A Scalable Multi-Strategy 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

In this paper we present DCI, a new data mining algorithm for frequent set counting. We also discuss in depth the parallelization strategies used in the design of ParDCI, the distributed and multi-threaded algorithm derived from DCI. Multiple heuristics strategies are adopted within DCI, so that the algorithm is able to adapt its behavior not only to the features of the specific computing platform, but also to the features of the dataset being processed. Our approach turned out to be highly scalable and very efficient for mining both short and long patterns present in real and synthetically generated datasets. The experimental results showed that DCI outperforms others previously proposed algorithms under a variety of conditions. ParDCI, the parallel version of DCI, is explicitly devised for targeting clusters of SMP nodes: shared memory and message passing paradigms were used at intra- and inter-node level, respectively. Due to the broad similarity between DCI and *Apriori*, we were able to adapt effective parallelization strategies previously proposed for *Apriori*.

Contact author information: Salvatore Orlando, Dipartimento di Informatica, Università Ca' Foscari di Venezia, - Via Torino 155, 30172 Venezia Mestre - Italy, Phone : +39 041 2348428, Fax : +39 041 2348419, e-mail: orlando@dsi.unive.it

1 Introduction

Association Rule Mining (ARM), one of the most popular topic in the KDD field [3, 10, 11, 18], regards the extractions of association rules from a database of transactions $\mathcal{D}$, where each 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. In this paper we are interested in the most computationally expensive phase of ARM, i.e the Frequent Set Counting (*FSC*) one. During this phase, the set $\mathcal{F} = \bigcup F_k$, including all the *frequent* k-itemsets, is built, where an itemset of k items (k-itemset) is frequent if its support is greater than a fixed 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 $\mathcal{D}$).

We introduce DCI (Direct Count & Intersect), a new algorithm to solve the FSC problem. We also discuss a parallel version of DCI, called ParDCI, which is explicitly targeted for clusters of SMPs. As *Apriori*, DCI builds at each iteration the set F_k of the frequent k-itemsets on the basis of a set of candidate itemsets C_k. However, DCI adopts a hybrid approach to determine the support of the candidate itemsets. In particular:

- during the first iterations DCI, exploits a novel *counting–based* technique, accompanied by a carefully pruning of the dataset, stored on disk in horizontal form. During this phase DCI counts how many times candidate k-itemsets occur in transactions of dataset $\mathcal{D}$;
- during the following iterations, DCI adopts a very efficient *intersection–based* technique. Hence, in this phase DCI determines the support of each itemset $c \in C_k$ by intersecting the tidlists associated with the k items of c. DCI starts using this technique as soon as the pruned dataset, whose layout has to be transformed from horizontal into vertical, fits into the main memory of the specific host machine. Tidlists are represented as bit-vectors.

DCI is able to adapt its behavior not only to the features of the specific computing platform, but also to the features of datasets processed. This ability of DCI is very important, since in the past many novel algorithms were devised, but often they outperformed others only for specific datasets. To adapt its behavior to the dataset peculiarities, DCI dynamically choose between different *heuristic strategies*. 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.

We will show how the sequential implementation of DCI outperforms previously proposed algorithms. In particular, under a number of different tests and independently of the dataset peculiarities, DCI results much better that FP–Growth [14], which is currently considered as one of the fastest algorithm for FSC. In particular, DCI performs very well with both synthetic and real datasets characterized by different density

features, e.g. datasets from which, due to the different correlations among items, either short or long frequent patterns can be mined.

ParDCI, which has been explicitly designed to target clusters of SMPs, adopts different parallelization strategies during the two phases of DCI, i.e. the counting-based and the intersection-based ones. Moreover, these strategies are slightly differentiated with respect to the two type of parallelism exploited: *inter-node* level within each SMP to exploit shared-memory cooperation, and *intra-node* levels among distinct SMPs, where message-passing cooperation is used. In particular, the MPI library is employed for inter-node parallelism, while the `pthread` library is used for intra-node parallelism.

Basically, at the inter-node level (coarse grain, message-passing) we use a *Count Distribution* technique during the counting-based phase, and a *Candidate Distribution* one during the intersection-based one [5]. The former technique requires the partitioning of dataset ($\mathcal{D}$), and the replication of candidates (C_k) and associated counters. The final values of the counters are derived by all-reducing the various local counters. The latter technique is used during the intersection-based phase. It requires an intelligent partitioning of C_k based on the prefixes of itemsets, but a partial/complete replication of the dataset $\mathcal{D}$.

At inter-node level, dataset and candidate distributions are decided once, and holds for the following iterations of the algorithm phase. During the counting-based phase of ParDCI we adopt a static distribution of $\mathcal{D}$, decided at loading time on the basis of the relative power on computational nodes. This decision holds until the algorithm switches to the next phase. This usually occurs after a few iterations. During the intersection-based phase, the intelligent distribution of C_k can also be decided on dynamic knowledge about the features of nodes involved, e.g., on the basis of dynamically collected information regarding workloads of nodes.

As regards intra-node level (fine grain, shared memory), we continue to use the same parallelization techniques adopted at the inter-node level. The main difference regards the dynamic load-balancing policy through which the various pieces of work are scheduled to threads. At this level, in fact, we have a logical shared queue from which the various thread picks and self-schedules pieces of work. When the *Count Distribution* technique is adopted, a piece of work corresponds to a block of transactions belonging the dataset partition assigned to the corresponding node. When the *Candidate Distribution* technique is adopted, a piece of work corresponds to a block of candidates, belonging to the specific candidate partition.

This paper is organized as follow. Section 2 shows some related results obtained in previous works on sequential and parallel FSC algorithms. In Section 3 we discuss the DCI algorithm while Section 4 sketches the solutions adopted to design ParDCI. In Section 5 we present a performance analysis based on theoretical results, and report our experimental results. Finally in Section 6 we present our conclusions and future works.

2 Related Works

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 $\mathcal{D}$. A way to prune $\mathcal{P}(M)$ is to restrict the search to itemsets whose subsets are all frequent. The *Apriori* algorithm [6] exactly exploits this pruning technique, and thus adopts a breadth-first visit of $\mathcal{P}(M)$ for counting itemset supports. Other algorithms [8, 2] adopt instead a depth-first visit of $\mathcal{P}(M)$. The goal in this case is to discover long frequent itemsets first, thus saving the work needed for discovering frequent itemsets included in long ones.

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 the supports of the various itemsets present in $\mathcal{P}(M)$: a *counting*-based [4, 6, 12, 17, 8, 1] and an *intersection*-based [19, 9, 23] one. The former one, also adopted by *Apriori*, exploits a *horizontal* dataset and *counts* how many times each candidate k-itemset occurs in every transaction. The latter method, on the other hand, exploits a *vertical* dataset, where a *tidlist* (list of transaction ids) is associated with each item.

The *counting*-based approach is quite efficient from the point of view of memory occupation, since only requires to maintain in main memory C_k along with data structures used to quickly access candidates (e.g. hash-trees or prefix-trees). On the other hand, the *intersection*-based method may be much more computational effective than its *counting*-based counterpart [19]. Unfortunately, to exploit efficient 2-way intersections, memory requirement increases since we need to buffer tidlists associated with previously computed frequent $(k-1)$-itemsets.

Other FSC algorithms that only mine the *maximal frequent itemsets* (e.g. *MaxEclat, MaxClique, Max-Miner*) [8, 23] have been proposed. In particular, the Max-Miner [8] algorithm, which has been explicitly devised to work well for problems characterized by long patterns, aims to find maximal frequent sets by looking ahead throughout the search space. It uses clever lower bound techniques to determine whether an itemset is frequent, without accessing the database and actually counting its support. Unfortunately while it is simple to derive all the frequent itemsets from the maximal ones, the same does not hold for their supports, which require a further counting step. Remember that the exact supports of all the frequent itemsets are needed to easily compute association rule confidences and other measures of rule interest.

The FP-growth algorithm [15] is an innovative and very fast algorithm that is able to efficiently find frequent itemsets without generating candidate sets. It builds in memory a compact representation of the dataset, where repeated patterns are represented once along with the associated repetition counters. The data structure used to store the dataset is called *frequent pattern tree*, or FP-tree for short. The algorithm recursively identifying tree paths which share a common prefix. These paths are intersected by considering the associated counters. The algorithm can exactly compute the support of all the frequent patterns discovered. A problem that has been recognized for FP-growth is the need to maintain the tree

in memory. Solutions based on a partition of the database, in order to permit problem scalability, are also illustrated. FP-growth is currently considered one of the fastest algorithms for solving the FSC problem. From our tests, FP-growth resulted particularly effective in mining dense datasets, which can efficiently be compacted in memory, and datasets for which *Apriori*-like algorithms cannot prune enough candidates from C_k, so that F_k turn out to be much smaller that C_k.

Another important topic of research in association mining regards the exploitation of *constraints* [21, 20] over rules to be generated. Using these constraints an algorithm becomes faster, since it avoid generating all the frequent itemsets characterized by a given uniform minimum support. In this paper we have only shown how our algorithm is able to generate efficiently the frequent itemsets using the *classic constraint* over minimum support, but we are confident about the possibility of introducing into DCI further constraints, for example in the candidate generation phase.

A number of parallel algorithms for solving the FSC have been proposed in the last years. Most of them can be considered parallelizations of the well-known *Apriori* algorithm [5, 13, 13]. Zaki authored a good survey on ARM algorithms and relative parallelization schemas [22]. In [5] Agrawal *et al.* proposed a broad taxonomy of the parallelization strategies that can be adopted for *Apriori*. The described approaches constitute a wide spectrum of trade–offs between computation, communication, memory usage, synchronization, and the use of problem–specific information:

- The *Count Distribution* approach follows a data-parallel according to which the only transaction database is statically partitioned among the processing nodes, while the candidate set C_k and associated counters are replicated. At the end of each 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.

- The *Data Distribution* approach 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. While this approach clearly maximizes the use of node aggregate memory, it requires large communications to transmit dataset partitions among nodes for subset counting purpose.

- The last approach, *Candidate Distribution*, incorporates problem–domain knowledge to partition both the data 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, subset of candidates supported by (possibly overlapping) subsets of different transactions. Candidates and relative frequent itemsets are subdivided on the basis of common prefixes shared by the itemsets within each partition. This trick is possible because candidates, frequent itemsets, and transactions, all contain lexicographically ordered item identifiers. From candidate partitioning also a dataset partitioning with limited replication

is derived. Even if the approach may suffer from poor load balancing, it is however very interesting. Once the partition schema for both C_k and F_k is decided, it in fact does not require further communications/synchronizations among the nodes.

The results of the experiments described in [5] 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 due to the need of redistributing the dataset.

3 The DCI algorithm

During its initial *counting*-based phase, DCI exploits a *horizontal* layout database with variable length records. DCI also 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. Some of the dataset pruning techniques were inspired by DHP [17]. Pruning the dataset may entail a reduction in I/O activity as the algorithm progresses, but 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, DCI adaptively changes its behavior, builds a *vertical* layout database in-core, and starts adopting an intersection-based approach to determine frequent sets. Note, however, that DCI continues to have a *Apriori*-like level-wise behavior.

DCI uses an *Apriori*-like technique to generate the candidate set C_k from F_{k-1}, by exploiting the lexicographic order of F_{k-1} to find pairs of $(k-1)$-itemsets sharing a common $(k-2)$-prefix. Due to this order, in fact, the various pairs occur in close positions within F_{k-1}, which can thus be read with high spatial and temporal locality. Only during the DCI counting-phase, C_k is further pruned by checking whether all the other subsets of a candidate are also 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 further pruning step over C_k, thus also preserving locality in accessing F_{k-1}.

While during its counting-based phase DCI has to maintain C_k in main memory to search candidates and increment associated counters, this is no longer needed during the intersection-based phase. As soon a candidate k-itemset is generated, DCI determines on-the-fly its support by intersecting the corresponding tidlists. This is an important improvement over other *Apriori*-like algorithms, which suffer from the possible huge memory requirements due to the explosion of the C_k size.

DCI makes use of a large body of out-of-core techniques, so that DCI is able to adapt its behavior also to machines with limited main memory. Datasets are read/written in blocks, to take advantage of I/O prefetching and system pipelining [7]. The outputs of the algorithms, e.g. the various frequent sets F_k, are written to files. These same files are then `mmap`-ped into memory in order to access them during the next

iteration for candidate generation. Note also ParDCI is able to adopts similar out-of-core techniques, since SMP nodes are all equipped with local disks.

3.1 Counting-based phase

The techniques used in this phase of DCI has already described in detail in [16], where the same authors proposed DCP (Direct Count & Pruning), an algorithm especially effective for miming short patterns. It is worth considering that DCI adopts this technique only for the first iterations, when very short patterns are mined: in all our experiments, DCI can start its intersection-based phase at the third or fourth iteration. In the following we only sketch the main features of the counting method adopted.

DCI exploits for its counting-based phase a simple, *directly accessible* data structure, which is an alternative to other more complex ones, like hash-trees or prefix-trees. In particular, for $k \geq 2$ DCI uses a *Direct Count technique*, which is based on a generalization of the technique adopted for $k = 1$. For $k = 1$, in fact, all *Apriori*-like algorithms simply exploits a vector of counters, which are "directly addressable" through item identifiers.

The novel counting technique of DCI is accompanied by an effective and simple pruning technique of $\mathcal{D}$. This pruning technique trims and removes transactions, and also reduces the cardinality of M_k ($\overline{m_k} = |M_k|$), i.e. the set of items that still appear in some transactions of $\mathcal{D}_k$.

The *Direct Count technique* employed by DCI uses a *prefix table*, PREFIX$_k$[ ], of size $\binom{\overline{m}_k}{2}$, which is re-initialized at each iteration. In particular, each location of PREFIX$_k$[ ] is associated with a distinct *ordered prefix* of two items belonging to M_k, and contains the pointer to the "first" candidate in C_k (and associated counters) characterized by this prefix. Note that itemsets in C_k are stored in lexicographically order, so that itemsets characterized by the same prefix appear in consecutive positions within C_k. Of course, for $k = 2$, PREFIX$_2$[ ] directly contains the counters associated to the various candidate 2-itemsets. To permit the various locations of 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 $t = \{t_1, \ldots, t_{|t|}\}$, select all the possible 2-prefixes of all k-subsets included in t;
- through PREFIX$_k$[ ], select the contiguous sections of C_k to be visited in order to check set-inclusion of candidates in transaction t. For $k = 2$, use PREFIX$_2$[ ] to directly access associated counters.

3.2 Intersection-based phase

Since the counting-based approach is not efficient for long patterns, DCI starts its intersection-based phase as soon as possible. Unfortunately, the intersection-based method needs to maintain in memory the vertical representation of the pruned dataset. So, at iteration k, $k \geq 2$, DCI checks whether the pruned dataset $\mathcal{D}_k$ may fit into main memory. When the dataset becomes small enough, its vertical in-core representation is

built on the fly, while the transactions are read and counted against C_k. The *intersection*-based method thus starts at the next iteration.

The vertical layout of the dataset is based on fixed length records (tidlists), stored as *bit-vectors*. The whole vertical dataset can thus be seen as a bidimensional bit-array $\mathcal{VD}[\,][\,]$, whose rows correspond to the bit-vectors associated with a *not pruned item* in M_k. In particular, given M_k ($\overline{m}_k = |M_k|$), the set of the not pruned items, and T_k ($\overline{n}_k = |T_k|$), the set of not pruned transactions, then the amount of memory required to store $\mathcal{VD}[\,][\,]$ is $\overline{m}_k \times \overline{n}_k$ bits. Starting from the iteration following the vertical database construction, DCI uses $\mathcal{VD}[\,][\,]$ to determine the support of candidate itemsets.

First, we give a simplified description of the intersection-based phase of DCI. At each iteration, DCI produces F_k as follows:

1. for each $c \in C_k$, *and*-intersect the k bit-vectors associated with the items included in c. Consider that a bit-vector intersection can be carried out very efficiently and with high spatial locality by using primitive Boolean *and* instructions with word operands. As previously stated, this method does not require C_k to be kept in memory: we can compute the support of each candidate c on-the-fly, as soon as it is generated;

2. compute the support of c by counting the 1's present in the resulting bit-vector. If this number is $\geq minsup$, write c to F_k.

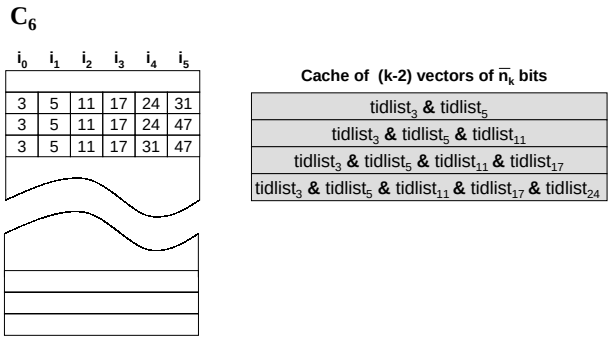
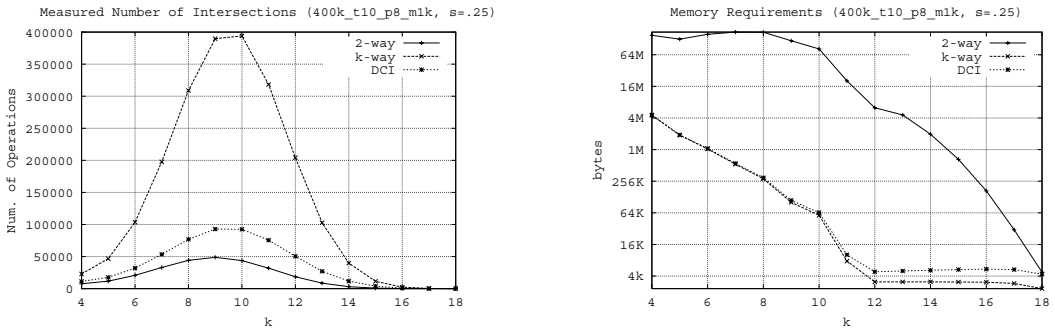


Figure 1: Example of DCI intersection caching for $k = 6$.

The simplified strategy above is highly inefficient, because it always needs a k-way intersection to determine a the support of each candidate c. Conversely, if we had enough memory to maintain the tidlists (bit-vectors) associated with all the frequent $(k-1)$-itemsets in F_{k-1}, we could carry out the same computation through 2-way intersection. Unfortunately, a pure 2-way intersection approach does not scale, due to the huge amount of memory required. Nevertheless, an effective caching policy could be exploited

in order to save work and speed up our k-way intersection method. DCI uses a small "cache" buffer to store all the intermediate intersections that have been computed to determine the support $c \in C_k$, where c is the last candidate that has been evaluated. The cache buffer used is a simple bidimensional bit-array $Cache[\,][\,]$ of size $(k-2) \cdot \overline{n}_k$, where the bit vector $cache[j][1 : \overline{n}_k]$, $2 \leq j \leq (k-1)$ is used to store the results of the intermediate intersections relative to the first j items of c. Since itemsets in C_k are generated in lexicographic order, with high probability two consecutive candidates, e.g. c and c', share a common prefix. Suppose that c and c' share a prefix of length $h \geq 2$. When we consider c', we can save intersection work by reusing the intermediate result stored in $cache[h][1 : \overline{n}_k]$. Figure 1 shows an example of use of the cache for $k = 6$. C_6 contains candidate itemsets $c = \{3, 5, 11, 17, 24, 31\}, c' = \{3, 5, 11, 17, 24, 47\}$, and $c'' = \{3, 5, 11, 17, 31, 47\}$. Thus, since c' shares a prefix of length 5 with c, in order to compute c' support, DCI only intersects $cache[5][1 : \overline{n}_k]$ with the tidlist associated with item 47. Analogously, when itemset c'' is processed, the intersections of the tidlists of its first 4 items are found in $cache[4][1 : \overline{n}_k]$.

To evaluate the effectiveness of our caching policy, we counted the actual number of intersections carried out by DCI on the synthetic dataset $400k\_t10\_p8\_m1k$, whose description is reported in Section 5. We compared this number with the best and the worst case. The former corresponds to the adoption of a 2-way intersection approach, which is only possible if we can fully cache the tidlists associated with all the frequent $(k-1)$-itemsets in F_{k-1}. The latter case regards the adoption of a pure k-way intersection method, i.e. a method that does not exploit caching at all. Figure 2.(a) plots the results of this analysis for support threshold $s = 0.25\%$. The caching policy of DCI turns out to be very effective, since the actual number of intersections performed results to be very close to the best case. Moreover, memory requirements for the three approaches are plotted in Figure 2.(b). As expected, DCI requires orders of magnitude less memory than a pure 2-way intersection approach, thus better exploiting memory hierarchies.



(a) (b)

Figure 2: Per iteration number of tidlist intersections performed (a), and memory requirements (b), for DCI, and the pure 2-way and k-way intersection-based approaches.

Others heuristics techniques are used within DCI to further reduce intersection costs. More specifically, two different optimization technique are exploited for *sparse* and *dense* datasets. Consider that dense datasets give rise to very long patterns even for large minimum support constraints.

In order to apply the right optimization, the vertical dataset is tested for checking its density as soon as it is built. To this end we simply compare the bit vectors associated with the *most frequent items* by using simple and inexpensive logical operation (*xor*, *not*, *and*). Note that we only consider the most frequent items since they occur in a large body of candidates, so that the corresponding bit-vectors need to be intersected multiple times. If a large part of these bit-vectors turns out to be identical, we deduce that the dataset is dense and adopt a specific heuristics which exploits similarities between these vectors. Otherwise the technique for sparse datasets is adopted.

- **Sparse datasets.** Sparse or moderately dense datasets originate bit-vectors with long runs of 0's. To speedup computation, while we compute the intersection of the bit vectors relative to the first two items c_1 and c_2 of a generic candidate itemset $c = \{c_1, c_2, \ldots, c_k\} \in C_k$, we also identify and maintain information about the the runs of 0's appearing in the resulting bit vector stored in the entry $Cache[2][1 : \overline{n}_k]$[1]. Further intersections that are needed to complete the processing of c (as well as intersections needed to compute other itemsets characterized by the same 2-item prefix) will skip these runs of 0's, so that only vector segments which may contain 1's are actually intersected. Since information about the runs of 0's are computed only once, and the same information is reused many times, this optimization results to be very effective.

  Moreover, sparse and moderately dense datasets offer the possibility of further pruning vertical datasets as computation progresses. The benefits of pruning regard the reduction in the length of the bit vectors and thus in the cost of intersections. Note that a transaction, i.e. a column of $\mathcal{VD}$, can be removed from the vertical dataset when it does not contain any of the itemsets included in F_k. This check can simply be done by *or*-ing the intersection bit-vectors computed by DCI for all the k-itemsets added to F_k. However, we observed that dataset pruning is expensive, since vectors must be compacted at the level of single bits. Hence DCI prunes the dataset only if turns out to be profitable, i.e. if we can obtain a large reduction in the vector length, and the number of vectors to be compacted is small with respect to the cardinality of C_k. Note that a relatively large cardinality of C_k entails an expensive iteration step, due to the repeated intersections of bit-vectors associated with the various items.

- **Dense datasets.** If the dataset turns out to be dense, we expect to deal with a dataset characterized by strong correlation among the most frequent items. This means not only that the bit-vectors associated with the *most frequent items* are characterized by long runs of 1's intermixed by few 0's, but also that they turn out to be very similar. Note that the presence of long runs of 1's discourages

[1] We maintain information about sections composed of zeroed memory words.

strategies that try to further prune the dataset, as the one adopted for sparse datasets.

Experimentally we verified that the DCI heuristic technique for dense dataset is advantageous only if the correlation among frequent items is very strong, i.e. when these items are at least the 30% of all the items in M_k, and they are associated with bit-vectors that are identical at the 80%.

The heuristic technique adopted by DCI for dense dataset cases works as follows:

- reorder the columns of the vertical dataset, in order to move identical segments of the bit vectors associated with the most frequent items to the first consecutive positions;
- since each candidate likely includes several of these most frequent items, avoid repeatedly intersecting the identical segments of the corresponding vectors. This technique may save a lot of work because (1) the intersection of an identical vector segment is done only once, (2) the identical segments are usually very large (80% or more), and (3) long candidate itemset likely contains several of these most frequent items.

4 ParDCI

In the following we describe the different parallelization techniques used in the *counting*- and *intersection*-based phases of ParDCI, the parallel version of the DCI algorithm discussed above. Since we exploited a cluster of SMP nodes, in both phases we will distinguish between an *intra-node* and an *inter-node* level of parallelism. At the inter-node level we used the message–passing paradigm through the MPI communication library, while at 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 *Candidate Distribution* approach [5].

4.1 The counting-based phase

At the inter-node level, the dataset is statically split in a number of partitions equal to the number of SMP nodes available. The sizes of partitions depend on the relative powers of nodes. At each iteration k, a copy of the whole candidate set C_k is generated independently by each nodes. Then each node p reads blocks of transactions its own dataset partition $\mathcal{D}_{p,k}$ stored on local disk, performs subset counting, prunes transactions read and appends them to $\mathcal{D}_{p,k+1}$. At the end of an iteration, an all-reduce (MPI-based) operation has to take place to update candidate counters. Finally, all nodes produce F_k, on the basis of which they will generate C_{k+1} at the next iteration. The replicated activities are relatively inexpensive, and their duplication does not jeopardize performances.

As depicted in Figure 3, 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 read from $\mathcal{D}_{p,k}$. The

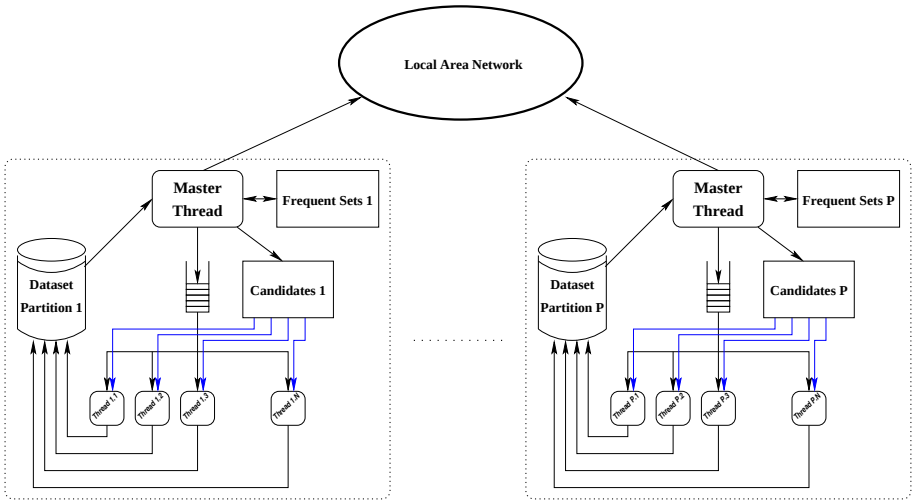


Figure 3: ParDCI: threads and processes interaction schema.

task of subdividing the local dataset into 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 *Queue of Empty positions* (*QE*), which contains pointers to free buffer positions, and a *Queue of Filled positions* (*QF*), which contains pointers to buffer positions that have been filled by the Master with blocks of transactions read from $\mathcal{D}_{p,k}$. The operations to 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 *QE*, and uses that section to read a block of transactions from disk. When the reading is completed, the Master inserts the reference to the buffer section into the (initially empty) *QF*. Symmetrically, each Worker thread self–schedules its work by extracting a reference to a chunk of transactions from the *QF*, and 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 $\mathcal{D}_{p,k+1}$. Once all transactions have been processed, the Worker writes to disk the block of pruned transactions, and reinserts the reference to the buffer section into the *QE*.

In order to avoid contention problems in updating counters associated with candidates, each Worker thread accesses and increment a logically private copy of the counters. When all transactions in $\mathcal{D}_{p,k}$ have been processed by a node p, the corresponding 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}.

4.2 The intersection-based phase

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: inter-node communications 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 in-core portions of the vertical dataset. The construction of the vertical dataset is carried on-the-fly by each node p, while transaction are read from $\mathcal{D}_{p,\overline{k}}$ for subset counting. Before starting the intersection-base phase, the partial vertical datasets are 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 by exploiting problem-domain knowledge. A disjoint partition $F_{p,\overline{k}+1}$ of $F_{\overline{k}+1}$ is thus assigned to each node p, where $\bigcup_p F_{p,\overline{k}+1} = F_{\overline{k}+1}$. It is worth remarking that this partitioning entails a Candidate Distribution schema for all the following iterations, according to which each node p will be able to generate a unique C_k^p $(k > \overline{k})$ independently of all the other nodes, where $C_k^p \cap C_k^{p'} = \emptyset$, $p \neq p'$, and $\bigcup_p C_k^p = C_k$.

$F_{\overline{k}}$ 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 are assigned to the same section. Since ParDCI, as all other *Apriori*-like algorithms, builds each candidate $(k+1)$-itemsets as the union of two k-itemsets sharing the first $k-1$ items, we are sure that each candidate k-itemset can independently be generated starting from one of the l disjoint sections of $F_{\overline{k}}$. Then the various partitions $F_{p,\overline{k}+1}$ are created by assigning round-robin the l sections to the various processing nodes. From our tests, this round-robin 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 supports by intersecting the corresponding tidlists of the replicated vertical dataset. Nodes continue to work according to the schema above also for the following iterations. Finally, note that, although at iteration $\overline{k}$ the whole vertical dataset is replicated on all the nodes, as the execution progresses, the implemented pruning technique may trim the vertical dataset in a different way on each node. The progressive reduction of dataset size introduces obvious benefits on the exploitation of memory hierarchies.

At the intra-node level, a Candidate Distribution parallelization schema is still employed, but at a finer level and by using dynamic scheduling to ensure load balancing. In particular, at each iteration k of the intersection-based phase, the Master thread of a node p initially splits the local partition of $F_{p,k-1}$ into x

segments, $x >> t$, where t is the total number of active threads. Like the inter-node level, this subdivision entails a partitioning of the candidates generated on the basis of these segments (Candidate Distribution). It is worth remarking that, unlike the inter-node level, segments of $F_{p,k-1}$ are not created on the basis of frequent itemset prefixes. We observed, in fact, that at this level the prefix method might produce only a few segments with unbalanced features. Therefore segments are created to contain about $\frac{|F_{p,k-1}|}{x}$ frequent itemsets. The information to identify the x segments are then inserted in a shared queue. Once the shared queue is initialized, 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. one segment of the local partition of $F_{p,k-1}$. If the queue is empty, go to step 4.

2. generate a new candidate k-itemset c from S_i. If it is not possible to generate further candidates, go to step 1. Note that in order to generate candidates, the thread cannot limit its visit of $F_{p,k-1}$ to segment S_i. Remember, in fact, that each candidates has to be generated from a pair of frequent $k-1$-itemsets sharing a common $(k-2)$ prefix, and that $F_{p,k-1}$ itemset are stored in lexicographic order. Thus, while the first element of is chosen from S_i, the second element of the pair has to be searched in the following position of $F_{p,k-1}$, even outside S_i. Therefore, the selection of segment S_i only forces the choice of the possible first element of the various itemset pairs.

3. compute on-the-fly the support of c by intersecting the vectors associated to the k items of c. Each thread exploits a private intersection cache to reuse previous work. If c turns out to be frequent, put c into $F_{p,k}$. Go to step 2.

4. write $F_{p,k}$ to disk and start new iteration.

5 Experimental Results

The sequential algorithm, DCI, is currently available in two versions, a MS-Windows one, for which we used the Visual Studio compiler suite, and a Linux one, for which we used the GNU compiler. ParDCI, which exploits the MPICH MPI and the *pthread* libraries, is currently available only for the Linux platform.

We used the MS-Windows version of DCI to compare its performance with other FSC algorithms. For test comparisons we used the FP-growth algorithm, currently considered on of the fastest algorithm for FSC[2], and the Christian Borgelt's implementation of *Apriori* [3].

[2]We acknowledge Prof. Jiawei Han for kindly providing us the latest binary version of FP-growth. This version of FP-growth was sensible optimized with respect to the one used for the tests reported in [14].

[3]`http://fuzzy.cs.uni-magdeburg.de/~borgelt`

For the DCI tests we used a Windows-NT workstation equipped with a Pentium II 350 MHz processor, 256 MB of RAM memory and a SCSI-2 disk. For testing ParDCI performance, we employed a small cluster of three Pentium II 233MHz two-way SMPs, for a total of six processors. Each SMP is equipped with 256 MBytes of main memory and a 3 GBytes SCSI disk.

We used both synthetic and real datasets by varying the minimum support threshold s. The synthetic datasets were created with one of the most commonly adopted synthetic dataset generator[4]. Other datasets, including real-world and dense ones, were instead downloaded from the Web. The characteristics of the datasets used are reported in Table 1.

Table 1: Datasets used in the experiments.

Dataset	**Description**
T25I10D10K	1K items and 10K transactions. The average transaction size is 25 and the average maximal potentially frequent itemset size is 10. Synthetic dataset available at `http://www.cs.sfu.ca/~peijian/personal/publications/T25I10D10k.dat.gz`
T25I20D100K	10K items and 100K transactions. The average transaction size is 25 and the average maximal potentially frequent itemset size is 20. Synthetic dataset available at `http://www.cs.sfu.ca/~peijian/personal/publications/T25I20D100k.dat.gz`
400k_t10_p8_m10k	10K items and 400K transactions. The average transaction size is 10 and the average maximal potentially frequent itemset size is 8. Synthetic dataset created with the IBM dataset generator [6].
400k_t30_p16_m1k	1K items and 400K transactions. The average transaction size is 30 and average maximal potentially frequent itemset size 16. Dataset size is about 50MB. Synthetic dataset created with the IBM dataset generator [6].
t20_p8_m1k	With this notation we identify a series of synthetic datasets characterized by 1K items, average transaction size 20 and average maximal potentially frequent itemset size 8. We vary the number of transactions for scaling measurements, while all the other parameters are kept constant.
t50_p32_m1k	A series of three datasets sharing the same number of items (1K), average transaction size (50 items) and average maximal potentially frequent itemset size (32). We used three such datasets with 1000K, 2000K and 3000K transactions.
connect-4	Dense dataset with 130 items and about 60K transactions. The maximal transaction size is 45 items. The dataset size is about 12MB. Available at `http://www.cs.sfu.ca/~wangk/ucidata/dataset/connect-4/connect-4.data`
BMS-WebView-1	497 items and 59K transactions containing click-stream data from an e-commerce web site. Each transaction is a web session consisting of all the product detail pages viewed in that session. Available at `http://www.ecn.purdue.edu/KDDCUP/data/BMS-WebView-1.dat.gz`

Computational cost of intersection-based phase. We analyzed the advantages of adopting the intersection-based approach, which is used by DCI after a few iterations, over the exploitation of the counting-based approach for all the iterations of the algorithm. The computational costs of each counting-based iteration

[4] `http://www.almaden.ibm.com/cs/quest.`

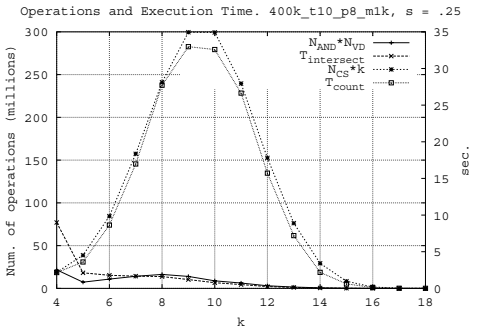


Figure 4: Theoretical and measured computational costs of intersection-based and counting-based iterations on dataset 400k_t10_p8_m1k ($s = 0.25\%$). Times $T_{intersect}$ and T_{count} are measured in seconds according to the scale on the right hand y axis. Millions of operations are instead reported on the left hand y axis.

is dominated by subset counting. Due to our 2-item prefix table, which allows us to select a section of C_k with a common prefix, at most $k-2$ comparisons are necessary in order to check whether a given candidate k-itemset appears in a transaction t or not. Hence, the number of operations performed at iteration k is approximately:

$$T_{count} = O(N_{CS} \cdot k) \tag{1}$$

where N_{CS} is the total number of candidates actually visited for counting the supports of all the transactions in $\mathcal{D}_k$. On the other hand, the computational cost of each intersection-based iteration is proportional to the number of *and* operations needed to determine the supports of all candidate itemsets. The number of *and* depends on both the average length of tidlists and the number of candidate itemsets. Therefore, the number of operations actually performed by DCI at iteration k is approximatively:

$$T_{intersect} = O(N_{AND} \cdot N_{VD}) \tag{2}$$

where N_{AND} is the total number of tidlist pairs actually intersected, while N_{VD} is the average number of operations needed for *and*-ing a pair of tidlists. In principle we can say that N_{VD} depends on the average length of tidlists, but we have to consider that DCI exploits several optimizations aimed to reducing the number of operations actually performed (see Section 3.2).

This simple analysis is confirmed by our experimental evaluation. In Figure 4 the measured per-iteration execution times, i.e. T_{count} and $T_{intersect}$, are plotted against their analytic estimates above, i.e. Equations (1) and (2), as a function of the iteration index k. The dataset considered was 400k_t10_p8_m1k, a sparse dataset, where the minimum support was fixed to $s = 0.25\%$. The actual values of N_{CS}, N_{AND} and N_{VD} were determined by profiling execution. These results shows the effectiveness of our intersection-based method and related optimization strategies versus its counting-based counterpart.

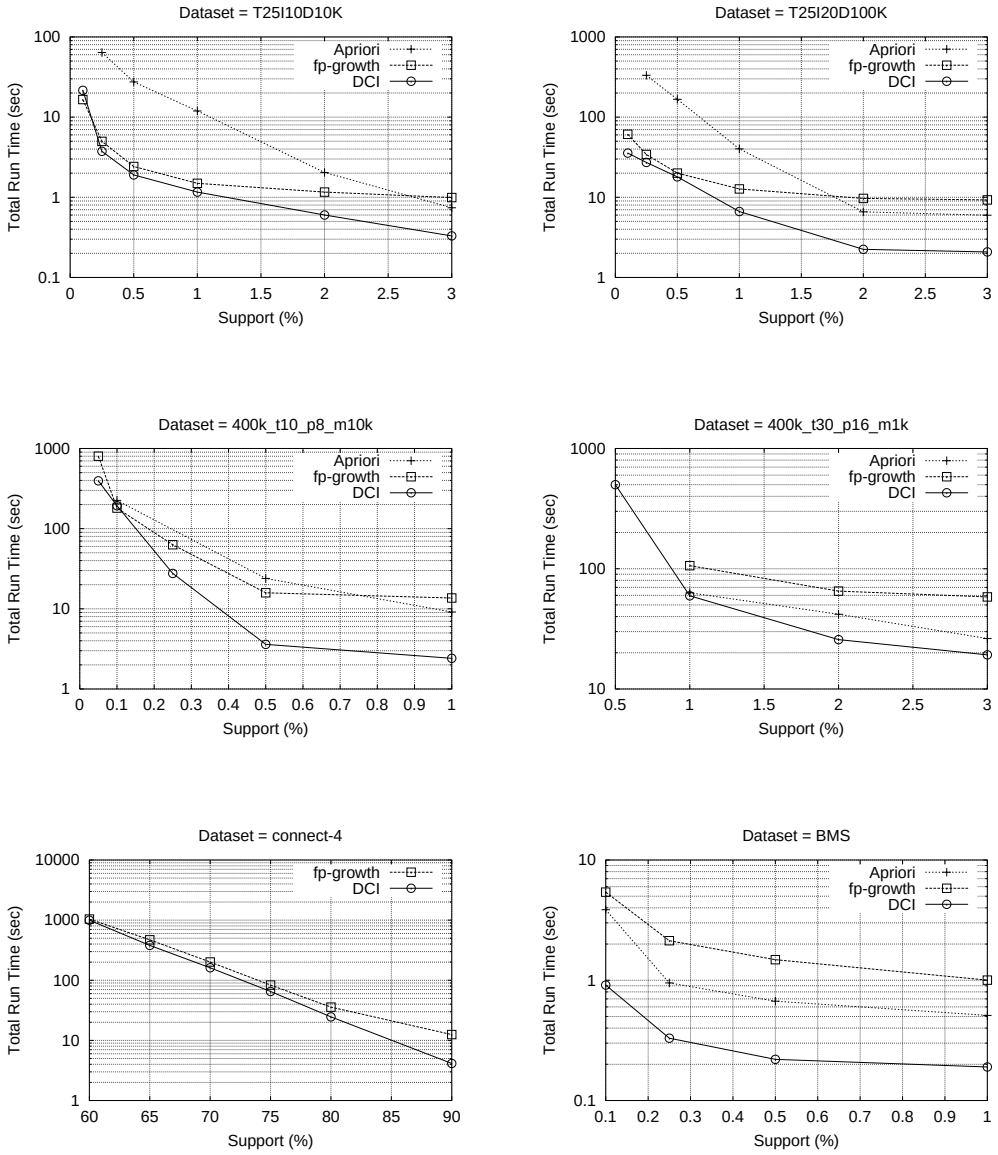


Figure 5: Total execution times for DCI, *Apriori*, and FP-growth on various datasets as a function of the support value.

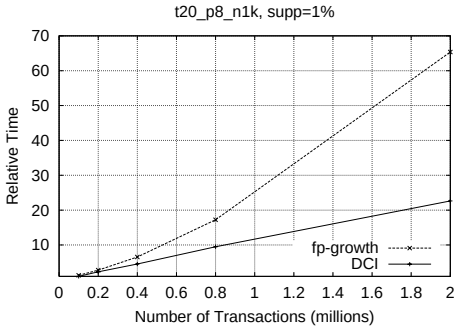


Figure 6: Relative execution times on datasets in the series t20_p8_m1k ($s = 1\%$) when varying the number of transactions (from $100K$ to $2M$.

DCI **performances and comparisons.** Figure 5 reports the total execution times obtained running *Apriori*, FP-growth, and DCI on the datasets described in Table 1 as a function of the support threshold s. In all the tests conducted, DCI outperforms FP-growth with speedups up to 8. Of course, DCI also remarkably outperforms *Apriori*, in some cases for one or more orders of magnitude. For connect-4, the dense dataset, the curve of *Apriori* is not shown, due to the relatively too long execution times. Note the for BMS, the real-world dataset, *Apriori* turns out to be better than FP-growth, accordingly to what discussed in [24].

The encouraging results obtained with DCI are due to 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. For only a dataset, namely T25I10D10K, FP-growth turns out to be better than DCI for $s = 0.1\%$. The cause of this behavior is the size of C_3, which in this specific case results much larger than the actual size of F_3. Hence, DCI has to carry out a lot of useless work to determine the support of many candidate itemsets, which will eventually result to be not frequent. In this case the FP-growth should be better than DCI since it does not require candidates generation.

We tested the scale-up behavior of DCI when the size of the dataset is increased. The various dataset samples have been obtained through the IBM generator, by keeping all the parameters constant except for the number of transactions. The samples belong to the series t20_p8_m1k. Figure 6 plots the execution times of FP-growth and DCI as a function of the number of transactions contained in the dataset processed, by keeping constant $s = 1\%$. The times reported are normalized with respect to the execution time of DCI on the smallest dataset sample of $100k$ transactions. DCI scale much better than FP-growth: it execution time curve corresponds to a quasi-liner function of the dataset size. For example, to process the dataset with 2 millions of transactions, DCI requires about 22 times the execution time spent on $100k$ transactions, while for FP-growth this ratio is about 65.

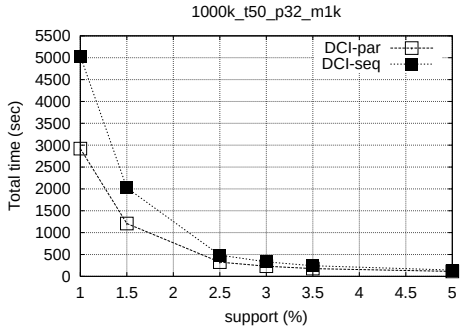
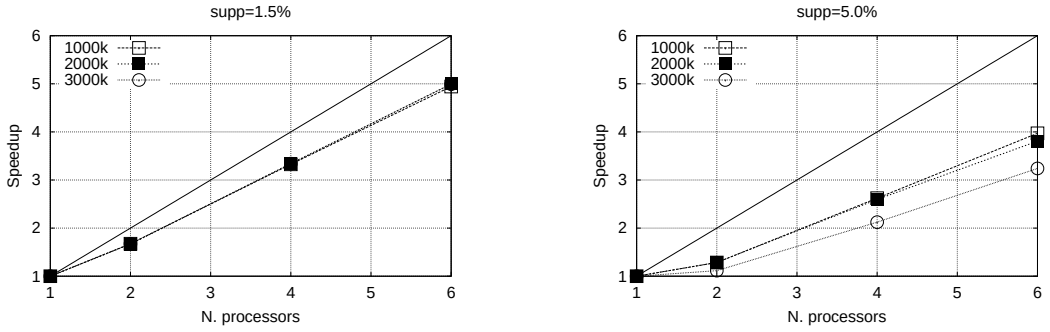


Figure 7: Sequential and multithreaded execution times for dataset 1000K as a function of s.

Performance evaluation of ParDCI. For these tests we used the synthetic dataset series identified as $t50\_p32\_m1k$ in Table 1. We varied the total number of transactions from 1000K to 3000K. In the following we will identify the various synthetic datasets on the basis of their number of transactions, i.e. 1000k, 2000k, and 3000k.

First we measured response time of ParDCI for the dataset 1000K. In this test we only used a single SMP node, so that we compared the sequential version (DCI) with the multi-threaded parallel version of ParDCI. Figure 7 plots the total execution times as a function of the support thresholds s (%). The reduction in the total execution time is quasi optimal (nearly optimal speedup) for support thresholds that involve expensive computations.



(a) (b)

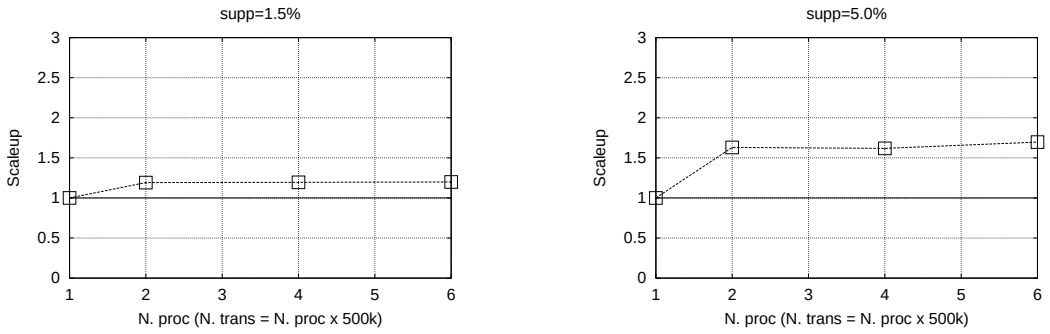
Figure 8: Speedup for datasets 1000K, 2000K and 3000K with $s = 1.5\%$ (a) and $s = 5\%$(b).

Figure 8 plots the speedups obtained on the three synthetic datasets for two fixed support thresholds ($s = 1.5\%$ and $s = 5\%$), as a function of number of processors used. Consider that, since our cluster is composed of three 2-way SMPs, we mapped tasks on processors always using the minimum number of nodes (e.g., when we used 4 processors, we actually employed 2 SMP nodes). This implies that experiments

performed on either 1 or 2 processors actually have identical memory and disk resources available, whereas the execution on 4 processors benefit from a double amount of such resources.

According to our experiments, ParDCI showed a quasi linear speedup. If you consider the results obtained with one or two processors, you note that 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 additional nodes are employed, the slope of the curve improves. For all the three datasets, when $s = 5\%$, a very small number of frequent itemsets is obtained. As a consequence, the CPU-time decreases, and becomes relatively smaller than I/O and also interprocess communication times.

Figure 9 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.



(a) (b)

Figure 9: Scaling behavior obtained varying the dataset size along with the processor number for $s = 1.5\%$ (a) and $s = 5\%$(b).

The strategies adopted for partitioning dataset and candidates on our homogeneous cluster of SMPs sufficed for balancing the workload. In out tests we observed very limited imbalance, below 0.5%. For targeting heterogeneous or non-dedicated clusters we plan to introduce in ParDCI more dynamic approaches to partitioning.

6 Conclusions and Future Works

DCI and ParDCI use a hybrid approach for extracting frequent patterns: a counting-based one during the first iterations and a very fast intersection-based one for the following iterations of the algorithm. One of the main innovative features of the two algorithms regards the ability to apply different heuristic strategies during the intersection-based phase, on the basis of the characteristics of a specific dataset. Different

techniques are thus used for dense and sparse datasets. These techniques are able to strongly reduce the complexity of intersections. Unlike other algorithms, such as maximal frequent set ones, even for dense datasets, from which very long patterns can be extracted, we are able to determine the exact support of frequent itemsets. Another important feature of DCI and ParDCI is the ability to adapt their behaviors to the features of the specific computing platform. For example, the intersection-based phase is started at iteration k only if the vertical layout representation of the dataset can be stored into main memory. Since our pruning technique strongly reduce dataset size as counting-based iterations progress, in our tests the optimized intersection-based phase always started at the third or fourth iteration.

As regards scalability of the approach, we have to consider that the in-core vertical dataset is created during the previous iteration $k-1$, i.e. the last iteration of the counting-based phase. During this iteration DCI needs enough memory space to store both the vertical "pruned" dataset and the candidate set C_{k-1}. Conversely, since ParDCI uses a Count Distribution approach for parallelization, the per-node memory requirement at iterations $k-1$ is reduced: each SMP node has only to build a vertical representation of its own partition of the pruned dataset. ParDCI then creates a complete in-core vertical dataset at the next iteration k, by joining the various partitions, so that nodes need enough memory to store a copy of the whole pruned dataset. Fortunately, during the intersection-based phase, candidates do not need to be kept in memory. DCI generates candidates by accessing with high locality the mmap-ped file containing F_{k-1}, while their supports are computed on-the-fly. In ParDCI, since a Candidate Distribution technique is adopted, each node need to produce and access (in the next iteration) a smaller partition $F_{p,k-1}$ of the whole F_{k-1}.

As a result of its optimized design, DCI significantly outperformed *Apriori* and FP-growth. For many datasets the performance improvement was impressive. The results were very good not only for synthetic datasets, when very low support thresholds are considered, but also for real-world datasets with medium/large support thresholds. The variety of datasets used and the large amount of tests conducted permit us to state that the design of DCI is not much focused on specific datasets, and that our optimizations are not over-fitted only to the features of these datasets [24].

ParDCI, the multi-threaded and distributed version of DCI, due to a number of optimizations and to the resulting effective exploitation of the underlying architecture, exhibited excellent scaleup and speedup under a variety of conditions. Our implementation of the Count and Candidate Distribution strategies for parallelization, used 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[5].

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

References

[1] R. C. Agarwal, C. C. Aggarwal, and V.V.V. Prasad. A Tree Projection Algorithm for Generation of Frequent Itemsets. *Journal of Parallel and Distributed Computing*. To appear.

[2] R. C. Agarwal, C. C. Aggarwal, and V.V.V. Prasad. Depth first generation of long patterns. In *Proc. of the 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 108–118, 2000.

[3] R. Agrawal, T. Imielinski, and A. Swami. Mining Association between Sets of Items in Massive Databases. In *ACM-SIGMOD 1993 Int'l Conf. on Management of Data*, pages 207–216. ACM, 1993. Washington D.C. USA.

[4] 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.

[5] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Transaction On Knowledge And Data Engineering*, 8:962–969, 1996.

[6] 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.

[7] 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.

[8] R. J. Bayardo Jr. Efficiently Mining Long Patterns from Databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 85–93, Seattle, Washington, USA, 1998.

[9] Brian Dunkel and Nandit Soparkar. Data organization and access for efficient data mining. In *Proc. of the 15th ICDE Int. Conf. on Data Engineering*, pages 522–529, Sydney, Australia, 1999. IEEE Computer Society.

[10] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smith, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI Press, 1998.

[11] V. Ganti, J. Gehrke, and R. Ramakrishnan. Mining Very Large Databases. *IEEE Computer*, 32(8):38–45, 1999.

[12] 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.

[13] Eui-Hong (Sam) Han, George Karypis, and Vipin Kumar. Scalable Parallel Data Mining for Association Rules. *IEEE Transaction on Knowledge and Data Engineering*, 12(3):337–352, may/june 2000.

[14] 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.

[15] 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.

[16] S. Orlando, P. Palmerini, and R. Perego. Enhancing the Apriori Algorithm for Frequent Set Counting. In *Proc. of the 3rd Int. Conf. on Data Warehousing and Knowledge Discovery, DaWaK 2001, LNCS 2114*, pages 71–82, Munich, Germany, 2001.

[17] 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.

[18] N. Ramakrishnan and A. Y. Grama. Data Mining: From Serendipity to Science. *IEEE Computer*, 32(8):34–37, 1999.

[19] 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.

[20] M. Seno and G. Karypis. Lpminer: An algorithm for finding frequent itemsets using length-decreasing support constraint. In *Proc. of the 1st IEEE Conf. on Data Mining*, 2001.

[21] K. Wang, Y. He, and J. Han. Mining Frequent Itemsets Using Support Constraints. In *Proc. of the 26th VLDB Conf.*, Cairo, Egypt, 2000.

[22] M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, 1999.

[23] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12:372–390, May/June 2000.

[24] Z. Zheng, R. Kohavi, and L. Mason. Real World Performance of Association Rule Algorithms. In *Proc. of KDD-2001*, 2001.