CA, Suffixes⟩ ∈ PL_{k-1} do
6      load = load + G(|Suffixes|);
7      if load > Ideal then
           /* Set block boundaries                                */
8          start[i] = stop[i − 1] = ⟨Prefix, CA, Suffixes⟩;
           /* Dynamically recompute Total and Ideal Work Load     */
9          Total = Total- load + G(|Suffixes|);
10         Ideal = Total / (MaxThreads − i);
11         load = G(|Suffixes|);
12         i++;
13     end
14 end
15 stop[i] = ⟨Prefix_{|PL_{k-1}|}, CA_{Prefix_{|PL_{k-1}|}}, Suffixes_{Prefix_{|PL_{k-1}|}}⟩;
```

The aforementioned partition strategy is one of the improvements ParCBMine introduces over its sequential counterpart, and this can be verified in the experimental results.

4.3 ParCBMine Algorithm

Considering a master-slave framework, typical of parallel clusters, in the first pass, the master node or coordinator determines the global L_1 and partitions the dataset D in N equitable segments and sends each one of them to the corresponding node that makes up the cluster, of this way ParCBMine like Count Distribution, adopts a horizontal partitioning of the dataset thus using "inter-node parallelism", in this case the communication among the nodes is made by means of message passing using the MPI library.

The first pass is special. For all other passes $k > 1$, the algorithm works as follows:

1. Each master-thread process $P_j(j = 1, N)$ generates all the set C_k, using all the frequent itemsets L_{k-1} created at the end of pass $k - 1$. Notice

that every process has the same L_{k-1}, so they will generate identical C_k. Threads $P_i(i = 1, N \times MaxThreads)$ running in the same node, share the same memory structure for L_{k-1}, C_k and $D_j(j = 1, N)$.

2. The master-thread process P_j creates $MaxThreads-1$ new threads and each one of these makes a pass over D_j data partition and develops local support counts for a portion of the candidates in C_k which was previously partitioned using the IBP strategy. With this, the local C_k at node j, is partitioned equitably and each thread of the process develops the support count of its candidates without making any synchronization to access the memory, since the support count is developed on a reserved memory structure for each candidate, taking advantage of the vertical representation of the dataset.
3. The master-thread process P_j sends the local counts of C_k to the master node or coordinator, in order to make an *all-reduce* operation to generate the global counts of C_k. Master-thread processes are forced to synchronize in this step.
4. The master node or coordinator computes L_k from C_k. If L_k is not empty the coordinator sends it to the master-thread process P_j and continues on to the next pass.

Notice that unlike Count Distribution we have replaced the word processor by process, since given the characteristics of the hardware of our cluster the amount of processes is greater than the amount of processors, and can be expressed by $N \times MaxThreads$, where: N is the number of nodes and $MaxThreads$ is the maximum number of threads per node, in our particular case $MaxThreads = 4$ considering the use of the Hyperthreading technology.

Unlike PAR-DCI algorithm [13], in which the local dataset is partitioned yet again into as many portions as threads that were possible to deploy, in step 2 of ParCBMine a more efficient solution was adopted. We distribute the candidate support count among the threads by partitioning the candidate set into disjoint parts of approximately the same size, without the need for semaphoric operations to control memory access.

4.4 Complexity Analysis

In this section we will evaluate the complexity of our algorithm in three different contexts, first assuming the use of shared memory model, second employing the distributed memory model, and lastly the solution proposed by us of fusing the models of shared memory and distributed memory.

Given that our algorithm is intended for a parallel framework based on an SMP cluster, it is important to indicate that if we had used only a distributed memory model based on message passing, like other authors have done, the performance of the algorithm would have suffered considerably.

It is well known that in any algorithm based on Count Distribution the scalability degrades as the number of dataset partitions increases, due to the amount of information that each MPI process receives in each pass when synchronizing the processes in order to develop global support counts of itemsets in the

candidate set. The amount of information is $(N-1) \times |C_k|$, where: N is the number of MPI processes with a dataset portion assigned to it and $|C_k|$ is the cardinality of the candidate set generated in each pass and for which each MPI process computes a local support count. Bear in mind that whatever the value of N is, the cardinality of C_k does not change. This is the reason why the reduction of local set C_k is an issue that has been and continues to be a research objective. From the literature one suggested approach involves the use of probabilistic estimations of local support, see [6,14].

Given that we are using SMP nodes the advantages offered by the data locality would be wasted since all the MPI processes running in each node will try to equitably distribute the total physical memory of the node, reserving equal amounts of memory for data structures to store L_{k-1} and C_k, as well as for the dataset partition assigned to the node. For the analysis lets assume that the problem size remains constant, so the amount of candidates will be the same in each case and will be denoted as $|C_k|$.

In the development of a parallel algorithm the most common notation for the execution times are (if we consider a problem of size m running in p processors): Sequential computation denoted by $\sigma(m)$, Parallel execution time (computation that can be performed in parallel) denoted by $\varphi(m)$ and Parallel overhead (communication and synchronization, etc) denoted by $\kappa(m)$. For the experiments performed, the sequential plus the parallel execution time was considered as the parallel execution time because of the characteristic of the CBMine, the part that can not be parallelized is less than 1% of total execution time. For that reason, the two times measured were: **Parallel execution time** and **Parallel overhead**.

Shared Memory: L_{k-1} is partitioned in disjoint sets using IBP, support is develop from the common data base.

Algorithm 3: Shared Memory

$while(L_{k-1} \neq \emptyset)$; `/* Level Iterator */`
$C_k^t = IBP(L_{k-1}, t)$; `/*` $C_k = \bigcup C_k^t$ `t=1,..,MaxThreads */`
$foreach(X \in C_k^t)$; `/* Count each X in the DB */`
$if(sup(X, DB) \geq minsup)$ $L_k = L_k \bigcup \{X\}$

where: $\varphi(m) = |C_k| * |DB|$, $\kappa(m) = \emptyset$.

$$ShTime = \sum_k \frac{|C_k| * |DB|}{p} = \ldots = \sum_k max|C_k^t| * |DB| \tag{6}$$

Distributed Memory: The DB is partitioned among the quantity process denoted by P, each one has a copy of L_{k-1}, count the local support and exchange it (i.e. Message Passing). For example: if the nodes are single processor $P = N$; in case that the nodes are SMP then $P = N$ x p, where: N is the quantity nodes, and p is the number of processors in each node.

Algorithm 4: Distributed Memory

$DB_{id} = Partition(DB, id)$; /* Horizontal partitioning */
$while(L_{k-1} \neq \emptyset)$; /* Level Iterator */
  $C_k = GenerateCandidate(L_{k-1})$; /* C_k is the same for each process */
  $foreach(X \in C_k)$; /* Count each X in the DB_{id} */
    $local[x] = sup(X, DB_{id})$; /* Local support */
  $global = InterchangeAndSum(local)$; /* All to All */
  $foreach(X \in C_k)$; /* global support */
    $if(global[X] \geq minsup)$ $L_k = L_k \bigcup \{X\}$

where: $\varphi(m) = |C_k| * |DB|$, $\kappa(m) = \sum_k InterchangeAndSum = \sum_k |C_k| * 2 * P = \sum_k |C_k| * 2 * N * p$.

$$DsTime = \sum_k \frac{|C_k| * |DB|}{N * p} + \kappa(m) = \sum_k |C_k| * max|DB_{id}| + \kappa(m) \quad (7)$$

Share + Distributed Memory Solution (Hybrid memory): In the previous cases the P processes were sharing the memory or completely distributed, in this case there will be N MPI-processes in correspondence with the quantity of nodes and in each node p processes sharing memory, for that reason $P = N$, because for the communication among nodes is not considering the quantity of processes in each node. In this case a process master for each node is in charge of the communication with the remaining nodes and of distributing tasks to the other processes that are in its node.

Algorithm 5: Hybrid Memory

$DB_N = Partition(DB, N)$; /* Horizontal partitioning */
$while(L_{k-1} \neq \emptyset)$; /* Level Iterator */
  $C_k^t = IBP(L_{k-1}, t)$; /* $C_k = \bigcup C_k^t$ t=1,..,MaxThreads */
  $foreach(X \in C_k^t)$; /* Count each X in the DB_N */
    $local[x] = sup(X, DB_N)$; /* Local support */
  $if(master(t))$ then $global = InterchangeAndSum(local)$
$foreach(X \in C_k^t)$; /* global support */
    $if(global[X] \geq minsup)$ $L_k = L_k \bigcup \{X\}$

where: $\varphi(m) = |C_k| * |DB|$, $\kappa(m) = \sum_k InterchangeAndSum = \sum_k |C_k| * 2 * N$.

$$HyTime = \sum_k \frac{|C_k| * |DB|}{N * p} + \kappa(m) = \sum_k max|C_k^t| * max|DB_N| + \kappa(m) \quad (8)$$

If we are using the same processes quantity for each memory model it is very simple to observe that:

$$ShTime < HyTime < DsTime \tag{9}$$

5 Results

All the experiments described in this section were performed on a SMP cluster of which we used 6 nodes: the master node and five working nodes. Each working node is equipped with two Intel Xeon processors at 2.4 GHz based on hyperthreading technology, 512 MB of RAM, 40 GB of disk space and 1 Gb/s Fast Ethernet card. The working nodes are connected to a master node by a network switch Gigabit Ethernet. The master node is equipped with two Intel Xeon processors at 3.06 GHz based on hyperthreading technology too, 2 GB of RAM, and a disk array of five disks, 36.4 GB of disk space each (total 145.6 GB).

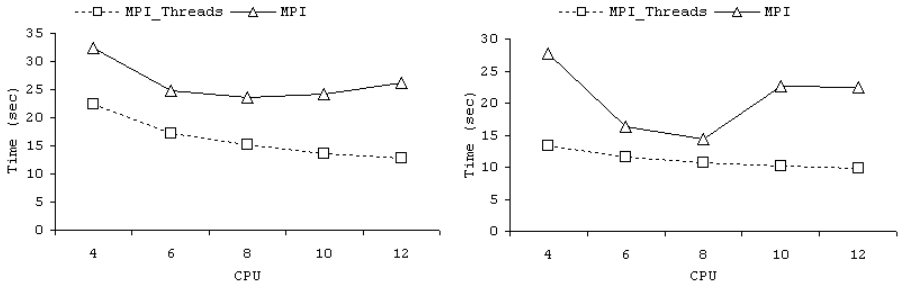
We ran two versions of the parallel algorithm: one using the distributed memory model implemented with MPI, so there were two processes for each node, i.e. one process by physical CPU; and the other combining the distributed memory model (MPI again) and the shared memory model implemented using Pthreads, in this case there were 4 threads per node sharing the same memory, considering the use of hyperthreading technology.

The experiments were made with one synthetic (T40I10D600K composed by 600000 transactions and 999 items) and one textual dataset (Kosarak composed by 990007 transactions and 41935 items) (available from FIMI repository-http://fimi.cs.helsinki.fi). The Kosarak Dataset was provided by Ferenc Bodon and contains (anonymized) click-stream data of a Hungarian on-line news portal. The T40I10D600K was created using an IBM generator(www.almaden.ibm.com/cs/quest/syndata.html).

In the first experiment we compared the execution times between ParCBMine using MPI plus Threads and ParCBMine using MPI only. The Figure 1 (a) and Figure 1 (b) show that the parallel execution time is reduced to half when the number of processors is doubled, for both implementations. The communication time overhead is stable in the first case (with the use of MPI + Tthreads) but increases linearly in the second (MPI only).

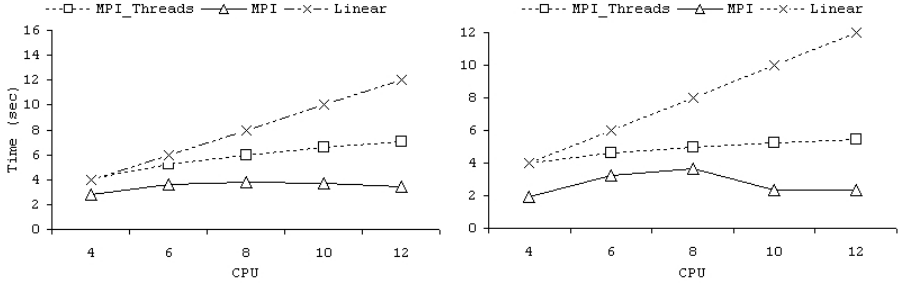
The second experiment was performed to analyze the SpeedUp (Figure 2) and Efficiency (Figure 3) of both implementations of ParCBMine algorithm. Figure 2 shows that the implementation of ParCBMine algorithm using MPI plus Threads scales better when the number of processors is increased in spite of the communication time overhead. Likewise, notice that in Figure 3 the degradation of the efficiency for ParCBMine implementation using MPI plus Threads is much slower with the increase of the communication time overhead.

In the third experiment we analyzed the algorithm scalability, thus we considered the case where both datasets were so big that they could not fit in the main memory of any node, increasing databases size in proportion with the number of nodes (N), these datasets were named T40I10D600KxN and KosarakxN. The



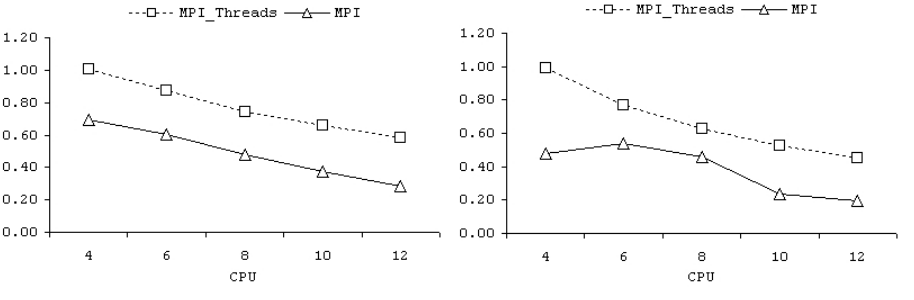
(a) (b)

Fig. 1. Execution time comparison: (a) T40I10D600K, $minsup = 0.01$, (b) Kosarak, $minsup = 0.002$



(a) T40I10D600K (b) Kosarak

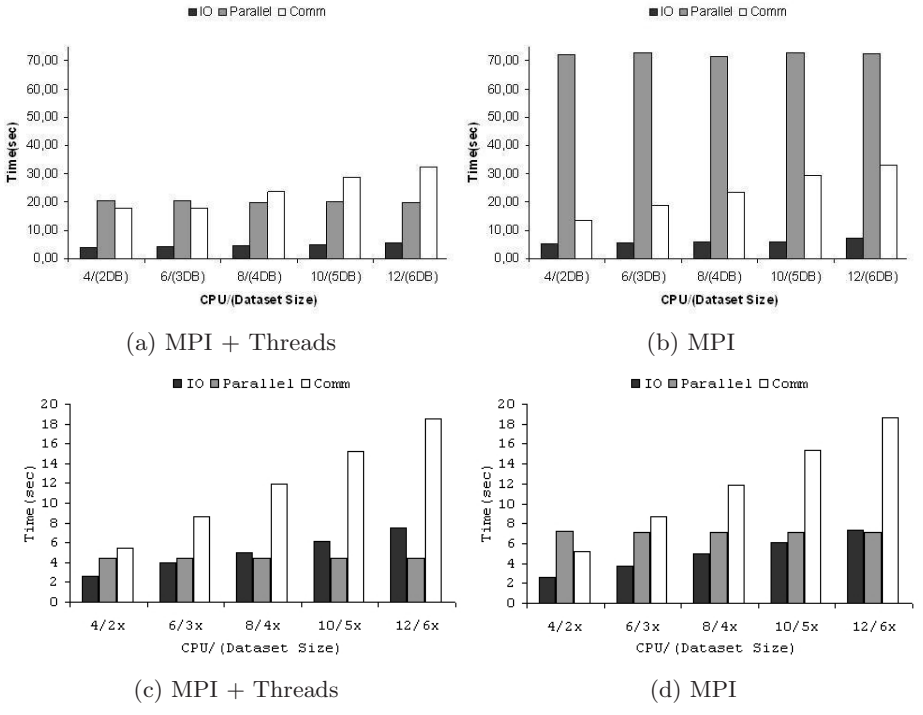
Fig. 2. SpeedUp comparison



(c) T40I10D600K (d) Kosarak

Fig. 3. Efficiency comparison

minimum support values used in each case were the smallest that the sequential version could process (for T40I10D600KxN the minimum support was set to 0.005 and for KosarakxN the minimum support was set to 0.003).



(a) MPI + Threads (b) MPI

(c) MPI + Threads (d) MPI

Fig. 4. Scalability Analysis of ParCBMine in T40I10D600KxN (a y b) and KosarakxN(c y d)datasets

Observe in Figure 4 that the parallel execution time remains constant, thus the scalability of the algorithm does not depend on the database used.

As a conclusion of these experiments we can affirm that the shared-distributed memory combination proved to be an effective way to avoid high traffic of data and drastic reduction of the efficiency of the parallel algorithm.

6 Conclusions

The algorithms proposed by Rakesh Agrawal and John Shafer in [2] are recognized as benchmarks for the development of parallel algorithms for Association Rules Mining.

Making a general assessment of these algorithms we can say that the Count Distribution reduces the communication overhead at the expense of ignoring the system physical memory. In a cluster of workstations environment, with monoprocessor nodes, this is probably the best approach; nevertheless it may not be the best solution in the case where nodes are SMP, because it would not take advantage of the combination of shared and distributed memory models. In order to reach efficient implementations based on Count Distribution, determining new

heuristics that allow a reduction of the cardinality of the local C_k obtained by each processor continues to be a latent problem.

The Data Distribution algorithm can help us to explore this feature by fully exploiting the physical memory with the risk of increasing communication overhead. The ability to count in a single pass T times as many candidates as Count Distribution makes this algorithm a strong contender.

If we include detailed background knowledge of the problem in the Candidate Distribution, the joint benefits of Count Distribution and Data Distribution [2] can be obtained. Yet, there are still some challenges for researchers in parallel algorithms for association rule mining: to find a heuristic that allows (from a step $k = l$) candidate itemsets partitioning so that synchronization among processors is not needed, and to obtain a suitable load balance among the processors.

In conclusion we suggest that the purposed parallel algorithm described here ParCBMine based on the sequential algorithm CBMine, and sustained on the principles of Count Distribution in which some features of Candidate Distribution are also introduced, suitably combines the parallel programming based on the message passing model with multithread programming. ParCBMine continues to be developed and in the future we expect to present new results oriented to the reduction of the computational effort at synchronization level and to reach a better load balance among processors.

References

1. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I.: Fast discovery of association rules. In: Fayyad, U., et al. (eds.) Advances in Knowledge Discovery and Data Mining, MIT Press, Cambridge (1996)
2. Agrawal, R., Shafer, J.: Parallel mining of association rules. In IEEE Trans. on Knowledge and Data Engg. 8(6), 962–969 (1996)
3. Park, J.S., Chen, M., Yu, P.S.: Efficient parallel data mining for association rules. In: ACM Intl. Conf. Information and Knowledge Management (November 1995)
4. Cheung, D., Ng, V., Fu, A., Fu, Y.: Efficient mining of association rules in distributed databases. In IEEE Trans. on Knowledge and Data Engg. 8(6), 911–922 (1996)
5. Cheung, D., Han, J., Ng, V., Fu, A., Fu, Y.: A fast distributed algorithm for mining association rules. In: 4th Intl. Conf. Parallel and Distributed Info. Systems (December 1996)
6. Cheung, D.W., Xiao, Y.: Effect of Data Skewness in Parallel Mining of Association Rules. In: Proceedings of the 2nd Pacific-Asia Conference on Knowledge Discovery and Data Mining, pp. 48–60, Melbourne, Australia (April 1998)
7. Ashrafi, M.Z., Taniar, D., Smith, K.A.: ODAM: An Optimized Distributed Association Rule Mining Algorithm. IEEE Distributed Systems Online (5) (2004)
8. Freitas, A.A., Lavington, S.H.: Mining Very Large Databases with Parallel Processing. Kluwer Academic Publishers, Boston (1998)
9. Freitas, A.A.: A Survey of Parallel Data Mining. In: Proc. 2nd Int. Conf. on the Practical Applications of Knowledge Discovery and Data Mining (1998)
10. Skillicorn, D.: Parallel Data Mining, Department of Computing and Information Science Queen's University, Kingston (1999)

11. Palancar, J.H., León, R.H., Pagola, J.M., Díaz, A.H.: Mining Frequent Patterns Using Compressed Vertical Binary Representations In: Lin, T.Y., Xie, Y. (eds.) Proceedings of a Workshop Foundation of Semantic Oriented Data and Web Mining, held in Conjunction with the Fifth IEEE International Conference on Data Mining, Houston, Texas, USA, pp. 29–33 (November 27-30, 2005) ISBN 0-9738918-7-4
12. Palancar, J.H., León, R.H., Pagola, J.M., Díaz, A.H.: A Compressed Vertical Binary Algorithm for Mining Frequent Patterns. In: the book Data Mining: Foundations and Practice Lin, T.Y., Wasilewska, A., Petry, F., Xie, Y.(eds.) Springer-Verlag, Accepted for publication (to appear)
13. Orlando, S., Palmerini, P., Perego, R., Silvestri, F.: A Scalable Multi-Strategy Algorithm for Counting Frequent Sets Washington, USA, pp. 19–30. In: Proceedings of the 5th Workshop on High Performance Data Mining, in conjunction with Second International SIAM Conference on Data Mining (April 2002)
14. Schuster, A., Wolff, R.: Communication-Efficient Distributed Mining of Association Rules. Data Mining and Knowledge Discovery, 8(2) (March 2004)
15. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New algorithms for fast discovery of association rules. In: Heckerman, D., Mannila, H., Pregibon, D., Uthurusamy, R. (eds.) Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97), p. 283. AAAI Press, Stanford (1997)
16. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New Parallel Algorithms for Fast Discovery of Association Rules. Data Mining and Knowledge Discovery 1(4), 343–373 (1997)
17. Zaki, M.J., Parthasarathy, S., Li, W.: A Localized Algorithm for Parallel Association Mining. In: Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures (1997)
18. Zaki, M.J.: Parallel and Distributed Association Mining: A Survey. IEEE Concurrency. (October- December 1999)
19. Zaki, M., Parthasarhaty, S., Ogihara, M., Li, W.: Parallel Data Mining for Association Rules on Shared Memory Systems. (February 28, 2001)
20. Zaiane, O.R., El-Hajj, M., Lu, P.: Fast Parallel Association Rule Mining without candidacy generation. Techical Report TR01-12. Department of Computing Sciences, University of Alberta, Canada (2001)
21. Shintani, T., Kitsuregawa, M.: Parallel Mining Algorithms for Generalized Association Rules with Classification Hierarchy. In: Proceedings ACM SIGMOD International Conference on Management of Data, SIGMOD 1998, Seattle, Washington, USA. (June 2-4, 1998)
22. Sam, E.-H., Karypis, H.G., Kumar, V.: Scalable Parallel Data Mining for Association Rules. Department of Computer Science. University of Minnesota (1997)