Asynchronous and Anticipatory Filter-Stream Based Parallel Algorithm for Frequent Itemset Mining*

Adriano Veloso[1], Wagner Meira Jr.[1], Renato Ferreira[1], Dorgival Guedes Neto[1], and Srinivasan Parthasarathy[2]

[1] Computer Science Department, Universidade Federal de Minas Gerais, Brazil
{adrianov,meira,renato,dorgival}@dcc.ufmg.br
[2] Department of Computer and Information Science, The Ohio-State University, USA
srini@cis.ohio-state.edu

Abstract. In this paper we propose a novel parallel algorithm for frequent itemset mining. The algorithm is based on the filter-stream programming model, in which the frequent itemset mining process is represented as a data flow controlled by a series of producer and consumer components (called filters), and the data flow (communication) between such filters is made via streams. When production rate matches consumption rate, and communication overhead between producer and consumer filters is minimized, a high degree of asynchrony is achieved. Following this strategy, our algorithm employs an asynchronous candidate generation, and minimizes communication between filters by transferring only the necessary aggregated information. Another nice feature of our algorithm is a look forward approach which accelerates frequent itemset determination. Extensive evaluation shows the parallel performance and scalability of our algorithm.

1 Introduction

The importance of data mining and knowledge discovery is growing. Fields as diverse as astronomy, finance, bioinformatics, cyber-security are among the many facing the situation where large amounts of data are collected and accumulated at an explosive rate. Analyzing such datasets without the use of some kind of data reduction/mining is getting more and more infeasible and thus there has been an increasing clamor for mining such data efficiently.

The problem is that mining such large and potentially dynamic datasets is a compute-intensive task, and even the most efficient of sequential algorithms may become ineffective. Thus, implementation of non-trivial data mining algorithms in high performance parallel computing environments is crucial to improving response times.

Mining frequent patterns/itemsets is the core of several data mining tasks. Much attention has gone to the development of parallel algorithms for such tasks[2, 7–9, 12, 16]. However, there are yet several challenges as yet unsolved.

First, parallelizing frequent itemset mining can be complicated and communication intensive. Almost all existing algorithms require multiple synchronization points.

* This work has been partially supported by CNPq-Brazil and by CNPq / CT-INFO / PTACS.

J.-F. Boulicaut et al. (Eds.): PKDD 2004, LNAI 3202, pp. 422–433, 2004.

Second, achieving good workload balancing in parallel frequent itemset mining is extremely difficult, since the amount of computation to be performed by each computing unit does not depend only on the amount of data assigned to it. In fact, equal-sized blocks of data (or partitions) does not guarantee equal (nor approximately equal) workloads, since the number of frequent itemsets generated from each block can be heavily skewed[7]. Thus, an important problem that adversely affects workload balancing is sensitivity to data skew.

Third, system utilization is an issue that is often overlooked in such approaches. The ability to make proper use of all system resources is essential in order to provide scalability when mining frequent itemsets in huge datasets.

In this paper we present a new parallel algorithm for frequent itemset mining, based on the filter-stream programming model. Essentially, the mining process is viewed as a coarse grain data flow controlled by a series of components, referred to as *filters*. A filter receives some data from other filters, performs specific processing in this data, and feeds other filters with the transformed/filtered data. Filters are connected via *streams*, where each stream denotes a unidirectional data flow from a producer filter to a consumer filter. This new approach for parallel frequent itemset mining results in interesting contributions, which can be summarized as follows:

- The candidate generation can work in an asynchronous way, yielding a very effective approach for determining frequent itemsets. Also, the algorithm communicates only the necessary aggregate information about itemsets.
- The parallel algorithm is also anticipatory, in the sense that it can look ahead if a candidate is frequent without the necessity of examining it over all partitions first. This ability becomes more effective when the dataset has a skewed itemset support distribution, a common occurrence in real workloads. In some sense, it compensates the general negative impact of data skewness in workload balancing.
- Finally, we demonstrate thought an extensive experimental evaluation that our algorithm utilizes the available resources very effectively and scale very well for huge datasets and large parallel configurations.

2 Definitions and Related Work

DEFINITION 1. [ITEMSETS] For any set $\mathcal{X}$, its size is the number of elements in $\mathcal{X}$. Let $\mathcal{I}$ denote the set of n natural numbers $\{1, 2, ..., n\}$. Each $x \in \mathcal{I}$ is called an item. A non-empty subset of $\mathcal{I}$ is called an itemset. An itemset of size k, $\mathcal{X} = \{x_1, x_2, ..., x_k\}$ is called a k-itemset.

DEFINITION 2. [TRANSACTIONS] A transaction $\mathcal{T}_i$ is an itemset, where i is a natural number called the *transaction identifier*. A transaction dataset $\mathcal{D} = \{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_m\}$, is a finite set of transactions, with size $|\mathcal{D}| = m$. The support of an itemset $\mathcal{X}$ in $\mathcal{D}$ is the number of transactions in $\mathcal{D}$ that contain $\mathcal{X}$, given as $\sigma(\mathcal{X}, \mathcal{D}) = |\{\mathcal{T}_i \in \mathcal{D} \,|\mathcal{X} \subseteq \mathcal{T}_i\}|$.

DEFINITION 3. [FREQUENT ITEMSETS] An itemset $\mathcal{X}$ is frequent in the dataset $\mathcal{D}$ iff $\sigma(\mathcal{X}, \mathcal{D}) \geq \sigma^{min}$, where σ^{min} is a user-specified minimum-support threshold, with values $0 < \sigma^{min} \leq |\mathcal{D}|$. The set of all frequent itemsets is denoted as $\mathcal{F}(\sigma^{min}, \mathcal{D})$.

PROBLEM 1. [MINING FREQUENT ITEMSETS] Given σ^{min} and a transaction dataset $\mathcal{D}$, the problem of mining frequent itemsets is to find $\mathcal{F}(\sigma^{min}, \mathcal{D})$.

Several parallel algorithms for frequent itemset mining were already proposed in the literature [2, 7–9, 16]. The majority of the proposed algorithms follow one of the three main parallelizing strategies:

1. COUNT DISTRIBUTION: This strategy follows a data-parallel paradigm in which the dataset is partitioned among the processing units (while the candidates are replicated). One drawback of this strategy is that at the end of each iteration all processing units must exchange local supports, incurring in several rounds of synchronization. We alleviate this problem by presenting algorithms [14] that need only one round of synchronization by employing an upper bound for the global *negative border* [10]. FDM [7] is another algorithm built on the COUNT DISTRIBUTION strategy. It employs new pruning techniques to reduce processing and communication[1]. Several other algorithms [2, 11] also follow this strategy.
2. CANDIDATE DISTRIBUTION: This strategy follows a paradigm that identifies disjoint partitions of candidates. A common strategy is to partition candidates based on their prefixes, and this strategy can incur in poor workload balancing. PARECLAT [16] is an algorithm that follows this strategy.
3. DATA DISTRIBUTION: This strategy attempts to maximize the use of all aggregate main memory, but requires to transfer the entire dataset at the end of each iteration, incurring in very high communication overheads.

Our parallel algorithm distributes both counts and candidates and has an excellent asynchrony. Further, it presents benefits due to the use of a novel anticipation method.

3 The Filter-Stream Programming Model

The filter-stream programming model was originally proposed for Active Disks [1], to allow the utilization of the disk resident processor in a safe and efficient way. The idea was to exploit the extra processor to improve application performance in two ways. First, alleviating the computation demand on the main processor by introducing the extra, mostly idle processor. Second, it was expected that such computation would reduce the amount of data that needed to be brought from the disk to the main memory. The proposed model, introduced the concept of disklets, or filters, which are entities that perceive streams of data flowing in, and after some computation it would generate streams of data flowing out. In a sense, it is very similar to the concept of UNIX pipes. The difference is that while pipes only have one stream of data coming in and one going out, in the proposed model, arbitrary graphs with any number of input and output streams are possible. Later, this concept was extended as a programming model suitable for the Grid environment [5]. A runtime system, called DATACUTTER (DC) was

[1] Other interesting proposal introduced by the same authors is a metric for quantifying data skewness. For an itemset $\mathcal{X}$, let $p_i(\mathcal{X})$ denote the probability that $\mathcal{X}$ occurs in partition i. The entropy of $\mathcal{X}$ is given as $\mathcal{H} = -\sum_i^n p_i(\mathcal{X}) \times log(p_i(\mathcal{X}))$. The skewness of $\mathcal{X}$ is given as $S(\mathcal{X}) = \frac{log(n)-\mathcal{H}(\mathcal{X})}{log(n)}$, where n is the number of partitions. A dataset's total data skewness is the sum of the skew of all itemsets weighted by their supports.

then developed to support such model. Applications from different domains have been successfully implemented using DC [4, 6, 13]. Creating an application in DC consists of decomposing the target application into filters. These filters are than scheduled for execution in the machines comprising a Grid environment.

Streams in DC represent unidirectional pipes. A filter can either write into or read from the stream. The units of communication are fixed size buffers, agreed upon by the two sides. Each stream has a name associated with it and the connecting of the endpoints of a stream is done at execution time. Once it is done, buffers that are written by the sender will eventually be available for reading on the recipient side. Delivery is guaranteed, and there is no duplication.

The instantiation of the filters is performed by the runtime environment. One of the most important concepts in DC is that of transparent filter copies. At execution time, many instances of the same filter can be created. This provides a simple way to express and implement parallelism, reduce the computation time and to balance the time spend on the several stages of the computation. So, while the concept of the decomposition of the application into filters is related to task parallelism, the possibility of having multiple replicas of the same filter, on the other hand, is related to data parallelism. DC nicely integrates both forms of parallelism into one orthogonal environment.

With respect to multiple copies of the same filter, or data parallelism, the difference between one copy and another is the portion of the entire data each copy has seen. If the filter needs to maintain a state, it is vital that data related to the same portion of the entire data to be always sent to the same copy. Moreover, the data buffers being sent onto the streams need to be transported from one copy of the originating filter to one specific copy of the recipient filter.

For most cases, the selection of the actual destination of any given message buffer actually consider the data in the message to be untyped. However, for applications that maintain some state, it is important to have some understanding of the contents of the message as to decide to which copy of the destination filter needs to be delivered. For these cases, DC implements *labelled streams* which extend the notion of the buffer to a tuple $< l, m >$ where l is a label and m is the message. Associated with each stream there is a label domain L and a hash function h which maps the label from L to $h(l)$. The label domain defines valid values for labels in that stream and the hash function defines a domain which may be associated with filter replicas. With the labeled stream, the stream can use a mapping from that value $h(l)$ to the set of replicas to decide to which copy of the filter should the buffer $< l, m >$ be delivered.

4 Filter-Stream Based Parallel Algorithm

In this section we present our parallel algorithm for frequent itemset mining. We start by discussing its rationale and then we raise some implementation issues.

For sake of filter definition, we distinguish three main tasks to determine whether a k-itemset is frequent or not:

1. verify whether its $(k-1)$-subsets are frequent; if so,
2. count its local supports; and,
3. check whether its global support is above the minimum-support.

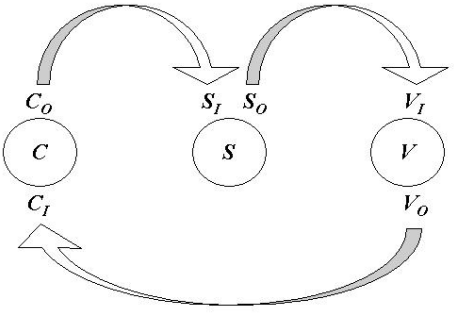

Fig. 1. Algorithm Execution and Data Flow.

The verification filter $\mathcal{V}$ receives as input (represented by the stream $\mathcal{V}i$) the itemsets found to be frequent so far and determines the itemsets that should be verified as being possibly frequent (the candidates), which are the output $\mathcal{V}o$. The counter filter $\mathcal{C}$ receives a candidate itemset through its input stream $\mathcal{C}i$, scans the dataset and determines the support of that candidate itemset, which is sent out using the stream $\mathcal{C}o$. The support checker filter receives the support associated with an itemset through the stream $\mathcal{S}i$, checks whether the counter is above the support threshold and notifies the proper verification filter via the stream $\mathcal{S}o$. We may express the computation involved in determining the frequent itemsets by instantiating the three filters for each itemset, as depicted in Figure 1. In this case, the stream $\mathcal{V}o$ and $\mathcal{C}i$ are connected, as well as the streams $\mathcal{C}o$ and $\mathcal{S}i$. The streams $\mathcal{S}o$ are connected to streams $\mathcal{V}i$ according to the itemset dependence graph[2]. Formally, we have two label domains $\mathcal{I}$ and $\mathcal{T}$ associated with itemsets and transactions, respectively. The streams $\mathcal{S}o$ and $\mathcal{V}i$ are associated with the domain $\mathcal{I}$, while the others with the itemset $\mathcal{T}$.

In the context of filters/streams, there are two dimensions where the parallelization of frequent itemset mining algorithms may be exploited: candidates and counts. We employ both strategies in our algorithm. The verification and counter filters, when created with multiple instances, employ a count distribution strategy, while the support checker filter adopts the candidate division among its instances. This strategy puts together filters from several levels of the dependence graph (that is, filters associated with itemsets of various sizes), using the label concept of our programming model. Although the mapping functions in this case may not be simple, this approach both uses the available plataform efficiently and does not require any replica of the transaction dataset. Further, the granularity of the parallelism that may be exploited is very fine, since we may assign a single transaction or itemset to a filter, without changing the algorithm nor even its implementation.

The execution of the algorithm starts with the counter filters. Each counter filter has access to its local dataset partition, and the first step is to count the 1-itemsets, by scanning its partition and building the *tidsets*[3] of the 1-itemsets. At this point, a label

[2] The vertices in the dependence graph are the itemsets and the edges represent which itemsets that are subsets of a given itemset and must be frequent so that the former may be also frequent.

[3] The set of all transaction identifiers in which a given itemset has occurred. Other data structures, such as diffsets [15], can also be used.

is assigned to each counted candidate, and such label is coherent across all filters (i.e., a candidate has the same label in all filters). The next step is to discover the frequent 1-itemsets, and so each counter filter sends a pair {*candidate label*, *local support*} to a support checker filter. For each candidate received, the support checker filter simply sums its local supports. When this value reaches the minimum-support threshold, the support checker filter determines that the candidate is frequent, and broadcasts its label to all verifier filters.

Each verifier filter receives the candidate label from the support checker filter and interprets that the candidate associated with the label is frequent. As the labels of frequent 1-itemsets arrive at verifier filters, it is possible to start counting the 2-itemsets that are enumerated from those frequent 1-itemsets. In order to control this process, each verifier filter maintains a prefix-tree that efficiently returns all candidates that must be counted. Note that, because the support checker filter communicates via broadcast and the prefixes are lexicographically ordered, all candidates are verified and counted according to the same order across the filters, being easy to label a candidate, since its label is simply a monotonically increasing number. As soon as a candidate is counted, the counter filter sends another pair {*candidate label, local support*} to the support checker filter, and the process continues until a termination condition is reached and all frequent itemsets were found. From this brief description we distinguish three major issues that should be addressed for implementing our algorithm: performance, anticipation, and termination condition.

Performance: Each filter produces and consumes data at a certain rate. The best performance occurs when the instances of filters are balanced with respect to each other and the communication overhead between the filters is minimized. That is, the data production rates of the producer filters should match the data comsuption rates of the consumer filters. In our case, the number of counter filters must be larger than the number of the other two filters, since counter filters perform a more computational intensive task than the other filters. The optimal number of instances for each filter may vary according to dataset characteristics (i.e., size, density etc.).

Anticipation: A support checker filter does not need to wait for all local supports of a given candidate to determine if it is frequent. In fact, the support checker filter can anticipate this information to the verifier filters, increasing the throughput of the support checker filter and consequently accelerating the whole process. Clearly, data distribution has a major hole in the effectiveness of the anticipation process. In fact, skewed distributions will provide the best gains.

Termination Condition: The execution terminates iff all filters have no more work to be done. However, it is difficult to detect this condition because of the circular dependence among filters. Fortunately, our algorithm has one property that facilitates the termination detection − the candidates are generated in the same order across filters, so that the candidate label may work as a local global clock, which is synchronized when all candidate labels are equal among all filters. At this point, there is no more work to be done.

5 Experimental Evaluation

In this section we present experimental results of our parallel algorithm. Sensitivity analysis on our algorithm was conducted on data distribution, data size, and degree of parallelism. We used both real and synthetic datasets as inputs to the experiments. The real dataset used is called KOSARAK, and it contains click-stream data of a Hungarian on-line news portal (KOSARAK has approximately 900,000 transactions). The synthetic datasets, generated using the procedure described in [3], have sizes varying from 560MB (D3.2MT16I12) to 2.2GB (D12.8MT16I12). To better understand how data distribution (i.e., data skewness) affects the performance of our parallel algorithm, we distributed the transactions among the partitions in two different ways:

- Random Transaction Distribution ($\mathcal{D}_R$): Transactions are randomly distributed among equal-sized partitions. This strategy tends to reduce data skewness, since all partitions have an equal probability to contain a given transaction.
- Original Transaction Distribution ($\mathcal{D}_O$): The dataset is simply splited into blocked partitions, preserving its original data skewness.

We start by analyzing the parallel efficiency of our algorithm. We define the parallel efficiency as: $\mu_{p,q} = \frac{\mathcal{T}_p}{q/p \times \mathcal{T}_q}$, where $\mathcal{T}_p$ is the total execution time when p processors are being employed. A parallel efficiency equals to 1 means linear speedup, and when it gets above 1 it indicates that the speedup is super-linear. Table 1 shows how the parallel efficiency varies as a function of dataset size, transaction distribution and degree of parallelism. Parallel efficiency gets much better when the anticipating procedure is used and dataset is larger. Parallel efficiency continues to be high even for larger degrees of parallelism, reaching 7% of improvement in the best case.

Table 1. Parallel Efficiency.

Dataset	Distribution	Anticipating	$\mathcal{T}_8$ (sec)	$\mu_{8,16}$	$\mu_{16,32}$
D6.4MT16I12	$\mathcal{D}_O$	NO	126.38	1.07	0.88
D6.4MT16I12	$\mathcal{D}_O$	YES	124.11	1.07	1.06
D6.4MT16I12	$\mathcal{D}_R$	NO	94.93	1.02	0.89
D6.4MT16I12	$\mathcal{D}_R$	YES	92.13	1.02	0.99
D12.8MT16I12	$\mathcal{D}_O$	NO	194.74	1.05	0.98
D12.8MT16I12	$\mathcal{D}_O$	YES	188.18	1.07	1.07
D12.8MT16I12	$\mathcal{D}_R$	NO	168.35	1.00	0.97
D12.8MT16I12	$\mathcal{D}_R$	YES	165.96	1.02	1.01
KOSARAK	$\mathcal{D}_O$	NO	642.82	0.96	0.85
KOSARAK	$\mathcal{D}_O$	YES	639.14	1.00	0.95
KOSARAK	$\mathcal{D}_R$	NO	640.87	0.97	0.94
KOSARAK	$\mathcal{D}_R$	YES	639.91	0.99	0.95

We also evaluated our algorithm by means of traditional speedup and scaleup experiments. Figure 2 shows speedup and scaleup numbers obtained from the synthetic

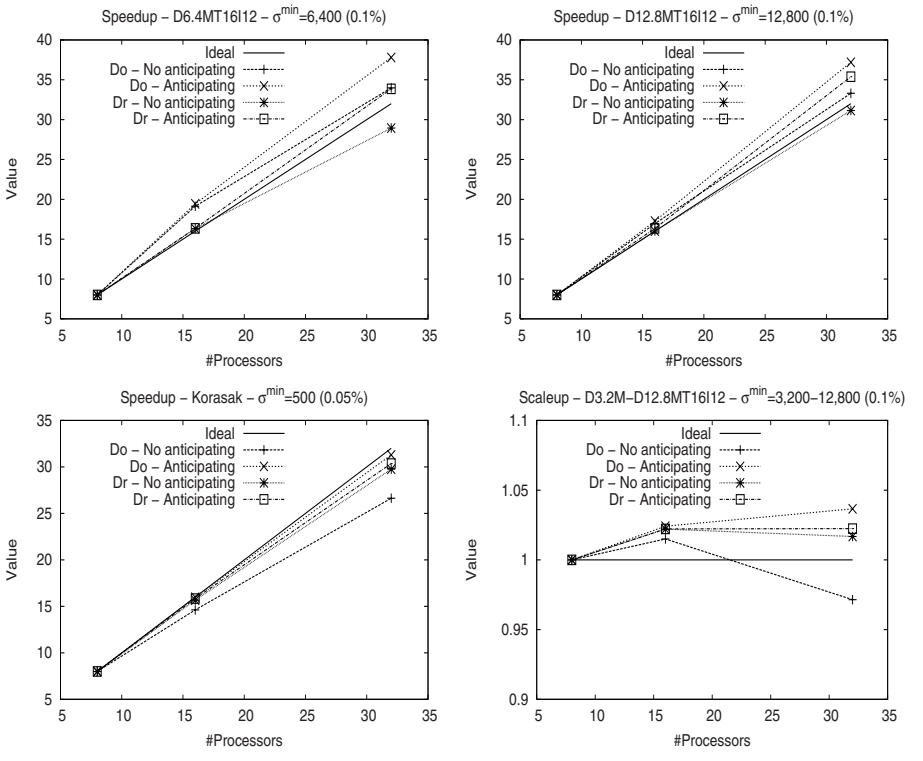

Fig. 2. Speedup and Scaleup Numbers (in relation to 8 processors).

and real datasets. Again, we varied the size, transaction distribution and degree of parallelism. For the speedup experiments with synthetic data we employed datasets with different sizes (6,400,000 and 12,800,000 transactions), and for the two datasets employed we observed superlinear speedups when the anticipation procedure is used. We also observed a superlinear speedup without the anticipation, but in this case the dataset has a random transaction distribution. Further, the speedup number tends to get better for larger datasets, since there is less variability. Impressive numbers were also observed with real data, and the best result was achieved with original (skewed) transaction distribution and using the anticipation procedure.

For the scaleup experiments, we varied dataset size and degree of parallelism in the same proportion. Dataset size ranges from 3,200,000 transactions (with 8 processors) to 12,800,000 transactions (with 32 processors). As we can see in Figure 2, our parallel algorithm also presents ultra scalability when the anticipation procedure is employed, or a random transaction distribution is used. Even when using original transaction distribution without anticipation, our algorithm shows to be very scalable, reaching approximately 95% of scalability.

As seem in both speedup and scaleup experiments, the anticipation is more effective in the presence of data skewness, since the probability of a frequent itemset be sent to the verifier filter using less partitions is higher. Figure 3 shows the average number

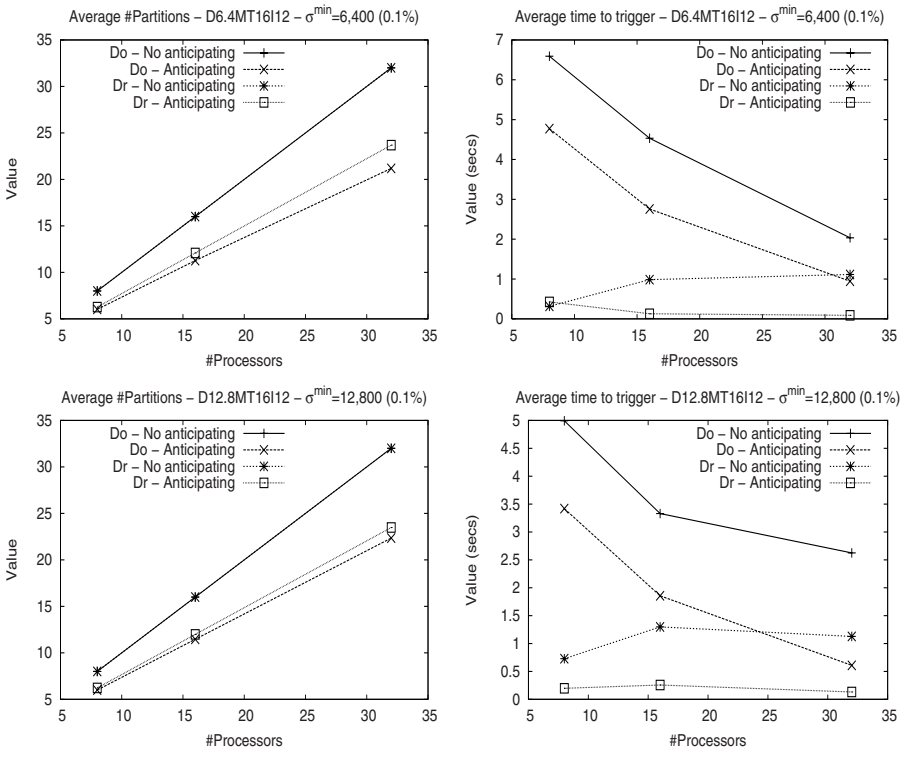

Fig. 3. Effects of Anticipating: Necessary Partitions and Average Time to Trigger.

of partitions necessary to trigger an itemset as frequent. Note that, without any anticipation, all partitions must be analyzed, but when the anticipation procedure is used, the number of partitions that must be analyzed to trigger a frequent itemset is smaller and varies with data skewness. As expected, the same trend is also observed in the average time to trigger a frequent itemset. Further, smaller itemsets usually are detected to be frequent earlier. In Table 2 we present the anticipation gain, that is, how long in advance the itemset is found to be frequent. In a 32-processor configuration using the D12.8MT16I12 dataset the gain varied from 60% to 7%. This observation is particularly interesting because the number of short-sized itemsets is much greater than large-sized itemsets, in particular 2- and 3-itemsets.

Table 2. Anticipation gain for D12.8MT16I12 using 32 processors.

Itemset size	1	2	3	4	5	6
Anticipation gain	60.6%	47.0%	44.5%	16.2%	6.0%	7.2%

In order to better understand the dynamics of the parallel algorithm, we introduce some metrics that quantify the various phases of the algorithm. We may divide the determination of the support of an itemset into four phases:

Activation: The various notifications necessary for an itemset become a candidate may not arrive at the same time, and the verification filter has to wait until the conditions for an itemset be considered candidate are satisfied.

Contention: After the itemset is considered a good candidate, it may wait in the processing queue of the counter filter.

Counting: The counter filters may not start simultaneously, and the counting phase is characterized by counter filters calculating the support of a candidate itemset in each partition.

Checking: The local supports coming from each counter filter may not arrive at the same time in the support checker filter, and the checking phase is the time period during which the notifications arrive.

Next we are going to analyze the duration of these phases in both speedup and scaleup experiments. The analysis of the speedup experiments explains the efficiency achieved, while the analysis of scaleup experiments shows the scalability.

In Table 3 we show the duration of the phases we just described for configurations employing 8, 16, and 32 processors for mining the D12.8MT16I12 dataset. The rightmost column also shows the average processing cost for counting an itemset, where we can see that this cost reduces as the number of processors increase, as expected. The same may be observed for all phases, except for the Activation phase, whose duration seems to reach a limit around 1 second. The problem in this case is that the number of processors involved is high and the asynchronous nature of the algorithm makes the reduction of the Activation time very difficult.

Table 3. Speedup Experiment: Profiling (secs).

Proc	Activation	Contention	Counting	Checking	Processing
8	2.741046	5.564751	9.412093	8.469050	0.001645
16	1.264842	2.058052	4.893773	4.691232	0.000759
32	1.229330	0.273229	1.129718	1.986129	0.000369

Verifying the timings for the scaleup experiments in Table 4, we verify the scalability of our algorithm. We can see that an increase in the number of processors and in the size of the dataset does not affect significantly the measurements, that is, the algorithm implementation does not saturate system resources (mainly communication) when scaled.

Table 4. Scaleup Experiment: Profiling (secs).

Proc	Activation	Contention	Counting	Checking	Processing
8	2.741046	5.564751	9.412093	8.469050	0.001645
16	2.628118	5.538353	9.349371	8.403360	0.001596
32	2.439369	5.021002	10.311501	8.906631	0.001594

6 Conclusion and Future Work

In this paper we proposed an algorithm to conduct parallel frequent itemset mining. The proposed algorithm is based on the filter-stream programming model, where the computation of frequent itemsets is expressed as a circular data flow between distinct components or filters. Parallel performance is optimized, and a high degree of asynchrony is achieved by using the right number of each filter. Further, we propose a very simple anticipation approach, which accelerates frequent itemset determination (specially in the presence of data skew). It was empirically showed that our algorithm achieves excellent parallel efficiency and scalability, even in the presence of high data skewness.

Future work includes the development of parallel filter-stream based algorithms for other data mining techniques, such as maximal/closed frequent itemsets and frequent sequential patterns. Utilization of our algorithms in real applications is also a possible target.

References

1. A. Acharya, M. Uysal, and J. Satlz. Active disks: Programming model, algorithms and evaluation. In *Proc. of the Intl. Conf. on Architectural Support for programming Languages and Operating Systems (ASPLOS VIII)*, pages 81–91. ACM Press, Oct 1998.
2. R. Agrawal and J. Shafer. Parallel mining of association rules. *Transactions on Knowledge and Data Engineering*, 8(6):962–969, 1996.
3. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the Intl. Conf. on Very Large Databases (VLDB)*, pages 487–499, SanTiago, Chile, June 1994.
4. M. Beynon, C. Chang, U. Catalyurek, T. Kurc, A. Sussman, H. Andrade, R. Ferreira, and J. Saltz. Processing large-scale multi-dimensional data in parallel and distributed environments. *Parallel Computing*, 28(5):827–859, 2002.
5. M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a framework for data-intensive wide-area applications. In *Proc of the Heterogeneous Computing Workshop (HCW)*, pages 116–130. IEEE Computer Society Press, May 2000.
6. U. Catalyurek, M. Gray, T. Kurc, J. Saltz, and R. Ferreira. A component-based implementation of multiple sequence alignment. In *Proc. of the ACM Symposium on Applied Computing (SAC)*, pages 122–126. ACM, 2003.
7. D. Cheung and Y. Xiao. Effect of data distribution in parallel mining of associations. *Data Mining and Knowledge Discovery*, 3(3):291–314, 1999.
8. E. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. *Transactions on Knowledge and Data Engineering*, 12(3):728–737, 2000.
9. M. Joshi, E. Han, G. Karypis, and V. Kumar. Efficient parallel algorithms for mining associations. *Parallel and Distributed Systems*, 1759:418–429, 2000.
10. H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.
11. S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. An efficient parallel and distributed algorithm for counting frequent sets. In *Proc. of the Intl. Conf. on Vector and Parallel Processing (VECPAR)*, pages 421–435, Porto, Portugal, 2002.
12. S. Parthasarathy, M. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems*, 3(1):1–29, 2001.
13. M. Spencer, R. Ferreira, M. Beynon, T. Kurc, U. Catalyurek, A. Sussman, and J. Saltz. Executing multiple pipelined data analysis operations in the grid. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, pages 1–18. IEEE Computer Society Press, 2002.

14. A. Veloso, M. Otey, S. Parthasarathy, and W. Meira. Parallel and distributed frequent itemset mining on dynamic datasets. In *Proc. of the High Performance Computing Conference (HiPC)*, Hyderabad, India, December 2003. Springer and ACM-SIGARCH.
15. M. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proc. of the Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*. ACM, August 2003.
16. M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery*, 4(1):343–373, December 1997.