An Approach to Parallel Class Expression Learning

An C. Tran, Jens Dietrich, Hans W. Guesgen, and Stephen Marsland

School of Engineering and Advanced Technology, Massey University
Palmerston North, New Zealand
{a.c.tran,j.b.dietrich,h.w.guesgen,s.r.marsland}@massey.ac.nz

Abstract. We propose a Parallel Class Expression Learning algorithm that is inspired by the OWL Class Expression Learner (OCEL) and its extension – Class Expression Learning for Ontology Engineering (CELOE) – proposed by Lehmann et al. in the DL-Learner framework. Our algorithm separates the computation of *partial definitions* from the aggregation of those solutions to an overall *complete definition*, which lends itself to parallelisation. Our algorithm is implemented based on the DL-Learner infrastructure and evaluated using a selection of datasets that have been used in other ILP systems. It is shown that the proposed algorithm is suitable for learning problems that can only be solved by complex (long) definitions. Our approach is part of an ontology-based abnormality detection framework that is developed to be used in smart homes.

Keywords: description logic learning, class expression learning, parcel, parallel learning, abnormal behaviour detection.

1 Introduction

Description logic (DL) is a popular formalism used in knowledge representation. Amongst its strengths are the availability of a formal semantics, the standardisation of description-logic-based languages by the W3C (RDFS and several versions and flavours of OWL [1]), and the availability of robust tools to edit and reason about ontologies.

The primary problem that motivates our research is the classification of normal and abnormal activities in a smart home environment, where elderly people are monitored by a system that can alert medical professionals if abnormal behaviour is detected [2]. It is important in this scenario that we do not miss any abnormal behaviours, in particular if these behaviours potentially pose a threat to the person living in the smart home. In technical terms, this means that we aim at avoiding false positives in the class of normal behaviours.

Using a symbolic (logic-based) approach in this context has the advantage that systems can be designed that are inherently more trustworthy than subsymbolic machine learning approaches, as system decisions are traceable through the proofs associated with classifications.

A common problem in symbolic AI is to find the "right" set of rules. Here, by rules we mean the expressions that define concepts such as normal and abnormal

A. Bikakis and A. Giurca (Eds.): RuleML 2012, LNCS 7438, pp. 302–316, 2012.

behaviour. It takes a significant effort to create and maintain such a set of rules, and comprehensive validation against real world data is needed to assess its accuracy. An alternative approach is to learn the rules directly from sample datasets. This has several advantages: if it can be demonstrated by means of a formal proof that the learning algorithm produces expected rules and all available data have been fed into the learning algorithm, then validation is no longer necessary. Also, if new training data becomes available, the algorithm can be easily reapplied and the new definitions can be created. In other words, the system can easily be re-calibrated as needed.

However, this implies that we need to apply the learning algorithm often, and on large datasets. Therefore, the scalability of the algorithm becomes a major concern. Benchmark tests performed by Hellmann [3] indicate that the DL-Learner [4] is a suitable starting point for the development of an expressive and scalable DL learning algorithm. Our experiments with the use cases described in [2] show that CELOE and its ancestor – OCEL – are the most suitable algorithms for solving this problem amongst the algorithms implemented in DL-Learner. These algorithms generate expressions of increasing complexity starting with explicitly defined classes in the ontology, and assess the accuracy of these expressions against a training set consisting of positive and negative examples. If an expression with a sufficiently high accuracy score is found, the algorithm terminates and the expression is returned as the result of the computation.

Unfortunately, our experiments suggest that these algorithms do not have the level of scalability necessary to be used in the smart home application domain, particularly for smart home datasets generated from the uses cases in [2]. An analysis of these algorithms reveals that the accuracy measure used to direct the generation and evaluation of descriptions is a combination of correctness (no negative examples are covered by the computed expression) and completeness (all positive examples are covered). Motivated by the need for a higher accuracy in learning normal behaviour, we propose a DL learning algorithm that separates this process into two steps: first the generation of correct rules (i.e. they do not cover any negative examples) but not necessarily cover all positive examples, and then the aggregation of those rules into a (sufficiently) complete solution. In addition, there is no need to serialise these two steps: they can be performed concurrently. In particular, multiple branches within the tree of possible descriptions can be traversed concurrently by multiple workers to find partial results, while a central reducer aggregates partial results to the overall solution until all positive examples are covered. The reducer also has the responsibility of removing redundant definitions covering overlapping sets of positive examples. We discuss several strategies to do this in section 3.

This approach follows the general ideas of the map-reduce architecture [5] and therefore lends itself to parallelisation using either multiple threads that can take advantage of multi-core processors, or may be developed for cloud computing platforms such as Amazon EC2 in the future. It also has the advantage that the resulting system shows anytime characteristics [6], which means that: i) it

can return a correct solution even if it is interrupted before a complete solution is computed, and ii) the solution is expected to improve (i.e., become more complete) with increasing runtime of the system.

2 Related Work

Description logic learning has its roots in inductive logic programming (ILP) [7,8,9]. In ILP, sets of positive and negative facts and some background knowledge are given, and an ILP algorithm is used to compute a logic program that describes all the positive and none of the negative examples. There are two fundamentally different strategies to compute this program: top-down and bottom-up [7]. Combined strategies have also been investigated by different authors [10].

In description logic learning we are interested to find concepts that describe all given positive examples, but do not describe any of the negative examples. Our work is directly based on the DL-Learner framework [4], particularly the CELOE and OCEL algorithms. Theses algorithms use a top-down strategy to learn concepts. Starting with the root of the concept class hierarchy, concepts are refined by means of specialisation until a suitable concept is found. The description learning space expansion is mainly directed by the accuracy (a combination of correctness and completeness) of concepts with respect to the positive and negative examples, the complexity of the expressions, the accuracy gained in each expansion step, and some other factors. Note that the refinement operator used by these algorithms also has some implicit support for the bottom-up strategy as it will generate complex expressions using disjunctions.

Lisi [11] has proposed an alternative top-down approach based on the hybrid $\mathcal{AL}$-log language which combines $\mathcal{ALC}$ description logic and Datalog for knowledge representation. This makes it possible to learn Datalog rules on top of ontologies.

Several other approaches to concept learning have been proposed. This includes LCSLearn [12], an early bottom-up approach that creates concepts by joining most specific concepts created for individuals (positive examples) using disjunction. This is a very simplistic approach that creates large concept definitions that are not truly intentional in a sense that those definitions are only enumerations of the sets of individuals they define. YinYang [13] is a hybrid learner that uses a combination of bottom-up (starting from most specific concepts) generalisation and top-down specialisation strategies.

Our contribution is similar to DL-FOIL [14]: we separate the computation of partial correct concepts from the computation of a complete concept. There are two main differences however: (i) the algorithm proposed by [14] is serial by nature as the computation of the partial correct concepts is executed in an inner loop, while we use a parallel computation model, and (ii) we propose an extra reduction step to compute an optimal set of partial concepts to be used in order to compute the overall (complete) result. The above differences aim to bring some

benefits: i) improved scalability of the algorithm due to parallelisation, ii) the any time characteristic of the algorithm, which means that the algorithm can produce correct (but not necessarily complete) solutions if interrupted prematurely, and iii) the flexibility gained through a separate reduction step that allows us to tradeoff completeness, number of partial definitions and the (average) lengths of the partial definitions.

3 Algorithm

Our algorithm is inspired by the popular map-reduce framework [5] that is widely used to process large amounts of data. Here, input data to be processed is divided into several pieces (sub-problems) and processed by multiple workers (map step) in parallel and then the intermediate results are aggregated (reduce step) into a final result.

In the context of our work, the problem of finding a correct and complete concept definition is mapped to workers responsible for refining and evaluating candidate concepts. The actual refinement operator used for this is the refinement operator proposed in [15], which is one of the refinement operators currently supported by the DL-Learner framework. However, the operator is customised by disabling the generation of disjunctions as the generalisation is done in a separate reducer step. In addition, its numerical data properties refinement has also been improved by using a better strategy for identifying the domain of the refinement.

The reduce step consists of combining the partial definitions until a complete (or at least sufficiently good) coverage of the positive examples is obtained. Often, this yields a set of partial definitions that is redundant in the sense that multiple definitions cover the same positive examples, and that a proper subset of definitions exists that is also complete with respect to the positive examples. Here, we propose the use of a set coverage algorithm [16] to find such a subset. This allows us to tailor the main algorithm, e.g. in order to compute smaller sets of concepts, or sets of concepts with a shorter average expression length (lower complexity).

An informal illustration of the algorithm is given in Figure 1. It shows the interaction between the two parts of the algorithm: the reducer that aggregates and compacts the partial definitions, and the worker(s) producing the partial definitions. The coordination is done using an agenda. The agenda contains the concepts to be refined, and an ordering of its nodes generated by an expansion heuristic. This means that there is always a top element representing the most promising concept for refinements based on the heuristic used. This element is assigned to workers for processing. The current search heuristic is based on the heuristic used in [17] that associates concepts with a score mainly based on their accuracy (combination of correctness and completeness). In addition, a level of penalty on complexity of the concepts (short expressions are preferred), bonus on accuracy gained, etc. are also applied. In our learning heuristic, we also penalise long descriptions to avoid infinite deep searches because the refinement

operator used in our learner is infinite. Instead, a description's score is mainly based on the correctness. Note that our learning heuristic can help to avoid infinite deep searches, but this does not avoid infinite loops in the whole learning process because the refinement operator itself is infinite. Therefore, the termination is controlled by the accuracy of the definitions generated and the timeout mechanism.

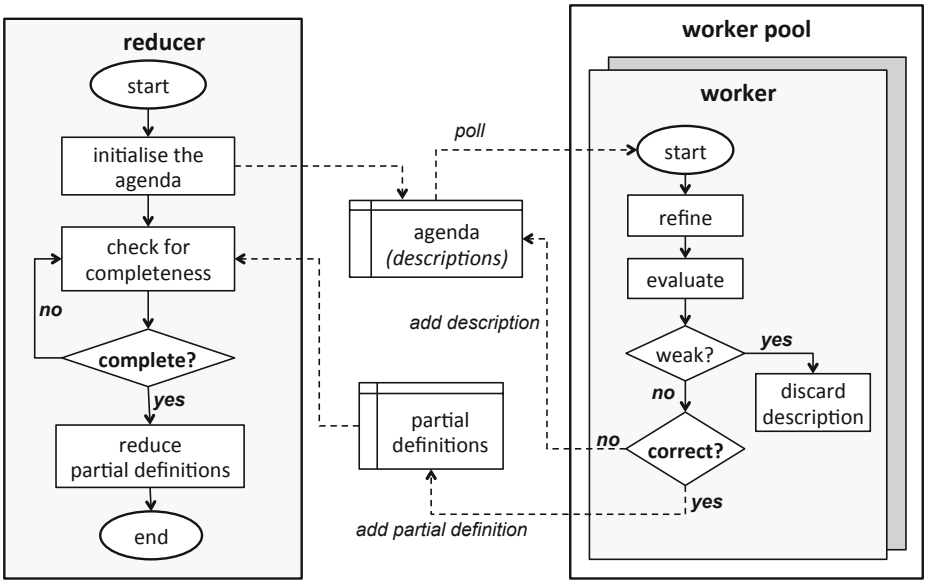

Fig. 1. Reducer-Workers interaction

For a more formal definition, we introduce some notations first. A *learning problem* is a structure (K, E^+, E^-) that consists of a knowledge base K, a set of positive examples E^+ and a set of negative examples E^-. We say that a concept C *covers* an example e iff $K \models C(e)$. A concept C is called *correct* if it does not cover any negative example and *weak* if it covers none of the positive examples. We also refer to correct concepts as *definitions*. A definition is called a *partial definition* if it covers at least one and less than all positive examples, and a *complete definition* if it covers all positive examples.

There are some useful metrics to measure the amounts of correctness and completeness of a concept C. Let $R(C)$ be the set of individuals covered by C. Then $un(C) = E^- \setminus R(C)$ is the set of negative examples not covered by C, and $cp(C) = E^+ \cap R(C)$ the set of covered positive examples. We can then define *correctness*, *completeness* and *accuracy* using predictive accuracy methodology as follows:

$$correctness(C) = \frac{|un(C)|}{|E^-|}$$

$$completeness(C) = \frac{|cp(C)|}{|E^+|}$$

$$accuracy(C) = \frac{|cp(C)| + |un(C)|}{|E^+ \cup E^-|}$$

Our algorithm can now be defined in two parts. The computational heavy part is done by the *multiple* workers: this is the refinement and the evaluation of concepts. In particular, the evaluation of (complex) concepts (i.e., the check whether a given example is defined by a concept) requires an ontology reasoner. By default, Pellet [18] is used for this purpose.

The reducer creates a worker pool, which manages a number of workers, and assigns new concepts for refinements and evaluations to worker pool until the completeness of the combined partial definitions is sufficient. Then the reducer tries to reduce the number of partial definitions in order to remove redundancies using a *reduction* function. While the reducer computes sets of concepts, these sets can be easily aggregated into a single concept using disjunction.

Algorithm 1 ***(Reducer Algorithm).*** *For a given learning problem* (K, E^+, E^-)*, a noise value* $\varepsilon \in [0, 1]$ *and a pool of workers, compute a set of partial correct solutions* $\{C_i\}$ *such that* $\text{completeness}(\sqcup_i(C_i)) \geq 1 - \varepsilon$*.*

1: *agenda* := $\{\top\}$
2: *solutions* := $\{\}$
3: *uncovered_positive_examples* := E^+
4: create a *worker_pool*
5: **while** $|uncovered\_positive\_examples| > |E^+| \times \varepsilon$ **do**
6: wait for new partial definition(s) produced by workers
7: reduce(*solutions*)
8: **return** *solutions*

The worker algorithm refines a concept and evaluates the results of the refinement. It will first check whether concepts are weak. If this is the case the concept can be safely removed from the computation as no partial definition can be computed through specialisation. If a concept is a partial definition (i.e., correct and not weak), it is added to the (shared) partial definitions set. If a concept is not weak, but also not correct (i.e., if it covers some positive and some negative examples), it is added back to the agenda and therefore scheduled for further refinement. Note that the concepts that have been refined can be scheduled for further refinement. This is necessary as each refinement step only computes a finite (and usually small) number of new concepts, usually constrained by a complexity constraint. For example, a concept of a given size N could first be refined to compute new concepts of a length $N+1$, and later it could be revisited to compute more concepts of length $N + 2$, etc. This technique is used in the original DL-Learner and discussed in detail in [17]. When implementing workers, an additional redundancy check takes place to make sure that the same concept computed from different branches in the search tree is not added twice to the agenda.

Algorithm 2 ***(Worker Algorithm).*** *For a set of positive examples E^+ and a set of negative examples E^-, refine a given concept C using a refinement operator ρ, and evaluate the refinements.*

```
1: refinements := ρ(C)
2: for all description ∈ refinements do
3:    positive_covered := positive examples covered by refinement
4:    if description is not weak then
5:       if description is correct then
6:          add description to solutions
7:          uncovered_positive_examples := uncovered_positive_examples \ positive_covered
8:       else
9:          add description to agenda
```

For the actual reduction step, we have investigated three simple algorithms:

- GMPC (greedy minimise partial definition count)
- GMPL (greedy minimise partial definition length)
- GOLR (greedy online algorithm - first in first out)

As the names suggest, they are all greedy optimisation algorithms that are based on sorting the partial definitions. Once the partial definitions are sorted, a new solution set (called the reduction set) is created and solutions are added to this set in descending order. A definition is added only if it covers at least one positive example not yet covered by any other solution in the reduction set. Details are given in algorithm 3.

We have used different sort criteria, resulting in the different algorithms. In GMPC, we sort partial definitions according to the number of positive examples they cover, preferring definitions that cover more positive examples. If two definitions cover the same number of positive examples, we use the lexicographical order of the respective string representations as a tiebreaker. This is important to make the results repeatable. Otherwise the order that is used when iterating over definitions could depend on internal system hash codes which the application does not control.

In GMPL, we sort definitions according to their expression lengths, preferring definitions with a shorter length. If two definitions have the same expression lengths, we again use the lexicographical order.

In GOLR, we use time stamps assigned to definitions when they are added to the solutions, preferring definitions that have been added earlier. While the other two heuristics have to be run in batch mode after a complete set of definitions has been computed, this algorithms can be employed just in time, the reduction can take place whenever a new definition is found and added. This algorithm is therefore very space efficient compared to the other two. On the other hand, how timestamps are assigned in an application may depend on thread scheduling. This again cannot be controlled completely by the application, causing variations in the results between runs.

More formally, we define a generic reduction algorithm that is based on a sort function as follows:

Algorithm 3 ***Generic greedy reduction algorithm based on sorting.*** *For a set of definitions D and a function* cover : $D \rightarrow 2^{E^+}$ *that associates partial definitions with the sets of covered positive examples, compute a subset $D' \subseteq D$ such that $\bigcup_{d \in D} \text{cover}(d) = \bigcup_{d \in D'} \text{cover}(d) = E^+$*

1: $D' := \emptyset$
2: *positive_covered* $:= \emptyset$
3: sort D
4: **while** D is not empty **and** *positive_covered* $\subset E^+$ **do**
5: $d := \text{poll}(D)$
6: **if** $\text{cover}(d) \not\subseteq$ *positive_covered* **then**
7: $D' := D' \cup d$
8: *positive_covered* $:=$ *positive_covered* $\cup \text{cover}(d)$
9: **return** D'

4 Validation

4.1 Methodology

We have implemented our algorithm using Java. The package is called the ParCEL (PARallel Class Expression Learning) and a minimal set of this package is available at `https://parcel-mu.googlecode.com/`. The algorithm is also integrated into the DL-Learner repository `http://dl-learner.svn.sourceforge.net`.

In the validation, we were not only interested in measuring the overall computation time of the benchmark learning problems, but also in measuring how quickly accuracy improved during the computation. We consider this to be important in scenarios where an application could intercept the learner once a sufficiently complete solution has been computed. For this purpose, two different sorts of experiments have been performed: i) a 10-fold cross validation to measure the learning time and accuracy, ii) a training run on different levels of parallelism to observe the accuracy improvement on the training set. The former follows the standard cross-validation methodology in statistics. In the later experiment, we start a background watcher thread that frequently takes probes from learner thread(s) and records them. This thread represents some overhead, so the net computation times are in fact slightly less than the values given below. We benchmarked our learner against the CELOE and OCEL algorithms.

In our experiments, we have used a number of datasets that have been used by other authors in similar experiments [19,20] to benefit comparisons. All datasets used in this paper, except the UCA1 which will be described later, can be found in any DL-Learner release or in the DL-Learner repository. An overview of these datasets is given in Tables 1 and 2. Note that DL-Learner is in the development process. New revisions have been being issued very regularly and the learning time and accuracy for the same dataset may change over the revisions. In our

Table 1. Experiment datasets summary

	Moral simple	Forte	Poker	Carcino genesis	Family benchmark	UCA1
Classes	43	3	2	142	4	65
Classes assertions	4646	86	374	22,372	606	300
Object properties	0	3	6	4	4	4
Object property assertions	0	251	1080	40,666	728	200
Data properties	0	0	0	15	0	11
Data property assertions	0	0	0	11,185	0	200
Examples	102p/ 100n	23p/ 163n	4p/ 151n	182p/ 155n	-	73p/ 77n

Table 2. Family benchmark datasets - Number of examples

	Aunt	Uncle	Cousin	Daughter	Father	Grandson	Brother
Examples	41p/ 41n	38p/ 38n	71p/ 71n	52p/ 52n	60p/ 60n	30p/ 30n	43p/ 30n

experiment, we have used DL-Learner version 1.0.1. In addition, we used the default learning configuration for CELOE/OCEL and ParCEL for all datasets.

In our experiment, we also used an additional dataset – UCA1 – which is extracted from the use case descriptions of the smart home domain we are primarily interested in [2]. This use case describes an over-long shower scenario in a smart home, in which the showering duration depends upon the season of year. This dataset is supported by an underlying smart home ontology which contains the basic concepts for describing the activities in smart homes and some context information, particularly the temporal and spacial information. The scenario was modelled using a Bayesian Network and then the network was used to generate the simulation dataset. The actual dataset contains a set of showering activities, their start times and durations.

For the experiments, we used a Linux server with a 8 x Intel Xeon E5440 @2.83GHz processor, 32GB memory and the Redhat 4.1.2 (Linux version 2.6.18) operating system with a JRE 1.6.0 (64-bit) Java Virtual Machine (JVM). The heap size of the JMV in our experiments is 5GB.

The length of definition reported is the length of the best description learnt so far.

4.2 Result Summary

Table 3 shows a summary of the results. The reduction mechanism used here is GMPC, i.e. we use a simple greedy algorithm to reduce the number of partial definitions.

Table 3. Experiment result summary *(averages ± standard deviations of 10 folds)*

problem	time (s)		accuracy (%)		(avg. partial)* def. length		no. of partial def.
	CELOE	ParCEL	CELOE	ParCEL	CELOE	ParCEL	ParCEL
Moral	0.148 ± 0.03	0.024 ± 0.012	100 ± 0	100 ± 0	3 ± 0	1.517 ± 0.053	2.1 ± 0.316
Forte	6.664 ± 3.524	0.064 ± 0.06	98 ± 4.216	100 ± 0	12 ± 1.054	9.167 ± 3.15	1.9 ± 0.738
Poker-straight	0.36 ± 0.709	0.274 ± 0.189	94.333 ± 13.152	93.238 ± 8.834	11.7 ± 0.675	10.9 ± 1.308	1.7 ± 0.675
UCA1	OOMem** @2259s	29.747 ± 5.768	91.238 ± 6.409	100 ± 0	9 ± 0	12.75 ± 0	4 ± 0
CarcinoGenesis	*int.**** @2000s	*int.**** @2000s	54.618 ± 2.711	55.865 ± 9.516	4.8 ± 0.422	55.865 ± 9.516	72.7 ± 3.433
Aunt	34.129 ± 14.94	0.256 ± 0.151	96.25 ± 11.859	100 ± 0	19 ± 0	8.267 ± 0.492	3.1 ± 0.316
Brother	0.191 ± 0.157	0.026 ± 0.015	100 ± 0	100 ± 0	6 ± 0	5.5 ± 0.707	1 ± 0
Uncle	34.129 ± 14.94	0.293 ± 0.178	96.25 ± 11.859	98.75 ± 3.953	19 ± 14.94	8.4 ± 0.378	3 ± 0
Cousin	471.555 ± 284.734	0.544 ± 0.199	94.286 ± 6.564	100 ± 0	23.4 ± 2.591	8.5 ± 0	2 ± 0
Daughter	0.023 ± 0.019	0.027 ± 0.025	100 ± 0	100 ± 0	5 ± 0	5.25 ± 1.087	1.1 ± 0.316
Father	0.023 ± 0.104	0.031 ± 0.03	100 ± 0	100 ± 0	5 ± 0	5.5 ± 0.527	1 ± 0
Grandson	0.054 ± 0.059	0.075 ± 0.066	100 ± 0	100 ± 0	7 ± 0	7.4 ± 0.459	1.3 ± 0.483

Note: *: *For ParCEL* **: *Out Of Memory* ***: *Interrupted*

In general, CELOE performed better on smaller datasets with simple definitions required like Daughter, Father, and Grandson in the Family dataset. This can be attributed to the more complex runtime architecture of ParCEL that requires additional overhead for thread creation and synchronisation. However, when either the data or the queries become more complex, the ParCEL outperforms CELOE. This is apparent in the Family dataset: CELOE is much better in answering simple queries that require less reasoning, while ParCEL performs better on complex queries on derived relationships. There are two reasons for this: (i) ParCEL obviously better utilises the multi-core processor(s) due to its parallel architecture, and (ii) different ParCEL workers explore different branches of the search tree at the same time, while CELOE may spend longer time to explore branches that at the end do not yield results, (iii) ParCEL plays a tradeoff between the readability of the learning result again the accuracy and learning time using the combination of specialisation and generalisation.

Most importantly for our application scenario, ParCEL outperformed CELOE on the UCA1 dataset, both on learning time and accuracy. Although the length of the definition produced by ParCEL is longer than CELOE, it is readability and describes well our scenario. For examples, one of the learnt results for the concept *normal showering* is the disjunction of the following partial definitions:

1. `EXISTS activityHasDuration.(hasDurationValue >= 4.5 AND hasDurationValue <= 15.5)`
2. `EXISTS activityHasDuration.(hasDurationValue >= 15.5 AND hasDurationValue <= 19.5) AND EXISTS activityHasStarttime.Spring`
3. `EXISTS activityHasDuration.(hasDurationValue >= 15.5 AND <= 19.5) AND EXISTS activityHasStarttime.Summer`
4. `EXISTS activityHasStarttime.Autumn AND ALL activityHasDuration. (hasDurationValue >= 4.5 AND hasDurationValue <= 19.5`

One of the most difficult learning problems in our experiment is the Carcino-Genesis dataset. Learning results for this dataset reported in [14,20] show that CELOE gives the best accuracy in comparison with other learners with a certain learning configuration. In our experiment, neither CELEO nor ParCEL could find an accurate definition on the training dataset before they ran out of memory. CELOE runs out of memory in around 2100 seconds and ParCEL can run for approximately 15800 seconds with the same JVM heap size. The experiment result shows that ParCEL outperformed CELOE on the training dataset by 36%. However, the accuracy for the testing dataset is not significantly different: 54.618% ± 2.711% for CELOE and 55.597% ± 9.516% for ParCEL. The above accuracy is obtained at 2000 seconds when CELOE is approaching the out of memory exception. Although ParCEL can run for more than 15800 seconds, we only let it run the same amount of time as CELOE since our experiments demonstrate that the accuracy does not improve significantly for the longer runs. Note that this result is generated by the default learning configuration and it may be different for the refined learning configuration. For example, show that the predictive accuracy can be improved by allowing a level of noise in training dataset. However, this has not yet been studied in our research.

A paired t-test rejected the null hypothesis (that the running times came from the same distribution) at the 5% confidence level, for both the running times and accuracies in Table 3. However, while an F test showed that the accuracies were normally distributed, this was not true for the running times, and so this result should be treated with caution.

4.3 Performance Improvement Comparison

We have used a monitoring thread as described above to investigate the level of approximation that the learners can achieve. This is shown in figure 2. The slightly odd values on the x-axis are due to the fact that they were taken from the timestamps when the monitoring thread returns data. CELOE computes a solution of about 0.55 accuracy very quickly (the first probe already returns this values), but then "stays flat". On the other hand, the ParCEL almost reaches

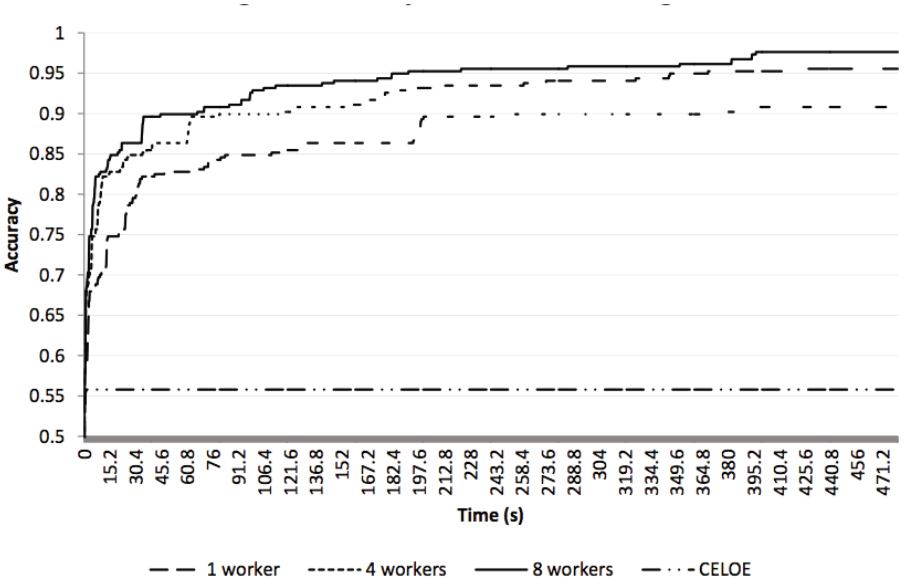

Fig. 2. Learning using the Carcino Genesis dataset

maximum accuracy, i.e., a level of completeness of more than 0.95. The figure also shows the impact of the number of threads: adding more threads can speed up the computation.

Figure 3 shows details for the UCA1 dataset. In this case, CELOE cannot compute a very accurate result before it times out, whereas ParCEL succeeds. Adding more threads can again speed up the computation significantly.

Fig. 3. Learning using the UCA1 dataset

4.4 Definition Aggregation

Finally, we conducted an experiment to compare the three reduction algorithms discussed earlier. Here we measured the length of descriptions and the number of descriptions defined. To compare the description length, we use the method defined above, i.e., we measure the lengths of the virtual disjunction we could create from the set of partial definitions. Comparison between reduction strategies is given in Table 4. The results shows that GMPC gives the shortest definition in most of the experiments and thus the definitions are more readable. On the other hand, GORL produces the longest definitions in all the experiments. However, GMPC requires all partial definitions to be kept until the learning finishes while the GORL can perform the reduction on the fly when the learning is happening. This may give us a selection on the tradeoff between the readability of the learnt definition and the memory used by the learner as well as the learning time.

Table 4. Definition length comparison between algorithms and reduction strategies *(averages ± standard deviations of 10 folds)*

	CELOE	GMPC		GOLR		GMPL	
dataset	def. length	no. of partial def.	avg. partial def. length	no. of partial def.	avg. partial def. length	no. of partial def.	avg. partial def. length
Moral	3 ± 0	2.1 ± 0.316	1.517 ± 0.053	3.4 ± 0.966	2.15 ± 1.263	3 ± 0	1.667 ± 0
Forte	12 ± 1.054	1.9 ± 0.738	9.167 ± 3.15	2.3 ± 0.675	7.708 ± 0.429	2.3 ± 0.675	7.617 ± 0.209
Poker-straight	11.7 ± 0.675	1.7 ± 0.675	10.9 ± 1.308	2.7 ± 0.483	9.883 ± 1.457	2.7 ± 0.483	9.483 ± 0.976
UCA1	9 ± 0 *@OOMem*	4 ± 0	12.75 ± 0	9.7 ± 1.059	13.479 ± 0.217	5.5 ± 1.179	13.063 ± 0.466
Aunt	19 ± 0	3.1 ± 0.316	8.267 ± 0.492	8.9 ± 2.079	7.75 ± 0.289	8.3 ± 1.418	7.477 ± 0.308
Brother	6 ± 0	1 ± 0	5.2 ± 0.422	1 ± 0	5.6 ± 0.516	1 ± 0	5.1 ± 0.316
Uncle	19 ± 14.94	3 ± 0	8.4 ± 0.378	7.1 ± 1.287	7.917 ± 0.163	6.8 ± 1.135	7.746 ± 0.482
Cousin	23.4 ± 2.591	2 ± 0	8.5 ± 0	8.2 ± 4.158	8.5 ± 0.575	5.7 ± 1.252	8.095 ± 0.208
Daughter	5 ± 0	1.1 ± 0.316	5.25 ± 1.087	1.5 ± 0.527	7.55 ± 2.409	1.4 ± 0.843	5.333 ± 0.471
Father	5 ± 0	1 ± 0	5.5 ± 0.527	1 ± 0	5.2 ± 0.422	1 ± 0	5.3 ± 0.483
Grandson	7 ± 0	1.3 ± 0.483	7.4 ± 0.459	2.9 ± 0.568	7.525 ± 0.553	2.5 ± 0.707	7.2 ± 0.502

5 Conclusion

Our approach to parallelising the class expression logic learning shows promising results on the datasets used in the evaluation. By dividing the learning process into two separate stages, one for generating correct but potentially incomplete definitions and another one for aggregating the partial definition to a complete (or nearly complete) solution, we were able to spread the task over several subprocesses that can run in parallel. As a result, we are able to utilise multi-core machines and potentially also cloud computing, which makes the task of description logic learning more scalable.

Since the aggregation of partial solutions is now not integrated in the refinement procedure anymore but runs as a separate thread concurrently to it, we are able to easily test different strategies for aggregating the partial definitions. The ones that we have tested are greedy strategies which avoid exhaustive search for an optimal aggregate and therefore scale more easily.

The main motivation for our research is the classification of normal and abnormal activities in a smart home environment, in which UCA1 is one of our simulation datasets. With this dataset, DL-Learner gave the best solution with 91.2% accuracy before it ran out of memory (with 5GB heap space allocated and 38 minutes run time). Describing this problem requires a description with the minimal length around 42 to 73 and this may be one of the potential causes that exploded DL-Learner memory. Generally, 91.2% accuracy is a good learning result. However, in this application domain, any false positive or false negative classifications may affect strongly on the inhabitant safety and thus an accurate definition is preferred to a readable one. With the combination of specialisation and generalisation and the parallelisation approach, we are preliminarily getting success with the first datasets in our research. In addition, completeness of the partial definitions may provide us an interesting dimension in our classification: belief of the classification.

In most of the datasets in our experiment, our learner algorithm gives a promising result both in accuracy and learning time. The only dataset that our learner could not give a better result is CarcinoGenesis. It shows that our learning currently does not deal well with noise data and this is a future development for our learner so that it can deal with various learning problems.

References

1. McGuinness, D., Van Harmelen, F., et al.: OWL web ontology language overview. W3C Recommendation 10, (2004) 2004–03
2. Tran, A.C., Marsland, S., Dietrich, J., Guesgen, H.W., Lyons, P.: Use Cases for Abnormal Behaviour Detection in Smart Homes. In: Lee, Y., Bien, Z.Z., Mokhtari, M., Kim, J.T., Park, M., Kim, J., Lee, H., Khalil, I. (eds.) ICOST 2010. LNCS, vol. 6159, pp. 144–151. Springer, Heidelberg (2010)
3. Hellmann, S., Lehmann, J., Auer, S.: Learning of OWL class descriptions on very large knowledge bases. Int. J. Semantic Web Inf. Syst. 5(2), 25–48 (2009)
4. Lehmann, J.: DL-Learner: learning concepts in description logics. The Journal of Machine Learning Research 10, 2639–2642 (2009)

5. Ghemawat, S., Dean, J.: Mapreduce: Simplified data processing on large clusters. In: Symposium on Operating System Design and Implementation, OSDI 2004, San Francisco, CA, USA (2004)
6. Zilberstein, S.: Using anytime algorithms in intelligent systems. AI Magazine 17(3), 73–83 (1996)
7. Muggleton, S.: Inductive logic programming. New Generation Computing 8(4), 295–318 (1991)
8. d'Amato, C., Fanizzi, N., Esposito, F.: Inductive learning for the semantic web: What does it buy? Semantic Web 1(1), 53–59 (2010)
9. Lavrac, N., Dzeroski, S.: Inductive Logic Programming: Techniques and Applications. Ellis Horwood, New York (1994)
10. Zelle, J.M., Mooney, R.J., Konvisser, J.B.: Combining top-down and bottom-up techniques in inductive logic programming. In: Proceedings of the 11th International Conference on Machine Learning, pp. 343–351 (1994)
11. Lisi, F.A.: Building rules on top of ontologies for the semantic web with inductive logic programming. Theory and Practice of Logic Programming 8(3), 271–300 (2008)
12. Cohen, W., Hirsh, H.: Learning the classic description logic: Theoretical and experimental results. In: Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning, Citeseer, pp. 121–133 (1994)
13. Iannone, L., Palmisano, I.: An Algorithm Based on Counterfactuals for Concept Learning in the Semantic Web. In: Ali, M., Esposito, F. (eds.) IEA/AIE 2005. LNCS (LNAI), vol. 3533, pp. 370–379. Springer, Heidelberg (2005)
14. Fanizzi, N., d'Amato, C., Esposito, F.: DL-FOIL Concept Learning in Description Logics. In: Železný, F., Lavrač, N. (eds.) ILP 2008. LNCS (LNAI), vol. 5194, pp. 107–121. Springer, Heidelberg (2008)
15. Lehmann, J., Hitzler, P.: Concept learning in description logics using refinement operators. Machine Learning 78(1), 203–250 (2010)
16. Cormen, T.: Introduction to algorithms. The MIT Press (2001)
17. Lehmann, J., Auer, S., Tramp, S., et al.: Class expression learning for ontology engineering. In: Web Semantics: Science, Services and Agents on the World Wide Web (2011)
18. Sirin, E., Parsia, B., Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical owl-dl reasoner. Web Semantics: Science, Services and Agents on the World Wide Web 5(2), 51–53 (2007)
19. Fahnrich, K., Lehmann, J., Hellmann, S.: Comparison of concept learning algorithms (2008)
20. Železný, F., Srinivasan, A., Page, D.L.: Lattice-Search Runtime Distributions May Be Heavy-Tailed. In: Matwin, S., Sammut, C. (eds.) ILP 2002. LNCS (LNAI), vol. 2583, pp. 333–345. Springer, Heidelberg (2003)