A Pipeline Model to Discover Frequent Itemsets in an Hierarchical Systems

Khedija Arour
Department of Computer Science and Mathematics, National Institute of Applied Science and Technology of Tunisia 1080 Tunis, Tunisia

khedija.arour@issatm.rnu.tn

Abstract

Like all the other fields of data processing, the modern information systems have integrated the results of the advanced technologies of the last decades. These systems contain implicit data which it will be necessary to extract and exploit, by using data Mining techniques. Mining association rules which trends to find interesting association or correlation relationships among large amounts of data is one of these techniques. It is a two-steps process, the first step finds all frequent itemsets and the second step constructs association rules from these frequent sets. The overall performance of mining association rules is determined by the first step which becomes the focus problem. This step is expensive with high demands for computation and data access. Parallel computing seems to have a natural role to play since parallel computers provides scalability. In this paper, we examine the issue of mining association rules among items in large databases transactions using the algorithm Apriori proposed by Agrawal. In this context, we propose a new parallel version of the Apriori algorithm of Agrawal, that is the main algorithm of each data mining technique. Parallel computing seems to have a natural role to play since parallel computers provides scalability. In fact, our objective of our work is to have an efficient parallel execution time that requires a delicate balance between program granularity and communication latency (synchronization overhead) between the different granules. Unlike previous work on parallelization of specific data mining algorithms, our approaches consist to discover the different granularity levels of parallelism and their impact on the performance. In this paper we focus on task and data parallelism (hybrid approach) under distributed memory. In particular, if communication latency is minimal then fine grain partitioning will yield the best performance. This is the case when data parallelism is used. If communication latency is large (as in a loosely coupled system), then coarse grain partitioning is more appropriate. For the target architecture used in this work (distributed-shared memory).), the problem of load balancing among the nodes becomes a more critical issue in attempts to yield high performance. We have carried out a detailed evaluation of the parallelization techniques and the impact of combining different types of parallelism (task, data and pipeline) on the effectiveness of the system.

Keywords: *Datamining, Task parallelism, Data parallelism, Pipeline model*

1. Introduction

With massive amounts of data continuously being collected and stored, many industries are becoming interested in the extraction of useful information from large databases. Association mining is one of the most important data mining tasks. Association rule mining (ARM) finds interesting correlation relationships among a large set of data items. A typical example of association rules mining is market basket analysis. This process analyses customer buying

habits by finding associations between the different items that customers place in their shopping baskets. Such information may be used to plan marketing or advertising strategies, as well as catalogue design. Each basket represents a different transaction in the transactional database, associated to this transaction the items bought by a specific customer. An example of an association rule over such a database could be that □80% of the customers that bought bread and milk, also bought eggs□. Given a transactional database D, an association rule has the form X ® Y, Where X and Y are two itemsets, and X) Y = □. The rule□s support is the joint probability of a transaction containing both X and Y at the same time, and is given as ∫(A ∗ B). The confidence of the rule is the conditional probability that a transaction contains Y given that it contains X and is given as ∫(A ∗ B)/ ∫(A). A rule is frequent if its support is greater than or equal to a pre-determined minimum support and strong if its confidence is more than or equal to a user specified minimum confidence. Association rule mining is a two-step process:

1. the first step consists of finding all frequent itemsets that occur at least as frequently as the fixed minimum support. The search space for enumerating all frequent itemsets is extremely large. For example, with m attributes there are, in the worst case, O(mk) potential sequences of length at most k;
2. the second step consists of generating strong implication rules (which satisfy the minimum confidence constraints) from these frequent itemsets.

The second step is the easiest of the two. The overall performance of mining association rules is determined by the first step. Because of that, we concentrate our attention in this paper on the frequent sets counting problem (FSC), which is the most time consuming phase of the association mining process.

Nevertheless, given the search complexity, sequential algorithms cannot provide scalability, in terms of the data size and the performance for large databases, we must rely on parallel multiprocessor systems to fill this role.

Several parallel algorithms have been developed for association mining for both distributed memory, shared memory and recently hierarchical systems. In this paper, we examine the issue of mining association rules among items in large databases transactions using the algorithm Apriori [2]. We aimed to study the impact of hybrid parallelism (i.e. the combination of data parallelism and task parallelism) on that algorithm. For that reason we propose three approaches for parallelizing Apriori. The first approach uses task parallelism, the second approach uses hybrid parallelism without pipeline model and the third one uses hybrid parallelism with pipeline model.

We have carried out a detailed evaluation of the parallelization techniques and the impact of combining different types of parallelism (task, data and pipeline). Our test bed is an IBM SP2 distributed memory machine with 32 processors under different types of data sets.

The paper is organized as follows. Section 2 briefly discusses sequential data mining algorithms. Sections 3 introduces parallel data mining algorithms. Section 4 presents the algorithm APRIORI. Section 5 proposes three approaches for parallelizing APRIORI. Section 6 presents the experimental results obtained from implementing these approaches. And finally, Section 7 draws some conclusions and future works.

2. Parallel and Distributed Association Rule Mining Algorithms

Most of the existing algorithms, use local heuristics to handle the computational complexity. The computational complexity of These algorithms are fast enough for application domains where N is relatively small. However, in the data mining domain where

millions of records and a large number of attributes are involved, the execution time of these algorithms can become prohibitive, particularly in interactive applications. Parallel algorithms have been suggested by many groups developing data mining algorithms. We discuss below two approaches that have been used.
Many parallel algorithms for solving the FSC problem have been proposed to provide scalability for association rule mining algorithms. Most of them use Apriori algorithm as fundamental algorithm, because of its success on the sequential setting [1]. The reader could refer to the survey of Zaki on ARM algorithms and relative parallelization schemas [11][10]. Agrawal et al. proposed a broad taxonomy of the parallelization strategies that can be adopted for Apriori [1].Two different parallelization of Apriori on distributed memory machine where presented in Agrawal et Al.

- distribution CD In CD, the database is paritioned and distributed across n processors.
- Data distribution DD In DD The candidates are partitioned over all the processor in a round-robin fashion. The communication overhead of broadcasting the database partitions can be reduced by asynchronous communication.

Experiments shows that algorithms based on count distribution outperforms the other algorithms. Data distribution is the worst approach, while candidate distribution obtained good performances but paid a high overhead due to the need of redistributing the dataset. In shared memory many algorithms are proposed by Zaki et Al [12].

- Common Candidate Partition Data CCPD algorithm, it is essentially the same as count distribution, but uses some optimization techniques to balance the candidate hash tree and to short-circuit the candidate search for fast support counting.
- Eclat algorithm was designed to overcome the shortcomings of the CD and CND algorithms. It utilizes the aggregate memory of the system by partitioning the candidates into disjoint sets using the equivalence class partitioning. Eclat uses the vertical database layout which clusters all relevant information in an itemset's tid-list. Each processor computes all the frequent itemsets from one equivalence class before proceeding to the next.

All these algorithms include data parallelism paradigm. These parallel algorithms differ in two aspects. The first one is whether further candidate pruning or efficient candidate counting techniques are employed or not. The second one is whether the database transactions is replicated or partitioned among the processors. The interesting problem to parallelize the Apriori algorithm, mentioned in the previous section, encouraged us to study it in depth in order to propose new approaches for parallelizing it. In fact, as we described above many parallel versions of Apriori are implemented but the same strategy is done in the parallel process: Data parallelism approach. In our work, we are interested to study the task parallelism and pipeline model in order to maximize the degree of parallelism. Pipeline computation is a common computational strategy in parallel processing, where a task is decomposed into several stages that are solved sequentially. Unlike previous work on parallelization of specific data mining algorithms, our approaches consist to discover the different granularity levels of parallelism and their impact on the performance. In this paper we focus on task and data parallelism with the integration of pipeline model.

3. Parallel Approach to Discover Frequent Itemsets

It is important to mention that in practice, parallelism appears in various forms in computing environment. All forms can be attributed to levels of parallelism, computational granularity, time and space complexities, communication latencies and load balancing.

Generally, parallelizing is not trivial, it is a costly process in both time and efforts. Our goal is to combine task and data parallelism to speed-up the generation of frequent itemsets. We wanted to focused our attention on task parallelism, i.e. parallelizing at the program instructions level. In the following, we are going to propose two approaches for parallelizing Apriori based on task. The first approach is based on task without use of pipeline model, and the second approach uses the hybrid parallelism (*i.e.*, the combination of data and task parallelism) with pipeline. At the end, the approaches will be compared.

3.1. Apriori algorithm

The main observation in the Apriori technique is that if an itemset occurs with frequency f, all the subsets of this itemset also occur with at least frequency f. The algorithm employs an iterative approach known as levelwise search. During the kth iteration, the algorithm scans the input dataset, and computes the frequent k-itemsets denoted by Lk. The sequential code was developed by Borglet [4] and uses a prefix tree to represent candidate itemset and hash tree to represent database transactions.

3.2. First approach: Task parallelism

Our approach is based on Christian Borgelt's implementation of Apriori. In this approach we aimed to study parallelism inside the program code. This could be done through searching inside the algorithm procedures for independent segments which necessitate analyzing loops to detect tasks (or instructions) that could be executed simultaneously. We try to exploit all degree of parallelism in Apriori algorithm. We start our work to describe algorithm dependencies and data flow graphs, to be able to make parallel sequential algorithms. Hence, we can execute certain computationally intensive parts of Apriori programs in parallel.

In the case of Apriori, we have many important loops (from the point of view number of operations to be executed) and tasks that they can be done independently. We can broken these loops and instructions into separate tasks and study dependencies between them based on Bernstein conditions [3].

3.2.1. Task Description

Based on the Borglet's implementation, the Apriori algorithm can be broken into principally six tasks. Table 1 illustrates the pseudo code of the principal tasks.

- T1 : lecture of database transactions and extraction of 1-itemsets candidates.
- T2 : clean data, this task has essentially two steps:
    - T2-1 : pruning and generation of frequent 1-itemsets.
    - T2-2 : pruning non frequent itemsets from database transactions.
- T3 : construction of transactions tree.
- T4 : construction of candidates tree.
- T5 : candidates generation based upon frequent (k-1)-itemsets.
- T6: pruning and generation of frequent k-itemsets.It has essentially two steps:
    - T6-1 : support generation of candidates k-itemsets.
    - T6-2 : pruning and generation of frequent k-itemsets.
- Test : pass to (k + 1) iteration.

Table 1. Task Decomposition of Apriori Algorithm

Task	Task Description

T1	3: in=fopen(dataBase,"r");
	4: for (tacnt=0;1 ;tacnt++) do
	5: k=s-read(itemset.in) ; // Reading itemsets.
	6: If(k > 0) break;
	7: k=itemset → cnt ;
	8: tas-add(taset,null,0); // Reading transactions.
	9: end for

T2	11: p=(int*)malloc(itemset → cnt * sizeof(int));
	12: k=(int)ceil(tacnt*supp);
	13: n=s-recode(itemset,k,1,p) ; // Creation of frequent 1-itemsets.
	14: tas-recode(taset);

T3	15: tatree=tat-create(taset);
T4	17: apps=(char*)malloc(n*sizeof(char)); 18: for (apps+=i=n ;-i ≥ 0 ;) do 19: *-apps=itemset → nimap → ids[i] → app; 20: istree=ist-create(n,supp,apps); 21: for (k=n ;-k ≥ 0 ;) do 22: ist-setcnt(istree,k, itemset → nimap → ids[i] → frq); 23: end for 24: end for

T5 T6	26: for (;istree → lvlcnt < maxcnt ;) do
	27: istree=ist-addlvl(istree); // Generation of candidate k-itemsets.
	28: ist-countx(istree,tatree); // Calcul of frequency and pruning.
	29: end for

3.3. Dependency graph

We present in Figure 1, the dependency graph of the different tasks. The adjacency matrix (see Table 2) and the dependence graph deducted will be as follows:

To construct dependence graph, we have to extract the following dependencies: (i) Between T1 and T2 we have a flow dependence (RAW: Read After Write) since O1 ∩ I2 □= □, where O1 stands for the Output of T1 and I2 stands for the Input of T2. Let this dependence be noted as δ. So T1 □□≪ T2 (T1 Preceede T2). (ii) Between T1 and T3, we have also a flow dependence δ, since O1 ∩ I3 □= □. Then T1 ≪ □ T3. (iii) Between T2 and T3,there is a δ dependence, since O2 ∩ I3 □= □. Then T2 □≪ T3.

Hence, we suggest the following dependencies:

- T1δT2-1,because S1 ∩ E2−1 □=□
- T1δT2-2,becauseS1 ∩ E2−1 □=□
- the two tasks T2-1 and T2-2 can be execute in parallel.
- also for T2-1 and T3.
- T3 and T4 may be execute in parallel.

- T2-1δT4δT5δT6-1δT6-2.
- T22δT3δT6-1.
- Although there exist dependencies between the the tasks T61 et T62, we could adopt the pipelining model to parallelize them [5].

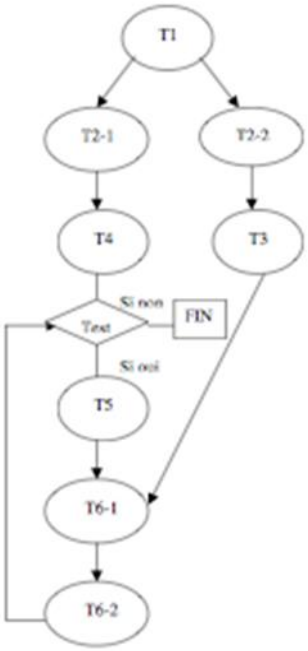

Figure 1. Dependency Graph of the Apriori algorithm

Table 2. Adjacency Matrix

	T1	T21	T22	T3	T4	T5	T61	T62
T1	0	1	1	0	0	0	0	0
T21	1	0	0	0	1	0	0	0
T22	1	0	0	1	0	0	0	0
T3	0	0	1	0	0	0	1	0
T4	0	1	0	0	0	1	0	0
T5	0	0	0	0	1	0	1	0
T61	0	0	0	1	0	1	0	1
T62	0	0	0	0	0	0	1	0

3. Data Parallelism

In this first approach, we focus on the data parallelism. We detailed below all the possible parallelism that be done in the inner tasks(intra-tasks parallelism).

4.1. Parallelization of the task T1

The database transactions is partitioned during the initial iteration among processors participating in the execution, and each processor calculates partial supports of its items from

its local database partition. Each processor calculates partial supports of its 1-itemsets candidates from its local database partition. At the end, the task does a sum reduction to obtain the global counts by exchanging local counts with all other processors. Once the global Fk has been determined, each processor builds the entire candidate Ck+1 in parallel, and the previous process is repeated until all frequent itemsets are found. The goal of this approach is minimizing communication, because only the counts are exchanged between processors. To minimize the communications phase, each processor Pi can maintain an n support array where n is the number of items and each case contains the corresponding support. This structure doesn't suffer from any of the overheads because the support array is sorted in increasing order, then the global count support of a kth candidate by simply adding the corresponding columns in each processor. This is

vectors of bits accessed with high locality, and without using expensive comparisons and conditional branch instructions. A column summation can be done to calculate the global support using "MPI AllReduce" command [7]. The Figure 2, presents a synchronization step between all processors. This synchronization can be done by "MPI BARRIER".

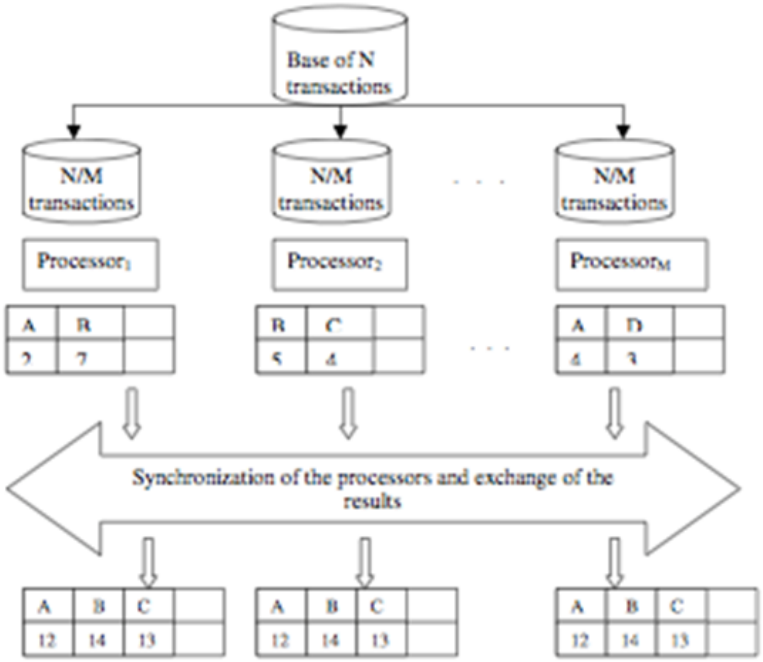

Figure 2. *1-itemsets* candidates generation

4.2. Parallelization of the task T2-1

This step consists to partition the 1-itemsets candidates, so that each processor can prune the non frequent 1-itemsets. After that, each processor can locally sort his support array according to the frequency. A step of global sort is necessary. To do that, we adopted a master-slave approach (see Figure 3), where we have one master and the rest of available processors are slaves. At the end of this step, the master processor send the global support array to the others processors.

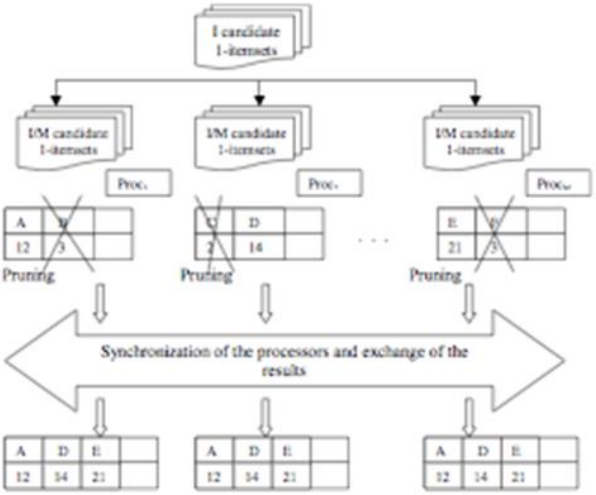

Figure 3. *1-itemsets* frequents generation: (tche T2-1)

4.3. Parallelization of task T2-2

In this task, we reduce the size of transactions database by pruning out the non frequent 1-itemsets. Hence, to do this step rapidly, we partition this database equitably between all processors and each one scans its local portion to skipped the non frequent (see Figure 4).

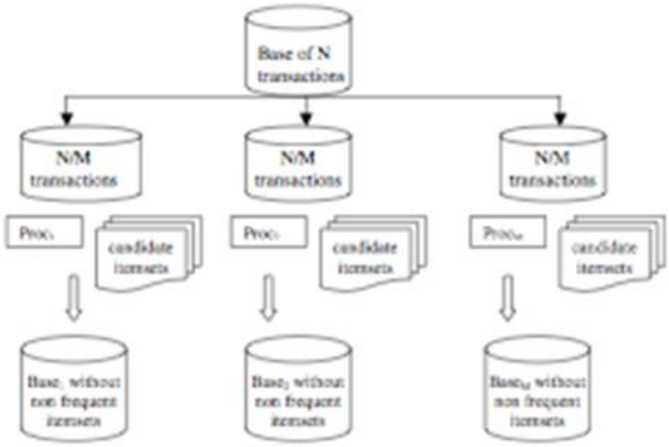

Figure 4. Pruning non frequents itemsets from the transactions database

4.4. Parallelization of task T4

The remaining frequent 1-itemsets is split between all the processors. Each processor builds a branch of the hash tree based on its local database (*i.e.*, local tree). This steps consists of constructing the whole of hash tree based on the individual local trees. Any slave processor Pi sends his local tree to a master one (for example P0). At the end, a global diffusion of the whole hash tree is done. The Figure 5 illustrates the different steps that are described.

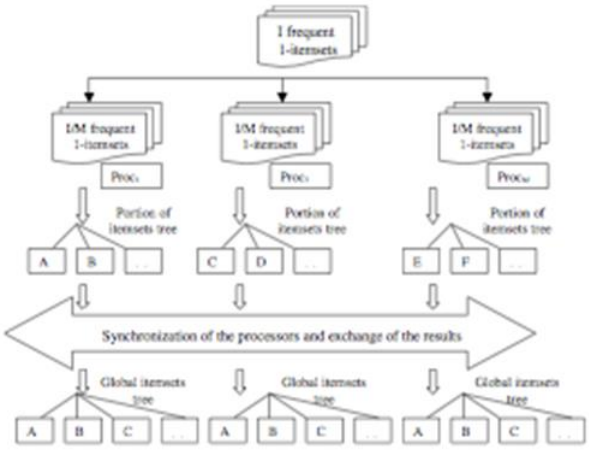

Figure 5. Candidate hash tree construction

4.5. Parallelization of task T5

The objectif of this step is to construct the candidates k-itemsets based on frequent (k-1)-itemsets. Each processor construct his local candidate k-itemsets. Based on the fact that (k-1)-itemsets is sorted, candidate k-itemsets can simply be generated from itemsets by joining an itemset with all successors one (see Figure 6).

To achieve an equal distribution of the candidate itemsets, we use a partitioning algorithm that is based on bin-packing [6] such that the sum of numbers of candidate itemsets are roughly equal. This gives about the same size hash tree in all the processors and thus provides a correct load balancing among processors.

For example, suppose that our tree is composed by four itemsets {A, B, C, D} and we have two processors {P1,P2}, the result of this distribution is :

- P1 is charged to do generate 2-itemsets by simply joining all pairs of items A et D with her successors, hence, we obtain {{AB },{AC },{AD }}.
- P2 generate all combinations based on the items B et C and we obtain {{BC},{BD},{CD}}.

Note that the equal assignment of candidates to the processors does not guarantee the perfect load balance among processors. This is because the cost of traversal of the hash tree are determined not only by the size of the tree, but also by the presence of items in the transactions.

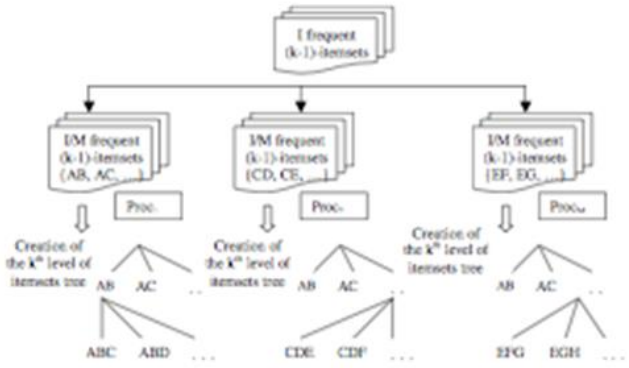

Figure 6. *k-itemsets* candidates generation (task T5)

4.6. Parallelization of task T6-1 and T6-2

The scop of this task is to calculate the support of the candidate k-itemsets. This task can be done in parallel (see Figure 7). Each processor can have a subset of candidate k-itemsets and count here support in the corresponding local database tree. A synchronous step is needed to calculate the global count of supports. This is done by a master processor. After that, Each processor prune the non frequent candidate k-itemsets from his local candidate hash tree. At the end, a step of synchronization is necessary to generate a global frequency.

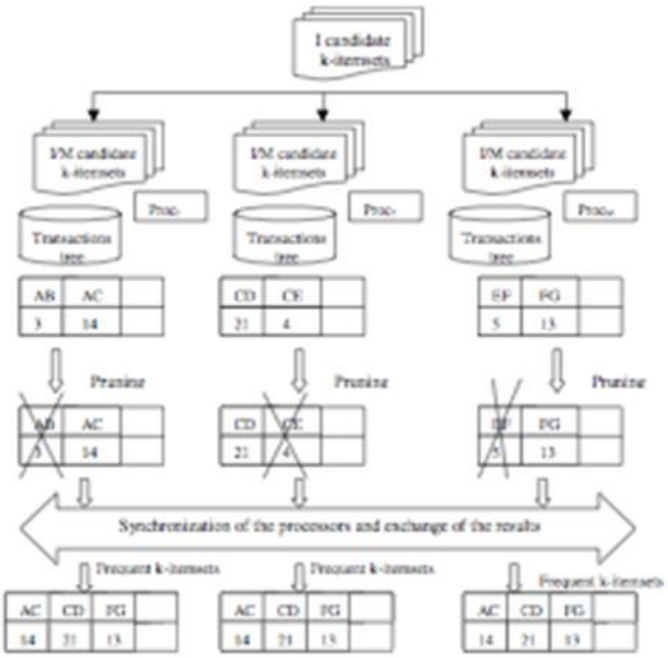

Figure 7. Support count of candidate and generation of frequent *onek-itemsets* (task T6-1)

5. Hybrid Approach

Experiments on sequential algorithm allowed us to estimate that, the last three tasks T5, T6-1 and T6-2 take approximatively 80% of all computational time. Although there exist dependencies between the three tasks, we could use the aspect of pipelining to parallelize them.

5.1. Hybrid Approach using pipeline model

In this subsection we describe the parallel version we have implemented and that exploit all the possibilities of parallelism. Although, there exist dependencies between T5, T6-1 and T6-2 (see Figure 1), we can parallelize them using the pipeline model [5]. In fact, when a new candidate k-itemset from task T5, is generated, we can anticipate and count his frequency (by task T6-1). A step of pruning this k-itemset can be done after that by task T6-2. To reiter and pass to the last iteration (k+1), all processors need to generate all the associated frequent k-itemsets. A step of global regroupment is necessary. To do that, we adopted a master-slave

approach, where, we have one master and the rest of available processors are slaves. At the end of this step, the master processor splits this set and sends these partitions to the others processors. The Figure 8 shows the parallelization of these three tasks.

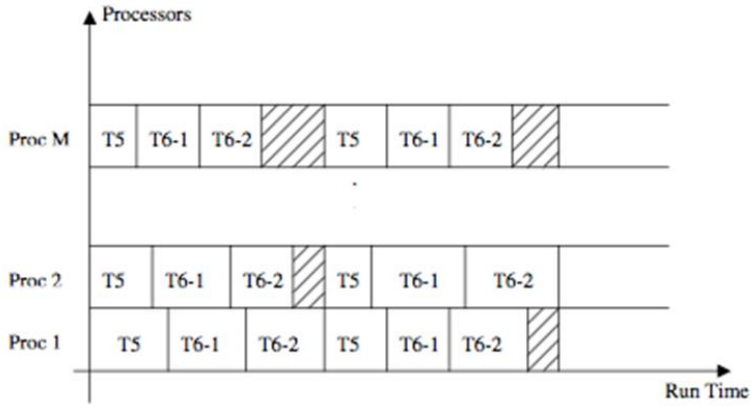

Figure 8. Parallelizing of tasks T5, T6-1 and T6-2

5.2. Hybrid Approach without using pipeline model

In this approach, we don't use the pipeline model between T5, T61 and T62. In the inner of theses tasks, we exploite all the degree of parallelism that we mentioned in the other sections. A step of synchronization is needed at the end of each task. For the task T5, each processor has its local frequent (k-1)-itemsets (including the master processor) and local frequencies of candidates are calculated in parallel. Only one phase of communication and synchronization is needed by the end of each iteration to obtain global frequencies that is done by a master processor. The candidates tree are constructed and split between all processors to achieve task T6-1. A global sum reduction is done to calculate global frequencies and generate the global frequents set.

6. Experimental Evaluation

In this section, we evaluate our implementation of different parallel approaches of Apriori algorithm.

6.1. Experimental Platform

The sequential and parallel versions of Apriori algorithm are implemented on an IBM SP2 distributed memory machine of 8 clusters. Each cluster contains four 375 MHz shared memory processors. We used different synthetic and dense databases. The synthetic databases are size ranging from 35MB to 275MB, which are generated using the procedure described in [8]. The following table illustrates the datasets used in the experiments:

Table 3. Adjacency Matrix

Database name	Transactions number	Items number	Average size of transaction
T20I4D50K	50000	884	20
T20I4D100K	100000	1000	20
T40I8D100K	100000	1000	40
Chess	3196	75	37
Connect	67557	129	43
Mushroom	8124	119	23

The transactions database ”T20I4D50K”, ”T20I4D100K” and ”T40I8D100K” are synthetic from [8] but the others one are dense [9].

Table 4. Adjacency Matrix

Notation	Signification
—T—	average size of transaction.
—I—	potential average size of frequent itemsets.
—D—	Number of transactions.

6.2. Comparative study between the different proposed approaches

In this subsection, we report a comparative study between different proposed approaches for different databases, different threshold support, number of clusters and processors. The different approaches are:

- Approach 1: Hybrid approach with pipeline,
- Approach 2: Hybrid approach without pipeline,
- Approach 3: data parallelism.

In this subsection, we analyse and compare the different approaches in terms of runtime, speed up and efficiency.

6.2.1. Parallel run time

Based on Figures 9 and 10, we can conclude the following remarks:

- All plots have the same allure for both databases.
- For a little number of clusters, the second approach outperforms the others one.

The study of these plots, we can conclude the following remarks:

- In both approaches, the parallel execution time is less than the sequential one.
- In all tests conducted, this parallelization approach outperforms the others one.
- The optimal execution time for both approaches is obtained by using 612 processors in the case of voluminous databases behind that number of processors there will be a communication overhead.

Based on Figures 9 and 10, we can conclude the following remarks:

- All plots have the same allure for both databases.
- For a little number of clusters, the second approach outperforms the first one.

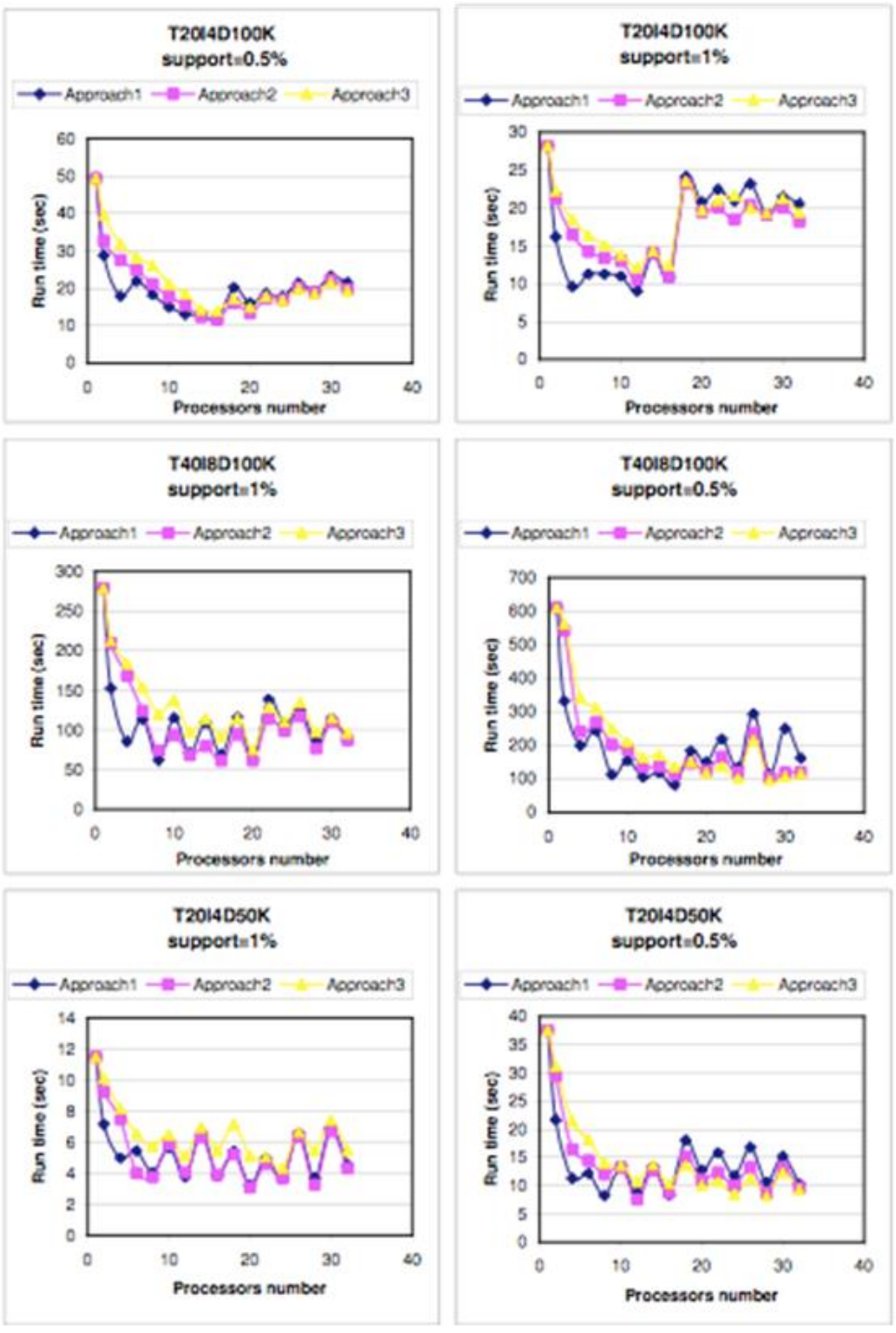

Figure 9. Comparative study of parallel runtime for synthetic databases

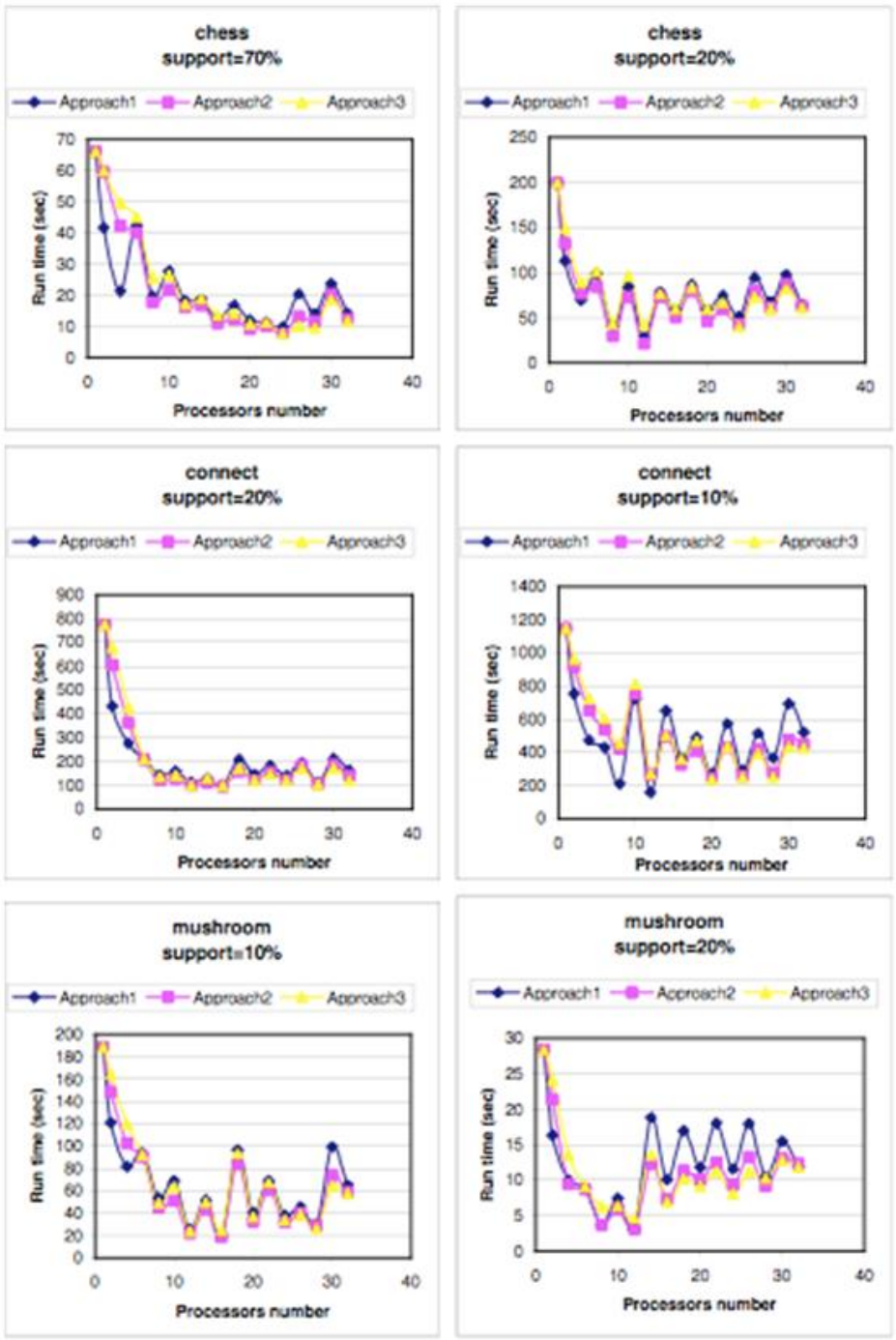

Figure 10. Comparative study of parallel runtime for dense databases

6.2.2. Speedup study

The plots from figures 11 and 12 show the speedup results for the different proposed approaches and for different databases.The speedup numbers are with respect to a sequential run of the algorithm on the given database.

Based on these plots, we can conclude the following remarks :

- The speedup is more interested in the case of dense databases.

- Compared both these approaches, first approach gives better speedup and scalability with increase in number of processors.

- Increasing the processors number, the data parallelism approach gives a better speedup.

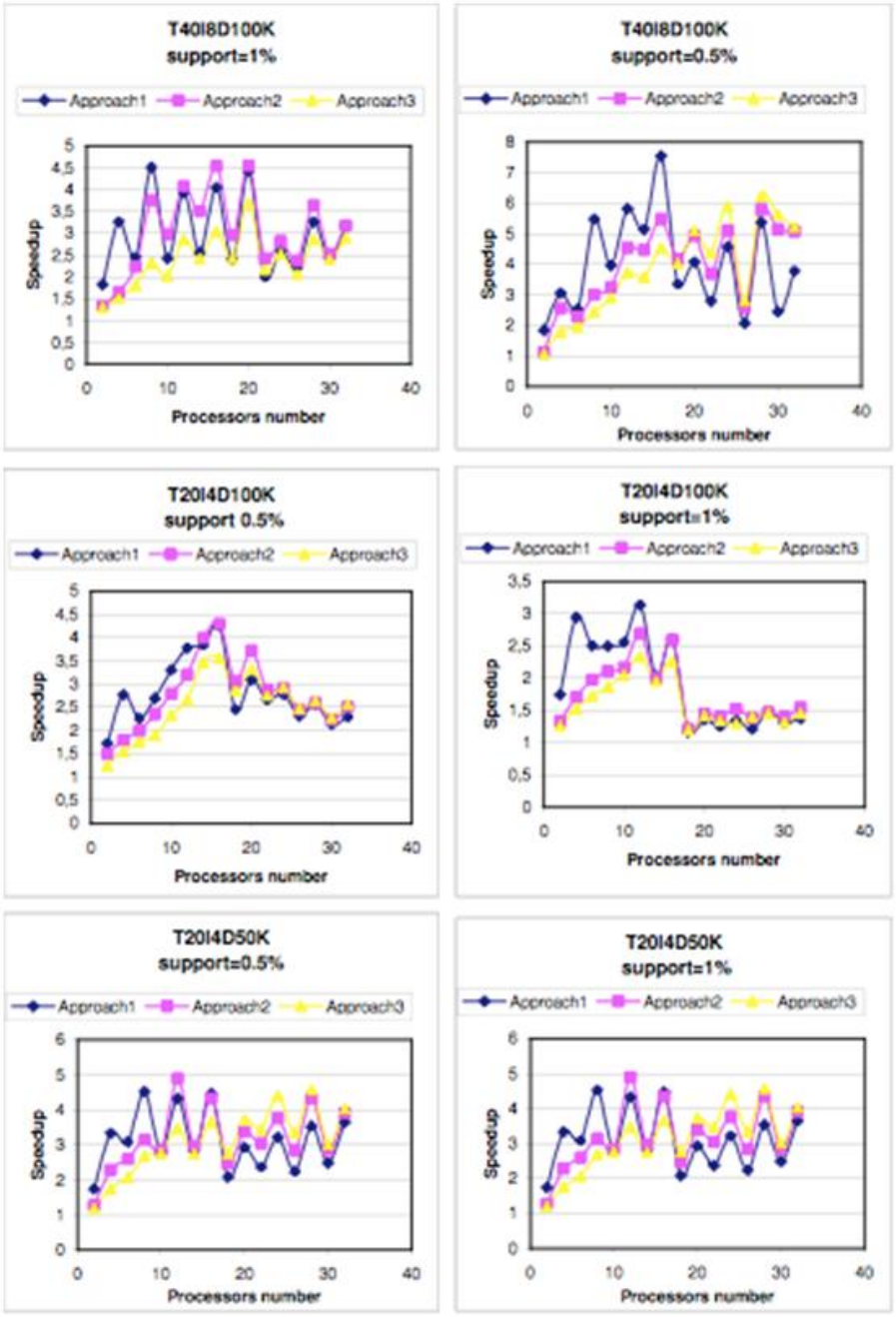

Figure 11. Speedup performance on synthetic databases

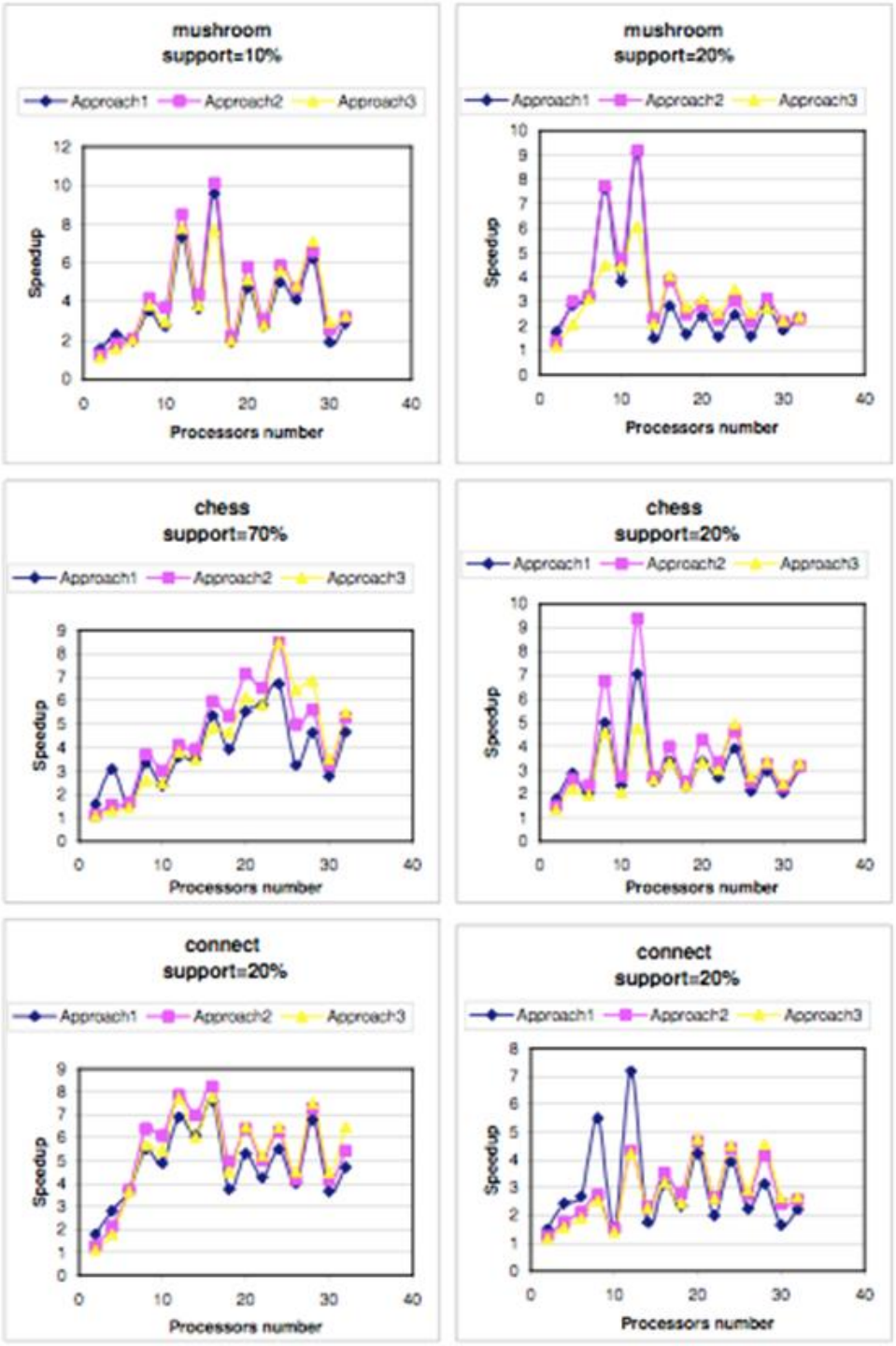

Figure 12. Speedup performance on dense databases

6.2.3. Efficiency study

The following Figures 13 and 14 give us the evolution of efficiency of the different approaches.

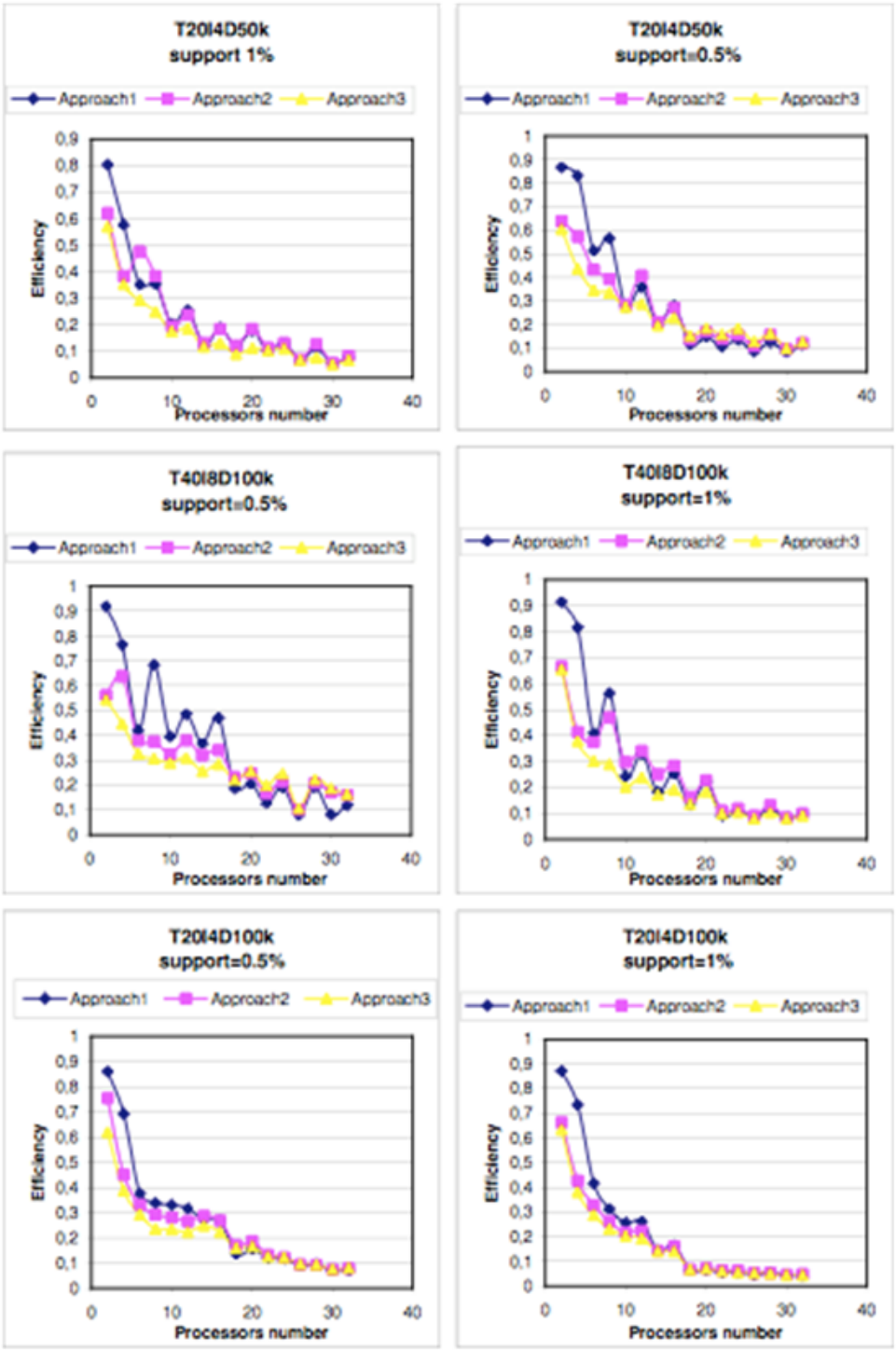

Figure 13. Efficiency Study for Synthetic databases

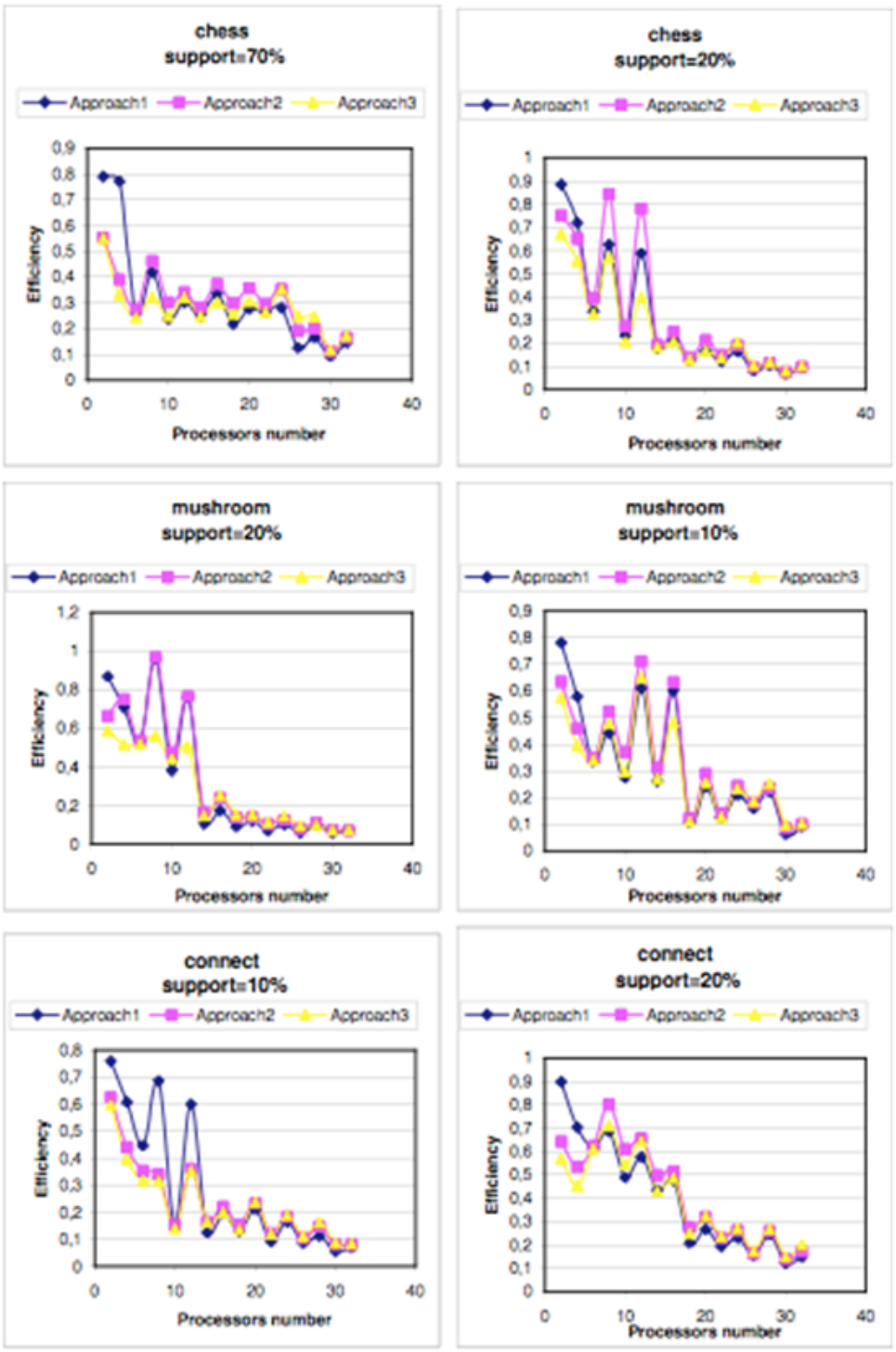

Figure 14. Efficiency Study for Dense databases

Based upon these plots, we can conclude the following remarks:

- The efficiency is mor20e interested in the case of dense databases.

- Task parallelism is better for dense databases with important number of clusters. For synthetic databases, the task parallelism is important inside a cluster.

7. Conclusion and Future Works

We considered the problem of mining association rules on a distributed memory machine on which data has been partitioned between processors. We presented three parallel approaches for this problem based upon Apriori algorithm. The first approach uses data parallelism and the second approach is a hybrid approach which uses both data and task parallelism using pipeline model. The third one an hybrid approach without the use of pipeline model. By analysing the obtained experimental results we find that the second approach outperforms the others. In the future, we plain to focus our attention to the problem of data size and memory management by using out of core algorithm. The performance of many algorithms does not scale well in the transition from the in-core to the out-of-core processing conditions. The hybrid approach is more scalable that the task parallelism paradigm and has lessened the insufficient main memory problem. This performance degradation is due to the high frequency of I/O operations that start dominating the overall running time. Also, the design of solutions adapted to modern multiprocessor systems like multicore and GPU sytems is under study.

References

[1] R. Agrawal and J. Shafer, "Parallel mining of association rules", IEEE Transaction On Knowledge and Data Engineering, vol. 8, **(1996)**, pp. 962–969.
[2] R. Agrawal and R. Skirant, "Fast algorithms for mining association rules in large databases", Proceedings of the 20th Int´l Conference of Very Large Databases (VLDB'94), **(1994)** June, pp. 478–499.
[3] A. J. Bernstein, "Program analysis for parallel processing", Proceedings of IEEE Trans. on Electronic Computers, **(1966)** October, pp. 757–762.
[4] C. Borgelt, "Efficient Implementations of Apriori and Eclat", Workshop of Frequent Item Set Mining Implementations, **(2003)**.
[5] T. Fosdick, *et al.*, "An Introduction to High Performance Scientific Computing", MIT Press, **(1996)**.
[6] E. Han, G. Karypis and V. Kumar, "Scalable parallel data mining for association rules", Proceedings ACM Conference of Management of Data, ACM Press, New York, **(1997)**, pp. 277–288.
[7] T. M. S., http://www.mhpcc.edu/training/workshop/mpi/main.html.
[8] G. of DataBases: Site http://www.almaden.ibm.com/cs/quest.
[9] G. of dense DataBases: Site http://fimi.cs.helsinki.fi/data.
[10] Y. Slimani, K. Arour and M. Jemni, "Informatique répartie: Chapitre Découverte parallèle de règles associatives", Hermes, Lavoisier, **(2005)** March.
[11] M. Zaki, "Parallel and distributed association mining: a survey", IEEE Concurrency, vol. 7, no. 4, **(1999)**, pp. 14–25.
[12] M. Zaki, M. Ogihara, S. Parthasarathy and W. Li, "Parallel data mining for association rules", IEEE Transactions Knowledge and Data Engineering, **(1996)** August, pp. 962–969.