Parallel Mining of Frequent Closed Patterns: Harnessing Modern Computer Architectures

Claudio Lucchese  
Ca’ Foscari University  
Venice, Italy  
clucches@dsi.unive.it

Salvatore Orlando  
Ca’ Foscari University  
Venice, Italy  
orlando@dsi.unive.it

Raffaele Perego  
National Researches Council  
Pisa, Italy  
r.perego@isti.cnr.it

Abstract

Inspired by emerging multi-core computer architectures, in this paper we present MT\_CLOSED, a multi-threaded algorithm for frequent closed itemset mining (FCIM). To the best of our knowledge, this is the first FCIM parallel algorithm proposed so far.

We studied how different duplicate checking techniques, typical of FCIM algorithms, may affect this parallelization. We showed that only one of them allows to decompose the global FCIM problem into independent tasks that can be executed in any order, and thus in parallel.

Finally we show how MT\_CLOSED efficiently harness modern CPUs. We designed and tested several parallelization paradigms by investigating static/dynamic decomposition and scheduling of tasks, thus showing its scalability w.r.t. to the number of CPUs. We analyzed the cache friendliness of the algorithm. Finally, we provided additional speed-up by introducing SIMD extensions.

1 Introduction

We can recognize some important trends in the design of future computer architectures. The first one indicates that the number of processor cores integrated in a single chip will continue to increase. The second one is that memory hierarchies in chip multiprocessors (CMPs) will become even more complex, and their organization will try to maximize the number of requests that can be served on-chip, thus reducing the overall number of cache misses that involve the access to memory levels outside the chip. Finally, nowadays modern processors have SIMD extensions, like Intel’s SSE, AMD’s 3DNow!, and IBM’s Altivec, and further instructions and enhancements are expected to be added in future releases. Thus algorithm implementations can take large performance advantages from their use.

In order to maximize the utilization of the computing resources provided by CMPs, and increase application performance, it is mandatory to use multiple threads within applications. Programs running on only one processor will not be able to take full advantage of such future processors, since they will only exploit a limited amount of implicit instruction-level parallelism (ILP). As a result, explicit parallel programming has suddenly become relevant even for desktop and laptop systems. Also, coarse-grained parallelism is mandatory in order to avoid conflicts among the core’s private caches.

Unfortunately, as the gap between memory and processor/CMPs speeds continues to widen, more and more cache efficiency becomes one of the most important components of performance. High performance algorithms must be conscious of the need of exploiting locality in data layout and, consequently, in data access. Furthermore, it is still unexplored the opportunity to exploit the SIMD co-processors that equip most modern CPUs. We instead think that the SIMD paradigm can be useful to speed-up several data mining sub-tasks operating on large data.

In this paper we discuss how a demanding algorithm to solve the general problem of Frequent Itemsets Mining (FIM) can be effectively implemented on modern computer architectures, by capturing all the technological trends mentioned above.

More specifically, the algorithm that will be the subject of our analysis (DCI\_CLOSED [6]) aims at extracting all the frequent closed itemsets from a transactional database.

This algorithm is deeply analyzed in this paper with respect to its capability to harness modern architectures. First of all, the core computational kernel of DCI\_CLOSED is mainly based on bitwise operations performed on long lists. Such operations are data parallel, and can be efficiently programmed using SIMD co-processors of modern CPUs.

Moreover, the data structures used by DCI\_CLOSED, and the way they are accessed, exhibit high locality, especially spatial one. We will see that the largest data structure exploited by DCI\_CLOSED is read-only. The concurrent accesses to such data by multiple processors thus do not need
synchronizations and do not cause cache coherence invalidations and misses.

These peculiarities of DCI\_CLOSED allowed us to design of MT\_CLOSED, an efficient, shared memory parallel FCIM algorithm for SMPs or multi-core CPUs. This is the main contribution of our work, since MT\_CLOSED is to the best of our knowledge, the first FCIM parallel algorithm proposed so far.

Note that a static scheduling/assignment of the mining sub-tasks to the various threads may involve load imbalance. In order to deal with this problem, we considered static scheduling as a sort of baseline, and investigated a dynamic scheduling policy of such statically created tasks, and also dynamic decomposition of the search space to provide adaptive load balancing. This resembles a receiver-initiated work-stealing policy, where a local round-robin strategy is used to select the partner from which a thread could steal some work. In our experiments, MT\_CLOSED obtained very good speed-ups on SMPs/multi-core architectures with up to 8 CPUs.

2 Preliminaries

Frequent Itemsets Mining (FIM) is a demanding task common to several important data mining applications that looks for interesting patterns within databases (e.g., association rules, correlations, sequences, episodes, classifiers, clusters). The problem can be stated as follows. Let \( \mathcal{I} = \{a_1, \ldots, a_M\} \) be a finite set of items or singletons, and let \( \mathcal{D} = \{t_1, \ldots, t_N\} \) be a dataset containing a finite set of transactions, where each transaction \( t \) is a subset of \( \mathcal{I} \). We call \( k \)-itemset a set of \( k \) items \( I = \{i_1, \ldots, i_k \mid i_j \in \mathcal{I}\} \). Given a \( k \)-itemset \( I \), let \( \sigma(I) \) be its support, defined as the number of transactions in \( \mathcal{D} \) that include \( I \). Mining all the frequent itemsets from \( \mathcal{D} \) requires to discover all the itemsets having a support greater or equal to a given minimum support threshold \( \sigma \). We denote with \( \mathcal{F} \) the collection of frequent itemsets, which is indeed a subset of the huge search space given by the power set of \( \mathcal{I} \).

State-of-the-art FIM algorithms visit a lexicographical tree spanning over such search space, by alternating candidate generation, and support counting steps. In the candidate generation step, given a frequent itemset \( X \) of \( |X| \) elements, new candidate \((|X| + 1)\)-itemsets \( Y \) are generated as supersets of \( X \) that follow \( X \) in the lexicographical order. During the counting step, the support of such candidate itemsets is evaluated on the dataset, and if some of those are found to be frequent, they are used to re-iterate the algorithm recursively. This two-step process stops when no more frequent itemsets are discovered.

The collection of frequent itemsets \( \mathcal{F} \) extracted from a dataset is usually very large. This makes the task of the analyst hard: since he has to extract useful knowledge from a huge amount of patterns, especially when very low minimum support thresholds are used.

Closed itemsets\cite{2} are a solution to this problem. For each collection of frequent itemsets \( \mathcal{F} \), there exist a condensed, i.e. both concise and lossless, representation of \( \mathcal{F} \) given by the collection of closed itemset \( \mathcal{C} \), \( \mathcal{C} \subseteq \mathcal{F} \). It is concise because \( |\mathcal{C}| \) may be orders of magnitude smaller than \( |\mathcal{F}| \). This allows to extract additional potentially interesting itemsets by using lower minimum support thresholds, which would make the extraction of all the frequent itemsets intractable. Moreover, it is lossless, because from \( \mathcal{C} \) it is possible to derive the identity and support of every frequent itemset in \( \mathcal{F} \). Finally, the extraction of association rules directly from closed itemsets has been shown to be more meaningful for analysts\cite{10,11}, since \( \mathcal{C} \) does not include many redundancies which are present in \( \mathcal{F} \).

The concept of closed itemset is based on the two following functions \( f \) and \( g \):

\[
f(T) = \{i \in \mathcal{I} \mid \forall t \in T, i \in t\} \]
\[
g(I) = \{t \in \mathcal{D} \mid \forall i \in I, i \in t\},
\]

where \( T \) and \( I \), \( T \subseteq \mathcal{D} \) and \( I \subseteq \mathcal{I} \), are subsets of all the transactions and item appearing in \( \mathcal{D} \), respectively. Function \( f \) returns the set of items included in all the transactions belonging to \( T \), while function \( g \) returns the set of transactions (called tid-list) supporting a given itemset \( I \).

**Definition 1.** An itemset \( I \) is said to be closed iff

\[
c(I) = f(g(I)) = f \circ g(I) = I
\]

where the composite function \( c = f \circ g \) is called Galois operator or closure operator.

The closure operator defines a set of equivalence classes over the lattice of frequent itemsets: two itemsets belong to the same equivalence class \textit{iff} they have the same closure, i.e. they are supported by the same set of transactions. Fig. 2(b) shows the lattice of frequent itemsets derived from the simple dataset reported in Fig. 2(a), mined with \( \sigma = 1 \). We can see that the itemsets with the same closure are grouped in the same equivalence class. Each equivalence class contains elements sharing the same supporting transactions, and closed itemsets are their maximal elements. Note that closed itemsets (six) are remarkably less than frequent itemsets (sixteen).

Usually inspired by popular FIM algorithms, also Closed Itemset Mining (FCIM) algorithms exploit a visit of the search space layered on top of a lexicographic spanning tree. In addition to the usual candidate generation and support counting, they execute a closure computation step. Given a closed itemset \( X \), new candidates \( Y \supset X \) are generated. For every candidate \( Y \) that is found to be frequent, the closure \( c(Y) \) is computed in order to find a new closed
Given two itemsets \( C \) and \( D \), there exists an equivalence class, any superset \( Y \) which will be later used to re-iterate the algorithm. Note that since a closed itemset is the maximal pattern of its equivalence class, any superset \( Y \) surely belongs to another equivalence class, and therefore its closure \( c(Y) \) will lead to a new closed itemset.

Unfortunately, the closure operator may involve backward jumps with respect to the underlying lexicographic tree. For instance, in Fig. 2, given the closed itemset \( \{C\} \) and the candidate \( \{C, D\} \), calculating the closure \( c(\{C, D\}) \) would conduct ourselves the itemset \( \{A, C, D\} \), which is not a descendant of \( \{C, D\} \). This may happen because, given any lexicographic order \( \preceq \), it is possible that \( c(Y) \preceq Y \).

The closure operator breaks the order given by the lexicographic tree spanning the search space, and thus we also loose the simple property of a visiting algorithm which guarantees that every node is visited only once. For example, the itemset \( \{A, C, D\} \) can be also reached by calculating the closure of \( \{A\} \).

In conclusion, FCIM algorithms may visit the same closed itemset multiple times, and they thus need to introduce some duplicate detection process. For efficiency reasons, this check is applied to candidate itemsets before calculating their closure. Therefore, rather than detecting a duplicate, FCIM algorithms try to avoid the generation of a duplicate.

Usually, such detection techniques exploit one of the following two lemmas.

**Lemma 1 (Subsumption Lemma).** Given two itemsets \( X \) and \( Y \), if \( X \subset Y \) and \( \text{supp}(X) = \text{supp}(Y) \) (i.e., \( |g(X)| = |g(Y)| \)), then \( c(X) = c(Y) \).

Let's focus on Fig. 2. Suppose that the sub-trees rooted in \( A, B, C \) and \( D \) are mined (recursively) in this order. Once the algorithm generates the new candidate \( Y = \{C, D\} \), it can apply the subsumption lemma to realize that \( Y \) is a subset of the closed itemset \( \{A, C, D\} \), which has the same support, and which was already mined. Therefore, it can avoid calculating \( c(Y) \) since this closed itemset (and all its supersets) have been already extracted.

Note that this Lemma 1 requires a global knowledge of the collection of frequent closed itemsets mined so far. As soon as a new closed itemset is discovered, it should be stored in a suitable data structure.

**Lemma 2 (Extension Lemma).** Given an itemset \( X \), if \( \exists i \in I \), \( i \not\in X \), such that \( g(X) \subseteq g(i) \), then \( X \) is not closed and \( i \in c(X) \).

This lemma provides a different method to perform duplicate check. Back again to the candidate itemset \( Y = \{C, D\} \) in Figure 2, it is easy to see that \( g(Y) \subseteq g(A) \), and therefore the item \( A \) belongs to the closure of \( Y \). On the other hand, by construction of the lexicographic tree, the item \( A \) does not belong to any descendant of \( Y \), and therefore neither \( c(Y) \) is descendant of \( Y \). This means that \( c(Y) \) is a duplicate closed itemset, since it should be visited by a different branch of the lexicographic tree.

The Extension lemma uses the tid-lists of every single item, i.e. a global knowledge on the dataset. Differently from the Subsumption Lemma, it does not need any additional dynamic data structure.

In the next section we will describe the implications of exploiting one of two lemma for a parallel FCIM algorithm.

### 3 Parallel Pattern Mining on Modern Computer Architectures

To the best of our knowledge, no parallel algorithm has been proposed so far to mine frequent closed itemsets. On the other hand, many sequential FCIM algorithms were proposed in the past, and also some interesting parallel implementation of FIM algorithms exist as well.

The lack of parallel FCIM solutions is probably due to the fact that mining closed itemsets is more challenging than mining other kind of patterns. As discussed before, during the exploration of the search space, a duplicate checking

<table>
<thead>
<tr>
<th>TID</th>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B D</td>
</tr>
<tr>
<td>2</td>
<td>A B C D</td>
</tr>
<tr>
<td>3</td>
<td>A C D</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
</tr>
</tbody>
</table>

Figure 1. (a) The input transactional dataset \( D \), represented in its horizontal form. (b) Lexicographic spanning tree of all the frequent itemsets (\( \text{min supp} = 1 \)), with closed itemsets and their equivalence classes.
process is involved, and deciding whether a pattern leads to a duplicate or not, needs a global view either of the dataset or of the collection of closed patterns mined so far. This makes tougher the decomposition of the mining task into independent, thus parallelizable, sub-problems.

In this section, we will take a guided tour of existing FCIM algorithms. We distinguish between algorithms exploiting Lemma 1 or Lemma 2. More than this, we will consider pros and cons of the parallelization of these algorithms, taking into consideration the capability to take advantage of modern CPUs.

### 3.1 First Generation Algorithms

Within this category of algorithms we include CHARM [12] and CLOSET [7]: these are the first depth-first algorithms providing really efficient performances.

The rationale behind them is to apply a divide-et-impera approach to the input dataset thus reducing the amount of data to be handled during each iteration. They both exploit recursive projections, where at each node of the spanning tree a small dataset is materialized, and used for the subsequent mining operations. This smaller dataset is pruned from all the non necessary information. When mining the supersets of some given closed itemsets $X$, two kinds of pruning can be applied: (1) remove transactions that do not contain $X$; (2) remove items that cannot be used to generate new candidates $Y \supset X$ in accordance to the underlying lexicographical order.

This latter pruning, removes many items from each projected dataset, and thus the Extension Lemma is not applicable. In fact, CHARM stores every newly discovered itemset in a historical collection of frequent closed itemsets arranged in a two-level hash structure. Duplicates are detected by extensive sub-sumption checks on the basis of Lemma 1.

CLOSET is very similar, but uses prefix-tree structures named fp-trees to store both projected datasets and historical closed itemsets. These fp-trees have become very popular and they have been used many mining algorithms and applied for different kind of patterns.

The last algorithm in this category is FP-CLOSE [5]. The algorithm comes chronologically much later than the other two, but it still uses the same approach based on the Subsumption Lemma. FP-CLOSE is very similar to CLOSET, but emphasized where the concept of recursive projection that is applied also to the historical collection of frequent closed itemsets. Not only a small dataset is associated to each node of the tree, but also a pruned subset of the closed itemsets mined so far is forged and used for duplicate detection. Indeed, this technique is called progressive focusing and it was introduced by [4] for mining maximal frequent itemsets. Together with other optimizations, this truly provides dramatic speed-ups, making FP-CLOSE order of magnitude faster than CHARM and CLOSET, and also making it worth to be celebrated as the fastest algorithm at FIMI workshop[3].

These approaches have several drawbacks when we think to their parallelization. Assume that some strategy is used, such that subtrees of the global lexicographic spanning tree are spread among a pool of mining threads. Such an algorithm will incur in the following problems:

**Spurious itemsets.** Since different portions of the search space are mined in parallel, closed itemsets are discovered in an unpredictable order. Recall that this order is fundamental for the correctness of the Subsumption Lemma. Suppose that our favorite candidate $Y = \{C, D\}$ is discovered before itemset $\{A, C, D\}$: the Subsumption Lemma will not detect $Y$ as a duplicate. On the other hand, the algorithm cannot compute the correct closure $c(Y) = \{A, C, D\}$ since no information about item $A$ will be present in the projected database. The algorithm will thus produce a spurious itemset $Y = \{C, D\}$ which is not closed.

**Search space overhead.** When a spurious itemsets is not detected as a duplicate, this is used to reiterate the mining. Therefore each thread will be likely to mine more itemsets than required, and therefore the cumulative cost of all the mining sub-tasks becomes larger than the serial cost of the algorithm.

**Maintenance.** In principle, the historical collection of closed itemsets should be shared among every task. Each of them will search for supersets and insert new itemsets. Since insertion requires exclusive access, this may become a limit the parallelism. Also consider that someone will be in charge of detecting and removing all the spurious itemsets generated. For what regards FP-CLOSE, this problem is replicated for each projected collection of historical closed itemsets.

### 3.2 Second Generation Algorithms

The FP-CLOSE algorithm has a large memory footprint: not only the dataset and the whole collection of frequent closed itemsets are stored into the main memory, but also as many projections as the number of levels of the spanning tree must be handled. It is not rare for FP-CLOSE to run out of memory, and start swapping projected fp-trees in and out from secondary memory.

In order to alleviate these large memory requirements, the authors of CLOSET+[8] introduced a new data projection technique associated with a different duplicated detection approach. Instead of materializing several projected datasets, the algorithm stores a set of pointers to the useful sub-trees of the original fp-tree. The algorithm always works on the global fp-tree representing the original datasets. This allows to exploit the Extension Lemma, with
a technique called upward checking by scanning the paths from the root to those useful sub-trees.

The upward checking nicely fits a parallel framework, overcoming the problems discussed above. Since the full dataset is always available, the Extension Lemma allows to detect duplicate itemsets correctly. Therefore spurious itemsets are never generated. Finally, the dataset is accessed read-only and no maintenance of any other shared data structure is needed. As a matter of fact, this approach was used in PAR-CSP [1] for the parallel mining of closed sequential patterns.

However the CLOSET- algorithm is orders of magnitude slower than its younger cousin FP-CLOSE. Also, the upward checking technique was actually used by the author only with sparse datasets. In fact, handling the large number of sub-tree arising in dense datasets is very inefficient compared to the original CLOSET algorithm.

For this reason it is worth for us to focus on DCI_CLOSET. This is yet another FCIM algorithm which sums the advantages of using a vertical bitmap to store the transactional dataset with the ones deriving for the exploitation of the Extension Lemma. Item tid-lists are stored in form of bit-vectors. Support counting and inclusions are performed with simple, and fast, bitwise operations. The result is an algorithm much more efficient then CLOSET-, almost as fast ad FP-CLOSE.

For the sake of completeness in Table 1 we report the running times and the memory footprint of FP-CLOSE and DCI_CLOSET. As usual there is not a clear winner in terms of running times. They have similar performances on accidents and pumsb*. Symmetrically, DCI_CLOSET is much faster on connect while FP-CLOSE is faster on USCE

| Table 1. FCIM Algorithms running times. |
|----------------|----------------|----------------|
|                | FP-CLOSE       | DCI_CLOSET     |
|                | Time | Memory | Time | Memory |
| dataset: ACCIDENTS |
| 10.0%          | 2.32 | 134 MB | 3.00 | 10 M   |
| 7.5%           | 3.32 | 204 MB | 6.24 | 11 M   |
| 5.0%           | 25:06 | 586 MB | 28:23 | 12 M   |
| 2.5%           | 4:44:29 | 2066 MB | 2:06:52 | 14 M   |
| dataset: CONNECT |
| 10.0%          | 2.29 | 47 MB | 0.30 | 5 M   |
| 7.5%           | 4:42 | 77 MB | 0.53 | 5 M   |
| 5.0%           | 10:09 | 141 MB | 1:45 | 5 M   |
| 4.0%           | 15:20 | 199 MB | 2:35 | 5 M   |
| dataset: USCENSUS1990 |
| 60%            | 1:24 | 35 MB | 4:55 | 49 M   |
| 50%            | 4:37 | 172 MB | 16:24 | 58 M   |
| 40%            | 36:19 | 1201 MB | 1:42:32 | 71 M   |
| 35%            | failed | 5:44:06 | 71 M   |
| dataset: PUMS*B |
| 7.5%           | 1:07 | 64 MB | 1:11 | 4 M   |
| 5.0%           | 3:40 | 112 MB | 3:15 | 5 M   |
| 4.0%           | 6:26 | 217 MB | 5:42 | 5 M   |
| 3.0%           | 12:56 | 374 MB | 14:03 | 6 M   |

However, the memory requirements of DCI_CLOSET are always much smaller and almost independent from the minimum support threshold. FP-CLOSE is very demanding and it may happen that the algorithm is not even able to run to completion in dataset such as USCE

The approach of DCI_CLOSET has three main advantages. First bit-wise tid-list allows high spatial locality and cache friendliness. Second, tid-list intersection and inclusion can easily exploit SIMD instructions. Finally, having item tid-lists immediately available makes it easy to use the Extension Lemma.

In conclusion, DCI_CLOSET is almost as fast as FP-CLOSE, but, while the parallelization of the latter would incur in the sever additional overheads described above, DCI_CLOSET can fruitfully be reshaped into a parallel algorithm.
4 The MT\_CLOSED algorithm

This section describes the main aspects of MT\_CLOSED, our parallel version of DCI\_CLOSED.

As in all pattern mining algorithms, also in MT\_CLOSED the initialization of the internal data structures occurs during the first two scans of the dataset. In the first scan frequent single items, denoted with \( \mathcal{F}^1 \), are extracted. Then, infrequent items are removed from the original dataset, and the new resulting set of transactions \( \mathcal{D} \) is stored in a vertical bitmap. The resulting bitmap has size \( |\mathcal{F}^1| \times |\mathcal{D}| \) bits, and row \( i \) is a bit-vector representation of the tid-list of the \( i \)-th frequent item.

Once the vertical representation of \( \mathcal{D} \) is materialized in the main memory, the real mining process performed according to Algorithm 1 starts. The input of the algorithm is a seed closed itemset \( X \), a set of items \( \mathcal{I}_X^- \) that can be added to \( X \) in order to generate new candidates or extending its closure, and another set of items \( \mathcal{I}_X^+ \) that cannot be used to calculate the closure, since they would lead to a duplicate closed itemset.

The above pseudo-code clarifies one important difference of our algorithm from prefix-tree based ones. Once the entire the subtree rooted in \( Y = X \cup i \) has been mined, FP\_CLOSE, CLOSET and CHARM, remove the singleton \( i \) from their recursive projections, thus loosing the chance to detect a jump to an already visited region of the search space by exploiting the Extension Lemma. Conversely, in MT\_CLOSED, the singleton \( i \) is not disregarded during the mining of subsequent closed itemsets. The item \( i \) is stored in the set of items \( \mathcal{I}_X^- \) that cannot be used anymore to extend new candidate patterns.

As recursion progresses in the visit of the lattice, the algorithm grows a local data structure where per-iteration information is maintained. Such information of course includes the vertical dataset \( \mathcal{D} \), the closed itemset \( X \), the sets of items \( \mathcal{I}_X^+, \mathcal{I}_X^- \), but also the tid-list \( g(X) \) and the item \( i \) used to generate a current candidate \( X \cup i \) (line 5). We will refer to \( J = (X, \mathcal{I}_X^+, \mathcal{I}_X^-, \mathcal{D}, g(X), i) \) as the job description, since it provides a exact snapshot of what the algorithm is doing and of what data it needs at a certain point in the execution. Moreover, due to the recursive structure of the algorithm, the job description is naturally associated with a mining sub-task corresponding to an invocation of the algorithm.

It is possible to partition the whole mining task into independent regions, i.e. sub-trees, of the search space, each of them described by a distinct job descriptor \( J \). In principle, we could split the entire search space into a set of disjoint regions identified by \( J_1, \ldots, J_m \), and use some policy to assign these jobs to a pool of parallel threads. Moreover, since the computation of each \( J_i \) does not require any cooperation with other jobs, and does not depend on any data produced by the other threads (e.g. the historical closed itemsets), each task can be executed in a completely independent manner. In the following we will see how to define and distribute \( J_1, \ldots, J_m \).

In Figure 2, we give a pictorial view of our parallel algorithm. We can imagine a public shared memory region, where the vertical dataset is stored: the bitmap is accessed read-only and can be thus shared without synchronizations among all the threads. At the same time, each thread holds
a private data structure, where new job descriptors are materialized for every new closed itemset encountered along the path of lattice visit.

We argue that this net separation between thread private and read-only shared data perfectly fits the multi-core architectures that inspired our work, and in particular their cache memory hierarchies. In the near future, not only it is expected that cache memories will increase their size, but we will also get used to quad-eight core CPUs with complicated and deep cache memory hierarchies, where different levels of caches will be shared among different subsets of CPUs.

4.1 Load partitioning and scheduling

Parallelizing frequent pattern mining algorithms has been proved to be a non trivial problem. The main difficulty resides in the identification of a set of jobs to be later assigned to a pool of threads.

We have already seen that it is easy to find a decomposition into independent task, however this may be not sufficient. One easy strategy would be to partition the frequent single items and assign the corresponding jobs to the pool of threads. Unfortunately, it is very likely, that one among such jobs has a computational cost that is much higher than all the other. The difference may be such that its load is not comparable with the cumulative cost of all the other tasks.

Among the many approaches to solve this problem, an interesting one is [1]. The rationale behind the solution proposed by the authors, is that there are two ways to improve the naïve scheduling described above. One option is to estimate the mining time of every single job, in order to assign the heaviest jobs first, and then the small ones to balance the load during the final stages of the algorithm. The other is to produce a larger number of tasks thus providing a finer partitioning of the search space. Their solution merges these two objectives in a single strategy. First the cost of the jobs associated to the frequent singletons is estimated by running a mining algorithm on significant samples of the dataset. Then, the largest jobs are split on the bases of the 2-itemsets they contain.

Our choice is to avoid the expensive pre-processing step where small samples have to be mined to determine the assignment to different threads.

We will materialize jobs on the basis of the 2-itemsets in \( F^1 \times F^1 \) thus providing a fine-grained partitioning of the search space. The only drawback of this approach is that some of these seed itemsets may lead to discover a duplicated closed itemset. However, they will be promptly recognized and discarded.

For what regards the load partitioning and scheduling policies, we investigated two different approaches and all their possible combinations. In particular, we considered Static and Dynamic solutions for both the load partitioning and scheduling problems. We thus implemented three slightly different versions of MT CLOSED:

Static Partitioning / Static Assignment (SS) Every frequent 2-itemset originates a different valid job descriptor from which to start independent mining sub-tasks. The sets \( I_X^- \) and \( I_X^+ \) are materialized consequently. Such statically fixed job descriptors are statically assigned to the pool of threads in a round-robin fashion. We will use this strategy as a sort of baseline.

Static Partitioning / Dynamic Assignment (SD) The job instances are the same as in the previous SS solution, but they are initially inserted in a virtual shared queue. Idle threads dynamically self-schedule the sub-tasks on demand, by accessing in mutual exclusion the queue to fetch the next job descriptor, thus exploiting a Work-Pool parallel programming paradigm. The SD strategy requires a limited number of synchronizations among the threads. Moreover, in most cases it should result in a quite good load balance. It is in fact important to notice that usually \( F^1 \times F^1 \) contains a quite large number of 2-itemsets with respect to the number of CPU/cores of modern architectures. Moreover, the last tasks to be taken from the shared queue are the least expensive since they are associated with sets \( I_X^- \) having a very low cardinality. These cheap sub-tasks thus help in balancing the load during the final stages of the algorithm.

Dynamic Partitioning / Dynamic Assignment (DD) This totally dynamic policy is based on a work stealing principle. Once a thread has finished its sub-task, it polls the other running threads until it steals from a heavy loaded thread: a closed itemset X, half the set of items in \( I_X^- \) not yet explored, and an adjusted \( I_X^+ \) from which to initiate a new mining sub-task. Thread polling order is established locally on the basis of a simple round-robin policy so that the most recent victim of a theft is the less likely to be stolen again by the same thread. This work stealing approach requires some data migration, since also the tid-list of \( X \) must be copied by the stealing thread in its own private data region. We designed this totally dynamic solution since experimental evidence showed that in some cases a very few sub-tasks results to be order of magnitudes more expensive than the others, and therefore they may harm the load balancing of the system. This DD solution has the great advantage that all the threads work until the whole mining process is finished, since there will be at least one other running thread from which to steal some workload. On the other hand, additional synchronization are required that may slightly slow down the algorithm.

Finally, it is worth noting that, as most FCIM algorithms, also our algorithm exploit projections of the dataset in order to improve its scalability with respect to dataset size. In our context, projecting the dataset introduces another challenge. In principle different threads may work on different pro-
jects, thus resulting in increased memory requirements and possible cache pollution. On the other hand, forcing every thread to work on the same projection would introduce many implicit barriers that remarkably reduce the parallelism degree. In the case of the DD we take advantage of the available degrees of freedom in the generation of new jobs. Our solution is to first force a thread to steal work from some other thread, and therefore to work on the same dataset projection. If this is not possible, i.e. there is nothing to steal, than the thread may build a new projection. During the construction of the new projection, the thread cannot be requested to split its workload until it does not start to mining it. However, we permit to more than one thread to create new projections at the same time, thus implicitly parallelizing also the dataset projecting process.

5 Harnessing Computer Architectures

Unfortunately, large multicore cpus are still not available. In order to asses the scalability of our algorithm we used a large SMP cluster. The experiments were conducted on an “IBM SP Cluster 1600” machine which was kindly provided us by Cineca (http://www.cineca.it). The cluster contains 60 SMP nodes, each equipped with 8 processors and 16 GB of RAM. Each processor is an IBM PowerPC 5, having only one core enabled out of two.

5.1 Scalability

We tested our three scheduling strategies on different datasets, whose characteristics are reported in Table 2. These strategies were plugged into two different versions of the same algorithm: one of them does not exploit projections. This is because we wanted to test the additional overheads introduced by the projection of the dataset. We just reported mining times, since the time to read the dataset is not parallelized.

As expected, the entirely static strategy SS was the one resulting in the worst performance in every experiment conducted. In some tests, it hardly reaches a sufficient speedup: in the tests conducted with the USCensus1990 dataset it never reaches a speedup of 3. It is well known that splitting the visit of the lattice of frequent patterns result in very unbalanced sub-tasks: a few tasks may be much more expensive than all the others. Therefore, our SS strategy fails in well balancing the load among the various threads because it considers every sub-task having the same cost.

On the other hand, our two dynamic strategies provide a considerable improvement. According to the SD strategy, the algorithm partitions the search space in a quite large number of independent sub-tasks, that are self-scheduled by the pool of concurrent threads. Threads receiving low cost sub-tasks will ask for additional workloads, while a thread receiving a demanding sub-task will not be forced to fulfill other requests.

The SD strategy may fail in well balancing the load only when a very few sub-tasks have a huge cost. In this case it may happen that it is not possible to compensate these few demanding tasks with a large set of small-sized ones. As proved by our experiments, our lastly proposed strategy DD helps to alleviate this problem. Once one thread has completed his task, he eagerly steals work of some other running thread. Therefore, even if an initial assignment of work is very unbalanced, large workloads may be dynamically split into smaller ones executed on other threads. The improvement of DD becomes significant when projections are performed, i.e. when a more dynamic and adaptive support is required from the pool of threads.

We also report the results achieved on the connect dataset, where none of the strategies proposed resulted in a performance less close to the ideal case. This is probably due to the additional synchronizations required by the DD strategy that do not allow to profit of its great adaptiveness. Further analysis are however needed to understand in depth this behavior.

Tough the results reported refer to a single minimum supports, in our experiments we witnessed an identical behavior for different support thresholds.

5.2 SIMD Instruction set

The three functions is\_frequent (Alg. 1.6), is\_duplicate (Alg. 1.9) and is\_in\_closure (Alg. 1.13) implement the three basic steps of a typical FCIM algorithm discussed in Section 2. Their cost determines the effectiveness of the algorithm. Our MT\_CLOSED algorithm behaves as follows:

\begin{itemize}
  \item is\_frequent\((X)\) The support of an itemset \(X\) is equal to the number of bits set in the bit-vector representing its tid-list. The function returns true if and only if \(\sigma(X) \geq \sigma\).
  \item is\_duplicate\((Y, I_X)\) According to the Extension Lemma, if there exist an item \(i \in I_X\) such that \(g(Y) \subseteq g(i)\), then the function returns true. The operation \(a \subseteq b\) is translated into a couple of bit-wise operations: \((a\&b) == a\). Note that it is usually not necessary to scan the whole tid-list to discover that the inclusion does not hold.
\end{itemize}
Figure 3. Speedup factors as a function of the number of threads.
is_in_closure(\(j, Y\)) This is actually another application of the Extension Lemma that requires a tid-list inclusion test as for the duplicate detection.

In [6] it is shown how to optimize these operations by reusing some information collected along the path of the lattice visit. However, they share the nice property of being very suitable for a SIMD approach. All these three basic operations are characterized by high spatial locality, and can effectively exploit the streaming instructions performing 128-bit operations that equip modern processor architectures, eg. AMD 3DNow! and Intel SSE2 instruction sets. The Itanium 2 processor also includes the \texttt{POPCNT} instruction, that counts the number of bits set in a 64-bit long string in one clock cycle, and this instruction will be included in newly released processors supporting SSE4.

We implemented both of the above three functions using Intel SSE2 extensions that allow not only to load and store quad-words with a single instruction, but also to perform the simple operations we need in parallel on each word. We can thus theoretically quadruplicate the throughput of each one of the above functions. For instance, we count the number of bits set in a 128 bit quad-word, still with only 16 operations. In general we do not expect to have a fourfold improvement in the performance of the algorithm, however the improvement in the mining time resulted to be significant on several datasets, as reported in Figure 5.2 which plots the relative speedups obtained with the exploitation of SSE2 instruction sets on a Intel Xeon processor.

Given this clear trend of integration of SIMD co-processors in modern CPU architectures, we think it is important to report on the exploitation of this interesting aspect.

6 Conclusions

In this paper we presented MT\textsubscript{CLOSED}, the first multi-threaded parallel algorithm for extracting closed frequent itemsets from transactional databases. MT\textsubscript{CLOSED} design was strongly inspired by nowadays trends in the field of processor architectures that indicate that the number of processor cores integrated in a single chip will continue to increase.

The choice of DCI\textsubscript{CLOSED} was motivated by two other important features. First, its peculiar duplicate checking method that allows an easy decomposition of the mining tasks into independent sub-tasks. Second, the vertical bitmap representation of the dataset allowed to exploit the streaming SIMD instructions that equip modern processor architectures. Finally, we took into account also the cache efficiency of the algorithm and the possibility to take advantage of memory hierarchies in a multi-threaded algorithm.

As a result of its accurate design, MT\textsubscript{CLOSED} performances are impressive. In many experiment we measured quasi-linear speedups with a pool of 8 processors. These promising result allow us to predict that the proposed solutions would exploit efficiently also hardware architectures with a larger number of processors/cores.

References