A Highly Parallel Algorithm for Frequent Itemset Mining

Alejandro Mesa[1,2], Claudia Feregrino-Uribe[2], René Cumplido[2], and José Hernández-Palancar[1]

[1] Advanced Technologies Application Center, CENATAV. La Habana, Cuba.
{amesa, jpalancar}@cenatav.co.cu
[2] National Institute for Astrophysics, Optics and Electronics, INAOE. Puebla,
México. {amesa, cferegrino, rcumplido}@inaoep.mx

Abstract. Mining frequent itemsets in large databases is a widely used technique in Data Mining. Several sequential and parallel algorithms have been developed, although, when dealing with high data volumes, the execution of those algorithms takes more time and resources than expected. Because of this, finding alternatives to speed up the execution time of those algorithms is an active topic of research. Previous attempts of acceleration using custom architectures have been limited because of the nature of the algorithms that have been conceived sequentially and do not exploit the intrinsic parallelism that the hardware provides. The innovation in this paper is a highly parallel algorithm that utilizes a vertical bit vector (VBV) data layout and its feasibility for making support counting. Our results show that for dense databases a custom architecture for this algorithm can perform faster than the fastest architecture reported in previous works by one order of magnitude.

1   Introduction

Nowadays, many data mining techniques have emerged to extract useful knowledge from large amounts of data. Finding correlations between items, specifically frequent itemsets, is a widely used technic in data mining. The algorithms that have been developed in this area require powerful computational resources and a lot of time to solve the combinatorial explosion of itemsets that can be found in a dataset. The high computational resources required to process large databases can render the implementation of this kind of algorithms impractical. This is mainly due to the presence of thousands of different items or the use of a very low threshold of support (minsup[3]).

Attempts to accelerate the execution of algorithms for mining frequent itemsets have been reported. The most common practice in this area is the use of parallel algorithms such as CD [1], DD [1], CDD [1], IDD [5], HD [5], Eclat [12] and ParCBMine [7]. However, all these efforts have not reported good execution

---

[3] Is the minimum number of times in a database an itemset must occur to be considered as frequent.

times given a reasonable amount of resources for some practical applications. Recently, hardware architectures have been used in order to speed up the execution time of those algorithms. These architectures were presented in [2, 3, 9–11], and improved the execution time of software implementations by some orders of magnitude.

Previously proposed parallel algorithms used a coarse granularity parallelism. Generally a partition of the database is made in order to process in a parallel fashion each block of data. In this paper an algorithm that exploits the inherent task parallelism of hardware implementation and the feasibility to perform bit-wise operations is proposed. A hardware architecture for mining frequent item-sets is developed to test the efficiency of the proposed algorithm.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 discusses the proposed algorithm. Section 4 describes the systolic tree architecture that supports the algorithm. The results are discussed in Section 5 and Section 6 presents the conclusions.

2 Related work

Algorithms that use data parallelism deal with load balancing, costs of communication and synchronization. These are problems not commonly present in algorithms that exploit task parallelism. This is because in this type of algorithms the efficiency is based on the high speed that can be achieved by a simple task executed in a multiprocess environment. Therefore, normally the data are accessed sequentially. Currently, hardware architectures for three frequent item-sets mining algorithms (Apriori, DHP and FP-Growth) have been proposed in the literature [2, 3, 10, 11], where sequential algorithms are used.

A systolic array architecture for the Apriori algorithm was proposed in [2] and [3]. A systolic array is an array of processing units that processes data in a pipelined fashion. This algorithm generates potentially frequent itemset (candidates itemsets) following a heuristic approach. Then, it prunes the candidates known to be infrequent, and finally it counts the support of the remaining item-sets. This is done for each size of itemset until there is no frequent itemset left. In the proposed architectures, each item of the candidate set and the database are injected to the systolic array. In every unit of the array, the subset operation and the candidate generation is performed in a pipelined fashion. The entire database has to be injected several times through the array. In [2], the hard-ware approach provides a minimum of a $4\times$ time performance advantage over the fastest software implementation running on a single computer. In [3], an enhancement to the architecture is explored, introducing a bitmaped CAM for parallel counting the support of several itemsets at once, achieving a $24\times$ time performance speedup.

In [11] the authors use as a starting point the architecture proposed in [3] to implement the DHP algorithm [8]. This introduces two blocks at the end of the systolic array to collect useful information about transactions and candidate

itemsets to perform a trimming operation over them and to reduce the amount of data that the architecture has to process.

In [10] a systolic tree architecture has been proposed to implement the FP-Growth algorithm [6]. This algorithm uses a compact representation of the data in an FP-Tree data structure. To construct the FP-Tree, two passes through the database are required and the remaining processing is made through this data structure. This tree architecture consists of an array of processing elements, arranged in a multidimensional tree to implement the FP-Tree. With this architecture a reduction of data workload is achieved.

All presented works in this section use a horizontal items-list data layout. This representation requires a number of bits per item to identify them (usually 32). Also, all architectures implement algorithms that have been conceived for software environments and do not take full advantage of the hardware intrinsic parallelism.

3   The proposed algorithm

Conceptually, a dataset can be defined as a two-dimensional matrix, where the columns represent the elements of the dataset and the rows represent the transactions. A transaction is a set of one or more items obtained from the domain in which the data is collected. Considering a lexicographical order over the items they can be grouped by equivalence classes. The equivalence class $(E(X))$ of an itemset X is given by:

$$E(X) = \{Y | Prefix_{k-1}(Y) = X \wedge |Y| = k\} \tag{1}$$

The proposed algorithm is based on a search over the solution space through the equivalence class, considering a lexicographical order over the items. This is a two-dimensional search, both breadth and depth is performed concurrently. Using the search through the equivalence class allows us to exploit a characteristic of VBV data layout. With this type of representation, the support of an itemset can be defined as the number of active bits of the vector that represents it $(\overline{X})$. This vector $\overline{X}$ represents the co-occurrence of items in the database and can be obtained as a consecutive bitwise *and* operation between all the vectors that represent each item of the itemset(see equation 2).

$$X = \{a, b, c, \ldots, n\}$$
$$\overline{X}_n = \overline{a} \text{ and } \overline{b} \text{ and } \overline{c} \text{ and } \ldots \text{ and } \overline{n} \tag{2}$$

The process to obtain $\overline{X}$ can be done by steps. First, an *and* operation can be performed between the first two vectors ($\overline{a}$ and $\overline{b}$) and then the results are accumulated. This accumulated value can be used to perform another *and* operation with the third vector to obtain $\overline{X}$ for $X = \{a, b, c\}$. To obtain $\overline{X}_n$ this must be done for all the items in X. This procedure is shown in Algorithm 1.

Defining the search space as a tree (Figure 1 shows a 5 items search space) in which each node represents an item of the database, an itemset is determined by

Algorithm 1 $\overline{X}_n$ calculation	**Algorithm 2** Data interchange

Initialize vector $\overline{X}_{accum}$ with 1's
 for all *items i in X* **do**
 $\quad \overline{X}_{accum} = \overline{X}_{accum}$ and $\overline{i}$
 end for
 $\overline{X}_n = \overline{X}_{accum}$
	for all *sections S in DB* **do**
	 $\quad$ **for all** *data_vector in S* **do**
	 $\qquad Process\_Data(data\_vector);$
	 $\quad$ **end for**
	 end for
	 $Process\_Minsup(min\_sup\_value);$
	 $Result\ = Get\_Data();$

the shortest path between two nodes of the tree. Using the procedure previously described, the vector $\overline{X}_n$ of an itemset can be obtained recursively as $\overline{X}_n = \overline{X}_{n-1}$ and $\overline{n}$, being $\overline{X}_{n-1}$ the accumulated value in the parent node. Once $\overline{X}_{n-1}$ is calculated, it can be used to obtain all the accumulated vectors on each child node. With this process, each node provides partial results for each one of the itemsets that includes it in the path of the tree.
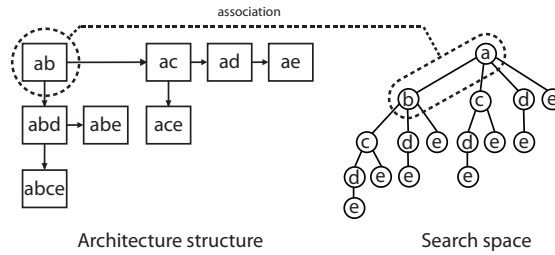

Architecture structure   Search space

Fig. 1. Structure for processing 5 item solution tree.

In this algorithm there is no candidate generation, but the search space is explored until reaching a node for which the support is zero. This process is sustained by the downward closure property, which establishes that the support of any itemset is greater than or equal to the support of any of its supersets. Because of this, if this node does not have active ones in the accumulated vector, the nodes in the lower subtree will not receive any contribution in the support value from that vector.

The lexicographical order of the items is established according to their frequency, ordering first the ones with the smallest values. This order causes the value of the support of itemsets to decrease rapidly when descending through the tree architecture.

Managing large databases with VBV data layout can be expensive because the size of $\overline{X}_n$ is determined by the number of transactions of the database. To solve this, a horizontal partition of the database is made. Each section is processed independently, as shown in Algorithm 2, and each node accumulates the itemsets section support. Every time a node calculates a partial support, it is

added to the accumulated support of the itemset. When the database processing finishes, each node has the global support of all the itemsets that it calculated.

A structure of processing elements is needed to implement the algorithm. This structure interchanges data as described in Algorithm 2. Two modes of data injection and one for data extraction are needed to achieve the task of frequent itemsets mining: "Data In", "Support Threshold" and "Data Flush". In the first mode ($Process\_Data()$) all data vectors are fed into the structure by the root element to process the itemsets. The second mode ($Process\_Minsup()$) injects the support threshold so the processing elements can determine which itemset is frequent and which is not. For the data extraction $Get\_Data()$, all the elements of the structure flush the frequent itemsets through their parent and the data exits the structure through the root element.

To implement the algorithm we use a binary tree structure of processing elements (PE). Figure 1 shows a five-item solution tree processing structure. The structure is a systolic tree and it was chosen because this type of construction allows to exponentially increase the concurrent operations at each processing step. For this, the number of concurrent operations can be calculated as $1 + \sum_{i=0}^{cc} 2^i$, being cc the number of processing steps that have elapsed since the process started.

Each node of the systolic tree (n_{arq}) is associated to a node of the solution tree (n_{sol}). This n_{arq} determines an itemset (IS_n) and it is formed with the path from the tree root to n_{sol}. The n_{arq} calculates the support of the supersets of IS_n. Those supersets are the ones that are determined by all the nodes in the path from the root to a leaf, following the node that is most to the left of the solution tree.

A PE has a connection from its parent and it is connected to a lower PE (PEu, with an upper entry) and to a right PE (PEl, with a left entry). In general, the amount of PEs a structure has, can be calculated as follows:

$$ST_n = 1 + PEu_{n-1} + PEl_{n-1}, \text{ for } n \geq 2,$$

with PEu_n and PEl_n:

$$
\begin{array}{l|l}
PEu_2 = 0, & PEl_1 = 0, \\
PEu_3 = 1, & PEl_2 = 1, \\
PEu_n = 1 + PEl_{n-2} + PEu_{n-1} & PEl_n = 1 + PEl_{n-1} + PEu_{n-1}
\end{array}
$$

3.1 Processing Elements

The itemsets that a PE calculates are determined by the data vectors that the parent feeds to it. This must be in such a way that the first vector to reach each PE is the $\overline{X}_n$ of the prefix of the equivalence class that the IS_n belongs to. Each PE calculates the support of the vector that receives from the parent, it accumulates the bitwise *and* operation in a local memory and propagates the data vector that it receives from the parent or the accumulated value correspondingly. To achieve this, the behavior of the two types of PEs is defined. In "Data In"

Algorithm 3 PEl in "Data In" mode	**Algorithm 4** PEu in "Data In" mode
if $(is\_first(data\_vector))$ **then**	**if** $(is\_first(data\_vector))$ **then**
$\quad and\_tmp = data\_vector$	$\quad and\_tmp = data\_vector$
$\quad Right\_vector = data\_vector$	$\quad Right\_vector = data\_vector$
else if $(is\_second(data\_vector))$ **then**	**else if** $(is\_second(data\_vector))$ **then**
$\quad and\_tmp = and\_tmp$ **and** $data\_vector$	$\quad Down\_vector = and\_tmp$ **and** $data\_vector$
$\quad Down\_vector = and\_tmp$	**else if** $(is\_third(data\_vector))$ **then**
$\quad mem[i] = calc\_sup(and\_tmp) + mem[i]$	$\quad and\_tmp = and\_tmp$ **and** $data\_vector$
else	$\quad Down\_vector = data\_vector$
$\quad and\_tmp = and\_tmp$ **and** $data\_vector$	$\quad mem[i] = calc\_sup(and\_tmp) + mem[i]$
$\quad Down\_vector = data\_vector$	**else**
$\quad Right\_vector = data\_vector$	$\quad and\_tmp = and\_tmp$ **and** $data\_vector$
$\quad mem[i] = calc\_sup(and\_tmp) + mem[i]$	$\quad Down\_vector = data\_vector$
end if	$\quad Right\_vector = data\_vector$
	$\quad mem[i] = calc\_sup(and\_tmp) + mem[i]$
	end if

mode, the main difference between PEl (described in Algorithm 3) and PEu (described in Algorithm 4) is that PEu does not accumulate the result of the bitwise *and* operation of the second data vector that it receives. This operation provides the down PE the prefix with the equivalence class of its IS_n.

3.2 Feedback Strategy

As the number of PEs of the tree is directly dependent on the number of frequent items that exist in the database, it is impractical to create this structure for databases with many frequent items. To solve this problem a structure for n items can be designed and taking into account the recursive definition of trees we could calculate the itemsets stepwise, see Figure 2. In the first step all the itemsets that can be calculated with this structure are obtained. Since it is known which level of the solution tree is processed for a given structure (Figure 2 shows it as the architecture processing border), data are injected back into this structure except that this time the first vector will not be the first item, but the prefix of the equivalence class of the itemset that is determined by the border tree node that was not processed.

In Figure 2, an example of a six-items search space is shown. The dotted line subtrees are examples of subtrees with different parents that have to be processed with the feedback strategy. In the example, to process the first subtree to the left, the first vector that has to be injected to the architecture is the vector that represents the itemset $X = \{a, b, c\}$.

This feedback is repeated for each solution tree node that is in the border of the nodes that were not processed. As each of these nodes defines a solution subtree (all the subtrees that are below the processing border in Figure 2), this structure is compatible with the entire tree and consequently can process a solution tree of any size. This process is defined recursively for the entire solution
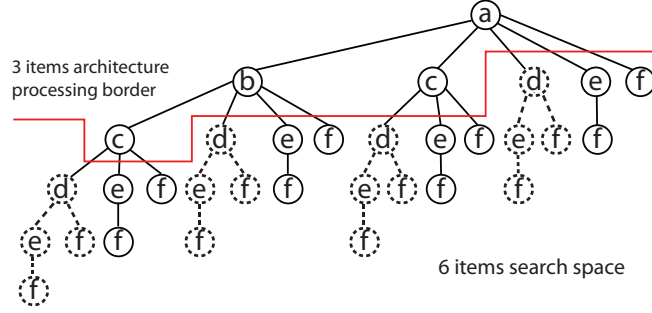
Fig. 2. Feedback strategy.

tree and as a result, we obtain a partition of the tree and each subtree of the partition represents a solution tree of n items.

4 Architecture structure

The proposed architecture has three modes of operation: "Data In", "Support Threshold" and "Data Flush". In the "Data In" mode, all $\overline{X}_n$ of each section of the database are injected. When the entire database is injected, the architecture enters in the "Support Threshold" mode and the support threshold value is provided and propagated through the systolic tree. Once a node receives the support threshold value, all the *PEs* change to "Data Flush" mode and the results are extracted from the architecture.

In this architecture, the amount of clock cycles for the entire process can be divided into two general stages: Data In (CC_{data}) and Data Flush (CC_{flush}). The CC_{data} is mainly defined by the number of frequent items in the database to specific support threshold (nf), the number of transactions of the database (T) and the size of the data vector that is chosen (bw). The data vector defines the number of sections (S) in the partition of the database. CC_{data} can be calculated as follows:

$$CC_{data} = S * (nf + 1) + 1, \text{ where } S = \lceil T/bw \rceil$$

For the data flush stage, CC_{flush} is mainly defined by the number of *PEs* that have frequent itemsets and the number of empty *PEs* in the path up to the next *PE* with useful data. In this strategy, once the data are extracted from a *PE*, data must be extracted from the *PEl*, since there is no information to determine if they will have frequent itemsets or not. In the case of the *PEu* in the lower subtree, if there are no frequent itemsets in the current *PE*, then no frequent itemsets will be found in the lower subtree. This is because all the itemsets calculated in the subtree will have as a prefix the first itemset of the current *PE*, and if this is not frequent, then no superset of it will be frequent.

The number of *PE* with and without useful data, and the position in the systolic tree that they occupy, only depends on the nature of the data and the

support threshold. In the worst case, all the *PEs* will have frequent itemsets and CC_{flush} could be calculated as $ST_n + |\text{FI set}|$.

4.1 Flush strategy

To obtain the data from the architecture, each *PE* enters in "Data Flush" operation mode. All *PE* have a connection to the parent so all the frequent itemsets can be flushed up (*Parent_out*). In this strategy each node of the systolic tree flushes the frequent itemsets stored in local memory, then it serves as a gateway to data from the lower *PE* (*Down_in*) and when it ends, it serves as a gateway to the data of the right *PE* (*Right_in*).

Algorithm 5 PE in "Data Flush" mode

> **if** $(mem[0] > min\_sup)$ **then**
>   **while** $(mem[i] > min\_sup)$ **do**
> $Parent\_out = mem[i]$
>   **end while**
> $Start\_flush(down\_child)$
>   **while** not_finish **do**
> $Parent\_out = Down\_in$
>   **end while**
> **end if**
> $Start\_flush(right\_child)$
> **while** not_finish **do**
> $Parent\_out = Right\_in$
> **end while**

Extracting the data through the root node of the systolic tree allows to take advantage of the characteristics of the itemsets. These characteristics permit to define heuristics to prune subtrees in the flush strategy and therefore shortens the amount of data that is necessary to flush from the architecture. Because of this, less time is needed to complete the task.

5 Implementation results

The proposed architecture was modeled using VHDL and it was verified in simulation with ModelSim SE 6.5 simulation tool. Once the architecture was validated, it was synthesized using ISE 9.2i. The target device was set to a Xilinx Virtex-4 XC4VFX140 with package FF1517 and -10 speed grade.

For the experiments, three datasets from [4] were used: Chess, Accidents and Retail. As explained in Section 4, the size of the architecture increases ruled by the number of frequent items that it is capable of processing, so the size of the device being used will highly affect the time it will take to finish the task. For the experiments we used a 32 bits data vector and the biggest architecture that fits the device was a structure to process 11 items with 264 *PE*s. This architecture
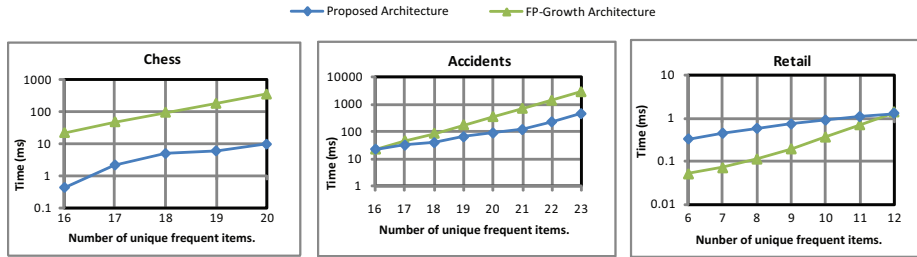
Fig. 3. Mining time comparison.

consumes 74.6% of the LUTs (94,248 out of 126,336) and $18,6\%$ of the flip-flops (23,496 out of 126,336) available in the device.

Since the architecture is completely decentralized, there are no global connections and thus the maximum operating frequency is not affected by the number of PEs of the architecture. The maximum frequency obtained for this architecture was 137 MHz. The proposed algorithm is sensitive to the number of frequent items more than the support threshold, so to show the behavior more precisely the experiments were carried out based on this variable.

In Figure 3 we compare the mining time of the architecture against the best time of the FP-Growth architecture presented in [9]. This Figure shows that the greatest improvement in execution time was for the Chess database, where more than one order of magnitude was obtained. In the other two databases it is shown how the execution time of the proposed architecture grows slower than the FP-Growth architecture when increasing the number of frequent items. Moreover, when increasing the ratio between the number of obtained frequent itemsets and the size of the database, the percentage of CC_{data} decreases. In the case of Accidents and Retail databases CC_{data} is 99% and 97% of total time correspondingly and for Chess database the percentage is between 62% and 68%. This is because the execution time of the "Data in" mode depends only on the amount of frequent items and the number of transactions that the databases have and the remaining time will depend on the number of frequent itemsets obtained.

This behavior shows a feature of the algorithm and its scalability. The architecture performs better when the density of the processed database increases and generally performs better when the number of frequent itemsets obtained also increases. This feature (better performance at higher density) has a great importance if it is considered that the denser the databases the more difficult to obtain frequent itemsets and the higher the number of frequent itemsets obtained.

For sparse databases and high supports (low number of frequent items), this task can be solved without the necessity of appealing to hardware acceleration in most cases. This need is more evident when dealing with large databases with low threshold of support or high density, or both. Because of this, this feature is desirable in the algorithms for obtaining frequent itemsets.

6 Conclusion

In this paper we proposed a parallel algorithm that is specially designed for environments which allow a high number of concurrent processes. This characteristic best suits a hardware environment and allows to use a task parallelism with fine granularity. Furthermore, a hardware architecture to validate the algorithm was developed. The experiments show that our approach outperforms the FP-Growth architecture presented in [10] by one order of magnitude when processing dense databases, and the mining time grows slower than the FP-Growth architecture mining time when processing sparse matrices.

References

1. R. Agrawal and J.C. Shafer. Parallel mining of association rules design, implementation and experience. Technical Report RJ10004, IBM Research Report, February 1996.
2. Z. K. Baker and V. K. Prasanna. Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs. In *Proc. of the 13th Annual IEEE Symposium on Field Programmable Custom Computing Machines 2005 (FCCM '05)*, pages 3–12, 2005.
3. Z. K. Baker and V. K. Prasanna. An Architecture for Efficient Hardware Data Mining using Reconfigurable Computing System. In *Proc. of the 14th Annual IEEE Symposium on Field Programmable Custom Computing Machines 2006 (FCCM '06)*, pages 67–75, 2006.
4. B. Goethals. Frequent itemset mining dataset repository. http://fimi.cs.helsinki.fi/data/.
5. E.H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proc. of the ACM SIGMOD Conference*, pages 277–288, 1997.
6. J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1–12. ACM Press, 2000.
7. J.H. Palancar, O.F. Tormo, J.F. Cárdenas, and R.H. León. Distributed and shared memory algorithm for parallel mining of association rules. In *MLDM '07: Proceedings of the 5th Intl. Conf. on Machine Learning and Data Mining in Pattern Recognition*, volume 4571 of *LNAI*, pages 349–363, 2007.
8. J. Park, M. Chen, and P. Yu. An effective hash based algorithm for mining association rules. In Michael J. Carey and Donovan A. Schneider, editors, *SIGMOD Conference*, pages 175–186. ACM Press, 1995.
9. S. Sun, M. Steffen, and J. Zambreno. A reconfigurable platform for frequent pattern mining. In *RECONFIG '08: Proc. of the 2008 Intl. Conf. on Reconfigurable Computing and FPGAs*, pages 55–60. IEEE Computer Society, 2008.
10. S. Sun and J. Zambreno. Mining association rules with systolic trees. In *Proc. of the Intl. Conf. on Field-Programmable Logic and its Applications (FPL)*, pages 143–148. IEEE, 2008.
11. Y. Wen, J. Huang, and M. Chen. Hardware-enhanced association rule mining with hashing and pipelining. *IEEE Trans. on Knowl. and Data Eng.*, 20(6):784–795, 2008.
12. M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proc. of the 3rd Intl. Conf. on KDD and Data Mining (KDD97)*, pages 283–286, 1997.