CONNER: A Concurrent ILP Learner in Description Logic

Eyad Algahtani and Dimitar Kazakov

University of York, Heslington, York YO10 5GH, UK,
ea922@york.ac.uk, kazakov@cs.york.ac.uk,
WWW home page: https://www-users.cs.york.ac.uk/kazakov/

Abstract. Machine Learning (ML) approaches can achieve impressive results, but many lack transparency or have difficulties handling data of high structural complexity. The class of ML known as Inductive Logic Programming (ILP) draws on the expressivity and rigour of subsets of First Order Logic to represent both data and models. When Description Logics (DL) are used, the approach can be applied directly to knowledge represented as ontologies. ILP output is a prime candidate for explainable artificial intelligence; the expense being computational complexity. We have recently demonstrated how a critical component of ILP learners in DL, namely, cover set testing, can be speeded up through the use of concurrent processing. Here we describe the first prototype of an ILP learner in DL that benefits from this use of concurrency. The result is a fast, scalable tool that can be applied directly to large ontologies.

Keywords: Inductive logic programming · description logics · ontologies · parallel computing · GPGPU

1 Introduction

Graphic processing units (GPU) can be used with benefits for general purpose computation. GPU-based data parallelism has proven very efficient in a number of application areas [1]. We have recently proposed a GPU-accelerated approach to the computation of the cover set for a given hypothesis expressed in $\mathcal{ALC}$ description logic, which results in a speed up of two orders of magnitude when compared with a single-threaded CPU implementation [2]. The present article combines this approach with an implementation of a well-studied refinement operator and a search strategy for the exploration of the hypothesis space. The result is the first version of a GPU-accelerated inductive learner, further on referred to as CONNER 1.0 (CONcurrent learNER).

In more detail, here we present the first complete description of the way binary predicates (or *roles* in description logic parlance) are handled by our cover set procedure, which is now extended beyond $\mathcal{ALC}$ to make use of cardinality restrictions (e.g. $\mathit{OffpeakTrain} \sqsubseteq \leq 6\ \mathit{hasCar}.\mathit{Car}$) and data properties (e.g. $\mathit{Bike} \sqsubseteq\ \mathit{numberOfWheels} = 2$). We test the speed and accuracy of our learner on a combination of synthetic and real world data sets. To emphasise the low

cost of adoption of this algorithm, we have run the tests on a commodity GPU, Nvidia GeForce GTX 1070.

The rest of this paper is structured as follows: Section 2 covers relevant work, Section 3 completes the previously published description of how the hypothesis cover set is computed [2] with the algorithms handling value restriction and existential restriction, and tests the speed of their execution. Section 4 extends the list of operators with the algorithms for handling the cardinality restriction and data property operators. Section 5 describes a complete GPU-accelerated ILP learner in DL, CONNER, and evaluates its performance, while Section 6 draws conclusions and outlines future work.

2 Background

CONNER lays at the intersection of ILP, parallel computation, and description logics (Fig. 1). In this section, we review the notable overlaps between these three areas. Algathani and Kazakov [2] can be further consulted for a suitable overview of the basics of GPU architecture.

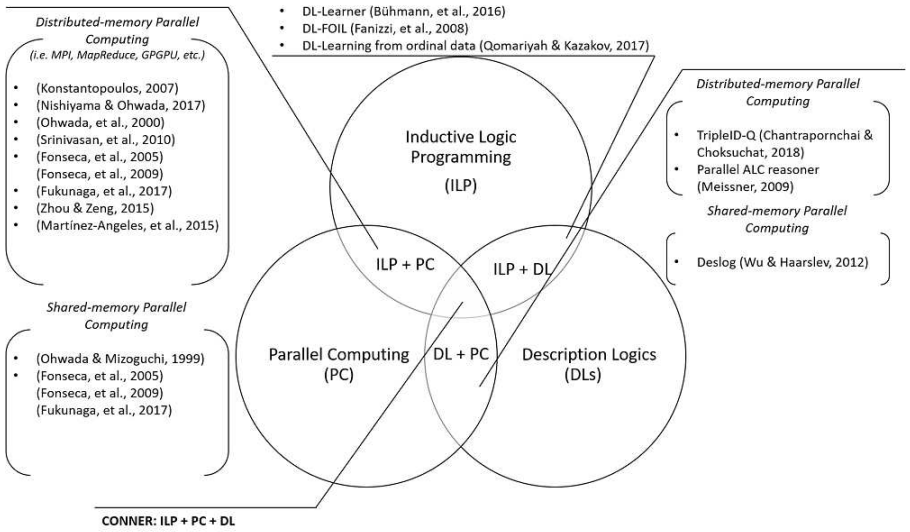

Fig. 1. Intersection of ILP, parallel computation and DL [3,5,8–10,16–24].

Arguably, the implementation of an ILP algorithm can be defined by the ways it approaches hypothesis cover set testing, the application of its refinement operator, and the search algorithm used to explore the hypothesis space. While all three are amenable to parallelisation, here we focus our efforts in this direction on the first component, and with a description logic as a hypothesis language.

Description Logics (DL) is a family of subsets of first order logic that are used to represent knowledge in the form of *ontologies*. $\mathcal{ALC}$ (Attributive Language with Complement) is a commonly used subset of DL, which makes use of propositional operators, as well as two binary operators, existential restriction and value restriction. As ontologies continue to gain popularity, e.g. in the form of linked open data, this strengthens the case for learners that can work directly with this type of data. A few notable examples of ILP learners in DL follow.

DL-FOIL [3] is an adaptation of the classical ILP algorithm FOIL [4] to DL as data and hypothesis language. It still uses a top-down refinement operator with search guided by information gain, but the latter is modified to accommodate the use of the Open World Assumption (OWA).

The DL-Learner [5] is a framework for learning in DL. The framework allows the user to select from different reasoners to handle inference. It is also possible to choose from four learning algorithms: OWL Class Expression Learner (OCEL), Class Expression Learning for Ontology Engineering (CELOE), EL Tree Learner (ELTL), and Inductive Statistical Learning of Expressions (ISLE). One can also choose from different refinement operators. The framework provides facilities to handle large datasets by the use of sampling.

APARELL [6] is an ILP algorithm for the learning of ordinal relations (e.g. *better_than* [7]) in DL. The algorithm borrows ideas from the Progol/Aleph pair of algorithms about the way hypothesis search operates. APARELL processes DL data directly from OWL ontologies, and can read any format which is supported by the OWL API.

There have been several attempts to implement reasoners in DL using parallel computation [8, 9]. The most relevant effort here is Chantrapornchai and Choksuchat's [10] recently proposed GPU-accelerated framework for RDF query processing, TripleID-Q. The framework maintains a separate hash table for each of the three arguments (*subject*, *predicate*, *object*) of the RDF triples. Each triple is represented through the three hash keys (all integers) and stored in the shared memory of the GPU. RDF queries are mapped onto that representation and the data is split among multiple GPU threads to retrieve the matching triples in parallel. In this context, the work of Martínez-Angeles *et al* on the use of GPU to speed up cover set computation for first-order logic ILP systems, such as Aleph, should also be noted [11].

3 Concurrent, GPU-Accelerated Cover Set Computation

A GPU can manipulate matrices very efficiently. Here we complete the description of the GPU-powered approach first presented by Algahtani and Kazakov [2], which aims at speeding up the calculation of the cover set of hypotheses expressed in $\mathcal{ALC}$ description logic. We also present experimental results on the performance of this algorithm.

DL allows one to describe *concepts* C_i defined over a universe of *individuals* I_i, and to define *roles* relating individuals to each other. Concept membership can be represented as a Boolean matrix M of size $|C| \times |I|$ (see Fig. 2). Using this

representation, it is possible to employ data parallelism for the application of logic operations to concepts. We have already shown how the three propositional operators $\{\sqcap, \sqcup, \neg\}$ can be implemented, and tested the speed with which they are computed [2]. Here we describe the concurrent implementation of the value restriction operator ($\forall r.C$) and the existential restriction operator ($\exists r.C$).

Either restriction operator takes a role and a concept as input, and makes use of the concept matrix M and another matrix, R, storing all role assertions (see Figure 2). The matrix is sorted by the role. As a consequence, all assertions of a given role are stored in a contiguous range of rows. This facilitates a more efficient GPU memory access pattern (namely, coalesced memory access). For each role, the start and end row indices corresponding to its range of rows are stored in a hash table, H, and can be retrieved efficiently using the role name as key.

Algorithm 1 shows the implementation of the existential operator. Its first step is to allocate memory for the output array `result` and set all values to 0 (representing `False`). This is done in a concurrent, multi-threaded fashion. The range of rows in R storing all assertions of *Role* is then looked up in H (in $\mathcal{O}(1)$ time). After that, the role assertions in R within the role range are divided among a number of threads, and for each such assertion, a check in matrix M is made whether `IndvB` belongs to *Concept* (i.e. the concept in the existential restriction). The result of this step is combined through OR with the current value in row `IndvA` of the output array `result` and stored back there.

This implementation avoids the use of conditional statement, which could slow down the process. At the same time, it is important that an atomic OR is used to avoid a race-condition situation between the individual threads. [1]

	C1	C2	C3
Indv1	0	1	0
Indv2	1	1	0
Indv3	0	0	1

(a) Table M

IndvA	Role	IndvB
6	1	46
6	1	5
9	2	14

(b) Table R

	DP1	DP2	DP3
Indv1	3	2.5	1
Indv2	7	3.7	1
Indv3	0	-0.5	0

(c) Table D

Fig. 2. Main data structures in the GPU memory

The implementation of the value restriction operator $\forall Role.Concept$ is analogous to the existential restriction, with the only difference being that all initial values in the *result* array are set to 1 (i.e. `True`), and an atomic AND operator is applied instead of OR.[2]

[1] Here we use the CUDA built-in atomic function `atomicOR(A,B)`, which implements the (atomic) Boolean operation `A := A OR B`.

[2] Note that this implementation returns the correct result for the following special case: if $\nexists IndvB : Role(IndvA, IndvB)$ then $IndvA \in \forall Role.Concept$.

Algorithm 1 Existential Restriction Cover Set ($\exists Role$.Concept)

procedure ParallelExistentialRestriction(Concept,Role)

```
Given:
M: Boolean 2D matrix of size (individuals x concepts)
R: Integer 2D matrix of size (# of property assertions x 3) // each row
                                 // representing a triple: subj,role,obj
HT: hash table // Role -> (Offsets for first and last entries in R)

Concept: Pointer to a column in M
Role: Integer

result: Boolean 1D array of size NumberOfIndividuals

Do:
parallel_foreach thread T_j
| for each individual i in result
| | set result[i] = 0
| endfor
endfor
set role_range := HT[Role] // get range of Role assertions in R from HT
parallel_foreach thread T_j
| foreach roleAssertion in role_range
| | set IndvA := R[row(roleAssertion),1] // first column of roleAssertion
| | set IndvB := R[row(roleAssertion),3] // third column of roleAssertion
| | set local_result := M[row(IndvB),Concept]
| | atomicOR(result[row(IndvA)],local_result)
| endfor
endfor
```

return result(1..numberOfIndividuals)

Table 1. Execution times for computing the cover sets of $\exists has\_car$.Long and $\forall has\_car$.Long (average of 10).

Data set size			Execution time [ms]	
Total number of individuals	Total number of has_car assertions	Total number of all role assertions	∃has_car.Long mean (stdev)	∀has_car.Long mean (stdev)
50	30	149	0.49(0.08)	0.48(0.08)
410	300	1,490	0.56(0.10)	0.50(0.10)
4,010	3,000	14,900	0.50(0.06)	0.51(0.08)
40,010	30,000	149,000	0.55(0.02)	0.55(0.03)
400,010	300,000	1,490,000	0.97(0.02)	1.02(0.03)
4,000,010	3,000,000	14,900,000	4.82(0.05)	5.85(0.28)
8,000,010	6,000,000	29,800,000	10.55(0.34)	11.38(0.48)

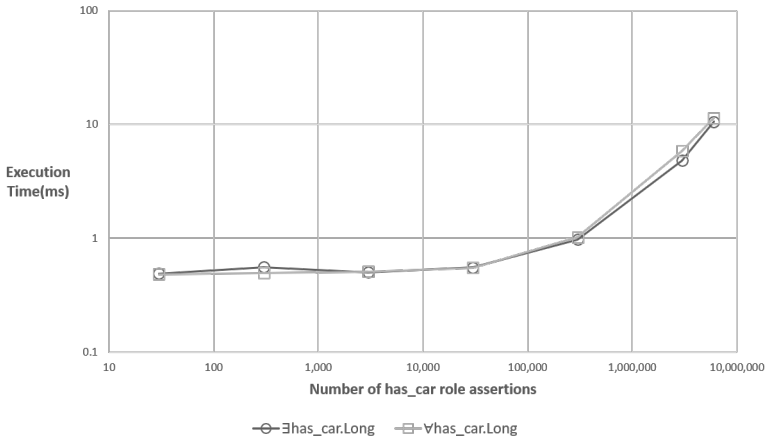

Fig. 3. Computing existential (∃) and value (∀) role restrictions

To evaluate the execution times of the two restriction operators, we use a dataset consisting of multiple copies of Michalski's eastbound/westbound trains dataset [12], here in its DL representation [5]. The dataset consists of 12 concepts (unary predicates), and 5 roles (binary predicates); these predicates describe 10 trains (5 eastbound and 5 westbound) and their respective cars. The results are shown in Table 1 and plotted in Figure 3. It should be noted that only the number of assertions for the given role has an impact on the execution times when matrix R is traversed: while matrix M grows with the number of individuals, access time to each row remains the same as, internally, an individual is represented directly with its row index in M. Also, the actual content of the assertions for the role in question makes no difference, which is why this dataset is appropriate here. The results show that the execution times remain virtually constant up to a point, possibly until the full potential for parallel computation in the GPU is harnessed, and then grow more slowly than $\mathcal{O}(n)$ for the range of dataset sizes studied.

4 Extending The Hypothesis Language

This section describes how the hypothesis language has now been extended to include cardinality and data property (also known as *concrete role*) restrictions. The result is a type of description logic referred to as $\mathcal{ALCQ}^{(\mathcal{D})}$.

4.1 Cardinality Restriction Support

The cardinality operator restricts for a given role the allowed number of assertions per individual. A cardinality restriction can be qualified (Q), or unqualified

(N), where N is a special case of Q. In Q, any concept can be used in the restriction. While in N, only the Top Concept is used. There are three kinds of cardinality restrictions: the minimum ($\geq$), maximum ($\leq$), and the exactly ($==$) restriction. Algorithm 2 implements the first of these three.

Algorithm 2 Qualified Cardinality Restriction Cover Set

procedure ParallelCardinalityRestriction(Concept,Role,n)

```
Given:
M: Boolean 2D matrix of size (individuals x concepts)
R: Integer 2D matrix of size (individuals x 3) // Storing (subj,role,obj)
H: hash table // Role -> (Offsets for first and last entries in R)
Concept: Pointer to a column in M
Role: Integer
n: restriction number
result: Boolean 1D array of size NumberOfIndividuals
Do:
parallel_foreach thread T_j
| for each individual i in result
| | set result[i] = 0
| endfor
endfor
set role_range := H[Role] // retrieve range of Role assertions in R from H
parallel_foreach thread T_j
| foreach roleAssertion in role_range
| | set IndvA := R[row(roleAssertion),1] // first column of roleAssertion
| | set IndvB := R[row(roleAssertion),3] // third column of roleAssertion
| | set local_result := M[row(IndvB),Concept]
| | atomicAdd(result[row(IndvA)],local_result)
| endfor
endfor

parallel_foreach thread T_i
| foreach individual I_j in thread T_i
| | result(row(I_j)):= result(row(I_j)) >= n // OR =< n  OR == n
| endfor
endfor
```

return result(1..numberOfIndividuals)

It first clears the result array (just as in the existential restriction). It then uses the CUDA atomic addition `atomicAdd(A,B)` to increment the counter of the corresponding `IndvA` for every assertion matching the role and concept. The values in the `result-array` are then compared with the cardinality condition, and the counter for each individual is replaced with 1 or 0 (representing True/False) according to whether the condition in question has been met. The condition in

this last loop determines the type of cardinality restriction: min ($\geq$), max ($\leq$), or exactly ($==$).

4.2 Data Property Restriction Support

Data properties (or concrete roles) map individuals on to simple values. In this implementation, we (currently) limit the range of values to numerical ones: integer, float and Boolean; supporting other types like Strings, is considered for future work. In order to handle such properties in the GPU, the individuals and their data properties are mapped on to a matrix, D (see Figure 2), in a way similar to matrix M. Each cell in the new 2D matrix, is of float datatype, as it is inclusive to integers and Booleans. As with the cardinality restrictions, there are three kinds of data property restrictions: min, max, and exactly. Algorithm 3 shows how the minimum data property restriction is implemented, with the other two requiring only a trivial variation.

Algorithm 3 Data Property Restriction Cover Set

procedure ParallelDataPropertyRestriction(Property,Value)

```
D := 2D matrix (individuals x data properties)
parallel_foreach thread T_i
| foreach individual I_j in thread T_i
| | result(row(I_j)) := D(row(I_j),column(Property)) >= Value //OR =< OR ==
| endfor
endfor
```

return result(1..numberOfIndividuals)

In Algorithm 3, a parallel for loop will iterate through all individuals, and the result— array will be set to 1 for all individuals matching the condition or to 0 otherwise. For the maximum and exactly restriction types, the condition will be changed to $\leq$ (for maximum), and $==$ (for exactly).

5 CONNER: All together now

The work described in this section was motivated by the desire to incorporate our approach to computing the DL hypothesis cover set in a learner in order to gauge the benefits this approach can offer.

5.1 TBox Processing

Every ontology consists of two parts, the so called ABox and TBox. Both of these need to be processed to establish correctly the membership of individuals

to concepts. It is possible to employ off-the-shelf reasoners for this purpose. Indeed, this is the approach employed by DL-Learner. While it is expected that CONNER will make the same provision in the next iteration of its development, we have decided to use our own implementation here in order to have full control over how the planned tests are run. The implementation described below is somewhat limited, but sufficient for the test data sets used.

The ABox explicitly lists individuals belonging to a given concept or relation. These are easily processed and matrices M and R updated accordingly. The TBox provides intensional definitions of concepts ($C \equiv \dots$) and their hierarchy (e.g. $C_1 \subset C_2$). Here we only handle subsumption between single concepts. This information is processed by a dedicated multi-pass algorithm which updates matrix M, and is repeated until no further changes in concept definitions occur. Cyclic references are also detected and flagged up as an error. For instance, if the membership of the concept `Man` is defined extensively, through the ABox, and the TBox contains the statement `Man` $\subset$ `Person`, the individuals in `Man` will also be marked in matrix M as belonging to the concept `Person` after the TBox is processed (Figure 4). The TBox needs only be processed once, when the ontology is imported, and represents no overhead on any subsequent computations. The hierarchy of concepts derived from the TBox statements is then used by the refinement operator to generate candidate hypotheses.

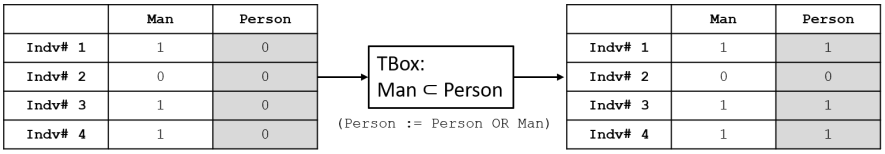

Fig. 4. Example of processing the TBox

5.2 Refinement Operator and Search Algorithm

CONNER borrows the top-down refinement operators used in DL-Learner. The operator used with $\mathcal{ALC}$ is complete, albeit improper ([13], p.69–70). Figure 5 shows a sample $\mathcal{ALC}$ refinement path produced by this operator for a data set discussed in this section. When the hypothesis language is extended to $\mathcal{ALCQ}^{(\mathcal{D})}$, i.e., to include cardinality restrictions and data properties, the corresponding operator is no longer complete ([13], p.72–73).

The original operator is capable of producing a hypothesis consisting of a single clause (making use of the disjunction operator, when needed). A refinement step of the type $\top \rightarrow C \sqcup C \sqcup \dots \sqcup C$ is used to produce a disjunction of concepts that are all subsumed by C, e.g. moving from $Car \sqcup Car$ to $Petrol \sqcup Electric$ (potentially excluding $Diesel$). Here the number of copies of C appears to be part of the user input reminiscent of, say, the limit on the maximum length of the

target clause used in Progol. We have experimented with alternatives, such as cautious learning where the above step is omitted from the refinement operator, and the disjunction of all consistent clauses found is used in the final hypothesis.

The refinement operator can be used under the closed world assumption (CWA), where test examples not covered by the hypothesis are labelled as negative examples. An example of such use was employed when the DL-Learner was tested by its author on Michalski's trains ([13], p.143–146). We have done the same to replicate the results, but we also implement the open world assumption (OWA), which is more commonly used with DL. In this case, two hypotheses H^+ and H^- are learned for the target class and its complement (by swapping the positive and negative examples). Test data is labelled as a positive, resp. negative example when either hypothesis is true, and an "I don't know" label is produced when neither or both hypotheses are true.

The learner uses informed search with a scoring function derived from the one used in the OCEL algorithm of the DL-Learner [13]:

$$ocel\_score(N) = accuracy(N) + 0.5.acc\_gain(N) - 0.02.n \tag{1}$$

Here $acc\_gain(N)$ is the increase in accuracy w.r.t. the parent of N, where N is the candidate hypothesis (e.g. conjunction of concepts and/or restrictions), and n is an upper bound on the length of child concepts, which we set to be equal to the number of concepts in the ontology. We extend this function to take into account the length of a hypothesis (i.e. #concepts + #operators) and its depth which represents (here) the refinement steps taken to reach the current hypothesis, not necessarily its depth in the search tree. Thus the scoring function in CONNER is:

$$conner\_score(N) = 10 * ocel\_score(N) - length(N) - depth(N) \tag{2}$$

The parser currently used to parse the hypotheses tested is Dijkstra's shunting-yard algorithm. The effect of its use here is equivalent to using a binary parse tree, so all conjunctions and disjunctions of three and more concepts are handled as series of applications of the given operator to a pair of concepts. This simplifies the parsing task, but results in a significant drop in performance when compared to simultaneously computing conjunctions or disjunctions of K concepts in the given hypothesis (cf. [2]). A more sophisticated parser or the use of lazy evaluation [2] can be employed with potential benefits, but are not discussed here for reasons of space. We do use *memoization* [14] in the evaluation of each hypothesis, where partial computations are stored and reused.

5.3 Evaluation

The overall run time of the learner is first tested under the CWA on data consisting of multiple copies of the Michalski train set (in its DL version distributed with DL-Learner's examples). While the task results in a relatively limited hypothesis space, this artificial data strains the cover set algorithm exactly as much as any real world data set with the same number of instances and assertions.

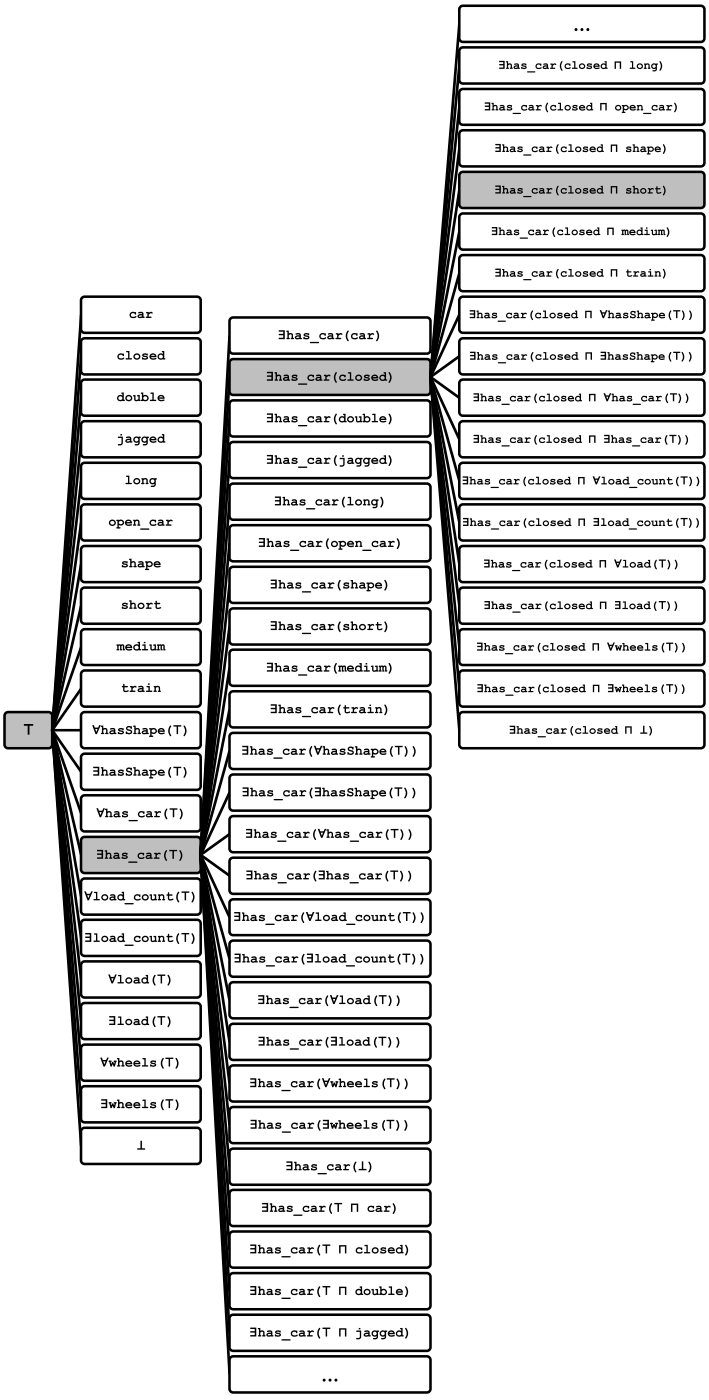

Fig. 5. A sample $\mathcal{ALC}$ refinement path to a solution: Michalski's trains

The results are shown in Table 2 and Figure 6. All experiments are deterministic and the results of multiple runs on the same data are remarkably consistent, so presenting results of single runs was deemed sufficient. Figure 5 shows the search path to the solution found. The solution itself is listed again in Table 3.

Table 2. Learning time vs size of data set (multiple copies of Michalski's trains)

Size (by factor)	Training examples	All individuals	Role assertions	Time [ms]
1x	10	50	149	227
10x	100	410	1,490	233
100x	1,000	4,010	14,900	292
1,000x	10,000	40,010	149,000	291
10,000x	100,000	400,010	1,490,000	712
100,000x	1,000,000	4,000,010	14,900,000	2,764
200,000x	2,000,000	8,000,010	29,800,000	4,836

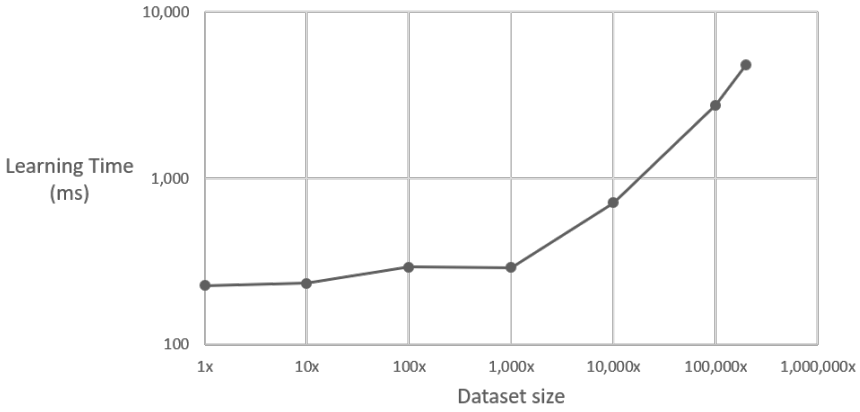

Fig. 6. Plot of results in Table 2: learning time vs size of data set

To confirm that all components of the learner have been correctly implemented, and to further test its speed, another artificial data set in the style of Michalski's trains (here with four cars each) has been generated and used in a second set of experiments.[3] There are 21,156 unique trains (5,184 eastbound and 15,972 westbound) in the data set, which are represented through 105,780 individuals and 148,092 role assertions. Table 4 shows the average run times of the learner for data sets of varying size using 10-fold cross-validation.

[3] The dataset is available from https://osf.io/kf4h6/.

Table 3. Solution to Michalski's trains task in DL and FOL

Description Logic	First Order Logic
∃has_car(Closed ⊓ Short)	eastbound(X) ← $has\_car(X, Y)$ ∧ closed(Y) ∧ short(Y)

Under CWA, out-of-sample accuracy of 100% was achieved for all reported samples except for one of the samples of the lowest reported size. The hypothesis found in most cases is `Train`⊓∃`has_car(Big`⊓∃`inFrontOf.Rectangle)`. The rule ∃`has_car.Rectangle`⊓∃`has_car(Big`⊓∃`inFrontOf.Rectangle)` is found by the DL-Learner as its first solution when applied to all data. The two hypotheses are functionally equivalent.

Table 4. Learning time vs sample size (21,156 unique trains, 10-fold cross-validation)

Proportion of data used for training [%]	0.09	0.9	90
Time [ms]: mean (stdev)	34,106 (90,035)	1,492 (700)	1,862 (66)
Out-of-sample accuracy [%]: mean (stdev)	97.64 (0.80)	100 (0.00)	100 (0.00)

We have also tested CONNER on the well-known mutagenesis ILP dataset [15] in its DL representation, using 10-fold cross-validation, with the following results:

Under CWA: $accuracy = 82.61(8.70)\%$

Under OWA: $precision = 96.00(4.63)\%$,
$recall = 80.43(9.22)\%$,
$F\text{-}score = 87.24(6.27)\%$.

6 Conclusion and Future Work

This article completes the implementation of the first working prototype of the CONNER algorithm. The results suggest that this GPU-powered ILP learner in DL has a lot of potential to scale up learning from ontologies. We have demonstrated how GPGPU can be used to accelerate the computation of the cover set of a hypothesis expressed in $\mathcal{ALC}$ and $\mathcal{ALCQ}^{(\mathcal{D})}$ description logics. The results add to our previous findings (cf. [2]) that even the use of a commodity GPU can provide the ability to process data sets of size well beyond what can be expected from a CPU-based sequential algorithm of the same type, and within a time that makes the evaluation of hypotheses on a data set with 10^7–10^8 training examples a viable proposition.

Future work should consider provisions for the use of external, off-the-shelf reasoners. However, extending the in-house facilities in this aspect is expected to play an important role when the use of concurrency in the search, and its possible

integration with cover set computation are considered. Finally, it should be said that the use of DL as hypothesis language simplifies the task of parallelising the cover set computation when compared to a Horn clause-based hypothesis language. It is clear that some of the problems traditionally tackled through learning in first-order logic can be effectively modelled in DL, and a broader evaluation of the trade-off between expressive power and potential speed up that this choice offers would certainly also provide useful insights.

References

1. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T.: A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113 (2007).
2. Algahtani, E. and Kazakov, D. *GPU-Accelerated Hypothesis Cover Set Testing for Learning in Logic*. CEUR Proceedings of the 28th International Conference on Inductive Logic Programming, (2018). CEUR Workshop Proceedings.
3. Fanizzi,N., d'Amato, C. and Esposito, F.: *DL-FOIL Concept Learning in Description Logics*. Proc. of the Intl Conf. on ILP (ILP2008), 107–121, 2008.
4. Quinlan, R.: Learning Logical Definitions from Relations. *Mach. Learn.* 5:239–266 (1990).
5. Bühmann, L., Lehmann, J. and Westphal, P. DL-Learner – A framework for inductive learning on the Semantic Web, Web Semantics: Science, Services and Agents on the World Wide Web, Volume 39, pp 15–24, 2016.
6. Qomariyah, N. and Kazakov, D.: *Learning from Ordinal Data with Inductive Logic Programming in Description Logic*. Proc. of the Late Breaking Papers of the 27th Intl Conf. on Inductive Logic Programming, 38–50 (2017).
7. Qomariyah, N. and Kazakov, D. *Learning Binary Preference Relations: A Comparison of Logic-based and Statistical Approaches*. Joint Workshop on Interfaces and Human Decision Making for Recommender Systems. Como, Italy (2017).
8. Wu, K. and Haarslev, V.: *A Parallel Reasoner for the Description Logic ALC*. Proc. of the 2012 International Workshop on Description Logics (DL-2012), 2012.
9. Meissner, A.: *A Simple Parallel Reasoning System for the ALC Description Logic*. International Conference on Computational Collective Intelligence - Semantic Web, Social Networks Multiagent Systems, 2009.
10. Chantrapornchai, C. and Choksuchat, C. TripleID-Q: RDF Query Processing Framework Using GPU, IEEE Transactions on Parallel and Distributed Systems, 29(9), pp. 2121–2135, 2018.
11. Martínez-Angeles, C. A., Wu, H., Dutra, I., Costa, V. and Buenabad-Chávez, J.: Relational Learning with GPUs: Accelerating Rule Coverage. *International Journal of Parallel Programming*, 2015.
12. Michalski, R. S., Pattern Recognition as Rule-Guided Inductive Inference, IEEE Transactions on Pattern Analysis and Machine Intelligence, **PAMI-2** (4), 349–361, 1980.
13. Lehmann, J.: Learning OWL Class Expressions, Amsterdam, IOS Press, 2010.
14. Michie, D.: Memo Functions and Machine Learning, *Nature* 218:19–22 (1968).
15. Lavrac, N., Zupanic, D., Weber, I., Kazakov, D., Stepankova, O., and Dzeroski, S. ILPNET repositories on WWW: Inductive Logic Programming Systems, Datasets and Bibliography. *AI Communications*, **9**(4), 1996.

16. Fonseca, N. A., Silva, F. and Camacho, R.: *Strategies to Parallelize ILP Systems.* Inductive Logic Programming: Proc. of the 15^{th} Intl Conf. (2005).
17. Fonseca, N. A., Srinivasan, A., Silva, F. and Camacho, R.: Parallel ILP for Distributed-memory Architectures. *Machine Learning* 74(3):257–279 (2009).
18. Fukunaga, A., Botea, A., Jinnai, Y., and Kishimoto, A.: A Survey of Parallel A*. *arXiv*:1708.05296, 2017.
19. Konstantopoulos, S. K..: *A Data-parallel Version of Aleph.* Proceedings of the Workshop on Parallel and Distributed Computing for Machine Learning (2007).
20. Nishiyama, H. Ohwada, H., Yet Another Parallel Hypothesis Search for Inverse Entailment, 25^{th} International Conference on ILP, 2017.
21. Ohwada, H., Mizoguchi, F.: *Parallel Execution for Speeding up Inductive Logic Programming Systems.* Proceedings of the 9^{th} Intl Workshop on Inductive Logic Programming, pp. 277–286 (1999).
22. Ohwada, H., Nishiyama, H., and Mizoguchi, F.: *Concurrent Execution of Optimal Hypothesis Search for Inverse Entailment*, Inductive Logic Programming, LNCS Vol. 1866, pp. 165-173, 2000.
23. Srinivasan, A., Faruquie, T. A. Joshi, S., Exact Data Parallel Computation for Very Large ILP Datasets, The 20th International Conference on ILP, 2010.
24. Zhou, Y. and Zeng, J.:*Massively parallel A* search on a GPU.* 29^{th} AAAI Conference on Artificial Intelligence, 2015.