AND Parallelism for ILP: The APIS System

Rui Camacho[1](✉), Ruy Ramos[1], and Nuno A. Fonseca[2]

[1] DEI and Faculdade de Engenharia and LIAAD-INESCTEC, Universidade do Porto, Porto, Portugal
rcamacho@fe.up.pt

[2] EMBL Outstation, European Bioinformatics Institute (EBI) and CRACS-INESCTEC, Cambridge, UK

Abstract. Inductive Logic Programming (ILP) is a well known approach to Multi-Relational Data Mining. ILP systems may take a long time for analyzing the data mainly because the search (hypotheses) spaces are often very large and the evaluation of each hypothesis, which involves theorem proving, may be quite time consuming in some domains. To address these efficiency issues of ILP systems we propose the APIS (**A**nd **P**arallel**IS**m for ILP) system that uses results from Logic Programming AND-parallelism. The approach enables the partition of the search space into sub-spaces of two kinds: sub-spaces where clause evaluation requires theorem proving; and sub-spaces where clause evaluation is performed quite efficiently without resorting to a theorem prover. We have also defined a new type of redundancy (Coverage-equivalent redundancy) that enables the prune of significant parts of the search space. The new type of pruning together with the partition of the hypothesis space considerably improved the performance of the APIS system. An empirical evaluation of the APIS system in standard ILP data sets shows considerable speedups without a lost of accuracy of the models constructed.

1 Introduction

Multi-Relational Data Mining (MRDM) addresses the important challenge of how to learn or mine the large multi-relational databases that are being developed by individuals and organizations. Inductive Logic Programming (ILP) is a well known approach to MRDM. It starts from a logic-based representation in order to induce theories that can describe common patterns in the data, or that discriminate between classes of examples. ILP benefits from the expressiveness and conciseness of logic and has been shown to be effective over a large range of applications.

As most other Multi-Relational Data Mining (MRDM) systems, ILP systems must *search* over a very large space. Controlling the running time is thus a key consideration and has become even more important as data-base size increase. Indeed, often ILP practitioners have to reduce the search space by using techniques such as sampling or strong language bias in order to actually obtain results.

G. Zaverucha et al. (Eds.): ILP 2013, LNAI 8812, pp. 93–106, 2014.
DOI: 10.1007/978-3-662-44923-3_7

There is therefore a strong motivation to making ILP systems *faster* [8]. Of the several approaches being considered, *parallelism* is a natural fit, given the widespread availability and the low-cost of modern parallel platforms. Indeed, one can argue that parallelism is nowadays fundamental in large-scale data mining. Therefore, it is unsurprising there has been much interest on parallel ILP [10].

We propose a novel algorithm for parallel ILP data-mining, APIS (**A**nd **P**aralel**IS**m). APIS takes advantage of previous work in logic programming for AND-parallelism, and takes it to the context to ILP. Results for our initial implementation look very promising.

The remainder of the paper is organized as follows. Section 2 provides a short introduction to ILP necessary to understand our proposal. It also explains the Logic Programming AND-parallelism foundation of the proposal. Section 3 describes the APIS systems. Section 4 presents an empirical evaluation of the proposed technique. We survey the parallel execution of ILP systems in Sect. 5. Finally, in Sect. 6, we draw some conclusions and describe future work.

2 Background

The fundamental goal of a predictive ILP system is to construct a model H given background knowledge B and observations E, usually called examples in the machine learning literature. The problem that a *predictive ILP* system must solve is to find a consistent and complete model H, i.e., find a set of hypotheses that *explain* all given positive examples, while being consistent with the given negative examples. More formally, given:

- B: background knowledge encoded as statements of a Logic Program.
- $\mathcal{L}$: a pre-defined language for acceptable hypotheses.
- E: a finite set of examples $= E^+ \cup E^-$ where the E^+ are named positive examples; E^- is an optional set of negative examples; and $B \not\models E^+$

the goal is to find a set of logical statements H from the set $\mathcal{L}$ of clauses that are sufficient and consistent with the examples. *Sufficiency* is defined as $B \cup H \models E^+$ and $B \cup \{H_i\} \models e_1 \vee e_2 \vee \cdots \vee e_p$ $(1 \leq i \leq k)$. *Consistency* is defined as $B \cup H \not\models \Box$ and $B \cup H \not\models \cup E^-$. The sufficiency requirements are designed to ensure that the theory H *predicts* the positive examples and that every clause h_i predicts at least one positive example. The consistency requirements try to ensure that the theory is consistent with the background knowledge, and that it is a good classifier. One particularly popular framework to constraint the clauses considered is *mode declarations*, where one assigns types to clause's arguments and says that some arguments must have been bind by a previous literal in the same clause.

Mode-Directed Inverse Entailment (MDIE) [15] takes advantage of mode declarations to constrain the ILP search space. The key idea in MDIE is to find all literals that could be used in rules that explain the example. This is achieved by selecting a seed example and then constructing the *saturated clause* from the set of all literals that could be used to prove (directly or indirectly) the example. Several ILP systems [2,9,15,22] use the saturated clause in order to anchor the search space lattice.

MDIE implementations such as Progol [15] or Aleph [22] start from a most general clause, and then enumerate the clauses that subsume the bottom-clause until finding a good clause that can be included in the theory. The search space is therefore bound by the combinations of literals in the bottom-clause and thus can grow very quickly, severely restricting the scalability of ILP systems. Several approaches have been proposed in order to address this important problem. Work has included faster evaluation of nodes in the search space [8,20], and reducing redundancy in the search space through more intelligent search or refining bias. A promising approach is to divide the search space and to use parallelism in order to improve running times, as discussed next. During the search each clause has to be evaluated by counting how many examples can be derived when the hypothesis is added to the background knowledge. This evaluation procedure requires a theorem prover and is most often the major time consuming step in the search procedure. For efficiency sake it is usual to keep track of the examples derivable by each clause (coverage lists). To avoid evaluating the refined clauses in the complete list of examples the coverage lists of the "parent clause" are usually used.

2.1 Parallel Execution of Logic Programs

There is a strong connection between parallelism in the context of ILP and parallelism in the context of logic programming (LP). Parallelism has been widely studied in LP [12], where it can be exploited implicitly, by parallelising the LP inference mechanism, or explicitly, by extending logic programs with primitives that create and manage tasks and allow for task communication.

Explicit parallelism is often implemented by interfacing to existing low-level primitives, such as Posix Threads [23], or MPI [11]. In contrast, implicit parallelism provides independency from the underlying low-level primitives. Two major sources of implicit parallelism have been recognized. In *or-parallelism*, the search in the LP system is run in parallel. Or-parallelism is known to achieve scalable speedups on current hardware [6] but it works better when we want to perform complete search, which may be expensive in the context of ILP.

And-Parallelism corresponds to running conjunctions of goals, or and-tasks, in parallel. If the goals communicate during the parallel computation, it is called *dependent and-parallelism*. Dependent and-parallelism may be used for concurrent languages or to implement pipelines [1]. On the other hand, *independent and-parallelism* (IAP) is useful in divide-and-conquer applications and often corresponds to coarse-grained tasks.

Modern IAP implementations support both shared-memory, such as thread-based systems [14], and distributed platforms [4]. Our approach is based on *independent and-parallelism* (IAP).

3 The APIS System

The APIS system is based on a new approach to the parallel execution of ILP systems. This approach establishes a partition on the hypothesis space enabling each sub-space to be executed in parallel. There are two types of sub-spaces: sub-spaces requiring theorem proving for clause evaluation; and sub-spaces that efficiently compute clause evaluation without the need of theorem proving. Not only the partition enables the parallel search but also achieves additional speedups resulting from the fact that some of the sub-spaces do not use theorem proving to evaluate the hypotheses. Although a partition is established on the hypothesis space the resulting sub-spaces are not completely independent as we explain later.

The theoretical foundation of our proposal is based on results from Logic Programming (LP) AND-parallelism.

And-Parallelism corresponds to running conjunctions of goals, or and-tasks, in parallel. *Independent and-parallelism* (IAP) is useful in divide-and-conquer applications and often corresponds to coarse-grained tasks.

It is well known in LP that if a clause has subsets of literals with literals in each subset not sharing variables with any literal of the other subsets, then each subset can be executed in parallel. When traversing the hypothesis space an MDIE-based ILP system constructs and evaluates clauses. Traditionally clause evaluation is done using a theorem prover[1]. Among the clauses constructed during the search, there are clauses that satisfy the LP IAP constraint: clauses with sets of literals that do not share variables. We can then think of a search procedure that generates in parallel each subset of literals in the "traditional" way (using theorem proving for evaluation) and then combines each sub-set to form a new clause and make the evaluation of the combined clause in a more efficient way. The coverage of the combined clause is computed by the intersection of the coverage lists of the clauses being combined. This result cannot, however, be efficiently applied in a traditional ILP system since it is computationally expensive to determine if the partition of the clause's literals into sub-sets that do not share variables exists. The key point of the APIS system approach is to analyze the mode declarations and establish the partition of the hypothesis space based on the mode declarations, thus avoiding the analysis of each clause for independent sets of literals at induction-time. Such partition can be computed as a pre-processing step in an efficient way. The overall process is therefore divided in two steps: a pre-processing step where mode declarations are used to establish the partition of the hypothesis space; and the execution in parallel of the sub-spaces resulting from the previous step. We now explain each step in detail.

Definition 1. *Island. An island is a set of mode declarations satisfying the following two conditions. Each mode declaration shares at least one type with other modes in the same island. Each mode declaration does not share any type*

[1] Counting the number of examples derivable from the hypothesis and the background knowledge.

with any other mode declaration outside the island. Types of the head literal are excluded from the above mentioned "type checking".

The core of the APIS system is the identification of the *islands* since they will be used in the partition of the hypothesis space. The algorithm for the automatic identification of the *islands* is described by Algorithm 1. The use of the islands in the the parallel search of the hypothesis space is described by Algorithm 2.

Algorithm 1. *Islands* computation from the mode declarations

```
 1: function ComputeIslands(AllModes)
 2:     IslandsSet $\leftarrow \emptyset$
 3:     Modes $\leftarrow$ removeHeadInputArguments(AllModes)        ▷ pre-processing step
 4:     while Modes $\neq \emptyset$ do                              ▷ process all modes
 5:         Mode = withoutInputArguments(Modes)
 6:         Modes = Modes $\setminus$ { Mode }
 7:         Island = ExtendIsland({Mode}, Modes)
 8:         IslandsSet $\leftarrow$ IslandsSet $\cup$ { Island }
 9:     end while
10:     return IslandsSet
11: end function
12:
13: function ExtendIsland(Island, Modes)
14:     repeat
15:         Mode = LinkedToTheIsland(Modes)          ▷ returns $\emptyset$ if no mode was found
16:         Modes = Modes $\setminus$ { Mode }
17:         Island $\leftarrow$ Island $\cup$ { Mode }
18:     until Mode = $\emptyset$
19:     return Island                                      ▷ Island as a set of modes
20: end function
```

The algorithm to compute the islands accepts as input a set of mode declarations and returns a set of *islands*. First, a pre-processing is done to remove the types appearing in the head mode declaration and the mode arguments that are constants. After the pre-processing the algorithm enters a cycle where each island is determined and terminates whenever there are no more mode declarations to process. In the main cycle a seed mode is chosen to start a new *island* and then the island is "expanded". Expanding an island consists in adding any mode declaration not yet in the island sharing a type with any mode already in the island. The expansion stops as soon as there is no mode outside the island sharing a type with the modes inside the island.

APIS execution algorithm is schematized in Algorithm 2. Algorithm 2 starts by computing the *islands* and each client node is instructed to upload the data set without the mode declarations. In the line of MDIE greedy cover ILP algorithms the main cycle generates hypotheses, adds the best discovered hypothesis to the final theory and removes the examples covered by the added hypothesis. The cycle is repeated until no uncovered positive examples are left. The specificity of APIS is evident in (steps 8 through 19). In this part of the algorithm APIS uses a pool of client nodes and a pool of sub-spaces of the hypothesis space to search (determined by the partition made on the mode declarations). Each node searches a sub-space. There are two kinds of sub-spaces: "saturation-based" sub-spaces; and "combination-based" sub-spaces. A saturation-based sub-space

Algorithm 2. The APIS parallel execution algorithm

```
1: function InduceTheory(DataSet, Clients)
2:     Islands ← ComputeIslands(GetModes(DataSet))
3:     Theory ← $\emptyset$
4:     Examples ← PositiveExamples(DataSet)                    ▷ initial positive examples
5:     broadCast(Clients, loadIslandsDataSets)
6:     while Examples $\neq \emptyset$ do                        ▷ while not covering all positives
7:         Samples = getSample(Examples)
8:         Jobs ← getJobs(Islands, Samples)
9:         while Jobs $\neq \emptyset$ do                        ▷ all islands processed in the cycle
10:            if Clients $\neq \emptyset$ then
11:                W ← client(Clients)                          ▷ get next available client
12:                Clients ← Clients \ { W }
13:                J ← nextJob(Jobs)                            ▷ select a non-processed job
14:                Jobs ← Jobs \ { J }
15:                sendMsg(W, J)                                ▷ client W processes job J
16:            end if
17:            if FinishedClient(C) $\neq \emptyset$ then Clients ← Clients ∪ { C }
18:            end if
19:        end while
20:        h = IslandsResults()                                 ▷ returns the best hupothesis
21:        Covered = Cover(h, Examples)                         ▷ compute h coverage
22:        Examples = Examples \ Covered
23:        if Examples $\neq \emptyset$ then broadcast(Clients, removeExamples(Covered))
24:        end if
25:        Theory ← Theory ∪ { h }
26:    end while
27:    return Theory
28: end function
```

is generated as in a typical saturation followed by reduction steps that characterize MDIE systems. The difference is that to generate the sub-space a sub-set of the mode declarations (an *island*) is used. All clauses constructed in this kind of subspace are evaluated by proving the examples from background knowledge and the hypothesis under evaluation. On the other hand in "combination-based" sub-spaces theorem proving is not required. Each clause constructed in a combination-based sub-space merges pairs of clauses each one coming from previously searched spaces that do not share islands. This restriction allows the evaluation of the new clauses by intersection of the parent's coverage lists. We can see that there is a dependency among combination-based sub-spaces. The saturation-based sub-spaces are the only ones completely independent. Let us further remark that in the main cycle of the algorithm we search several hypothesis spaces at the same time[2]. We have an hypothesis space for each example of the seed. All of the jobs to execute (sub-spaces to be searched) are in a common pool but only sub-spaces belonging to the same example are combined. The number of jobs associated with each example is equal to the number of all possible combinations of the islands up to the clause length. First the saturation-based sub-spaces are generated, then these sub-spaces are combined in pairs them in groups of three and so on up to the "clause length" value. The combinations are all computed once before execution of the algorithm and each sub-space is schedule to run as soon as the two "parents" finish.

[2] As many as the size of the sample.

3.1 Redundancy Avoidance

It is well known that there is a lot of redundancy among the hypotheses in an ILP search space. Several types and remedies have been identified and proposed, see [19]. With the APIS approach there is a another redundancy situation that can be avoided and therefore improving the search.

As explained previously, if a clause is "constructed" by combining two clauses from different *islands*, its coverage is computed by the intersection of the two coverage lists of the clauses being combined. The coverage result depends only on the coverage lists of the combining clauses and not on the clauses *per se*[3]. If we have clause C_1 and clause C_2 with the same positives and negatives coverage lists originated from the same *island* and we try to combine each of them with clause C_3, from a different island, we will necessarily obtain two clauses (C_1 "+" C_3 and C_2 "+" C_3) with the same coverage lists (positives and negatives). Combining each of C_1 or C_2 with clauses from other *islands* will always result in clauses with equal coverage lists. We call such clauses (C_1 and C_2) **coverage equivalent clauses**.

Definition 2. Coverage-equivalent clauses. *Two clauses C_1 and C_2 are coverage equivalent if both cover exactly the same positive and negative examples.*

Although coverage equivalent clauses may not be equivalent in the logic sense, a coverage-based ILP system will always report only one exemplar of the coverage equivalent class. In the APIS system we keep only one exemplar of each coverage equivalent classes (the shortest clause).

Coverage equivalence is used in APIS for pruning in the following way. During the search of a sub-space the inconsistent clauses are stored in a file. The purpose is to combine them with other inconsistent clauses from other sub-spaces. Pruning takes place at saving time. From each coverage equivalence class only a single clause is saved.

4 Experiments and Results

4.1 Experimental Settings

We have used four data sets to evaluate the APIS system. DBPCAN is part of the water disinfection by-products database and contains predicted estimates of carcinogenic potential for 178 chemicals. The goal is to provide informed estimates of carcinogenic potential to be used as one factor in ranking and prioritizing future monitoring, testing, and research needs in the drinking water area [24]. The second data set is CPDBAS, the Carcinogenic Potency Data Base that contains detailed results and analyzes of 6540 chronic, long term carcinogenesis bio assays.

[3] Opposite from what happens when literals share variables.

A description of the background knowledge for these two data sets[4] can be found in [3]. Other two data sets used in this study are the carcinogenesis and mutagenesis well known in ILP and can be found in the Oxford University Machine Learning repository[5] along with an explanation of the domain that produced the data.

The data sets are characterized in Table 1 together with the associated Aleph's parameters used in the experiments. The nodes limit parameter indicated in the table concern the sequential execution value. When running APIS we have divided the nodes limit among the saturation-based sub-spaces. For each saturation-based sub-space the nodes limit is a weighted proportion of the nodes limit of the sequential execution. The weight used is based on the number of mode declaration of the corresponding island. For instance, let us consider the carcinogenesis data set. The nodes limit is set to 1 million (1M) clauses in the sequential execution. Four islands where identified hence the nodes limits in the saturation-based sub-spaces were the following ones: 3/34 * 1M for island 1; 24/34 * 1M for island 2; 4/34 * 1M for island 3 and 3/34 * 1M for island 4. In carcinogenesis there are 34 mode declarations, 3 in island 1, 24 in island 2, 4 in island 3 and 3 in island 4. The nodes limit used in the sequential execution is the overall nodes limit used by APIS for each example. In the current experiments the overall nodes limit is split among the saturation-based sub-spaces according to the number of mode declaration in their island. If the saturation-based sub-spaces did not reach their nodes limit (what happens frequently for some of them) the combination-based sub-spaces can run and use the number of nodes not used by the saturation-based sub-spaces. As said before, for each example the global limit, used in the sequential execution, is never surpassed by the complete set of sub-spaces searched.

Table 1. Characterization of the data sets used in the study. In the cells of the second column P/N represents the number of positive examples (P) and negative examples (N). The 5 right most columns are the values for Aleph's parameters.

data set name	number of examples	number of *islands*	clause length	nodes (Millions)	noise	minimum positives	sample size
carcinogenesis	162/136	4	5	0.5	10	12	30
mutagenesis	125/63	5	6	1	4	9	25
dbpcan	80/98	37	7	1	2	5	30
cpdbas	843/966	37	6	0.1	150	150	5

All the experiments were carried out on a cluster of 8 nodes having two quad-core Xeon 2.4 GHz and 32 GB of RAM per node and running Linux Ubuntu 8.10.

[4] Source data for both data sets is available from the Distributed Structure-Searchable Toxicity (DSSTox) Public Data Base Network from the U.S. Environmental Protection Agency http://www.epa.gov/ncct/dsstox/index.html,accessedDec2008.

[5] http://www.cs.ox.ac.uk/activities/machlearn/applications.html

Table 2. Speedups (a) and accuracy (b) obtained in the experiments numbers in each cell correspond to average and standard deviation (in parenthesis). There is no statistical difference ($\alpha \leq 0.05$) between the sequential execution accuracy values and the parallel execution for each data set.

data set	**number of worker nodes**			
	2	4	6	7
carcinogenesis	4.8(2.1)	5.6(2.7)	6.7(2.6)	6.1(2.6)
mutagenesis	76.5(32.9)	138.9(82.7)	188.4(119.3)	231.3(148.6)
dbpcan	13.8(2.5)	26.7(4.3)	36.5(5.5)	41.1 (5.9)
cpdbas	18.3(7.0)	31.4(16.1)	36.2(26.9)	28.5(11.8)

(a)

data set	**sequential execution**	**number of worker nodes**			
		2	4	6	7
carcinogenesis	53.7(3.8)	58.9(5.5)	57.8(3.8)	57.8(4.8)	58.0(7.6)
mutagenesis	84.1(6.9)	80.7(5.4)	82.0(4.8)	80.9(5.2)	81.3(4.7)
dbpcan	87.9(5.0)	89.8(4.1)	89.3(5.1)	89.3(5.1)	89.3(5.1)
cpdbas	54.0(1.8)	51.2(1.4)	53.6(1.2)	53.5(1.2)	53.4(1.0)

(b)

To estimate the predictive quality of the classification models we compute the average values (speed-up and accuracy) of 10 (70 %/30 %) train/test splits. The ILP system used was Aleph 5.0 [22].

4.2 Results and Discussion

Overall, the results show that significant speedups were achieved by APIS, well beyond the number of processors (Table 2(a))[6] without affecting accuracy (no statistical significant difference for $\alpha \leq 0.05$), Table 2(b). To understand the results a second set of experiments were performed with several sorts of countings on all parts of the APIS system. In these second set of experiments we have measured the execution times of all sub-spaces, we have counted the number of constructed clauses and the number of pruned clauses (shown in Table 3).

We have focus our initial attention on the saturation-based sub-spaces since their running times and number of nodes searched are much larger than the intersection-based sub-spaces. Results in Table 3 concern the saturation-based sub-spaces only.

We can observe that the number of clauses constructed by APIS (in the saturation-based sub-spaces) is smaller than in the sequential execution. For example, in the mutagenesis data set the whole number of clauses constructed in saturation-based sub-spaces are 20 % of the number in sequential runs. This is due to the lower limit imposed in each sub-space and because some of those sub-spaces do not reach the nodes limit. The accuracy values are similar (Table 2(b)) despite the reduction in the total number of nodes searched.

[6] Except for the carcinogenesis data set.

The major contribution for the speedups is, however, from the parallel search of the sub-spaces. We identified two sources of the parallel execution on the speedups. With enough CPUs (number of workers larger than the number of the islands) the execution time would be broadly determined by the slower sub-space search. For example, in mutagenesis data set, if we have more than 5 CPU workers we can search the five saturation-based sub-spaces in parallel. The overall time is determined by the slower search. With this effect alone we would expect the speedups to be close to the speedup of the search in the slower subspace. In the first result's column of Table 3 we can see, for example, that the slowest sub-space in mutagenesis has a speedup of 10.8 when compared with the sequential run.

Looking at the global data sets speedup results we see that the speedup of the slowest sub-space search alone does not explain the global speedups obtained. Again, looking at the results columns 4th and 5th in Table 3, we can see in column 4 the number of "slow" sub-spaces (1 in mutagenesis and 3 in dbpcan, for example) and can also see in column 5 that the other sub-saves use less than 10 % of the time of the slower ones. The is there are a one or few "slow" sub-spaces and their run time is much larger than the others. This means that we can start processing the next example much earlier than the finish time of the slower sub-space. In practice we can run several examples in parallel. This is also a significant contribution for the global speedup.

Another contribution, although weaker, for the speedup results is the use of intersection of coverage lists instead of theorem-proving. The number of clauses evaluated using intersection of coverage lists is rather small (when compared with the theorem-proving case) but represent also a faster method to evaluate clauses.

Table 3. Execution statistics. Column two shows the average (and standard deviation) of the quotient between the sequential run time of an example and the slowest sub-space (speedup). Column three sown the percentage of nodes constructed by APIS in the saturation-subspaces and the nodes constructed in the sequential run. Column four shows the number of "slow" subs-paces (left) and total number of saturation-based subspaces (right). Column 5 shows the average run time of all sub spaces (except the slowest ones) as a percentage of the slowest run time. The last column shows the coverage equivalence pruned nodes as a percentage of the total number of nodes constructed. Results concern the saturation-based sub-spaces only. Execution times and nodes constructed are negligible when compared with saturation-based subspace's values.

data set name	Slowest sbsp. speedup	nodes constructed in the sbspcs. (%)	Number of "slow" sbsps.	Av. other sub-spaces (%)	Coverage Equiv. pruning
carcinogenesis	2.7(1.8)	29	1/4	2(5)	12(4)
mutagenesis	10.8(8.7)	20	1/5	7(0)	16(10)
dbpcan	15.3(11.1)	80	3/37	1(1)	27(8)
cpdbas	5.3(5.8)	16	1/37	1(1)	18(5)

Table 4 show the island's membership of predicates that appear in the sequential execution theories.

Table 4. Island's membership of the predicates found in the clauses of the theories of sequential execution. N means a clause with all predicates in island N, N-M means a clause with predicates belonging to islands N and M, and N-M-L means a clause with predicates belonging to islands N, M and L. A list of the island's predicates can be found in Table 5 of the Appendix.

data set name	**islands ids**
carcinogenesis	1 , 1-3, 1-4, 2-3, 3-4, 1-2-3
mutagenesis	3, 4, 2-3, 2-4, 3-4, 1-3-4
dbpcan	1, 1-2
cpdbas	1, 2, 1-2

5 Parallel Execution of ILP Systems

Based on the principal performance bottlenecks for ILP systems identified in Sect. 1, we classify three main sources of parallelism in ILP systems [10].

Search parallelism arises from the need to enumerate clauses. We can further distinguish between parallel execution of multiple searches, and the parallel execution within a search. The granularity of the latter is substantially finer than the former. This strategy was the first to be exploited, as an extension of Dehaspe and De Raedt's Claudien system [7]. It is also exploited by Ohwada *et al.* [18] and by Wielemaker and Srinivasan in the context of randomised search [23].

Evaluation parallelism arises from the need to compute the utility of a clause. This usually requires determining the subset of E entailed by the D_i given B and H_{i-1}. A coarse-grained strategy involves partitioning E into blocks. The blocks are then provided to individual processors, which compute the examples covered in the block. Ohwada and Mizoguchi [17] implement evaluation and search parallelism in the context of inverse entailment.

Data parallelism arises when individual processors are provided with subsets of the examples prior to invoking the search procedure in Figure. Wang and Skillicorn [21] use this technique to parallelise the Progol algorithm [16]. They also use search and evaluation parallelism. Matsui *et al.* [13] compared search and evaluation parallelism, with initial promise for data-parallelism.

Notice that other classification criteria can be used. For example, as for LP systems, we can divide strategies into those that expect to use shared memory and those that expect to use distributed memory. Clare and King's Polyfarm [5] is an example of a system designed for distributed environments. Fonseca*et al.*'s survey of parallel ILP systems [10], reports that most of the best results for parallel ILP were obtained on shared-memory architecture, but argues that there is scope for experimenting with distributed-memory "clusters".

6 Conclusions

A new ILP system based on the partition of the hypothesis space and parallel search of the generated sub-spaces was presented. The partition of the hypothesis space results in two types of sub-spaces: “saturation-based and “combination-based” sub-spaces. Saturation-based sub-spaces are searched as in a “traditional” MDIE-based system. Combination-based sub-spaces combine clauses from two previously searched sub-spaces and evaluate them efficiently by intersection of the coverage lists of the clauses being combined. Using the process of combination of clauses a new type of redundancy was identified and implemented. Results of the APIS system, on well known data sets, show very good speed-ups without lost in accuracy. We are currently performing further runs in order to achieve good speedups without any decrease in accuracy. The procedure taken consists in finding a reasonable way of determining the “nodes” limit for the sub-spaces. This limit is specially critical for the saturation-based sub-spaces since they produce the initial set of clauses that are being combined in other sub-spaces. If node limit is too small we may loose crucial (sub-)clauses important for the combination process.

Acknowledgments. This work has been partially supported by Fundação para a Ciência e Tecnologia (FCT) through the project ADE (PTDC/EIA-EIA/121686/2010 (FCOMP-01-0124-FEDER-020575)). The work was also partial supported by project NORTE-07-0124-FEDER-000059, financed by the North Portugal Regional Operational Programme (ON.2 O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by national funds, through the Portuguese funding agency, FCT.

A Composition of the Dataset’s Islands

Table 5 shows the partial composition of the islands that where used to define the hypothesis sub-spaces. In the table we show only the predicates that appear in the models constructed in the sequential execution runs.

Table 5. Island's membership of the predicates that appear in the final theories induced by the APIS system.

data set name	**island**			
	1	**2**	**3**	**4**
carcinogenesis	ames/1 has_property/3 mutagenic/1	ashby_alert/3 ether/2 ar_halide/2 non_ar_6c_ring/2 non_ar_hetero_5_ring/2	atm/5 lteq/2 gteq/2	ind/3 lteq/2
mutagenesis	ring_size_5/2	logp/2 gteq/2	lumo/2 lteq/2	atm/5 bond/4 gteq/2 lteq/2
dbpcan	chemical_fingerprint/2 rotatable_bondcount/2 primary_carbon/2 atLeastOneOfFuncGroups/2 resonant_count/2 tertiary_carbon/2 primary_carbon/2 secondary_carbon	pharmacophore_fingerprint/4 ltPharmacophoreArg3/2 ltPharmacophoreArg2/2 gtPharmacophoreArg2/2		
cpdbas	atLeastOneOfFuncGroups/2 heteroaromatic_ringcount/2 fusedaliphatic_ringcount/2 tertiary_carbon/2 tautomer_count/2 ringcount/2 tertiary_carbon/2	pharmacophore_fingerprint/4 ltPharmacophoreArg2/2 gtPharmacophoreArg/2		

References

1. Bone, P., Somogyi, Z., Schachte, P.: Estimating the overlap between dependent computations for automatic parallelization. TPLP **11**(4–5), 575–591 (2011)
2. Camacho, R.: IndLog — induction in logic. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 718–721. Springer, Heidelberg (2004)
3. Camacho, R., Pereira, M., Costa, V.S., Fonseca, N.A., Adriano, C., Simoes, C.J.V., Brito, R.M.M.: A relational learning approach to structure-activity relationships in drug design toxicity studies. J. Integr. Bioinform. **8**(3), 182 (2011)
4. Casas, A., Carro, M., Hermenegildo, M.V.: A high-level implementation of non-deterministic, unrestricted, independent and-parallelism. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 651–666. Springer, Heidelberg (2008)
5. Clare, A.J., King, R.D.: Data mining the yeast genome in a lazy functional language. In: Dahl, V. (ed.) PADL 2003. LNCS, vol. 2562, pp. 19–36. Springer, Heidelberg (2002)
6. Costa, V.S., de Castro Dutra, I., Rocha, R.: Threads and or-parallelism unified. TPLP **10**(4–6), 417–432 (2010)
7. Dehaspe, L., De Raedt, L.: Parallel inductive logic programming. In: Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases (1995)
8. Fonseca, N.A., Costa, V.S., Rocha, R., Camacho, R., Silva, F.: Improving the efficiency of inductive logic programming systems. Softw. Pract. Exper. **39**(2), 189–219 (2009)

9. Fonseca, N.A., Silva, F., Camacho, R.: April – an inductive logic programming system. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 481–484. Springer, Heidelberg (2006)
10. Fonseca, N.A., Srinivasan, A., Silva, F.M.A., Camacho, R.: Parallel ilp for distributed-memory architectures. Mach. Learn. **74**(3), 257–279 (2009)
11. The MPI Forum: Mpi: a message passing interface (1993)
12. Gupta, G., Pontelli, E., Ali, K.A.M., Carlsson, M., Hermenegildo, M.V.: Parallel execution of prolog programs: a survey. ACM Trans. Program. Lang. Syst. **23**(4), 472–602 (2001)
13. Matsui, T., Inuzuka, N., Seki, H., Itoh, H.: Comparison of three parallel implementations of an induction algorithm. In: 8th International Parallel Computing Workshop, Singapore, pp. 181–188 (1998)
14. Moura, P., Crocker, P., Nunes, P.: High-level multi-threading programming in logtalk. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 265–281. Springer, Heidelberg (2008)
15. Muggleton, S.: Inverse entailment and Progol. New Gener. Comput., Spec. Issue Induct. Log. Program. **13**(3–4), 245–286 (1995)
16. Muggleton, S., Firth, J.: Relational rule induction with CProgol4.4: a tutorial introduction. In: Džeroski, S., Lavrač, N. (eds.) Relational Data Mining, pp. 160–188. Springer, Heidelberg (2001)
17. Ohwada, H., Mizoguchi, F.: Parallel execution for speeding up inductive logic programming systems. In: Arikawa, S., Nakata, I. (eds.) DS 1999. LNCS (LNAI), vol. 1721, pp. 277–286. Springer, Heidelberg (1999)
18. Ohwada, H., Nishiyama, H., Mizoguchi, F.: Concurrent execution of optimal hypothesis search for inverse entailment. In: Cussens, J., Frisch, A.M. (eds.) ILP 2000. LNCS (LNAI), vol. 1866, pp. 165–173. Springer, Heidelberg (2000)
19. Costa, V.S., Srinivasan, A., Camacho, R., Blockeel, H., Demoen, B., Janssens, G., Struyf, J., Vandecasteele, H., Van Laer, W.: Query transformations for improving the efficiency of ILP systems. J. Mach. Learn. Res. **4**, 465–491 (2003)
20. Camacho, R., Blockeel, H., Demoen, B., Janssens, G., Struyf, J., Vandecasteele, H., van Laer, W.: Query Transformations for Improving the Efficiency of ILP Systems. J. Mach. Learning Res. Ashwin Srinivasan **4**, 465–491 (2003)
21. Skillicorn, D.B., Wang, Y.: Parallel and sequential algorithms for data mining using inductive logic. Knowl. Inf. Syst. **3**(4), 405–421 (2001)
22. Srinivasan, A.: The Aleph Manual (2003). http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph
23. Wielemaker, J.: Native preemptive threads in SWI-prolog. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 331–345. Springer, Heidelberg (2003)
24. Woo, Y.T., Lai, D., McLain, J.L., Manibusan, M.K., Dellarco, V.: Use of mechanism-based structure-activity relationships analysis in carcinogenic potential ranking for drinking water disinfection by-products. Environ. Health Perspect. **110**, 75–87 (2002)