Parallel algorithms for multi-relational data mining: application to life science problems

Rui Camacho[1], Jorge G. Barbosa[1], Altino Sampaio[2], João Ladeiras[1], Nuno A. Fonseca[3], and Vítor S. Costa[4]

[1] DEI & Faculty of Engineering of University of Porto, Portugal
[2] IPP, Escola Superior de Tecnologia e Gestão de Felgueiras, CIICESI
[3] EMBL-European Bioinformatics Institute, Wellcome Trust Genome Campus, Hinxton, CB10 1SD, UK
[4] DCC & Faculty of Sciences of University of Porto, Portugal

Abstract. Data Mining (DM) algorithms are able to construct models from available data that can be very useful both for business and for science. However, a powerful representation language is required to express the highly complex models and that stem from structured data. Multi-relational algorithms are adequate for such tasks since they use a rich representation language for both data and models. However, for very large or highly complex domains multi-relational algorithms may require long running times. This drawback can be substantially reduced by using parallel implementations. In this chapter we present a survey on parallel approaches to run Inductive Logic Programming (ILP), a flavour of multi-relational algorithms. We also analyze different scheduling approaches for those implementations and describe two applications where the proposed approaches may be very useful.

1 Introduction

The amount of data stored nowadays in data bases is huge and increases every year at a very fast pace. The analysis of such data can be very useful for both business and research. However, in order to analyze large amounts of data or address highly complex problems, computational-based tools are required. *Knowledge Discovery in Databases* (KDD)[15] aims at the discovery of patterns that are both novel and potentially useful. In some applications the comprehensibility of the pattern is also a requirement. The KDD process encompasses a series of steps, one of them being the Data Mining (DM) step. In the DM step, algorithms based on Machine Learning (ML) and Statistics, among others, are used to construct models from the data. One may classify the ML algorithms into two groups. Those that require the input data to be contained in a single table of a relational database, and those that can handle directly all the tables in a database. For simplicity, let Us denominate the former ones as *propositional learners* and the latter ones, as *multi-relational learners*. Multi-relational learners are the focus of this chapter and, for simplicity sake, we will use, from now on, a shorter name, that is *relational learners*.

One of the most well known flavours of relational learning is Inductive Logic Programming [37,42] (ILP). ILP has been used to construct highly sophisticated models that address very diverse tasks. Examples include Structure-activity prediction [26,59], a major challenge in rational drug design; Natural language understanding with Grammar acquisition [62]; Proteim secondary structure prediction [27]; Qualitative model identification in naive qualitative physics [6]; Workload prediction in computer networks [1]; Expected survival time of kidney transplanted patients [52].

The success of ILP in the above mentioned applications is due to the following features. First, ILP can very naturally accept background knowledge that can be integrated into the constructed models. ILP learning can harmoniously combine numerical and symbolic computations, while being able to handle structured data. And finally, and very important, it has capacity of building highly comprehensible models even for complex tasks.

In the remaining of this chapter we introduce the basic concepts of ILP necessary to understand the rest of the text (Section 2). We then survey the main approaches to take advantage of parallel execution to speedup ILP systems (Section 3). A scheduling algorithm is proposed in Section 4 to improve execution of current ILP implementations. In Section 5 we present applications where the scheduling techniques proposed here will most benefit the parallel execution. Finally, we present a summary of the chapter and draw some conclusions in Section 6.

2 ILP Basic Concepts

We shall adress ILP within the broader area within Machine Learning (ML) called *supervised learning*. This learning task may be stated in a set-theoretic perspective. It aims at learning an intensional description of a certain set in a universe of elements (U), given that we are aware that some elements belong in this set (positive examples), and of others that do not (negative examples). The intensional description we strive to lern is called the **concept**. In an ILP setting the concept is usually referred to as the **hypothesis**. Elements known to be in the target set are called instances of the concept . Let Q^+ denote the target set. Elements in $Q^- = \{\neg x \mid x \in U \setminus Q^+\}$ are called **negative instances** or **counter-examples** and are elements of the universe that are not instances of the target concept.

Note that both or either Q^+ and Q^- may be infinite. Learning systems usually consider finite subsets of Q^+ and Q^-. These finite subsets will be denoted by E^+ ($\subseteq Q^+$) and E^- ($\subseteq Q^-$). If not stated otherwise E^+ will be referred as the *positive examples* and E^- will be referred as the *negative examples*.

ILP framework

ILP is concerned with the generation and justification of hypotheses from a set of examples making use of prior knowledge. The representation most often

used is Horn clauses, a subset of First-Order Predicate Calculus. The induced hypotheses are represented as a finite set or conjunction of clauses denoted by H. H is of the form $\mathrm{h}_1 \wedge \ldots \wedge \mathrm{h}_l$ where each h_i is a nonredundant clause. The prior knowledge, also called *background knowledge*, will be denoted by B. B is described in the same language, that is, it is a finite set or conjunction of clauses, $\mathrm{B} = \mathrm{C}_1 \wedge \ldots \wedge \mathrm{C}_m$, typically definite clauses. In the ILP setting a positive example is often represented by a positive unit ground clause, also known as a *ground atom*; this largely corresponds to a data-base tuple. E^+ is a conjunction of ground atoms. $\mathrm{E}^+ = \mathrm{e}_1^+ \wedge \mathrm{e}_2^+ \wedge \ldots \wedge \mathrm{e}_n^+$ where e_i^+ is an individual positive example.

Negative examples are typically negative unit ground clauses. E^- represents the conjunction of negated ground atoms. If we denote a negative example by $\overline{f_i}$ then $\mathrm{E}^- = \overline{f_1} \wedge \overline{f_2} \wedge \ldots \wedge \overline{f_r}$. $\mathrm{E}^+ \wedge E^-$ is the training set.

Notice that ILP systems can use non-ground examples and not all of them need negative examples to arrive at a concept description [40].

In order to induce or learn, in the ILP framework we must meet the following conditions. First, we must ensure **consistency** conditions, so that the background B and the training set are consistent.The first two conditions are:

$$B \not\models \square$$

$$E^+ \wedge E^- \not\models \square$$

In other words, B should not logically imply any of the negative examples, and the positive and negative examples shsould be disjoint. This condition is called **prior satisfiability** [38].

Moreover, the positive examples must be consistent with the background, and there must be a need for a model:

$$B \wedge E^+ \not\models \square.$$

$$B \not\models E^+.$$

says that there is no point in learning if the examples already explain the model, or **prior necessity** condition[38].

The induced hypotheses should satisfy the **posterior satisfiability** condition [38]:

$$B \wedge H \wedge E^- \not\models \square.$$

That means that the hypotheses found should be consistent with the negative examples.

The set of hypotheses should not be vacuous and explain the positive examples, as stated by the **posterior sufficiency** condition [38]:

$$B \wedge H \models E^+$$

The last condition states that each hypothesis $\mathrm{h}_i \in \mathrm{H}$ should not be vacuous. This condition is called **posterior necessity** condition:

$$B \wedge h_i \models e_1^+ \vee e_2^+ \vee \ldots \vee e_n^+ \quad (\forall h_i, h_i \in H)$$

As an example imagine that the system is learning the concept of a virtuoso player. Consider that the information given to the system is the following.

$$\mathbf{B} \begin{cases} plays\_instrument(glenn\_gould, piano) \leftarrow \\ plays\_instrument(david\_oistrach, violin) \leftarrow \\ plays\_instrument(fisher, piano) \leftarrow \\ plays\_instrument(john, violin) \leftarrow \\ performance(glenn\_gould, piano, superb) \leftarrow \\ performance(david\_oistrach, violin, superb) \leftarrow \\ performance(fisher, piano, lousy) \leftarrow \\ performance(fisher, chess, superb) \leftarrow \\ performance(john, violin, lousy) \leftarrow \end{cases}$$

$$\mathbf{E}^{+} \begin{cases} virtuoso(glenn\_gould) \leftarrow \\ virtuoso(david\_oistrach) \leftarrow \end{cases}$$

$$\mathbf{E}^{-} \begin{cases} \leftarrow virtuoso(fisher) \\ \leftarrow virtuoso(john) \end{cases}$$

The previously stated prior conditions are met. B is trivially consistent. $E^{+} \wedge E^{-}$ are also trivially consistent. B does not logically entail the negative examples or any of the positive examples since the predicate symbol *virtuoso* does not appear in B.

A possible hypothesis generated by an ILP system could be the following single clause:

virtuoso(Player) ←
    plays_instrument(Player, Instrument) ∧
    performance(Player, Instrument, superb).

The hypothesis found satisfies the posterior conditions. $B \wedge H$ does not logically entail E^{-} since neither *fisher* nor *john* are capable of a superb performance when playing an instrument.

A generalisation ordering

The number of hypotheses satisfying the previously stated conditions is, in general, very large and even infinite in most cases. However, it is possible to constrain the hypothesis space by imposing an ordering on the set of clauses. The learning algorithm may then take advantage of the existence of that ordering. The search space may be systematically searched and some parts of the space may be justifiably ignored during the search.

One such ordering over the set of clauses was mentioned originally in [53] and is called **subsumption**:

Definition 1. *If C and D are two distinct nonempty clauses, then C subsumes D and we write $C \preceq D$ iff there is a substitution θ such that $C\theta \subseteq D$*[5].

Definition 2. *A clause C is* **subsumption equivalent** *to a clause D and we write $C \equiv_s D$ iff $C \preceq D$ and $D \preceq C$. A clause is* **reduced** *if it is not subsumption equivalent to any proper subset of itself.*

Subsumption is the most common ordering over the set of clauses used in ILP systems. θ-subsumption ([46]) is usually the name used in ILP to refer to the concept of subsumption. The ordering over the set of clauses is sometimes called a **generalisation model** [7]. If not stated otherwise, the generalisation ordering assumed in the definitions of the rest of the chapter is subsumption.

The concept of redundancy follows naturally from the idea of an ordering over the set of clauses and the concept of equivalence between clauses or sets of clauses. Note, that there are two kinds of redundancy. A literal may be redundant within a clause and a clause may be redundant within a set of clauses.

Definition 3. *A literal l is* **redundant** *in clause $C \vee l$ relative to background theory B iff*

$$B \wedge (C \vee l) \equiv B \wedge C.$$

Definition 4. *A clause C is redundant in the theory $B \wedge C$ iff*

$$B \wedge C \equiv B.$$

The subsumption ordering imposes a lattice over the set of clauses.

Definition 5. *A* **lattice** *is a partially ordered set in which every pair of elements a,b has a greatest lower bound (glb) (represented by $a \sqcap b$) and least upper bound (lub) (represented by $a \sqcup b$).*

Definition 6. *A* **generalisation ordering** *(or* **generalisation model***) is a partial order*[6] *over the set of clauses. The lattice imposed by a generalisation ordering is called a* **generalisation lattice**.

The top element of the subsumption lattice is $\Box$, the empty clause. The **glb** of two clauses C and D is called the most general instance (**mgi**) and is the union of the two clauses mgi(C,D) = C ∪ D. The **lub** of two clauses C and D is called the **least general generalisation** (lgg) [46] of C and D. Under subsumption the glb and lub of clauses are unique up to a renaming of variables.

As pointed out by Mitchell [36] the task of concept learning can me mapped into a search through a space of hypothesis. A generalisation ordering is a crucial concept in ILP for it is the basis of an organised search of the hypothesis space. The search for an hypothesis is mapped into the traversal of the generalisation lattice. The traversal of the generalisation lattice is, in general, what leads to the computationally expensive nature of the learning task.

[5] We refer to John Loyd's book [30] for basic concepts and definitions of Logic Programming.

[6] A partial order is a reflexive, anti-symmetric and transitive binary relation.

generalise$(B, E, \mathcal{L}, \rho, f)$: Given background knowledge B; a finite training set $E = E^+ \cup E^-$; a predefined hypotheses language $\mathcal{L}$; a refinement operator ρ; and an utility function f, returns a hypothesis H that explains the E.

1. $i = 0$
2. $E_i^+ = E^+$, $H_i = \emptyset$
3. if $E_i^+ = \emptyset$ return H_i otherwise continue
4. increment i
5. $Train_i = E_{i-1}^+ \cup E^-$
6. $D_i = search(B, H_{i-1}, Train_i, \mathcal{L}, \rho, f)$
7. $H_i = H_{i-1} \cup \{D_i\}$
8. $E_p = \{e_p : e_p \in E_{i-1}^+ \; s.t. \; B \cup H_i \models \{e_p\}\}$.
9. $E_i^+ = E_{i-1}^+ \backslash E_p$
10. Go to Step 3

Fig. 1: An ILP implementation using a greedy cover set procedure. The final set H is constructed by progressively finding the next best clause in Step 6 (this is the clause with the highest utility). The search for this clause is some generic search procedure that returns the best clause that meets the requirements (previoulsy stated in this section)

3 Parallel algorithms for ILP

Based on the principal performance bottlenecks for ILP systems, we classify three main sources of parallelism in ILP systems [16].

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 & King's Polyfarm [9] is an example of a system designed for distributed environments. Fonseca*et al.*'s survey of parallel ILP systems [16], 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".

Search. We can distinguish here between parallel execution of multiple searches, and the parallel execution within a search. The granularity of the latter is substantially finer than the former.

Data. In this, individual processors are provided with subsets of the examples prior to invoking the search procedure in Figure 1. We distinguish between two forms of parallel execution, with different communication requirements. In the first, each processor completes its search and returns the best clause. The set of all clauses are then examined in conjunction and a final result constructed by re-computing the utility of each clause using all the data. In the second approach, as each processor finds a good clause, its utility in the final set is re-computed using all the data. The granularity of the second approach is finer than the first, but it has the advantage that all clauses found will also be in the final set of clauses (in both cases, recursive clauses cannot be identified reliably).

Evaluation. The search procedure invoked in Figure 1 evaluates the utility of a clause. This usually requires its "coverage", which means 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. The final coverage is obtained by the union of examples entailed in each block. There is a similarity to the coarse data-parallelism strategy described above. There too processors are provided with subsets of the data. There, the subsets are used to identify different clauses. Here, the subsets are used to evaluate a given set of clauses. A fine-grained strategy would involve determining subsets of literals in each D_i that can be evaluated independently (this could be identified, for example, using the "cut" transformation described in [54]). Each such independent subset is then evaluated on a separate processor and the final result obtained by the intersection of examples entailed by the subsets.

The three strategies above are not mutually exclusive. In fact, a parallel algorithm may exploit more than one. Furthermore, it should also be evident that the parallel algorithms can be classified in many different ways. For instance, we could also classify the parallel algorithms regarding their *correct*, i.e. do they produce the same solution (correct) as the corresponding sequential algorithm. Next, we will focus our classification of previous work on parallel ILP systems on the three strategies mentioned above, along with the hardware architecture. Figure 2 shows a brief summary of the entries that follow.

Parallelism	Architecture	
	Shared-Memory	Distributed-Memory
Search	Dehaspe & De Raedt [13] Ohwada & Mizoguchi [43] Ohwada *et al.* [44] Wielemaker [61]	No reports
Data	Wang & Skillicorn [55] Graham *et. al* [22]	Matsui *et al.* [32] Clare & King [9] Blaták & Popelínský [4]
Evaluation	Ohwada & Mizoguchi [43] Graham *et al.* [22]	Matsui *et al.* [32] Konstantopoulos [28]

Fig. 2: Parallel ILP systems reported in the literature.

Dehaspe and De Raedt [13] developed the first parallel ILP system that we are aware of. The system is a parallel implementation of Claudien, an ILP system capable of discovering general clausal constraints. The strategy is based on the parallel exploration of the search space where each processor keeps a pool of clauses to specialize, and shares part of them to idle processors (processors with an empty pool). In the end, the sets of clauses found in each processor are

combined. The system was evaluated on a shared-memory computer with two datasets and exhibited a linear speedup up to 16 processors.

Ohwada and Mizoguchi [43] have implemented an algorithm based on inverse entailment [39]. The implementation uses a parallel logic programming language and explored the parallel evaluation of clause coverage, and two strategies for search parallelisation (parallel exploration of independent hypotheses and parallel exploration of each refinement branch of a search space arising from each hypothesis). The system was applied to three variants of an email classification data set and the experiments performed evaluated each strategy. The results on a shared-memory parallel computer showed a sub-linear speedup in all strategies, although parallel coverage testing appeared to yield the best results.

An algorithm that explores the search space in parallel was first implemented by Ohwada *et al.* [44]. The set of nodes to be explored is dynamic and implemented using contract-net communication [56]. Their paper investigated two types of inter-process communication, with results showing near-linear speedups on a 10 processor machine.

Wielemaker [61] implemented a parallel version of a randomized search found in the Aleph system. The parallel implementation executes concurrently several randomized local searches using a multi-threaded version of the SWI Prolog engine. Experiments examined performance as the number of processors was progressively increased. Near-linear speedups were observed up to 4 processors, however the speedup was not sustained as the number of processors increased to 16.

Wang & Skillicorn [55] have implemented a parallel version of the Progol algorithm [41] by partitioning the data and applying a sequential search algorithm to each partition. Data are partitioned by dividing the positive examples among all processors and by replicating the negative examples at each processor. Each processor then performs a search using its local data to find the (locally) best clause. The true utility of each clause found is then re-computed by sharing it among all processors. Note that this algorithm exploits the parallelization strategies identified (parallel search, data and evaluation) mentioned above, thus being an example that the strategies are not mutually exclusive. Experiments with three data sets suggest linear speedups on machines with 4 and 6 processors.

A study by Matsui *et al.* [32] evaluates and compares two algorithms based on data parallelism and parallel evaluation of refinements of a clause (the paper calls this parallel exploration of the search space, although it really is a parallelisation of the clause evaluation process). The two strategies are used to examine the performance of a parallel implementation of the FOIL [47] system. Experiments are restricted to a small synthetic data set (the "trains" problem [35]) and the results show poor speedups from parallelisation of clause evaluation. Data parallelism showed initial promise, with near linear speedups up to 4 processors. Above 4 processors, speedup was found to be sub-linear due to increased communication costs.

PolyFarm was a parallel ILP system specifically designed for the discovery of first-order association rules on distributed memory machines developed by Clare

& King [9]. Data are partitioned amongst multiple processors and the system follows a master-worker strategy. The master generates the rules and reports the results and workers perform the coverage tests of the set of rules received from the master on the local data. Counts are aggregated by a special type of worker that reports the final counts to the master. No performance evaluation of the system is available.

An implementation of a parallel ILP system, using the PVM message passing library, was done by Graham *et al.* [22]. Parallelisation is achieved by partitioning the data and by parallel coverage testing of sets of clauses (corresponding to different parts of the search space) on each processor. Near-linear speedups are reported up to 16 processors on a shared memory machine.

Konstantopoulos [28] has investigated a data parallel version of a deterministic top-down search implemented within the Aleph ILP system [57]. The parallel implementation uses the MPI library and performs coverage tests in parallel on multiple machines. This strategy is quite similar to that reported in Graham *et al.*, with the caveat that testing is restricted to one clause at a time (Graham *et al.* look at sets of clauses). Results are not promising, probably due to the over-fine granularity of testing one clause at a time.

dRap was developed by Blaták & Popelínský [4]. This was a parallel ILP system specifically designed for the discovery of first-order association rules on distributed memory machines. Data are partitioned amongst multiple processors and the system follows a master-worker strategy. The master generates the partitions and each worker then executes a sequential first-order association rules learner. The master collects the rules found by the workers and then redistributes the rules by all the workers to compute the support on the whole data set. No performance evaluation of the system is reported.

Angel-Martinez & Dutra & S. Costa & Buenabad-Chávez [31] describe the use of GPUs to perform parallel evaluation of hypotheses. The authors extended the widely used Aleph system, parallelism is implemented by evaluating clauses as Datalog queries, and taking advantage of prior work in implementing the main database primitives in GPUs. Results show a one or two-order of magnitude in large data sets, where the overhead of sending a clause to a GPU is significantly less than the benefits of parallel execution.
Results reported by these papers are summarized in Figure 3. The principal points that emerge are these:

1. Most of the effort has been focused on shared-memory machines where the communication costs are lower than for distributed-memory machines.
2. Speedups observed on shared-memory machines are higher than those observed on distributed memory ones. Maximum disparity is observed with parallel execution of the coverage tests: this is undoubtedly due to the fact that communication costs are high for distributed-memory machines, and the granularity of the task is finer than other forms of parallelism.

Despite the apparently discouraging results observed to date on distributed-memory machines, we believe that a further investigation is warranted for several reasons. First, the results are not obtained from a systematic effort to investigate

Parallelism	Speedup	
	Shared-Memory	Distributed-Memory
Search	Dehaspe & De Raedt: Linear (16) Ohwada & Mizoguchi: 2–3 (6) Ohwada *et al.*: 8 (10) Wielemaker: 7 (16)†	No reports
Data	Wang & Skillicorn: Linear or better (6) Graham *et. al*:linear (16)	Matsui *et al.*: 4 (15)‡ Clare & King: – Blaták & Popelínský: –
Evaluation	Ohwada & Mizoguchi: 4 (6) Graham *et al.*: 5 (8)	Matsui *et al.*: 1 (15) Konstantopoulos: none

† linear up to 4 processors

‡ linear up to 5 processors

Fig. 3: Summary of speedups reported of parallel ILP systems. The numbers in parentheses refer to the number of processors. Neither Clare & King nor Blaták & Popelínský report any speedups.

the effect of the different kinds of parallelism. That is: results that are available are obtained from a mix of fine and coarse-grained parallelisation, on differently configured networks and with different communication protocols. Second, the availability and parallelism of shared-memory architecture machines continues to be substantially lower than distributed-memory ones (for example, distributed-memory "clusters" comprised of 10s or 100s of machines are relatively easy, and cheap, to construct). There is, therefore, practical interest in examining if significant speedups are achievable on distributed-memory architectures. In this paper, we present a systematic empirical evaluation of coarse-grained search, data and evaluation parallelisation for such architectures using a well-established network of machines (a Beowulf cluster) and a widely accepted protocol for communication (an implementation of the Message Passing Interface, or MPI [17], that can be used by applications running in heterogeneous distributed-memory architectures).

The APIS ILP system

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 [23], 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.

Two major sources of implicit parallelism have been recognized withi ILP. In *or-parallelism*, the search in the LP system is run in parallel. Or-parallelism is known to achieve scalable speedups on current hardware [10] 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 [5]. On the other hand, *independent and-parallelism* (IAP) is useful in divide-and-conquer applications and often corresponds to coarse-grained tasks. Our approach is based on *independent and-parallelism* (IAP).

The APIS system introduces a new approach to the parallel execution of ILP systems. APIS partitions the hypothesis space so that each sub-space can be executed in parallel. WE define two types of sub-spaces: standard 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. Unfortunately, although a partition is established on the hypothesis space the resulting sub-spaces are not completely independent as we discuss later.

The theoretical foundation of our proposal derives from the early results in Logic Programming's (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[7]. 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. In this case, We can then apply bottom-up techniques. We generate in parallel each subset of literals in the "traditional" way (using theorem proving for evaluation) and then combine 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

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

parallel of the sub-spaces resulting from the previous step. We now explain each step in detail.

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

The algorithm accepts as input a set of mode declarations and returns a set of *islands*. First, a pre-processing step removess 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 as Algorithm 2. Algorithm 2 starts by computing the *islands*: 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 repeats until no uncovered positive examples are left. The specificity of

Algorithm 2 The APIS parallel execution algorithm

```
1: function InduceTheory(DataSet, Clients)
2:     Islands ← ComputeIslands(GetModes(DataSet))
3:     Theory ← ∅
4:     Examples ← PositiveExamples(DataSet)                    ▷ initial positive examples
5:     broadCast(Clients, loadIslandsDataSets)
6:     while Examples ≠ ∅ do                                   ▷ while not covering all positives
7:         Samples = getSample(Examples)
8:         Jobs ← getJobs(Islands, Samples)
9:         while Jobs ≠ ∅ do                                   ▷ all islands processed in the cycle
10:            if Clients ≠ ∅ 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) ≠ ∅ 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 ≠ ∅ then broadcast(Clients, removeExamples(Covered))
24:        end if
25:        Theory ← Theory ∪ { h }
26:    end while
27:    return Theory
28: end function
```

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 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-bases 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 [8]. 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

[8] As many as the size of the sample

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.

4 Scheduling and Load balancing

Parallel implementations have been employed to significantly enhance and speed up solution search, allowing to reach high quality results with reasonable execution times even for hard-to-solve optimization problems. In this section we first address the computing platforms for parallel computing and then several approaches for accelerating ILP algorithms are discussed.

4.1 Parallel computing platforms

The execution of parallel applications demand for substantial number of computing resources, which were traditionally deployed by dedicated high-performance computing (HPC) infrastructures such as clusters, and later by Grid computing [18]. Even though Grids introduce new capabilities such as larger number of resources belonging to different administrative domains and the ability to select the best set of machines meeting the requirements of applications, there are limitations related to runtime environments for applications and accomplishment of applications' needs.

Rather than owning physical, fixed-capacity clusters, organizations have recently shifted onto Cloud computing [19] paradigm. Compared to aforementioned traditional networked computing environments, Cloud computing offers to end users a variety of services covering the entire computing stack. Clouds represent a new kind of computational model, providing better use of distributed resources, while offering dynamic flexible infrastructures and quality of service (QoS) guaranteed services. It supports various configurations (e.g., CPU, memory, I/O networking, storage) and scale capacities while abstracting resource management. In spite of this, Cloud computing has recently gained popularity as a resource platform for on-demand, high-availability, and high-scalability access to resources, using a pay-as-you-go model [3]. Some of today's major commercial Cloud providers are Amazon EC2 [2], and Google Cloud Platform [21]. Clouds rely on virtualization technology for the management of traditional data center resource provisioning. Virtualization has several benefits for scientific computing, such as provisioning of isolated computing environments on shared multicore machines. Huang et al. [24] have conducted a performance evaluation regarding the use of virtualization and have concluded that HPC applications (which are performance oriented) can achieve almost the same performance as those running in a native, non-virtualized environment.

By means of virtualization, a collection of virtual machines (VMs) run on top of physical machines (PMs) to create virtual clusters [33]. Also, a VM can be suspended and later resumed on either the same or on a different PM, which

is useful for fault tolerance and load balancing. A virtual cluster is created so a user has exclusive access to a customized virtual execution environment. The provisioning of elastic virtual clusters as on-demand pay-as-you-go resources is an essential characteristic of Clouds, endowing great flexibility and scalability for end users and their applications. In this context, resources can be dynamically allocated, expanded, shrunk, or moved, according to applications demand.

Parallel and distributed applications, as is the case of parallel implementation of ILP systems, exploit the processing power of a cluster of processors. As such, these applications can utilize the Cloud to rapidly deploy an application-specific virtual cluster infrastructure to achieve new levels of availability and scalability. Although Clouds were built primarily with business computing needs in mind, their value has been already recognized within the scientific community as a easy way to store, process, and retrieve huge data without worrying about the hardware needed. For example, Delgado et al. [14] have explored the use of Cloud computing to execute scientific applications, with a specific focus on medical image processing and computational fluid dynamics applications. Also Juve and Deelman [25] discussed many possible ways to deploy scientific applications on a Cloud, ranging from astronomy to earthquake science. The Magellan project, which takes place at the Argonne Leadership Computing Facility and the National Energy Research Scientific Computing Facility, aimed at investigating the use of cloud computing for science [48]. A diverse set of scientific data parallel applications, with no tight coupling between tasks, was running on the Magellan resources, and the results allowed to conclude that current cloud software can be used for science clouds.

4.2 Load balancing for ILP algorithms

In the specific case of parallel ILP systems considered in this chapter, islands are characterized by having diverse sizes and requiring different processing capacity needs. A task parallel approach is adequate to this problem and homogeneously scheduling the same amount of CPU resource to jobs in charge of processing diverse sized islands will result in jobs' finish time imbalances, with consequent non optimized makespan. Furthermore, if we take into account that Cloud resources are usually billed by hour [2], an inefficient schedule of parallel ILP jobs will result in increased costs to solve the parallel ILP search problem. Therefore, it is important to produce schedules of jobs that consider their diverse needs, and the heterogeneity of resources, in order to achieve the objectives of reducing the makespan (i.e., the finishing time of the last job in the system) and maximizing load balancing. To minimize makespan it is essential to allocate jobs correctly so that computer loads and communication overheads will be well balanced. In turn, load balancing aims to distribute workload between available machines to obtain as good throughput and resource utilization as possible. Despite load balancing and makespan concepts are somehow related, a good load balancing does not always lead to minimal makespan and vice versa.

Several papers address the problem of static and dynamic minimization of makespan and maximization of load balancing in parallel and distributed sys-

tems. A good review and classification of state-of-the art load balancing methods for jobs dispatched to run independently on multiple computers can be found in [45][64][67]. This section points out some relevant scheduling algorithms and strategies that we believe can improve load balancing and optimize makespan of independent and-parallelism ILP approach. When managing resources in a cloud computing environment, scheduling can be made in different layers, as described next.

Scheduling at the application level Concerning the scheduling of jobs onto VMs, Dhinesh and Krishna [29] proposed to minimize the makespan and maximize load balancing of applications in the Cloud. They use Swarm Intelligence (SI) to propose an Artificial Bee Colony (ABC) algorithm, named Honey Bee Behavior inspired Load Balancing (HBB-LB), which aims to achieve well balanced load across VMs for maximizing the throughput and minimization of makespan in a Cloud infrastructure. Cumulatively, HBB-LB algorithm not only dynamically balances the load but also considers the priorities of tasks in the waiting queues of VMs in order to minimize the time spent in the queues. The scheduling problem is solved considering that a job is a honey bee and VMs are the food sources. First, VMs are grouped on three sets: (i) overloaded VMs; (ii) underloaded VMs; and (iii) balanced VMs. Processing time of a job varies from one VM to another based on VM's capacity. Then, jobs removed from overloaded VMs have to find suitable underloaded VMs to get placed in. If there are several suitable VMs to allocate the task in, the task chooses the VM which as a less number of tasks with the same kind of priority. The winning task is allocated to the selected VM and state information is updated, which includes the workload on all VMs, number of various jobs in each VM, jobs priority in each VM, and the number of VMs in each set. These details will be helpful for other jobs, i.e., whenever a high priority job is submitted, it will consider the VM that has less number of high priority jobs to execute earlier. Once the jobs switching process is over, the balanced VMs are included into the balanced VM set. The load balancing process ends when this set contains all the VMs. This solution was successfully tested with Cloudsim [8] by means of simulation. The results have showed a good performance for heterogeneous Cloud computing systems, in terms of average execution time and waiting time of jobs on queue.

Ramezani et al. [50] contributes with a Task-based System Load Balancing method using Particle Swarm Optimization (TBSLB-PSO) that achieves system load balancing in Cloud environments by only transferring extra tasks from an overloaded VM instead of migrating the entire overloaded VM (which is known to be time- and cost-consuming). The problem of finding an optimal solution for allocating these extra tasks from an overloaded VM to appropriate host VMs is solved by using Particle Swarm Optimization (PSO) heuristic. PSO is a population-based search algorithm based on the simulation of the social behavior of birds. The scheduling algorithm is multi-objective, aiming at minimizing task execution time and task transfer time. TBSLB-PSO method considers both computing intensive and data intensive tasks. Computing intensive tasks de-

mand extensive computation (e.g., scientific applications) while data intensive tasks are characterized by high volumes of data to be published and maintained over time. The scheduling optimization model takes into account the number of CPUs on host VMs to schedule computing intensive tasks, and the bandwidth as a variable to minimize the tasks transferring time for data intensive applications. To solve this multi-objective optimization problem, the PSO algorithm is applied to find an optimal way to allocate extra tasks to the new (underloaded) VMs with less task execution and task transfer time. The TBSLB-PSO algorithm works in 6 steps, ranging from monitoring and analysis of PMs, VMs, and tasks, determining overloaded VMs, finding optimal homogeneous VMs to transfer the tasks to and optimal task migration schema, and transferring tasks and updating the scheduler information. Authors simulated their proposal with Cloudsim toolkit, which was able to schedule a set of 10 tasks onto 5 VMs over 3 PMs, in 0.224 seconds. The authors have also solved this problem with Multi-Objective Genetic Algorithm (MOGA) [49] as an alternative to PSO, although no comparison information of the two alternatives was provided.

Scheduling at the virtualization level In the case of scheduling VMs onto PMs, Dasgupta et al. [11] proposed a novel load balancing strategy using Genetic Algorithms (GA) aiming at balancing the load of the cloud infrastructure while trying to minimizing the makespan of a given tasks set. GAs is a stochastic searching algorithm based on the mechanisms of natural selection and genetics, widely used in complex and vast search space and known to be very efficient in searching out global optimum solutions. The results showed that the proposed algorithm outperformed the existing approaches like First Come First Serve (FCFS), Round Robing (RR) and the local search algorithm Stochastic Hill Climbing (SHC).

Scheduling at the programming level Aiming at optimizing load balancing, Xu et al. [65] proposed a novel model to balance data distribution to improve Cloud computing performance in data-intensive applications, such as distributed data mining. Their work consists in extending the classic MapReduce model [12] in order to fight its computational imbalance problem. MapReduce provides a set of frameworks that aim to enable productive programming of computer clusters. The frameworks are efficient in the processing of huge data sets with flexible job decomposition and sub-tasks allocation. Using distributed programming frameworks such as Hadoop [60], the application programmers can construct parallel dataflows using map and reduce functions to achieve portability and scalability of their applications. However, these frameworks are not suitable for all scientific workloads since they are designed for data-intensive workloads. In fact, the original load balance in Hadoop considers only static storage space balance, without considering the workloads attached to the data blocks, which is of paramount importance for the unbalanced ILP jobs. To tackle this issue, authors have proposed a completely distributed approach for adjusting load balancing based on agent-workers running on nodes, unlike original Hadoop which relies on an agent-

master for centralized task processing such as task allocation. By representing each node, agents can communicate with each other to cooperatively manage and adjust the balance of working loads. These series of agents monitor the data nodes (e.g., hardware performance, working load in real time), and jointly make decisions on how to move data blocks to maximize load balancing. In the process of load balancing, the overload nodes request computing resources from other nodes, and copy their data to the new nodes who have more resources. Initially, agents have no knowledge of their neighbors' working load distribution, and so a token-based heuristic algorithm is proposed as well. The goal of balancing working load is to reduce calculation and free storage variance. In the optimal case, each node holds at most one computation task and has almost the same size of free storages. Authors have evaluated their proposal through simulation and concluded that agents can improve load balancing efficiency with limited communication costs among agents.

5 Life Science Applications

The Life Sciences have a lot of scientific problems where Machine Learning technique may be very helpful. However, some of the important problems have to deal with data from quite different sources, encoded in difference representation schemes and with structure. To analyze data sets in those kinds of problems/domains using algorithm such Decision Trees, SVMs or Artificial Neural Nets[9], the data has to be "enclosed" into a single table of a Relational Data base [10]. Most often this reduction procedure leads to information loss or an extensive amount of pre-processing. The advantages of such algorithm's analysis are that, usually, they are much faster than ILP's analysis.

To analyze data with structure, encoded in different encoding schemes, ILP have been recognized to have advantages over propositional learners. ILP can handle "naturally" with several relations, data with structure, encoded in different representation schemes, can combine harmoniously symbolic and numerical computations and, most importantly the constructed models and most often comprehensible to the domain experts. This last feature may be of capital importance to scientific applications where understating the phenomena that produced the data is required. ILP models may provide clues for such explanations.

In this section we visit two applications where ILP have give very good results, produced comprehensible models and showed several advantages over propositional learners. We discuss also the the advantage of the application of the parallel execution of ILP in these type of applications.

[9] Propositional level algorithms
[10] or a sheep of a spreadsheet

5.1 Structure-Activity Relationship experiments

ILP have been extensively used in Structure-Activity Relationship[11] (SAR) problems. Published studies include [26,59,51] where a ILP systems predicted mutagenicity and [58] caricnogenicity activity. In order to assess the impact of the parallel approach to ILP implemented in the APIS system (See Section 3) we have used four data sets originated from SAR problems[12]. Two of them are the ones just mentioned for predicting mutagenicity and carcinogenicity. The other two are toxicity data sets (DBPCAN and CPDBAS). 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 [63]. The second data set is CPDBAS, the Carcinogenic Potency Data Base (CPDB) that contains detailed results and analyzes of 6540 chronic, long term carcinogenesis bio assays[13]. The other two data sets used in this study were the carcinogenesis and mutagenesis mentioned above [14].

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 same nodes limit was used. The background provided for the data sets were as follows. Far all data sets, the structure of each molecule (atoms and bonds) was available in the background knowledge. For the toxicity data sets (DBPCAN and CPDBAS) a set of molecular descriptors for each molecule was also available in the background knowledge.

data set name	number of examples	number of *islands*	clause length	nodes limit (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

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.

[11] An approach where the activity of a compound, for example, is predicted only based on its structure features.

[12] A detailed description of this study can be found in [66]

[13] 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, accessed Dec 2008.

[14] Available from the Oxford University Machine Learning repository http://www.cs.ox.ac.uk/activities/machlearn/applications.html

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**	**number of worker nodes**			
name	**execution**	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)

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.

Overall, the results show that significant speedups were achieved by APIS, well beyond the number of processors (Table 2 (a))[15] without affecting accuracy (no statistical significant difference for $\alpha \leq 0.05$), Table 2 (b).

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.

However, we have also noticed that the speedup of the slowest sub-space search alone does not explain the global speedups obtained. In a deeper analysis we can see that the number of "slow" sub-spaces (1 in mutagenesis and 3 in dbpcan, for example) that the other sub-spaces use less than 10% of the time of the slower ones. That is, there are 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

[15] Except for the carcinogenesis data set

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.

As seen from the above results, the APIS approach provides several "opportunities" for the parallel execution to produce very good speedups. In the next application we describe and example of a chemoinformatics application where the number of islands that can be identified is quite large making that kind of applications adequate for the use of the APIS approach.

5.2 Predicting drug efficiency in Cancer cells treatment

The data used in the experiments is the result of the work done in the "Genomics of Drug Sensitivity in Cancer" project [20], and its pre-processing follows the same approach used in [34]. The original data set, publicly available in the "Genomics of Drug Sensitivity in Cancer" project website, consists of measured IC50 values for various cell lines and compound pairs. It contains 639 different cell lines, each with 77 gene mutation properties. Each cell line also has information about its microsatellite instability status (MIS), cancer type and correspondent tissue. The IC50 value is available in its natural logarithmic form, ranging from -18.92 (6.07E-9 raw form) to 15.27 (4.28E6 raw form). Each gene mutation is described by its sequence variation and copy number variation.

The data set contains 131 drugs that were applied to the cell lines leading to 83709 potential IC50 values. Each example was characterized with the cell line feature together with the features of the drug that was used in that cell line and the IC50 value obtained. Cell features are the cell mutation properties and drug feature are molecular descriptors and fingerprints generated with the same version of PaDEL used in [34]. The final amount of cell line features was 142, and the final amount of drug features was 790, resulting in a total of 932 features plus the IC50 value. The final data set resulted in 40691 instances.

An experiment was done using the Aleph ILP system in the classification task of predicting "good" drugs in the cell line data set described above. We have transformed the original regression problem into a binary classification problem. We have sorted the examples by their IC50 value and established a lower threshold below which examples are in class "good" and a upper threshold above which example where in class "bad". Examples in the "gray zone" between the lower and upper thresholds were discarded. The discretization resulted in a total of 27120 examples. The background knowledge included the molecular descriptors, fingerprints as well as the cell lines features.

Some simple rules found by Aleph include:

body of Rule 1: pubchemfp567(Compound), pubchemfp516(Compound), pubchemfp692(Compound) ; Positive cover = 3571, Negative cover = 77.

"If the compound molecule has the substructure O-C-C-O, the substructure [#1]-C=C-[#1] and the substructure O=C-C-C-C-C-C, then the IC50 is considered as good (98% of the covered examples)."

body of Rule 2: pubchemfp188(Compound) ; Positive cover = 4069, Negative cover = 2915.

"If the compound molecule has 2 or more saturated or aromatic heteroatom-containing ring of size 6, then the IC50 is considered good (58% of the covered examples)."

Aleph was able to construct very simple rules that easily to understand by the experts. The accuracy was of 91%.

The background knowledge used in the just described experiment is typical in chemoinformatics data analysis. There is a large number of molecular descriptors and fingerprints. When encoding such formation in the background knowledge a large number of [APIS] islands. These type of chemoinformatic data analysis can profit a lot from the type of parallelization available in the APIS system.

6 Conclusions

In this chapter we have discussed how ILP systems can profit from parallel execution. We have surveyed parallel and distributed executions of ILP systems. We have payed special attention to the parallel approach implemented in the APIS ILP system. A survey on parallel an distributed computation was also presented. To link ILP to parallel execution we have also discussed how a distributed system scheduler framework could be used to improve the execution of ILP systems by running in parallel several of the task involved in the execution of an ILP system. We have presented several applications where ILP have been successfully used and discussed how those applications can profit from a parallel approach like the one of the APIS system.

As a conclusion we may state that ILP systems have several advantages when applied to data analysis in Life Sciences domains. Those applications can profit even more if a parallel execution is used. A parallel execution can substantially improve execution time or improve the quality of the models by searching larger regions of the hypothesis space in the same time as the sequential execution. Parallel execution can be used in a very large number of parts of an ILP algorithms. We can use it at the theory-level search (coarse level) to the hypothesis space search to even use ILP with a parallel execution of the Prolog engine, for highly non-deterministic background knowledge.

References

1. Alexessander Alves, Rui Camacho, and Eugenio Oliveira. Discovery of functional relationships in multi-relational data using inductive logic programming. In *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 2004), 1-4 November 2004, Brighton, UK*, pages 319–322, 2004.
2. EC Amazon. Amazon elastic compute cloud (amazon ec2). 2010.

3. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.
4. Jan Blaták and Lubomíŕ Popelínský. dRAP: A Framework for Distributed Mining Firts-Order Frequent Patterns. In *Proceedings of the 16th Conference on Inductive Logic Programming*, pages 25–27. Springer-Verlag, 2006.
5. Paul Bone, Zoltan Somogyi, and Peter Schachte. Estimating the overlap between dependent computations for automatic parallelization. *TPLP*, 11(4-5):575–591, 2011.
6. I. Bratko, S. Muggleton, and A. Varsek. Learning qualitative models of dynamic systems. In *Proceedings of the Eighth International Machine Learning Workshop*, San Mateo, Ca, 1991. Morgan-Kaufmann.
7. Wray Buntine. Generalised subsumption and its applications to induction and redundancy. *Artificial Intelligence journal*, 36(2):149–176, 1988. revised version of the paper that won the A.I. Best Paper Award at ECAI-86.
8. Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.
9. Amanda Clare and Ross D. King. Data mining the yeast genome in a lazy functional language. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages*, pages 19–36, 2003.
10. Vítor Santos Costa, Inês de Castro Dutra, and Ricardo Rocha. Threads and or-parallelism unified. *TPLP*, 10(4-6):417–432, 2010.
11. Kousik Dasgupta, Brototi Mandal, Paramartha Dutta, Jyotsna Kumar Mandal, and Santanu Dam. A genetic algorithm (ga) based load balancing strategy for cloud computing. *Procedia Technology*, 10:340–347, 2013.
12. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
13. L. Dehaspe and L. De Raedt. Parallel inductive logic programming. In *Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, 1995.
14. Javier Delgado, Anas Salah Eddin, Malek Adjouadi, and S Masoud Sadjadi. Paravirtualization for scientific computing: Performance analysis and prediction. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 536–543. IEEE, 2011.
15. Usama M. Fayyad and Ramasamy Uthurusamy, editors. *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95), Montreal, Canada, August 20-21, 1995*. AAAI Press, 1995.
16. Nuno A. Fonseca, Ashwin Srinivasan, Fernando M. A. Silva, and Rui Camacho. Parallel ilp for distributed-memory architectures. *Machine Learning*, 74(3):257–279, 2009.
17. Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee, Knoxville, TN, USA, 1994.
18. Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.
19. Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28:13, 2009.

20. Mathew J Garnett, Elena J Edelman, Sonja J Heidorn, Chris D Greenman, Anahita Dastur, King Wai Lau, Patricia Greninger, I Richard Thompson, Xi Luo, Jorge Soares, et al. Systematic identification of genomic markers of drug sensitivity in cancer cells. *Nature*, 483(7391):570–575, 2012.
21. Cloud Google. Google cloud platform.
22. James Graham, David Page, and Ahmed Kamal. Accelerating the drug design process through parallel inductive logic programming data mining. In *Proceeding of the Computational Systems Bioinformatics (CSB'03)*. IEEE, 2003.
23. Gopal Gupta, Enrico Pontelli, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Trans. Program. Lang. Syst.*, 23(4):472–602, 2001.
24. Wei Huang, Jiuxing Liu, Bulent Abali, and Dhabaleswar K Panda. A case for high performance computing with virtual machines. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 125–134. ACM, 2006.
25. Gideon Juve and Ewa Deelman. Scientific workflows and clouds. *Crossroads*, 16(3):14–18, 2010.
26. R. King, S. Muggleton R. Lewis, and M. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Sciences*, 89(23), 1992.
27. R. King and M.J.E. Sternberg. A machine learning approach for the prediction of protein secondary structure. *Journal of Molecular Biology*, 216:441–457, 1990.
28. Stasinos K. Konstantopoulos. A Data-Parallel version of Aleph. In *Proceedings of the Workshop on Parallel and Distributed Computing for Machine Learning, co-located with ECML/PKDD'2003*, Dubrovnik, Croatia, 2003.
29. P Venkata Krishna. Honey bee behavior inspired load balancing of tasks in cloud computing environments. *Applied Soft Computing*, 13(5):2292–2303, 2013.
30. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.
31. Carlos Alberto Martinez-Angeles, Inês de Castro Dutra, Vítor Santos Costa, and Jorge Buenabad-Chavez. A datalog engine for gpus. In *Declarative Programming and Knowledge Management - Declarative Programming Days, KDPD 2013, Unifying INAP, WFLP, and WLP, Kiel, Germany, September 11-13, 2013, Revised Selected Papers*, volume 8439 of *Lecture Notes in Computer Science*, pages 152–168. Springer, 2013.
32. T. Matsui, N. Inuzuka, H. Seki, and H. Itoh. Comparison of three parallel implementations of an induction algorithm. In *8th Int. Parallel Computing Workshop*, pages 181–188, Singapore, 1998.
33. Viktor Mauch, Marcel Kunze, and Marius Hillenbrand. High performance cloud computing. *Future Generation Computer Systems*, 29(6):1408–1416, 2013.
34. Michael P Menden, Francesco Iorio, Mathew Garnett, Ultan McDermott, Cyril H Benes, Pedro J Ballester, and Julio Saez-Rodriguez. Machine learning prediction of cancer cell sensitivity to drugs based on genomic and chemical properties. *PloS one*, 8(4):e61318, 2013.
35. R.S. Michalski. Pattern recognition as rule-guided inductive inference. In *Proceedings of IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 349–361, 1980.
36. T. M. Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.
37. S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–317, 1991.

38. S. Muggleton. Inductive logic programming:derivations, successes and shortcomings. In *Proceedings of the European Conference on Machine Learning: ECML-93.*, pages 21–37, Vienna, Austria, April 1993.
39. S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
40. Stephen Muggleton. Learning from positive data. In *Inductive Logic Programming, 6th International Workshop, ILP-96, Stockholm, Sweden, August 26-28, 1996, Selected Papers*, pages 358–376, 1996.
41. Stephen Muggleton and John Firth. Relational Rule Induction with CProgol4.4: A Tutorial Introduction. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, pages 160–188. Springer-Verlag, 2001.
42. Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *JOURNAL OF LOGIC PROGRAMMING*, 19(20):629–679, 1994.
43. H. Ohwada and F. Mizoguchi. Parallel execution for speeding up inductive logic programming systems. In *Proceedings of the 9th International Workshop on Inductive Logic Programming*, number 1721 in LNAI, pages 277–286. Springer-Verlag, 1999.
44. H. Ohwada, H. Nishiyama, and F. Mizoguchi. Concurrent execution of optimal hypothesis search for inverse entailment. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *LNAI*, pages 165–173. Springer-Verlag, 2000.
45. Elina Pacini, Cristian Mateos, and Carlos García Garino. Distributed job scheduling based on swarm intelligence: A survey. *Computers & Electrical Engineering*, 40(1):252–269, 2014.
46. G. D. Plotkin. *A note on inductive generalisation*, pages 153–163. Edinburgh University Press, Edinburgh, 1969. eds. Meltzer, B. and Michie, D.
47. J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, volume 667, pages 3–20. Springer-Verlag, 1993.
48. Lavanya Ramakrishnan, Piotr T Zbiegel, Scott Campbell, Rick Bradshaw, Richard Shane Canon, Susan Coghlan, Iwona Sakrejda, Narayan Desai, Tina Declerck, and Anping Liu. Magellan: experiences from a science cloud. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 49–58. ACM, 2011.
49. Fahimeh Ramezani, Jie Lu, and Farookh Hussain. Task based system load balancing approach in cloud environments. In *Knowledge Engineering and Management*, pages 31–42. Springer, 2014.
50. Fahimeh Ramezani, Jie Lu, and Farookh Khadeer Hussain. Task-based system load balancing in cloud computing using particle swarm optimization. *International Journal of Parallel Programming*, 42(5):739–754, 2014.
51. King RD1, Muggleton SH, Srinivasan A, and Sternberg MJ. Structure-activity relationships derived by machine learning: the use of atoms and their bond connectivities to predict mutagenicity by inductive logic programming. *Proc Natl Acad Sci U S A*, 9(93(1)):438–42, 1996.
52. Francisco Reinaldo, Carlos Fernandes, Md. Anishur Rahman, Andreia Malucelli, and Rui Camacho. Assessing the eligibility of kidney transplant donors. In *Machine Learning and Data Mining in Pattern Recognition, 6th International Conference, MLDM 2009, Leipzig, Germany, July 23-25, 2009. Proceedings*, pages 802–809, 2009.
53. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, Jan 1965.

54. Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik Blockeel, Bart Demoen, Gerda Janssens, Jan Struyf, Henk Vandecasteele, and Wim Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 4:465–491, 2003.
55. David B. Skillicorn and Yu Wang. Parallel and sequential algorithms for data mining using inductive logic. *Knowl. Inf. Syst.*, 3(4):405–421, 2001.
56. R.G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Trans. Computers*, 29(12):1104–1113, 1980.
57. A. Srinivasan. The Aleph Manual, 2003. Available from http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph.
58. Ashwin Srinivasan, Ross D. King, Stephen Muggleton, and Michael J. E. Sternberg. Carcinogenesis predictions using ILP. In *Inductive Logic Programming, 7th International Workshop, ILP-97, Prague, Czech Republic, September 17-20, 1997, Proceedings*, pages 273–287, 1997.
59. *Nuno A. Fonseca*, Max Pereira, Vitor Santos Costa, and Rui Camacho. Interactive discriminative mining of chemical fragments. In *Proceedings of the 2010 International Conference on Inductive Logic Programming (ILP 2010)*, number 6489 in Lecture Notes in Artificial Intelligence, pages 59–66. Springer-Verlag, 2011.
60. Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.
61. Jan Wielemaker. Native preemptive threads in SWI-Prolog. In Catuscia Palamidessi, editor, *Proceedings of the 19th International Conference on Logic Programming*, volume 2916 of *LNAI*, pages 331–345. Springer-Verlag, 2003.
62. R. Wirth. Learning by failure to prove. In *Proc. Third European Working Session on Learning*, pages 237–251, London, 1988. Pitman.
63. Y.T. Woo, D. Lai, J.L. McLain, M.K. Manibusan, and V. Dellarco. Use of mechanism-based structure-activity relationships analysis in carcinogenic potential ranking for drinking water disinfection by-products. *Environ. Health Perspect*, 110((Suppl 1)):75–87, 2002.
64. Fuhui Wu, Qingbo Wu, and Yusong Tan. Workflow scheduling in cloud: a survey. *The Journal of Supercomputing*, pages 1–46, 2015.
65. Yang Xu, Lei Wu, Liying Guo, Zheng Chen, Lai Yang, and Zhongzhi Shi. An intelligent load balancing algorithm towards efficient cloud computing. In *Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.
66. Gerson Zaverucha, Vítor Santos Costa, and Aline Paes, editors. *Inductive Logic Programming - 23rd International Conference, ILP 2013, Rio de Janeiro, Brazil, August 28-30, 2013, Revised Selected Papers*, volume 8812 of *Lecture Notes in Computer Science*. Springer, 2014.
67. Zhi-Hui Zhan, Xiao-Fang Liu, Yue-Jiao Gong, Jun Zhang, Henry Shu-Hung Chung, and Yun Li. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Computing Surveys (CSUR)*, 47(4):63, 2015.