Adaptive and Resource-Aware Mining of 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

The performance of an algorithm that mines frequent sets from transactional databases may severely depend on the specific features of the data being analyzed. Moreover, some architectural characteristics of the computational platform used – e.g. the available main memory – can dramatically change the runtime behaviors of the algorithm. In this paper we present DCI (Direct Count & Intersect), an efficient data mining algorithm for discovering frequent sets from large databases, which effectively addresses the issues mentioned above. DCI adopts a classical level-wise approach based on candidate generation to extract frequent sets, but uses a hybrid method to determine candidate supports. The most innovative contribution of DCI relies on the multiple heuristics strategies employed, which permits DCI to adapt its behavior not only to the features of the specific computing platform, but also to the features of the dataset being mined, so that it results effective in mining both short and long patterns from sparse and dense datasets. The large amount of tests conducted permit us to state that DCI sensibly outperforms state-of-the-art algorithms for both synthetic and real-world datasets. Finally we also discuss the parallelization strategies adopted in the design of ParDCI, a distributed and multi-threaded implementation of DCI. ParDCI explicitly targets clusters of SMP nodes, so that shared memory and message passing paradigms are exploited at the intra- and inter-node levels, respectively. The adopted parallelization strategies result very effective and allow ParDCI to reach near optimal speedups.

1 Introduction

Association Rule Mining (ARM), one of the most popular topic in the KDD field [3, 11, 12, 19], regards the extractions of association rules from a database of transactions $\mathcal{D}$. 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 contains 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 of all the *frequent* itemsets is built. An itemset of k items (k-itemset) is frequent if its support is greater than the fixed threshold s, i.e. the itemset occurs in at least *minsup* transactions ($minsup = s/100 \cdot n$, where n is the number of transactions in $\mathcal{D}$).

In this paper we present DCI (Direct Count & Intersect), a new algorithm for solving the FSC problem. We also discuss the parallelization strategies used in the design of ParDCI, a distributed and multi-threaded implementation of DCI. We will show that DCI outperforms other state-of-the algorithms, and that ParDCI reaches near optimal speedups. Before introducing the innovative features of our algorithms and the original contributions of our paper, we survey the main related works in the field.

Related work. The computational complexity of the FSC problem derives from the exponential 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 visits breadth-first $\mathcal{P}(M)$ for counting itemset supports. At each iteration k, *Apriori* generates C_k, a set of candidate k-itemsets, and counts the occurrences of these candidates in the transactions. The candidates in C_k for which the the minimum support constraint holds are then inserted into F_k, i.e. the set of frequent k-itemsets, and the next iteration is started. Other algorithms adopt instead a depth-first visit of $\mathcal{P}(M)$ [9, 2]. In this case the goal is to discover long frequent itemsets first, thus saving the work needed for discovering frequent itemsets included in long ones. 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. Note that the exact knowledge of the supports of all the frequent itemsets is needed to derive association rule confidences and other measures of interest.

The FSC problem has been extensively studied in the last years. Several variations to the original

Apriori algorithm, as well as many parallel implementations, have been proposed. We can recognize two main methods for determining the supports of the various itemsets present in $\mathcal{P}(M)$: a *counting*-based approach [4, 6, 13, 17, 9, 1, 8], and an *intersection*-based one [20, 10, 22]. The former one, also adopted in *Apriori*, exploits a *horizontal* dataset and *counts* how many times each candidate k-itemset occurs in every transaction. When the number of candidates does not explode, the memory needs of this approach are limited, since only C_k, along with data structures used to quickly access the candidates (e.g. hash-trees or prefix-trees), are required to be kept in-core. The latter method, on the other hand, exploits a *vertical* dataset, where a *tidlist*, i.e. a list of transaction identifiers (tids), is associated with items (or itemsets), and itemset supports are determined through tidlist intersections. The support of a k-itemset c can thus be computed either by a *k-way* intersection, i.e. by intersecting the k tidlists associated with the k items included in c, or by a *2-way* intersection, i.e. by intersecting the tidlists associated with a pair of frequent $(k-1)$-itemsets whose union is equal to c. While the intersection-based approaches may be more computational effective than their counting-based counterparts [20], pro and cons of *2-way* and *k-way* methods must be evaluated. While the *2-way* methods involve less intersections than *k-way* ones, they require much more memory to buffer all the intersections corresponding to (previously computed) frequent $(k-1)$-itemsets. As discussed below, in our DCI algorithm we tried to find a tradeoff between the two methods: we use a *k-way* intersection method to reduce main memory usage, and exploit caching strategies to reduce the number of tidlist intersections actually carried out. On the other hand, in [20, 22] algorithms based on the *2-way* intersection method are proposed. They try to address the memory-related scalability issues of the *2-way* method by partitioning the dataset. While the *Partition* algorithm [20] relies on a blind subdivision of the dataset, Zaki's techniques [22] are based on a clever dataset partitioning method that relies on a lattice-theoretic approach for decomposing the search space. For example, in the Zaki's *Eclat* algorithm, the search space decomposition entails several subproblems, each concerned with finding all the frequent itemsets which share a common prefix. On the basis of a given common prefix is possible to determine a partition of the dataset, which will be composed of only those transactions which constitute the support of the prefix. By recursively applying the Eclat's search space decomposition we can thus obtain subproblems which can entirely fit into the main memory.

Recently another category of methods, i.e. the *pattern growth ones*, have been proposed [1, 15, 18]. FP-growth [15] is the best representant of this kind of algorithms. It is not based on candidate generation as *Apriori*, but 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 identifies tree paths which

share a common prefix. These paths are intersected by considering the associated counters. FP-growth is currently considered the fastest FSC algorithm for dense or moderately dense datasets, for which it is able to construct very compressed FP-trees. Note that dense datasets are exactly those for which an explosion in the number of candidates is usually observed. in this case FP-growth is more effective than *Apriori*-like algorithms, which need to generate and store in main memory a huge number of candidates for subset-counting. Unfortunately, FP-growth does not perform well on sparse datasets [23], so the same authors recently proposed a new pattern-growth algorithm, H-mine [18], based on an innovative hyper-structure that allows the in-core dataset to be recursively projected by selecting those transactions that include a given pattern prefix. Since H-mine works well only for sparse datasets, FP-growth, which generates physically projected datasets, should still be used for dense ones. The authors suggest to switch between the two algorithm on the basis of the dataset characteristics discovered at run-time.

Contribution of the paper. In this paper we discuss in depth DCI (Direct Count & Intersect), a new algorithm to solve the FSC problem. We also introduce a parallel version of DCI, called ParDCI, which is explicitly targeted for clusters of SMPs. Several considerations concerning the features of real datasets to be mined, the characteristics of modern hw/sw system, as well as scalability issues of DM algorithms have motivated the design of DCI:

- Transactional databases may have different peculiarities in terms of the correlations among items, so that they may result either dense or sparse. Hence, a desirable characteristic of a new algorithm should be the ability to adapt its behavior to these features. DCI, which supports this kind of adaptiveness, thus constitutes an innovation in the arena of previously proposed FSC algorithms, which often outperformed others only for specific datasets.

- Modern hw/sw systems need high locality for effectively exploiting memory hierarchies and achieving high performances. Large dynamic data structures with pointers may lack in locality due to unstructured memory references. Other sources of performance limitations may be unpredictable branches. DCI tries to take advantages of modern systems by using simple array data structures, accessed by tight loops which exhibit high spatial and temporal locality. In particular, DCI exploits such techniques for intersecting tidlists, which are actually represented as bit-vectors that can be intersected very efficiently with simple loops exploiting primitive bitwise *and* instructions. Another issue regards I/O operations, which must be carefully optimized in order to allow DM algorithms to efficiently man-

age large databases. Even if the disk-stored datasets to be mined may be very large, DM algorithms usually access them sequentially with high spatial locality, so that suitable out-of-core techniques to access them can be adopted, also taking advantage of prefetching and caching features of modern OSs [7]. DCI adopts these out-of-core techniques to access large databases, prunes them as execution progresses, and starts using more efficient in-core strategies as soon as possible.

- Scalability is the main concern in designing DM algorithms that aim to efficiently mine large databases. Therefore, it is important to be able to parallelize these algorithms without introducing large overheads. We will show how our DCI algorithm can be easily parallelized with near optimal speedups by exploiting techniques similar to those already proposed for *Apriori*.

Once motivated the design requirements of DCI, we can now detail how it works. As *Apriori*, at each iteration DCI builds the set F_k of the frequent k-itemsets on the basis of the set of candidates C_k. However, DCI adopts a hybrid approach to determine the support of the candidates. During its first iterations, DCI exploits a novel counting–based technique, accompanied by an effective pruning of the dataset, stored to disk in horizontal form. During the following iterations, DCI adopts a very efficient intersection–based technique. DCI starts using this technique as soon as the pruned dataset fits into the main memory.

As previously stated, DCI is able to adapt its behavior not only to the features of the specific computing platform, e.g. to the main memory available, but also to the features of the datasets processed. DCI deals with dataset peculiarities by dynamically choosing between distinct *heuristic strategies*. For example, when a dataset is dense, identical sections appearing in several bit-vectors 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 bit-vectors to be intersected are promptly identified and skipped.

We will show how the sequential implementation of DCI significantly outperforms previously proposed algorithms. In particular, under a number of different tests and independently of the dataset peculiarities, DCI results to be faster than *Apriori* and FP-growth. By comparing our experimental results with the published ones obtained on the same sparse dataset, we deduced that DCI is also faster than H-mine [18]. DCI performs very well on both synthetic and real-world datasets characterized by different density features, i.e. datasets from which, due to the different correlations among items, either short or long frequent patterns can be mined.

ParDCI, the parallel version of DCI, 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 levels of parallelism exploited: *intra-node* level within each SMP to exploit shared-memory cooperation, and *inter-node* level among distinct SMPs, where message-passing cooperation is used. Basically, ParDCI uses a *Count Distribution* technique during the counting-based phase, and a *Candidate Distribution* one during the intersection-based one [5, 14, 21]. The former technique requires the partitioning of the dataset, and the replication of candidates and associated counters. The final values of the counters are derived by all-reducing the various local counters. The latter technique is instead 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.

The rest of the paper is organized as follows. Section 2 describes the DCI algorithm and discusses the various adaptive heuristics adopted, while Section 3 sketches the solutions adopted to design ParDCI. In Section 4 we report our experimental results. Finally in Section 5 we present concluding remarks.

2 The DCI algorithm

During its initial counting-based phase, DCI exploits an out-of-core, *horizontal* database with variable length records. DCI, by exploiting effective pruning techniques inspired by DHP [17], trims the transaction database as execution progresses. In particular, a pruned dataset $\mathcal{D}_{k+1}$ is written to disk at each iteration k, and employed at the next iteration. Let m_k and n_k be the number of items and transactions that are included in the pruned dataset $\mathcal{D}_k$, where $m_k \geq m_{k+1}$ and $n_k \geq n_{k+1}$. 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 level-wise behavior.

At each iteration, DCI generates the candidate set C_k by finding all the pairs of $(k-1)$-itemsets that are included in F_{k-1} and share a common $(k-2)$-prefix. Since F_{k-1} is lexicographically ordered, the various pairs occur in close positions, and candidate generation is performed with high spatial and temporal locality. Only during the DCI counting-phase, C_k is further pruned by checking whether also all the other subsets of a candidate 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 itemset, we found much more profitable to avoid this further check.

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 as a candidate k-itemset is generated, DCI determines its support on-the-fly 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 size of C_k [15].

DCI makes use of a large body of out-of-core techniques, so that it 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 algorithm, e.g. the various frequent sets F_k, are written to files that are `mmap`-ped into memory during the next iteration for candidate generation.

2.1 Counting-based phase

The techniques used in the counting-based phase of DCI are detailed in [16], where the same authors proposed an effective algorithm for mining short patterns. Since the counting-based approach is used only for few iterations, in the following we only sketch the main features of the counting method adopted.

In the first iteration, similarly to all FSC algorithms, DCI exploits a vector of counters, which are directly addressed through item identifiers. For $k \geq 2$, instead of using complex data structures like hash-trees or prefix-trees, DCI 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*, PREFIX$_k$[ ], of size $\binom{m_k}{2}$. In particular, each entry of PREFIX$_k$[ ] is associated with a distinct *ordered prefix* of two items. For $k = 2$, PREFIX$_k$[ ] can directly contain the counters associated with the various candidate 2-itemsets, while, for $k > 2$, each entry of PREFIX$_k$[ ] contains the pointer to the contiguous section of ordered candidates in C_k sharing the same prefix. To permit the various entries 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 $t = \{t_1, \ldots, t_{|t|}\}$, we select all the possible 2-prefixes of all k-subsets included in t. We then exploit PREFIX$_k$[ ] to find the sections of C_k which must be visited in order to check set-inclusion of candidates in transaction t.

2.2 Intersection-based phase

Since the counting-based approach becomes less efficient as k increases [20], DCI starts its intersection-based phase as soon as possible. Unfortunately, the intersection-based method needs to maintain in memory the

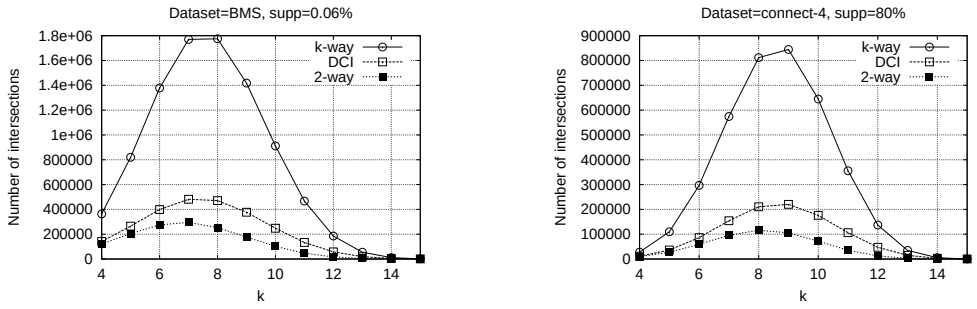
vertical representation of the pruned dataset. So, at iteration k, $k \geq 2$, DCI checks whether the pruned dataset $\mathcal{D}_k$ may fit into the main memory. When the dataset becomes small enough, its vertical in-core representation is built on the fly, while the transactions are read and counted 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 non pruned items. Therefore, the amount of memory required to store $\mathcal{VD}[\,][\,]$ is $m_k \times n_k$ bits.

At each iteration of its intersection-based phase, DCI computes F_k as follows. For each candidate k-itemset c, we *and*-intersect the k bit-vectors associated with the items included in c (*k-way* intersection), and count the 1's present in the resulting bit-vector. If this number is greater or equal to *minsup*, we insert c into F_k. Consider that a bit-vector intersection can be carried out very efficiently and with high spatial locality by using primitive bitwise *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.

The strategy above is, in principle, highly inefficient, because it always needs a *k-way* intersection to determine the support of each candidate c. Nevertheless, a caching policy could be exploited in order to save work and speed up our *k-way* intersection method. To this end, DCI uses a small "cache" buffer to store all the $k-2$ intermediate intersections that have been computed for the last candidate evaluated. Since candidate itemsets 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 process c', we can avoid performing the first $h-1$ intersections since their result can be found in the cache.

To evaluate the effectiveness of our caching policy, we counted the actual number of intersections carried out by DCI on two different datasets: BMS, a real-world sparse dataset, and connect-4, a dense dataset (the characteristics of these two datasets are reported in Table 1). We compared this number with the best and the worst case. The best case 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 worst case regards the adoption of a pure *k-way* intersection method, i.e. a method that does not exploit caching at all. Figure 1.(a) plots the results of this analysis on the sparse dataset for support threshold $s = 0.06\%$, while Figure 1.(b) regards the dense dataset mined with support threshold $s = 80\%$. In both the cases the caching policy of DCI turns out to be very effective, since the actual number of intersections performed



(a) (b)

Figure 1: Per iteration number of tidlist intersections performed on datasets BMS and connect-4 for DCI, and the pure *2-way* and *k-way* intersection-based approaches.

results to be very close to the best case. Moreover, DCI requires orders of magnitude less memory than a pure *2-way* intersection approach, thus better exploiting memory hierarchies.

We have to consider that while caching reduces the number of tidlist intersections, we also need to reduce intersection cost. To this end, further heuristics, differentiated w.r.t. *sparse* or *dense* datasets, are adopted by DCI. In order to apply the right optimization, the vertical dataset is tested for checking its density as soon as it is built. In particular, we compare the bit-vectors associated with the *most frequent items*, i.e., the vectors which likely need to be intersected several times since the associated items occur in many candidates. If large sections of these bit-vectors turn out to be identical, we deduce that the items are highly correlated and that the dataset is dense. In this case we adopt a specific heuristics which exploits similarities between these vectors. Otherwise the technique for sparse datasets is adopted. In the following we illustrate the two heuristics in more detail:

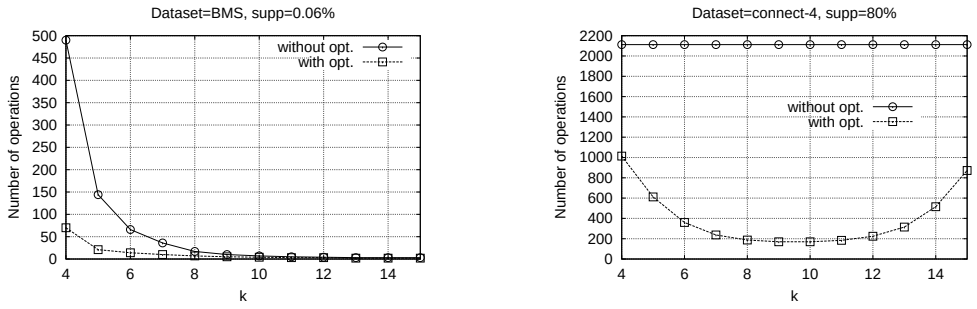
- **Sparse datasets.** Sparse or moderately dense datasets originate bit-vectors containing long runs of 0's. To speedup computation, while we compute the intersection of the bit-vectors relative to the first two items c_1 and c_2 of a generic candidate itemset $c = \{c_1, c_2, \ldots, c_k\} \in C_k$, we also identify and maintain information about the runs of 0's appearing in the resulting bit-vector stored in cache. The further intersections that are needed to determine the support of c (as well as intersections needed to process other k-itemsets sharing 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 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 for all the frequent k-itemsets. 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.

- **Dense datasets.** If the dataset turns out to be dense, we expect to deal with a dataset characterized by strong correlations among the most frequent items. This not only means that the bit-vectors associated with the *most frequent items* contain long runs of 1's, but also that they turn out to be very similar. The heuristic technique adopted by DCI for dense dataset thus 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 is likely to include several of these most frequent items, we avoid repeatedly intersecting the identical segments of the corresponding vectors. This technique may save a lot of work because (1) the intersection of identical vector segments is done once, (2) the identical segments are usually very large, and (3), long candidate itemsets presumably contains several of these most frequent items.

The plots reported in Figure 2 show the effectiveness of the heuristic optimizations discussed above in reducing the average number of bitwise *and* operations needed to intersect a pair of bit-vectors. In particular, Figure 2.(a) regards the *sparse* BMS dataset mined with support threshold $s = 0.06\%$, while Figure 2.(b) regards the *dense* dataset connect-4 mined with support threshold $s = 80\%$. In both cases, we plotted the per-iteration cost of each bit-vector intersection in terms of bitwise *and* operations when either our heuristic optimizations are adopted or not. The two plots show that our optimizations for both sparse and dense datasets have the effect of reducing the intersection cost up to an order of magnitude. Note that when no optimizations are employed, the curves exactly plot the bit-vector length (in words). Finally, from the plot reported in Figure 2.(a), we can also note the effect of the pruning technique used on sparse datasets. Pruning has the effect of reducing the length of the bit-vectors as execution progresses. On the other hand,



(a) (b)

Figure 2: Average number of bitwise *and* operations actually performed to intersect two bit-vectors as a function of k.

when datasets are dense, the vertical dataset is not pruned, so that the length of bit-vectors remains the same for all the DCI iterations.

3 ParDCI

In the following we describe the different parallelization techniques exploited for the counting- and intersection-based phases of ParDCI, the parallel version of DCI. Since our target architecture is a cluster of SMP nodes, in both phases we distinguish between *intra-node* and *inter-node* levels 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].

3.1 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 size of partitions depend on the relative powers of nodes. At each iteration k, a identical copy of C_k is generated independently by each node. Then each node p reads blocks of transactions from its own dataset partition $\mathcal{D}_{p,k}$, performs subset counting, and writes pruned transactions to $\mathcal{D}_{p,k+1}$. At the end of the iteration, an all-reduce operation is performed to update the counters associated to all candidates of C_k, and all the nodes produce an identical set F_k.

At the intra-node level each node uses a pool of threads. They have the task of checking in parallel

candidate itemsets against chunks of transactions read from $\mathcal{D}_{p,k}$. The 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 unnecessary data copying overheads, we used an optimized producer/consumer schema for the cooperation among the Master and Worker threads. Through two shared queues, the Master and Worker threads cooperate by exchanging pointers to empty and full buffers storing block of transactions to be processed.

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 thread-private 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 and writes F_k to the local disk.

3.2 The intersection-based phase

During the intersection-based phase, a 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 on-the-fly the bit-vectors representing their own in-core portions of the vertical dataset. Before starting the intersection-based 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 partitioned by exploiting problem-domain knowledge. A disjoint partition $F_{p,\overline{k}}$ of $F_{\overline{k}}$ is assigned to each node p, where $\bigcup_p F_{p,\overline{k}} = F_{\overline{k}}$. 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 $C_k^p \cap C_k^{p'} = \emptyset$ if $p \neq p'$, 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 we build each candidate $(\overline{k}+1)$-itemset as the union of two frequent $\overline{k}$-itemsets sharing the first $\overline{k}-1$ items, we are sure that each candidate can be independently generated

starting from one of the l disjoint sections of $F_{\overline{k}}$. Then the various partitions $F_{p,\overline{k}}$ are created by assigning the l sections to the np processing nodes with a simple greedy strategy that considers the number of itemsets contained in each section, and builds well-balanced partitions. Once completed the partitioning of $F_{\overline{k}}$, the nodes independently generate the associated candidates and determine their supports by intersecting the corresponding bit-vectors. Nodes continue to work according to the schema above also for the following iterations, without any communication exchange.

At the intra-node level, a similar Candidate Distribution approach is employed, but at a finer granularity by using dynamic scheduling to ensure load balancing. In particular, at each iteration k the Master thread of a node p initially splits the local partition of $F_{p,k-1}$ into x disjoint partitions S_i, $i = 1, \ldots, x$, where $x >> t$, and t is the number of active threads. The boundaries of these x partitions are then inserted in a shared queue. Once the shared queue is initialized, also the Master thread becomes a Worker. Hereinafter, 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 partition of the local $F_{p,k-1}$. If the queue is empty, write $F_{p,k}$ 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 to 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 (see Section 2.2). If c turns out to be frequent, put c into $F_{p,k}$. Go to step 2.

4 Experimental Results

The DCI algorithm is currently available in two versions, a MS-Windows one, and a Linux one. 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 one of the fastest algorithm for FSC[1], and the Christian Borgelt's implementation of *Apriori*[2]. For the sequential tests we used a Windows-NT

[1]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 [15].

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

Table 1: Datasets used in the experiments.

Dataset	Description
T25I10D10K	1K items and 10K transactions. The average size of transactions is 25, and the average size of the maximal potentially frequent itemsets 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 size of transactions is 25, and the average size of the maximal potentially frequent itemsets 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 size of transactions is 10, and the average size of the maximal potentially frequent itemsets is 8. Synthetic dataset created with the IBM dataset generator [6].
400k_t30_p16_m1k	1K items and 400K transactions. The average size of transactions is 30, and the average size of the maximal potentially frequent itemsets is 16. 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. The average transaction size is 20, and the average size of maximal potentially frequent itemsets is 8. The number of transactions is varied for scaling measurements.
t50_p32_m1k	A series of three synthetic datasets with the same number of items (1K), average transaction size of 50, and average size of maximal potentially frequent itemsets equal to 32. We used three datasets of this series with 1000k, 2000k and 3000k transactions.
connect-4	Dense dataset with 130 items and about 60K transactions. The maximal transaction size is 45. Available at `http://www.cs.sfu.ca/~wangk/ucidata/dataset/connect-4/connect-4.data`
BMS	Also known as *Gazelle*. 497 items and 59K transactions containing click-stream data from an e-commerce web site `gazelle.com`. 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`

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 2-way SMPs, for a total of six processors. Each SMP is equipped with 256 MBytes of main memory and a SCSI disk. For the tests, we used both synthetic and real datasets by varying the minimum support threshold s. The characteristics of the datasets used are reported in Table 1.

DCI performances and comparisons. Figure 3 reports the total execution times obtained running *Apriori*, FP-growth, and DCI on some 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 more than one order of magnitude. For connect-4, the dense dataset, the curve of *Apriori* is not shown, due to the relatively too long execution times. Note that, accordingly to [23], on the real-world sparse dataset BMS (also known as *Gazelle*), *Apriori* turned out to be faster than FP-growth. To overcome such bad performance results on sparse datasets, the same authors of FP-growth recently proposed a new pattern-growth algorithm, H-mine [18]. By comparing our experimental results with

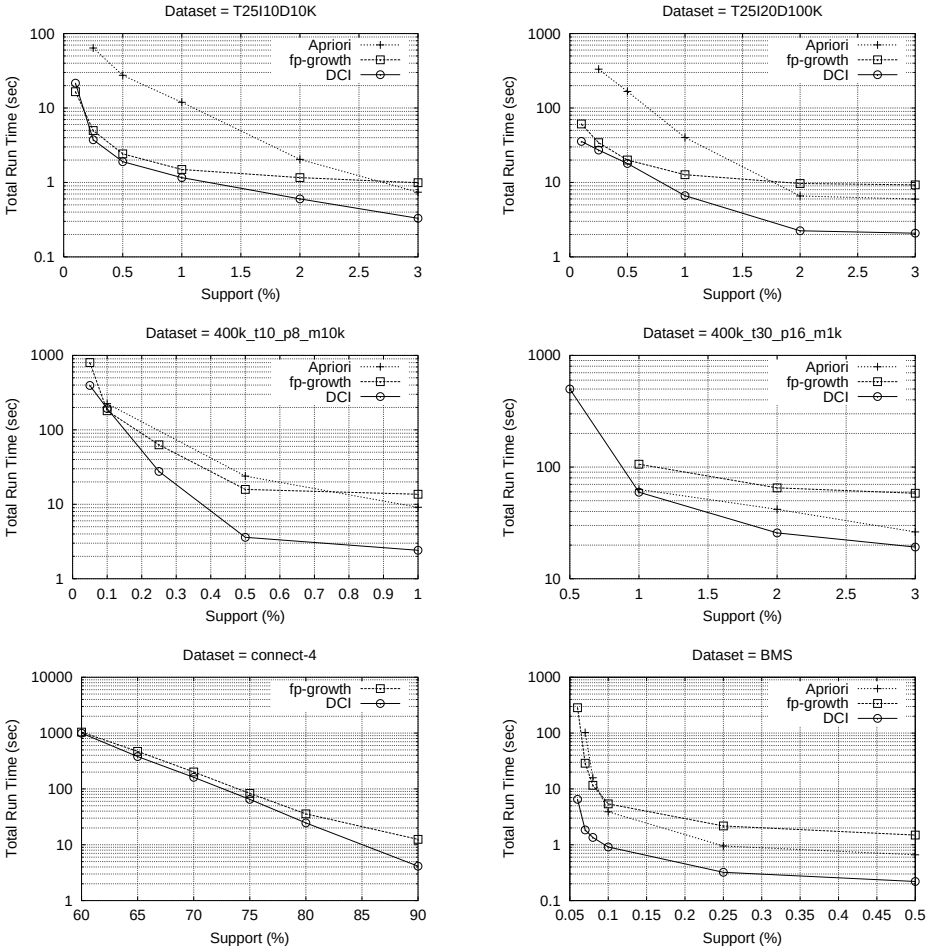
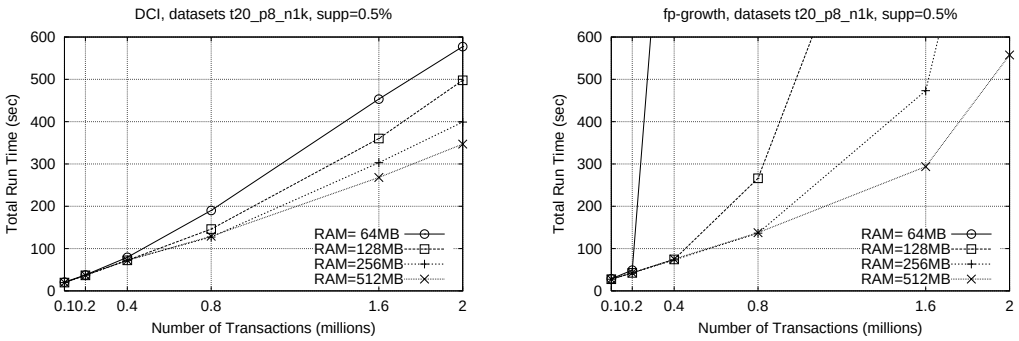


Figure 3: Total execution times for DCI, *Apriori*, and FP-growth on various datasets as a function of the support threshold.

the published execution times on the BMS dataset, we deduced that DCI is also faster than H-mine. For $s = 0.06\%$, we obtained an execution time of about 7 sec., while H-mine completes in about 40 sec. on a faster machine.

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 slightly faster 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 final 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 FP-growth is faster than DCI since it does not require candidate generation.



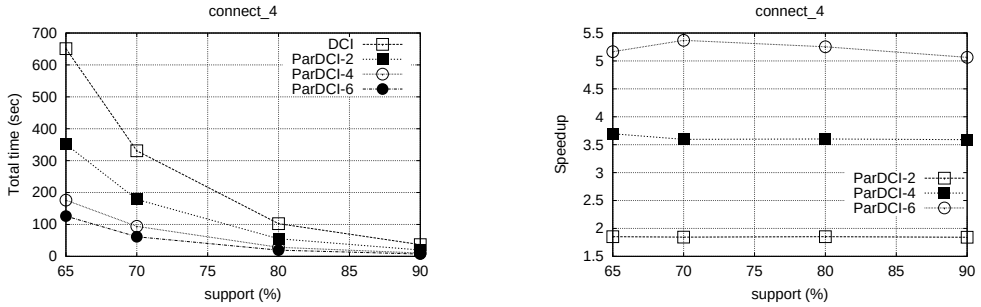
(a) (b)

Figure 4: Total execution times of (a) DCI, and (b) FP-growth, on datasets in the series t20_p8_m1k mined with a support threshold $s = 0.5\%$ on a PC equipped with different RAM sizes as a function of the number of transactions (ranging from $100K$ to $2M$).

We also tested the scale-up behavior of DCI when both the size of the dataset and the size of RAM installed in the PC vary. The datasets employed for these tests belong to the series t20_p8_m1k (see Table 1) mined with support threshold $s = 0.5\%$, while the available RAM was changed from 64MB to 512MB by physically plugging additional memory into the PC main board. Figure 4.(a) and 4.(b) plot several curves representing the execution times of DCI and FP-growth, respectively, as a function of the number of transactions contained in the dataset processed. Each curve plotted refers to a series of tests conducted with the same PC equipped with a different amount of memory. As it can be seen from Figure 4.(a), DCI scales linearly also on machines with a few memory. Due to its adaptiveness and the use of efficient out-of-core techniques, it is able to modify its behavior in function of the features of the dataset mined and the computational resources available. For example, in the tests conducted with the largest dataset containing two millions of transactions, the in-core intersection-based phase was started at the sixth iteration when only 64MB of RAM were available, and at the third iteration when the available memory was 512MB. On the other hand the results reported in Figure 4.(b) show that FP-growth requires much more memory than DCI, and is not able to adapt itself to memory availability. For example, in the tests conducted with 64MB of RAM, FP-growth requires less than 30 seconds to mine the dataset with $200k$ transactions, but when we double the size of the dataset to $400k$ transactions, FP-growth execution time becomes 1303 seconds, more than 40 times higher, due to an heavy page swapping activity.

Performance evaluation of ParDCI. We evaluated ParDCI on both dense and sparse datasets. First we compared the performance of DCI and ParDCI on the dense dataset `connect-4`. Figure 5 plots total execution

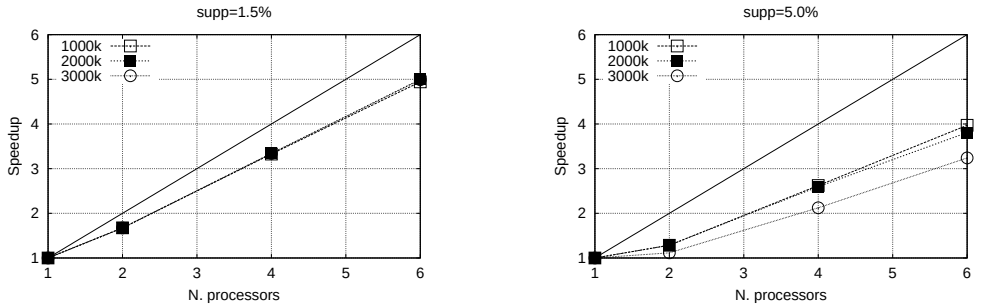
times and speedups (nearly optimal ones) as functions of the support thresholds s. ParDCI-2 corresponds to the pure multithread version running on a single 2-way SMP, while ParDCI-4 and ParDCI-6 also exploit inter-node parallelism, and run, respectively, on two and three 2-way SMPs.



(a) (b)

Figure 5: Dense dataset `connect-4`: completion times of DCI and ParDCI (a) and speedups of ParDCI (b), varying the minimum support threshold.

For what regard sparse datasets, 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.



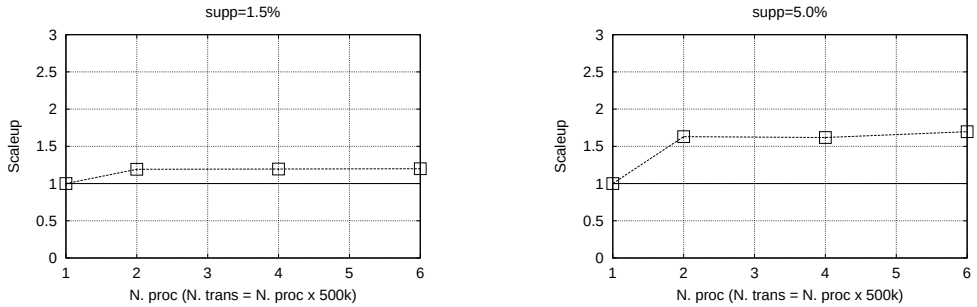
(a) (b)

Figure 6: Speedup for datasets 1000K, 2000K and 3000K with $s = 1.5\%$ (a) and $s = 5\%$(b).

Figure 6 plots the speedups obtained on the three synthetic datasets for two fixed support thresholds ($s = 1.5\%$ and $s = 5\%$), as a function of the 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 tests, ParDCI showed a speedup close to the optimal one. Considering the results obtained with one or two processors, one can note that the slope of the speedup curve is 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.

Finally, Figure 7 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 worse than the theoretical limit.



(a) (b)

Figure 7: 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 our tests we observed a very limited imbalance. The differences in the execution times of the first and last node to end execution were always below the 0.5%. In the near future we plan to extend ParDCI with adaptive *work stealing* policies for load balancing aimed at efficiently exploiting also heterogeneous/grid environments.

5 Conclusions

DCI and ParDCI use different approaches for extracting frequent patterns: counting-based during the first iterations and intersection-based for the following ones. Adaptiveness and resource awareness are the main innovative features of the two algorithms. On the basis of the characteristics of the dataset mined, DCI and ParDCI choose at run–time which optimization to adopt for reducing the cost of mining. Dataset pruning and

effective out-of-core techniques are exploited during the counting-based phase, while the intersection-based phase works in core, and is started only when the pruned dataset can fit into the main memory. As a result, our algorithms can manage efficiently, also on machines with limited physical memory, very large datasets from which, due to the different correlations among items, either short or long frequent patterns can be mined.

ParDCI, the multi-threaded and distributed version of DCI, uses a *Count Distribution* strategy for parallelizing the counting-based phase, and a *Candidate Distribution* one for the intersection-based phase. Our implementation of these parallelization strategies minimizes the communication overheads and achieves a good load balancing. ParDCI employs different granularities at the two levels of parallelism exploited: fine grain at the *intra-node* level where shared-memory cooperation is used, and coarse grain at the *inter-node* level where SMP nodes cooperate with message-passing.

The experimental evaluations demonstrated that DCI significantly outperforms *Apriori* and FP-growth on both synthetic and real-world datasets. In many cases the performance improvements are impressive. ParDCI, on the other hand, exhibits excellent scaleups and speedups on our homogeneous cluster of SMPs.

The variety of datasets used and the large amount of tests conducted permit us to state that the performances of DCI and ParDCI are not influenced by dataset characteristics, and that our optimizations are very effective and general.

To share our efforts with the data mining community, we made DCI and ParDCI binary codes available for research purposes at `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*, 2000. Special Issue on Hig Performance Data Mining.

[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] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining frequent patterns with counting inference. *ACM SIGKDD Explorations Newsletter*, 2(2):66–75, December 2000.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[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 3^{rd} 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] J. Pei, J. Han, H. Lu, S. Nishio, and D. Tang, S. amd Yang. H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases. In *Proc. The 2001 IEEE International Conference on Data Mining (ICDM'01)*, San Jose, CA, USA, 2000.

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

[20] 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.

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

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

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