An Efficient Parallel and Distributed Algorithm for Counting Frequent Sets

S. Orlando[1], P. Palmerini[1,2], R. Perego[2], F. Silvestri[2,3]

[1] Dipartimento di Informatica, Università Ca' Foscari, Venezia, Italy
[2] Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy
[3] Dipartimento di Informatica, Università di Pisa, Italy

Abstract Due to the huge increase in the number and dimension of available databases, efficient solutions for counting frequent sets are nowadays very important within the Data Mining community. Several sequential and parallel algorithms were proposed, which in many cases exhibit excellent scalability. In this paper we present ParDCI, a distributed and multithreaded algorithm for counting the occurrences of frequent sets within transactional databases. ParDCI is a parallel version of DCI (Direct Count & Intersect), a multi-strategy algorithm which is able to adapt its behavior not only to the features of the specific computing platform (e.g. available memory), but also to the features of the dataset being processed (e.g. sparse or dense datasets). ParDCI enhances previous proposals by exploiting the highly optimized counting and intersection techniques of DCI, and by relying on a multi-level parallelization approach which explicitly targets clusters of SMPs, an emerging computing platform. We focused our work on the efficient exploitation of the underlying architecture. Intra-Node multithreading effectively exploits the memory hierarchies of each SMP node, while Inter-Node parallelism exploits smart partitioning techniques aimed at reducing communication overheads. In depth experimental evaluations demonstrate that ParDCI reaches nearly optimal performances under a variety of conditions.

1 Introduction

Association Rule Mining (ARM) [6,7] is one of the most popular topic in the Data Mining field. The process of generating association rules has historically been adopted for *Market-Basket Analysis*, where transactions are records representing point-of-sale data, while items represent products on sale. The importance for marketing decisions of association rules like "the 80% of customers who buy products X with high probability also buy Y" is intuitive, and explains the strong interest in ARM.

Given a database of transactions $\mathcal{D}$, an association rule has the form $X \Rightarrow Y$, where X and Y are sets of items (*itemsets*), such that $(X \cap Y) = \varnothing$. A rule $X \Rightarrow Y$ holds in $\mathcal{D}$ with a minimum confidence c and a minimum support s, if at least the c% of all the transactions containing X also contain Y, and $X \cup Y$ is present in at least the s% of all the transactions of the database. It's important to distinguish among support and confidence. While confidence measures the probability of having Y when X is given, the support of a rule measures the number of transactions in $\mathcal{D}$ that contain both X and Y.

The ARM process can be subdivided into two main steps. The former is concerned with the Frequent Set Counting (*FSC*) problem. During this step, the set $\mathcal{F}$ of all the *frequent* itemsets is built, where an itemset is frequent if its support is greater than a fixed support threshold s, i.e. the itemset occurs in at least *minsup* transactions ($minsup = s/100 \cdot n$, where n, is the number of transaction in $\mathcal{D}$). In the latter step the association rules satisfying both minimum support and minimum confidence conditions are identified. While generating association rules is straightforward, the first step may be very expensive both in time and space, depending on the support threshold and the characteristic of the dataset processed. The computational complexity of the FSC problem derives from the size of its search space $\mathcal{P}(M)$, i.e. the power set of M, where M is the set of items contained in the various transactions of $\mathcal{D}$. Although $\mathcal{P}(M)$ is exponential in $m = |M|$, effective pruning techniques exist for reducing it. The capability of effectively pruning the search space derives from the intuitive observation that none of the superset of an infrequent itemset can be frequent. The search for frequent itemsets can be thus restricted to those itemsets in $\mathcal{P}(M)$ whose subsets are all frequent. This observation suggested a level-wise, or breadth-first, visit of the lattice corresponding to $\mathcal{P}(M)$, whose partial order is specified by the subset relation ($\subseteq$) [16].

Apriori [3] was the first effective algorithm for solving FSC. It iteratively searches frequent itemsets: at each iteration k, the set F_k of all the frequent itemsets of k items (k-itemsets), is identified. In order to generate F_k, a *candidate* set C_k of potentially frequent itemsets is first built. By construction, C_k is a superset of F_k, and thus in order to discover frequent k-itemsets, the supports of all candidate sets are computed by scanning the entire transaction database $\mathcal{D}$. All the candidates with minimum support are then included in F_k, and the next iteration is started. The algorithm terminates when F_k becomes empty, i.e. when no frequent set of k or more items is present in the database. *Apriori* strongly reduces the number of candidate sets generated on the basis of a simple but very effective observation: a k-itemset can be frequent only if all its subsets of $k-1$ items are frequent. C_k is thus built at each iteration as the set of all k-itemsets whose subsets of $k-1$ items are all included in F_{k-1}. Conversely, k-itemsets that at least contain an infrequent $(k-1)$-itemset are not included in C_k.

Several variations to the original *Apriori* algorithm, as well as many parallel implementations, have been proposed in the last years. We can recognize two main methods for determining itemset supports: a *counting*-based [1,3,5,8,13] and an *intersection*-based [14,16] one. The former one, also adopted by *Apriori*, exploits a *horizontal* dataset, where the transactions are stored sequentially. The method is based on *counting* how many times each candidate k-itemset occurs in every transaction. The intersection–based method, on the other hand, exploits a *vertical* dataset, where a *tidlist*, i.e. a list of the identifiers of all the transactions which contain a given item, is associated with the identifier of the item itself.

In this case the support of any k-itemset can be determined by computing the cardinality of the tidlist resulting from the k-way intersection of the k tidlists associated with the corresponding k items. If we are able to buffer the tidlists of previously computed frequent $(k-1)$-itemsets, we can speedup the computation since the support of a generic candidate k-itemset c can be simply computed by intersecting the tidlists of two $(k-1)$-itemsets whose union produces c. The *counting*-based approach is, in most cases, quite efficient from the point of view of memory occupation, since it only requires enough main memory to store C_k along with the data structures exploited to make the access to candidate itemsets faster (e.g. hash-trees or prefix-trees). On the other hand, the *intersection*-based method is more computational effective [14]. Unfortunately, it may pay the reduced computational complexity with an increase in memory requirements, in particular to buffer the tidlists of previously computed frequent $(k-1)$-itemsets.

In this paper we discuss ParDCI, a parallel and distributed implementation of DCI (Direct Count & Intersect), which is an effective FSC algorithm previously proposed by the same authors [12]. DCI resulted faster than previously proposed, state-of-the-art, sequential algorithms. The very good results obtained for sparse and dense datasets, as well as for real and synthetically generated ones, justify our interest in studying parallelization and scalability of DCI. ParDCI works similarly to DCI. It is based on a level-wise visit of $\mathcal{P}(M)$, and adopts a *hybrid* approach to determine itemset supports: it exploits an effective *counting*-based method during the first iterations, and a very fast *intersection*-based method during the last ones. When the counting method is employed, ParDCI relies on optimized data structures for storing and accessing candidate itemsets with high locality. The database is partitioned among the processing nodes, and a simple but effective database pruning technique [11,12] is exploited which allows to trim the transaction database partitions as execution progresses. When the pruned dataset is small enough to fit into the main memory, ParDCI changes its behavior, and adopts an intersection-based approach to determine frequent sets. The representation of the dataset is thus transformed from horizontal into vertical, and the new dataset is stored in-core on each node. Using this approach, candidates are partitioned in a way that grants load balancing, and the support of each candidate itemset is locally determined on-the-fly by intersecting the corresponding tidlists. Tidlists are actually represented as vectors of *bits*, which can be accessed with high locality and intersected very efficiently. Differently from other proposals, our intersection approach only requires a *limited* and *configurable* amount of memory. To speedup the intersecting task, ParDCI reuses most of the intersections previously done, by caching them in a fixed-size buffer for future use. Moreover, ParDCI adopts several heuristic strategies to adapt its behavior to the features of datasets processed. For example, when a dataset is dense, the sections of tidlists which turn out to be identical are aggregated and clustered in order to reduce the number of intersections actually performed. Conversely, when a dataset is sparse, the runs of zero bits in the intersected tidlists are promptly identified and skipped. More details on these strategies can be found in [12].

ParDCI implementation is explicitly targeted towards the efficient use of clusters of SMP nodes, an emerging computing platform. Inter-Node parallelism exploits the MPI communication library, while Intra-Node parallelism uses multithreading and out-of-core techniques in order to effectively exploit the memory hierarchies of each SMP node. To validate our proposal we conducted several experiments on a cluster of three dual-processor PCs running linux. Different synthetic datasets and various support thresholds were used in order to test ParDCI under different conditions. The results were very encouraging since the performances obtained were nearly optimal.

This paper is organized as follows. Section 2 introduces FSC parallelization techniques and discusses same related work. In Section 3 we describe ParDCI in depth, while Section 4 presents and discusses the results of the experiments conducted. Finally in Section 5 we draw future works and some conclusions.

2 Related Work

Several parallel algorithms for solving the FSC problem have been proposed in the last years [2,8]. Zaki authored a survey on ARM algorithms and relative parallelization schemas [15]. Most proposals can be considered parallelizations of the well-known *Apriori* algorithm. Agrawal *et al.* in [2] proposes a broad taxonomy of the parallelization strategies that can be adopted for *Apriori* on distributed-memory architectures. These strategies, summarized in the following, constitute a wide spectrum of trade–offs between computation, communication, memory usage, synchronization, and exploitation of problem–domain knowledge.

The *Count Distribution* strategy follows a data-parallel paradigm according to which the transaction database is statically partitioned among the processing nodes, while the candidate set C_k is replicated. At each iteration every node counts the occurrences of candidate itemsets within the local database partition. At the end of the counting phase, the replicated counters are aggregated, and every node builds the same set of frequent itemsets F_k. On the basis of the global knowledge of F_k, candidate set C_{k+1} for the next iteration is then built. Inter-Node communication is minimized at the price of carrying out redundant computations in parallel.

The *Data Distribution* strategies attempts to utilize the aggregate main memory of the whole parallel system. Not only the transaction database, but also the candidate set C_k are partitioned in order to permit both kinds of partitions to fit into the main memory of each node. Processing nodes are arranged in a logical ring topology to exchange database partitions, since every node has to count the occurrences of its own candidate itemsets within the transactions of the whole database. Once all database partitions have been processed by each node, every node identifies the locally frequent itemsets and broadcasts them to all the other nodes in order to allow them to build the same set C_{k+1}. This approach clearly maximizes the use of node aggregate memory, but requires a very high communication bandwidth to transfer the whole dataset through the ring at each iteration.

The last strategies, *Candidate Distribution*, exploits problem–domain knowledge in order to partition both the database and the candidate set in a way that allows each processor to proceed independently. The rationale of the approach is to identify, as execution progresses, disjoint partitions of candidates supported by (possibly overlapping) subsets of different transactions. Candidates are subdivided on the basis of their prefixes. This trick is possible because candidates, frequent itemsets, and transactions, are stored in lexicographical order. Depending from the resulting candidate partitioning schema, the approach may suffer from poor load balancing. The parallelization technique is however very interesting. Once the partitioning schema for both C_k and F_k is decided, the approach does not involve further communications/synchronizations among the nodes.

The results of the experiments described in [2] demonstrate that algorithms based on Count Distribution exhibits optimal scale-up and excellent speedup, thus outperforming the other strategies. Data Distribution resulted the worst approach, while the algorithm based on Candidate Distribution obtained good performances but paid a high overhead caused by the need of redistributing the dataset.

3 ParDCI

During its initial *counting*-based phase, ParDCI exploits a *horizontal* database with variable length records. During this phase, ParDCI trims the transaction database as execution progresses. In particular, a pruned dataset $\mathcal{D}_{k+1}$ is written to the disk at each iteration k, and employed at the next iteration [11]. Dataset pruning is based on several criteria. The main criterium states that transactions that do not contain any frequent k-itemset will not surely contain larger frequent itemsets and can thus be removed from $\mathcal{D}_{k+1}$. Pruning entails a reduction in I/O activity as the algorithm progresses, since the size of $\mathcal{D}_k$ is always smaller than the size of $\mathcal{D}_{k-1}$. However, the main benefits come from the reduced computation required for subset counting at each iteration k, due to the reduced number and size of transactions. As soon as the pruned dataset becomes small enough to fit into the main memory, ParDCI adaptively changes its behavior, builds a *vertical-layout* in-core database, and adopts the intersection-based approach to determine larger frequent sets. Note, however, that ParDCI continues to have a level-wise behavior, so that the search space for finding frequent sets is still traversed breadth-first [10].

ParDCI uses an *Apriori*-like technique to generate C_k starting from F_{k-1}. Each candidate k-itemset is generated as the union of a pair of F_{k-1} itemsets sharing a common $(k-2)$-prefix. Since itemsets in F_{k-1} are lexicographically ordered, the various pairs occur in close positions within F_{k-1}, and ParDCI can generate candidates by exploiting high spatial and temporal locality. Only during the counting-based phase, C_k is pruned by checking whether all the other $(k-1)$-subsets of a candidate k-itemset are frequent, i.e. are included in F_{k-1}. Conversely, during the intersection-based phase, since our intersection method is able to quickly determine the support of a candidate itemsets, we found more profitable to avoid this check. As soon a candidate k-itemset is generated, ParDCI determines on-the-fly its support by intersecting the corresponding tidlists. As a consequence, while during its counting-based phase ParDCI has to maintain C_k in main memory to search candidates and increment associated counters, this is no longer needed during its intersection-based phase. This is an important improvement over other FSC algorithms, which suffer from the possible huge memory requirements due to the explosion of the size of C_k[9].

In designing ParDCI we exploited effective out-of-core techniques, so that the algorithm is able to adapt its behavior to the characteristics of the dataset and of the underlying architecture. The efficient exploitation of memory hierarchies received particular attention: datasets are read/written in blocks, to take advantage of I/O prefetching and system pipelining [4]; at each iteration the frequent set is written to a file which is then `mmap`-ped into memory in order to access it for candidate generation during the next iteration.

In the following we describe the different parallelization techniques used in the *counting*- and *intersection*-based phases of ParDCI. In both phases we have to further distinguish between the *Intra-Node* and the *Inter-Node* level of parallelism exploitation. At the Inter-Node level we used the message–passing paradigm through the MPI communication library, while in the Intra-Node level we exploited multi-threading through the *Posix Thread* library. A *Count Distribution* approach is adopted to parallelize the *counting*-based phase, while the *intersection*-based phase exploits a very effective and original implementation of the *Candidate Distribution* approach [2].

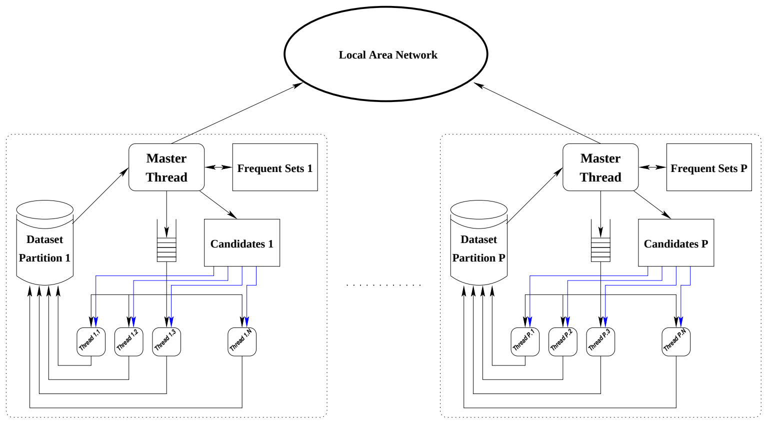


Figure1. ParDCI: threads and processes interaction schema.

3.1 The counting-based phase

A *Count Distribution* approach was adopted for the *counting*-based phase of ParDCI. Since the counting-based approach is used only for a few iterations (in all the experiments conducted ParDCI starts using intersections at the third or fourth iteration), in the following we only sketch the main features of the counting method adopted (interested readers can refer to [11]). In the first iteration, as all FSC algorithms, ParDCI directly counts the occurrences of items within all the transactions. For $k \geq 2$, instead of using complex data structures like hash-trees or prefix-trees, ParDCI uses a novel *Direct Count technique* that can be thought as a generalization of the technique used for $k = 1$. The technique uses a *prefix table*, $\mathrm{PREFIX}_k[\,]$, of size $\binom{m_k}{2}$, where m_k is the number of different items contained in the pruned dataset $\mathcal{D}_k$. In particular, each entry of $\mathrm{PREFIX}_k[\,]$ is associated with a distinct *ordered prefix* of two items. For $k = 2$, $\mathrm{PREFIX}_k[\,]$ directly contains the counters associated with the various candidate 2-itemsets, while, for $k > 2$, each entry of $\mathrm{PREFIX}_k[\,]$ points to the contiguous section of ordered candidates in C_k sharing the associated prefix. To permit the various entries of $\mathrm{PREFIX}_k[\,]$ to be directly accessed, we devised an order preserving, minimal perfect hash function. This prefix table is thus used to count the support of candidates in C_k as follows. For each transaction $t = \{t_1, \ldots, t_{|t|}\}$, we select all the possible 2-prefixes of all k-subsets included in t. We then exploit $\mathrm{PREFIX}_k[\,]$ to find the sections of C_k which must be visited in order to check set-inclusion of candidates in transaction t.

At the Inter-Node level, candidate set C_k is replicated, while the transaction database is statically split in a number of partitions equal to the number of SMP nodes available. Every SMP node at each iteration performs a scan of the whole local dataset partition. When the occurrence of a candidate itemset $c \in C_k$ is discovered in a transaction, the counter associated with c is incremented. At the end of the counting step, the counters computed by every node are aggregated (via a MPI_Allreduce operation). Each node then produces the same set F_k and generates the same candidate set C_{k+1} employed at the next iteration. These operations are however inexpensive, and their duplication does not degrade performances.
As depicted in Figure 1, at the Intra-Node level each node uses a pool of threads. They have the task of checking in parallel each of the candidate itemset against chunks of transactions of the local dataset partition. The task of subdividing the local dataset in disjoint chunks is assigned to a particular thread, the *Master Thread*. It loops reading blocks of transactions and forwarding them to the *Worker Threads* executing the counting task. To overlap computation with I/O, minimize synchronization, and avoid data copying overheads, we used an optimized producer/consumer schema for the cooperation among the Master and Worker threads. A prod/cons buffer, which is logically divided into *Npos* sections, is shared between the Master (producer) and the Workers (consumers). We also have two queues of pointers to the various buffer positions: a *Writable Queue*, which contains pointers to free buffer positions, and a *Readable Queue*, which contains pointers to buffer positions that have been filled by the Master with transactions read from the database. The operations that modify the two queues (to be performed in critical sections) are very fast, and regard the attachment/detachment of pointers to the various buffer positions. The Master thread detaches a reference to a free section of the buffer from the *Writable*

Queue, and uses that section to read a block of transactions from disk. When reading is completed, the Master inserts the reference to the buffer section into the (initially empty) *Readable Queue*. Symmetrically, each Worker thread self–schedules its work by extracting a reference to a chunk of transactions from the *Readable Queue*, and by counting the occurrences of candidate itemsets within such transactions. While the transactions are processed, the Worker also performs transaction pruning, and uses the same buffer section to store pruned transactions to be written to $\mathcal{D}_{k+1}$. At the end of the counting step relative to the current chunk of transactions, the worker writes the transactions to disk and reinserts the reference to the buffer section into the *Writable Queue*. When all transactions (belonging to the partition of $\mathcal{D}_k$) have been processed, each Master thread performs a local reduction operation over the various copies of counters (reduction at the Intra-Node level), before performing via MPI the global counter reduction operation with all the other Master threads running on the other nodes (reduction at the Inter-Node level). Finally, to complete the iteration of the algorithm, each Master thread generates F_k, writes this set to the local disk, and generates C_{k+1}.

3.2 The intersection-based phase

When the size of the pruned dataset $\mathcal{D}_k$ becomes small enough to fit into the main memory of all nodes, ParDCI changes its behavior, and starts the *intersection*-based phase, by first transforming the dataset from horizontal into vertical. Tidlists in the vertical dataset are actually represented as bit-vectors, where the bit i is set within tidlist j if transaction i contains item j. This representation enhances locality exploitation, and allows intersections to be efficiently performed by using simple Boolean *and* instructions.

In order to speedup tidlist intersection, ParDCI stores and reuses most of the intersections previously done by caching them in a fixed-size buffer. In particular, it uses a small “cache” buffer to store all the intermediate intersections that have been computed to determine the support of $c \in C_k$, where c is the last evaluated candidate. The cache buffer used is a simple bi-dimensional bit-array $Cache[\,][\,]$, where the bit vector $Cache[j][\,]$, $2 \leq j \leq (k-1)$ is used to store the results of the intersections relative to the first j items of c. Since candidate itemsets are generated in lexicographic order, with high probability two candidates consecutively generated, say c and c', share a common prefix. Suppose that c and c' share a prefix of length $h \geq 2$. When we consider c' to determine its support, we can save work by reusing the intermediate result stored in $Cache[h][]$. Even if, in the worst case, the tidlists corresponding to all the k items included in a candidate k-itemset have to be intersected (k-way intersection), our caching method is able to strongly reduce the number of intersections actually performed [12].

During the intersection-based phase, an effective Candidate Distribution approach is adopted at both the Inter and Intra-Node levels. This parallelization schema makes the parallel nodes completely independent: communications and synchronization are no longer needed for all the following iterations of ParDCI.

Let us first consider the Inter-node level, and suppose that the intersection-based phase is started at iteration $\overline{k}+1$. Therefore, at iteration $\overline{k}$ the various nodes build the bit-vectors representing their own portions of the vertical in-core dataset. The construction of the vertical dataset is carried out on-the-fly, while transactions are read from the horizontal dataset for subset counting. The partial vectors are then broadcast to obtain a complete replication of the whole vertical dataset on each node. The frequent set $F_{\overline{k}}$ (i.e., the set computed in the last counting-based iteration) is then statically partitioned among the SMP nodes by exploiting problem-domain knowledge, thus entailing a Candidate Distribution schema for all the following iterations. Partitioning is done in a way that allows each processing node P^i to generate a unique C_k^i $(k > \overline{k})$ independently of all the other nodes, where $C_k^i \cap C_k^j = \emptyset, i \neq j$, and $\bigcup_i C_k^i = C_k$. To this end $F_{\overline{k}}$ is partitioned as follows. First, it is split into l *sections*, on the basis of the prefixes of the lexicographically ordered frequent itemsets included. All the frequent $\overline{k}$-itemsets that share the same $\overline{k}-1$ prefix (i.e. those itemsets whose first $\overline{k}-1$ items are identical) are assigned to the same section. Since ParDCI builds each candidate $(\overline{k}+1)$-itemsets as the union of two $\overline{k}$-itemsets sharing the first $\overline{k}$ items, we are sure that each candidate $\overline{k}$-itemset can be independently generated starting from one of the l disjoint sections of $F_{\overline{k}}$. The various partitions of $F_{\overline{k}}$ are then created by assigning the l sections to the various processing nodes, by adopting a simple greedy strategy that considers the number of itemsets contained in each section in order to build well-balanced partitions. From our tests, this policy suffices for balancing the workload at the Inter-Node level. Once completed the partitioning of $F_{\overline{k}}$, nodes independently generate the associated candidates and determine their support by intersecting the corresponding tidlists of the replicated vertical dataset. Finally they produce disjoint partitions of $F_{\overline{k}+1}$. Nodes continue to work according to the schema above also for the following iterations. It is worth noting that, although at iteration $\overline{k}$ the whole vertical dataset is replicated on all the nodes, as the execution progresses, the implemented pruning techniques trim the vertical dataset in a different way on each node.

At the Intra-Node level, the same Candidate Distribution parallelization schema is employed, but at a finer granularity and by exploiting a dynamic scheduling strategy to balance the load among the threads. In particular, at each iteration k of the intersection-based phase, the Master thread initially splits the local partition of F_{k-1} into x sections, $x >> t$, where t is the total number of active threads. This subdivision entails a partitioning of the candidates generated on the basis of these sections (Candidate Distribution). The information to identify every section S_i are inserted in a shared queue. Once this initialization is completed, also the Master thread becomes a Worker. Thereinafter, each Worker thread loops and self-schedules its work by performing the following steps:

1. access in mutual exclusion the queue and extract information to get S_i, i.e. a section of the local partition of F_{k-1}. If the queue is empty, write F_k to disk and start a new iteration.

2. generate a new candidate k-itemset c from S_i. If it is not possible to generate further candidates, go to step 1.
3. compute on-the-fly the support of c by intersecting the vectors associated with the k items of c. In order to reuse effectively previous work, each thread exploits a private cache for storing the partial results of intersections. If c turns out to be frequent, put c into F_k. Go to step 2.

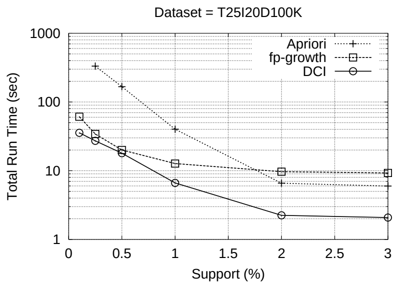
The various sections S_i are not created on the basis of prefixes. So, since in order to generate candidates we have to pick pairs of frequent itemsets, once selected a section S_i only the first element of each pair must belong to S_i, while the second element must be searched in the following elements of the local partition of F_k.

4 Experimental Evaluation

ParDCI is the parallel version of DCI, a very fast sequential FSC algorithm previously proposed by the same authors [12]. In order to show both the efficiency of the counting method exploited during early iterations, and the effectiveness of the intersection-based approach used when the pruned vertical dataset fits into the main memory, we report the result of some tests where we compared DCI performances with those of two other FSC algorithms: FP-growth, currently considered one of the fastest algorithm for FSC[1], and the Christian Borgelt's implementation of *Apriori*[2]. For these tests we used publicly available datasets, and a Windows-NT workstation equipped with a Pentium II 350 MHz processor, 256 MB of RAM memory and a SCSI-2 disk. The datasets used are the `connect-4` dense dataset, which contains many frequent itemsets also for high support thresholds, and the T25I20D100K synthetic dataset[3].

Figure 2 reports the total execution times of *Apriori*, FP-growth, and DCI on these two datasets as a function of the support threshold s. As it can be seen, DCI significantly outperforms both FP-growth and *Apriori*. The efficiency of our approach is also highlighted by the plots reported in Figure 3, which show per iteration execution times for DCI and *Apriori* on dataset T25I20D100K with support thresholds equal to 0.3 and 1. As it can be seen DCI significantly outperforms *Apriori* in every iteration but the second one due to the additional time spent by DCI to build the vertical dataset used for the following intersection-based iterations. Complete performance tests and comparisons of DCI are discussed in depth in [12]. In that work we analyzed benchmark datasets characterized by different features, thus permitting us to state that the design of DCI is not focused on specific datasets, and that our optimizations are not over-fitted only to the features of these datasets.

[1] We acknowledge Prof. Jiawei Han for kindly providing us the last version of FP-growth, significantly optimized with respect to the one used for the tests reported in [9].
[2] `http://fuzzy.cs.uni-magdeburg.de/∼borgelt`
[3] `http://www.almaden.ibm.com/cs/quest.`



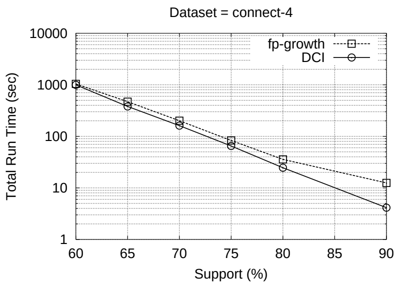
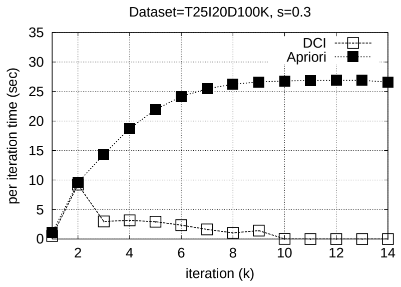


Figure2. Total execution times for DCI, *Apriori*, and FP-growth on datasets T25I20D100K and connect-4 as a function of the support threshold.



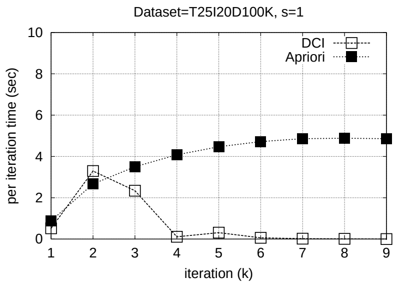
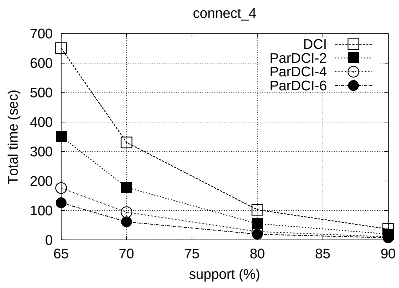


Figure3. Per iteration execution times for DCI and *Apriori* on dataset T25I20D100K with support thresholds 0.3 and 1.

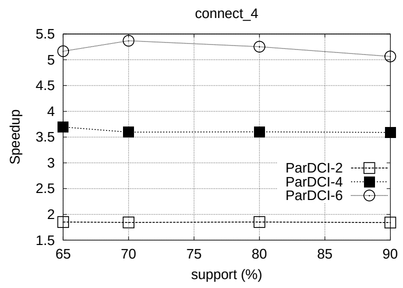
For what regards parallelism exploitation, we report an experimental evaluation of ParDCI on a Linux cluster of three two-way SMPs, for a total of six processors. Each SMP is equipped with two Pentium II 233MHz, 256 MB of main memory, and a SCSI disk.

First we compared the performance of DCI and ParDCI on the dense dataset `connect-4`. Figure 4 plots total execution times and speedups (nearly optimal ones) as functions of the support thresholds s (%). ParDCI-2 corresponds to the pure multithread version running on a single 2-way SMP, while ParDCI-4 and ParDCI-6 also exploit inter-node parallelism, and run, respectively, on two and three 2-way SMPs. Note that, in order to avoid exponential explosion in the number of frequent itemsets, we used relatively high supports for the tests with the dense `connect-4` dataset.

Figure 5 plots the speedups obtained on three synthetic datasets for two fixed support thresholds ($s = 1.5\%$ and $s = 5\%$), as a function of the number of processors used. The datasets used in these tests were all characterized by an average transaction length of 50 items, a total number of distinct items of 1000, and a size of the maximal potentially frequent itemset of 32. We only varied the total number of transactions from 500K to 3000K, so we can identify them on the basis of their number of transactions. It is important to remark that since our cluster is composed of 2-way SMPs, we mapped tasks on processors always



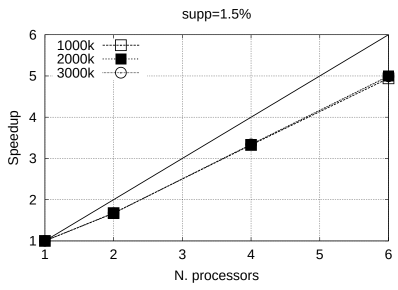
(a)



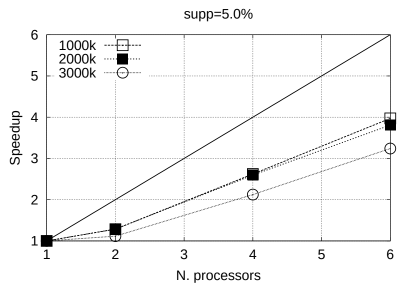
(b)

Figure4. Dense dataset `connect-4`: completion times of DCI and ParDCI (a) and speedups of ParDCI (b), varying the minimum support threshold.

using the minimum number of nodes (e.g., when we use 4 processors, we actually use 2 SMP nodes). This implies that experiments performed on either 1 or 2 processors actually have the same memory and disk resources available, whereas the execution on 4 processors benefits from double amount of such resources. According to these experiments, ParDCI shows a quasi linear speedup. As it can be seen by considering the results obtained with one or two processors, the slope of the speedup curve turns out to be relatively worse than its theoretical limit, due to resource sharing and thread implementation overheads at the Inter-Node level. Nevertheless, when several nodes are employed, the slope of the curve improves. For all the three datasets, when we fix $s = 5\%$, we obtain a very small number of frequent itemsets. As a consequence, the CPU-time decreases, and becomes relatively smaller than I/O and interprocess communication times.



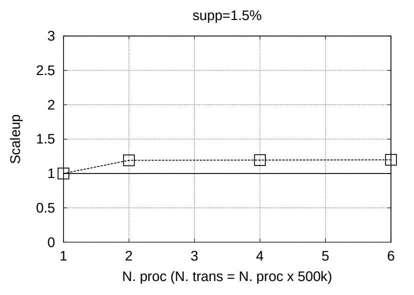
(a)



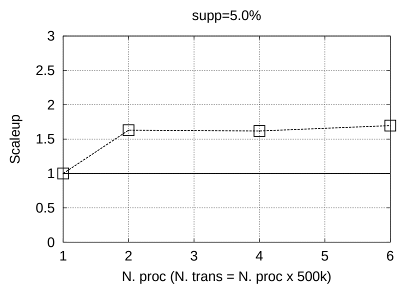
(b)

Figure5. Speedup for the three datasets `1000K` `2000K` and `3000K` with $support = 1.5\%$ (a) and $support = 5\%$(b).

Figure 6 plots the scaleup, i.e. the relative execution times measured by varying, at the same time, the number of processors and the dataset size. We can observe that the scaling behavior remains constant, although slightly above one. This is again due to thread management overheads and resource sharing (mainly disk sharing).



(a)



(b)

Figure6. Scaling behavior obtained varying the dataset size together with the number of processors for *support* = 1.5% (a) and *support* = 5%(b).

The strategies adopted for partitioning the dataset and the candidates well balanced the workload among the processing nodes. In all the tests conducted on our homogeneous cluster of SMPs used as a dedicated resource, the differences in the completion time between the fastest and the slowest processing were always lower than the 1% of the total execution time. Clearly, since we used a static partitioning strategy at the inter-node level, in the case of a heterogeneous pool of computational resources, or a non-dedicated environment, a different partitioning strategy should be necessary.

5 Conclusions and Future Works

Originally used as a *Market Baskets* analysis tool, ARM is today used in various fields such as Web Mining, where it is adopted to discover the correlations among the various pages visited by users, Web Searching, where association rules can be used to build a statistical thesaurus or to design intelligent caching policies. Due to the impressive growth rate of data repositories, only efficient parallel algorithms can grant the needed scalability of ARM solutions.

ParDCI originates from DCI, a very fast sequential FSC algorithm previously proposed [12]. Independently of the dataset peculiarities, DCI outperforms not only *Apriori*, but also FP-growth [9], which is currently considered one of the fastest algorithm for FSC. ParDCI, the multithreaded and distributed version of DCI, due to a number of optimizations and to the resulting effective exploitation of the underlying architecture, exhibits excellent Scaleup and Speedup under a variety of conditions. Our implementation of the Count and Candidate Distribution parallelization approaches at both Inter and Intra-Node levels resulted to be very effective with respect to main issues such as load balancing and communication overheads. In the near future we plan to extend ParDCI with adaptive *work stealing* policies aimed to efficiently exploit heterogeneous/grid environments. To share our efforts with the data mining community, we made DCI and ParDCI binary codes available for research purposes[4].

References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast Discovery of Association Rules in Large Databases. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, 1996.
2. R. Agrawal and J. C. Shafer. Parallel Mining of Association Rules. *IEEE Transaction On Knowledge And Data Engineering*, 8:962–969, 1996.
3. R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proc. of the 20th VLDB Conf.*, pages 487–499, 1994.
4. R. Baraglia, D. Laforenza, S. Orlando, P. Palmerini, and R. Perego. Implementation Issues in the Design of I/O Intensive Data Mining Applications on Clusters of Workstations. In *Proc. of the 3rd Work. on High Performance Data Mining, (IPDPS-2000), Cancun, Mexico*, pages 350–357. LNCS 1800 Spinger-Verlag, 2000.
5. R. J. Bayardo. Efficiently Mining Long Patterns from Databases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 85–93, 1998.
6. U.M. Fayyad, G. Piatetsky-Shapiro, P. Smith, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining.* AAAI Press, 1998.
7. V. Ganti, J. Gehrke, and R. Ramakrishnan. Mining Very Large Databases. *IEEE Computer*, 32(8):38–45, 1999.
8. E. H. Han, G. Karypis, and Kumar V. Scalable Parallel Data Mining for Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):337–352, May/June 2000.
9. J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 1–12, Dallas, Texas, USA, 2000.
10. Hipp, J. and Güntzer, U. and Nakhaeizadeh, G. Algorithms for Association Rule Mining – A General Survey and Comparison. *SIGKDD Explorations*, 2(1):58–64, June 2000.
11. S. Orlando, P. Palmerini, and R. Perego. Enhancing the Apriori Algorithm for Frequent Set Counting. In *Proc. of the 3^{rd} Int. Conf. on Data Warehousing and Knowledge Discovery, LNCS 2114*, pages 71–82, Germany, 2001.
12. S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and Resource-Aware Mining of Frequent Sets. In *Proc. of the 2002 IEEE Int. Conference on Data Mining (ICDM'02)*, Maebashi City, Japan, Dec. 2002.
13. J. S. Park, M.-S. Chen, and P. S. Yu. An Effective Hash Based Algorithm for Mining Association Rules. In *Proc. of the 1995 ACM SIGMOD Int. Conf. on Management of Data*, pages 175–186, 1995.
14. A. Savasere, E. Omiecinski, and S. B. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. In *Proc. of the 21th VLDB Conf.*, pages 432–444, Zurich, Switzerland, 1995.

[4] Interested readers can download the binary codes at address `http://www.miles.cnuce.cnr.it/~palmeri/datam/DCI`

15. M. J. Zaki. Parallel and Distributed Association Mining: A Survey. *IEEE Concurrency*, 7(4):14–25, 1999.
16. M. J. Zaki. Scalable Algorithms for Association Mining. *IEEE Transactions on Knowledge and Data Engineering*, 12:372–390, May/June 2000.