Concurrent Classification of $\mathcal{EL}$ Ontologies

Yevgeny Kazakov, Markus Krötzsch, and František Simančík

Department of Computer Science, University of Oxford, UK

Abstract. We describe an optimised consequence-based procedure for classification of ontologies expressed in a polynomial fragment $\mathcal{ELH}_{\mathcal{R}+}$ of the OWL 2 EL profile. A distinguishing property of our procedure is that it can take advantage of multiple processors/cores, which increasingly prevail in computer systems. Our solution is based on a variant of the 'given clause' saturation algorithm for first-order theorem proving, where we assign derived axioms to 'contexts' within which they can be used and which can be processed independently. We describe an implementation of our procedure within the Java-based reasoner ELK. Our implementation is light-weight in the sense that an overhead of managing concurrent computations is minimal. This is achieved by employing lock-free data structures and operations such as 'compare-and-swap'. We report on preliminary experimental results demonstrating a substantial speedup of ontology classification on multi-core systems. In particular, one of the largest and widely-used medical ontologies SNOMED CT can be classified in as little as 5 seconds.

1 Introduction

Ontology classification is one of the key reasoning services used in the development of OWL ontologies. The goal of classification is to compute the hierarchical representation of the subclass (a.k.a. 'is-a') relations between the classes in the ontology based on their semantic definitions. Ontology classification, and ontology reasoning in general, is a computationally intensive task which can introduce a considerable delay into the ontology development cycle.

Many works have focused on the development of techniques to reduce classification times by optimizing the underlying (mostly tableaux-based) procedures so that they produce fewer inferences. In this paper we study another way of reducing the classification time, which is achieved by performing several inferences in parallel, i.e., concurrently. Concurrent algorithms and data structures have gained substantial practical importance due to the widespread availability of multi-core and multi-processor systems.

Nonetheless, concurrent classification of ontologies is challenging and only few works cover this subject. Approaches range from generic 'divide-and-conquer' strategies when the ontology is divided into several independent components [18] to more specific strategies involving parallel construction of taxonomies [1], concurrent execution of tableau rules [11,12], distributed resolution procedures [15], and MapReduce-based distribution approaches [14]. The practical improvements offered by these strategies, however, remain yet to be demonstrated, as

L. Aroyo et al. (Eds.): ISWC 2011, Part I, LNCS 7031, pp. 305–320, 2011.

empirical evaluation of the proposed approaches is rather limited. As of today, none of the commonly used ontology reasoners, including CEL, FaCT++, HermiT, jCEL, Pellet, and RACER, can make use of multiple processors or cores.

In this paper, we consider a 'consequence-based' classification procedure, which works by deriving logical consequences from the axioms in the ontology. Such procedures were first introduced for the $\mathcal{EL}$ family of tractable description logics (DLs) [2], which became the basis of the OWL 2 EL profile [13]. Later the $\mathcal{EL}$-style classification procedures have been formulated for more expressive languages, such as Horn-$\mathcal{SHIQ}$ [7] and $\mathcal{ALC}$ [16]. Consequence-based procedures have several distinguished properties, such as optimal worst-case complexity, 'pay-as-you-go' behaviour, and lack of non-determinism (even for expressive DLs such as $\mathcal{ALC}$). In this paper we will demonstrate that such procedures are also particularly suitable for concurrent classification of ontologies.

The contributions of this paper can be summarised as follows:

(i) We formulate a consequence-based procedure for the fragment $\mathcal{ELH}_{\mathcal{R}^+}$ of an OWL 2 EL profile. The procedure does not require the usual axiom normalisation preprocessing [2] but works directly with the input ontology. Although normalisation is usually fast and its effect on the overall running time is negligible, the removal of this preprocessing step simplifies the presentation of the algorithm and reduces the implementation efforts.

(ii) We describe a concurrent strategy for saturation of the input axioms under inference rules. The strategy works by assigning axioms to 'contexts' in which the inferences can be performed independently, and can be used with arbitrary deterministic inference systems.

(iii) We describe an implementation of our concurrent saturation strategy for $\mathcal{ELH}_{\mathcal{R}^+}$ within a Java-based reasoner ELK. We demonstrate empirically that the concurrent implementation can offer a significant speedup in ontology classification (e.g., a factor of 2.6 for SNOMED CT on a 4-core machine) and can outperform all existing highly-optimised (sequential) reasoners on the commonly used $\mathcal{EL}$ ontologies. The improved performance is achieved by minimising the overheads for managing concurrency and is comparable to the 'embarrassingly parallel' algorithm on the pre-partitioned input.

2 Preliminaries

The vocabulary of $\mathcal{ELH}_{\mathcal{R}^+}$ consists of countably infinite sets N_R of *(atomic) roles* and N_C of *atomic concepts*. Complex *concepts* and *axioms* are defined recursively using the constructors in Table 1. We use the letters R, S, T for roles, C, D, E for concepts and A, B for atomic concepts. A *concept equivalence* $C \equiv D$ stands for the two inclusions $C \sqsubseteq D$ and $D \sqsubseteq C$. An *ontology* is a finite set of axioms. Given an ontology $\mathcal{O}$, we write $\sqsubseteq^*_{\mathcal{O}}$ for the smallest reflexive transitive binary relation over roles such that $R \sqsubseteq^*_{\mathcal{O}} S$ holds for all $R \sqsubseteq S \in \mathcal{O}$.

$\mathcal{ELH}_{\mathcal{R}^+}$ has Tarski-style semantics. An *interpretation* $\mathcal{I}$ consists of a non-empty set $\Delta^{\mathcal{I}}$ called the domain of $\mathcal{I}$ and an interpretation function $\cdot^{\mathcal{I}}$ that

Table 1. Syntax and semantics of $\mathcal{ELH}_{\mathcal{R}^+}$

	Syntax	Semantics
Roles:		
atomic role	R	$R^{\mathcal{I}}$
Concepts:		
atomic concept	A	$A^{\mathcal{I}}$
top	$\top$	$\Delta^{\mathcal{I}}$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
existential restriction	$\exists R.C$	$\{x \mid \exists y : \langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
Axioms:		
concept inclusion	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
role inclusion	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
transitive role	$\mathrm{Trans}(T)$	$T^{\mathcal{I}}$ is transitive

Table 2. Inference rules for $\mathcal{ELH}_{\mathcal{R}^+}$

$$\mathbf{R}_{\sqsubseteq}\ \frac{C \sqsubseteq D}{C \sqsubseteq E} : D \sqsubseteq E \in \mathcal{O} \qquad \mathbf{R}^{+}_{\sqcap}\ \frac{C \sqsubseteq D_1 \quad C \sqsubseteq D_2}{C \sqsubseteq D_1 \sqcap D_2} : D_1 \sqcap D_2 \text{ occurs in } \mathcal{O}$$

$$\mathbf{R}^{-}_{\sqcap}\ \frac{C \sqsubseteq D_1 \sqcap D_2}{\begin{array}{c} C \sqsubseteq D_1 \\ C \sqsubseteq D_2 \end{array}} \qquad \mathbf{R}^{+}_{\exists}\ \frac{C \sqsubseteq D}{\exists S.C \rightarrow \exists S.D} : \exists S.D \text{ occurs in } \mathcal{O}$$

$$\mathbf{R}^{-}_{\exists}\ \frac{C \sqsubseteq \exists R.D}{D \sqsubseteq D} \qquad \mathbf{R}_{\mathcal{H}}\ \frac{D \sqsubseteq \exists R.C \quad \exists S.C \rightarrow E}{D \sqsubseteq E} : R \sqsubseteq^{*}_{\mathcal{O}} S$$

$$\mathbf{R}^{+}_{\top}\ \frac{C \sqsubseteq C}{C \sqsubseteq \top} : \top \text{ occurs in } \mathcal{O} \qquad \mathbf{R}_{\mathcal{T}}\ \frac{D \sqsubseteq \exists R.C \quad \exists S.C \rightarrow E}{\exists T.D \rightarrow E} : \begin{array}{l} R \sqsubseteq^{*}_{\mathcal{O}} T \sqsubseteq^{*}_{\mathcal{O}} S \\ \mathrm{Trans}(T) \in \mathcal{O} \end{array}$$

assigns to each R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ and to each A a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$. The interpretation function is extended to complex concepts as shown in Table 1.

An interpretation $\mathcal{I}$ *satisfies* an axiom α (written $\mathcal{I} \models \alpha$) if the corresponding condition in Table 1 holds. If an interpretation $\mathcal{I}$ satisfies all axioms in an ontology $\mathcal{O}$, then $\mathcal{I}$ is a *model* of $\mathcal{O}$ (written $\mathcal{I} \models \mathcal{O}$). An axiom α is a *consequence* of an ontology $\mathcal{O}$ (written $\mathcal{O} \models \alpha$) if every model of $\mathcal{O}$ satisfies α. A concept C is *subsumed* by D w.r.t. $\mathcal{O}$ if $\mathcal{O} \models C \sqsubseteq D$. *Classification* is the task of computing all subsumptions $A \sqsubseteq B$ between atomic concepts such that $\mathcal{O} \models A \sqsubseteq B$.

3 A Classification Procedure for $\mathcal{ELH}_{\mathcal{R}^+}$

Table 2 lists the inference rules of our classification procedure, which are closely related to the original completion rules for $\mathcal{EL}^{++}$ [2]. The rules operate with two types of axioms: (*i*) *subsumptions* $C \sqsubseteq D$ and (*ii*) *(existential) implications* $\exists R.C \rightarrow \exists S.D$, where C and D are concepts, and R and S roles. The implications $\exists R.C \rightarrow \exists S.D$ have the same semantic meaning as $\exists R.C \sqsubseteq \exists S.D$; we use the symbol $\rightarrow$ just to distinguish the two types of axioms in the inference rules.

$$\mathsf{KneeJoint} \equiv \mathsf{Joint} \sqcap \exists \mathsf{isPartOf}.\mathsf{Knee} \tag{1}$$
$$\mathsf{LegStructure} \equiv \mathsf{Structure} \sqcap \exists \mathsf{isPartOf}.\mathsf{Leg} \tag{2}$$
$$\mathsf{Joint} \sqsubseteq \mathsf{Structure} \tag{3}$$
$$\mathsf{Knee} \sqsubseteq \exists \mathsf{hasLocation}.\mathsf{Leg} \tag{4}$$
$$\mathsf{hasLocation} \sqsubseteq \mathsf{isPartOf} \tag{5}$$
$$\mathrm{Trans}(\mathsf{isPartOf}) \tag{6}$$

Fig. 1. A simple medical ontology describing some anatomical relations

Note that we make a distinction between the premises of a rule (appearing above the horizontal line) and its side conditions (appearing after the colon), and use axioms from $\mathcal{O}$ as the side conditions, not as the premises.

It is easy to see that the inference rules are sound in the sense that every conclusion is always a logical consequence of the premises and the ontology, assuming that the side conditions are satisfied. For all rules except the last one this is rather straightforward. The last rule works similarly to the propagation of universal restrictions along transitive roles if we view axioms $\exists S.C \to E$ and $\exists T.D \to E$ as $C \sqsubseteq \forall S^-.E$ and $D \sqsubseteq \forall T^-.E$ respectively. Before we formulate a suitable form of completeness for our system, let us first consider an example.

Example 1. Consider the ontology $\mathcal{O}$ in Fig. 1 expressing that a knee joint is a joint that is a part of the knee (1), a leg structure is a structure that is a part of the leg (2), a joint is a structure (3), a knee has location in the leg (4), has-location is more specific than part-of (5), and part-of is transitive (6).

Below we demonstrate how the inference rules in Table 2 can be used to prove that a knee joint is a leg structure. We start with a tautological axiom saying that knee joint is a knee joint and then repeatedly apply the inference rules:

$$\mathsf{KneeJoint} \sqsubseteq \mathsf{KneeJoint} \quad \text{input axiom,} \tag{7}$$
$$\mathsf{KneeJoint} \sqsubseteq \mathsf{Joint} \sqcap \exists \mathsf{isPartOf}.\mathsf{Knee} \quad \text{by } \mathbf{R}_{\sqsubseteq}\ (7) : (1), \tag{8}$$
$$\mathsf{KneeJoint} \sqsubseteq \mathsf{Joint} \quad \text{by } \mathbf{R}^-_{\sqcap}\ (8), \tag{9}$$
$$\mathsf{KneeJoint} \sqsubseteq \exists \mathsf{isPartOf}.\mathsf{Knee} \quad \text{by } \mathbf{R}^-_{\sqcap}\ (8). \tag{10}$$

In the last axiom, we have obtained an existential restriction on knee, which now allows us to start deriving subsumption relations for this concept thanks to $\mathbf{R}^-_{\exists}$:

$$\mathsf{Knee} \sqsubseteq \mathsf{Knee} \quad \text{by } \mathbf{R}^-_{\exists}\ (10), \tag{11}$$
$$\mathsf{Knee} \sqsubseteq \exists \mathsf{hasLocation}.\mathsf{Leg} \quad \text{by } \mathbf{R}_{\sqsubseteq}\ (11) : (4). \tag{12}$$

Similarly, the last axiom lets us start deriving subsumptions for leg:

$$\mathsf{Leg} \sqsubseteq \mathsf{Leg} \quad \text{by } \mathbf{R}^-_{\exists}\ (12). \tag{13}$$

This time, we can use (13) to derive existential implications using the fact that $\exists \mathsf{hasLocation.Leg}$ and $\exists \mathsf{isPartOf.Leg}$ occur in $\mathcal{O}$:

$$\exists \mathsf{hasLocation.Leg} \rightarrow \exists \mathsf{hasLocation.Leg} \qquad \text{by } \mathbf{R}^+_\exists \text{ (13)}, \tag{14}$$

$$\exists \mathsf{isPartOf.Leg} \rightarrow \exists \mathsf{isPartOf.Leg} \qquad \text{by } \mathbf{R}^+_\exists \text{ (13)}. \tag{15}$$

The last implication, in particular, can be used to replace the existential restriction in (12) using $\mathsf{hasLocation} \sqsubseteq^*_{\mathcal{O}} \mathsf{isPartOf}$, which is a consequence of (5):

$$\mathsf{Knee} \sqsubseteq \exists \mathsf{isPartOf.Leg} \qquad \text{by } \mathbf{R}_{\mathcal{H}} \text{ (12), (15)}. \tag{16}$$

Similarly, we can derive a new implication using $\mathsf{hasLocation} \sqsubseteq^*_{\mathcal{O}} \mathsf{isPartOf} \sqsubseteq^*_{\mathcal{O}} \mathsf{isPartOf}$ and transitivity of $\mathsf{isPartOf}$ (6):

$$\exists \mathsf{isPartOf.Knee} \rightarrow \exists \mathsf{isPartOf.Leg} \qquad \text{by } \mathbf{R}_{\mathcal{T}} \text{ (12), (15)}. \tag{17}$$

This implication can now be used to replace the existential restriction in (10):

$$\mathsf{KneeJoint} \sqsubseteq \exists \mathsf{isPartOf.Leg} \qquad \text{by } \mathbf{R}_{\mathcal{H}} \text{ (10), (17)}. \tag{18}$$

Finally, we "construct" the definition of leg structure (2) using (3) and the fact that the concept $\mathsf{Structure} \sqcap \exists \mathsf{isPartOf.Leg}$ occurs in $\mathcal{O}$:

$$\mathsf{KneeJoint} \sqsubseteq \mathsf{Structure} \qquad \text{by } \mathbf{R}_\sqsubseteq \text{ (9)} : \text{(3)}, \tag{19}$$

$$\mathsf{KneeJoint} \sqsubseteq \mathsf{Structure} \sqcap \exists \mathsf{isPartOf.Leg} \qquad \text{by } \mathbf{R}^+_\sqcap \text{ (18), (19)}, \tag{20}$$

$$\mathsf{KneeJoint} \sqsubseteq \mathsf{LegStructure} \qquad \text{by } \mathbf{R}_\sqsubseteq \text{ (20)} : \text{(2)}. \tag{21}$$

We have thus proved that a knee joint is a leg structure.

In the above example we have demonstrated that a consequence subsumption $\mathsf{KneeJoint} \sqsubseteq \mathsf{LegStructure}$ can be derived using the inference rules in Table 2 once we start with a tautology $\mathsf{KneeJoint} \sqsubseteq \mathsf{KneeJoint}$. It turns out that all implied subsumption axioms with $\mathsf{KneeJoint}$ on the left-hand-side and a concept occurring in the ontology on the right-hand-side can be derived in this way:

Theorem 1. *Let* $\mathbf{S}$ *be any set of axioms closed under the inference rules in Table 2. For all concepts* C *and* D *such that* $C \sqsubseteq C \in \mathbf{S}$ *and* D *occurs in* $\mathcal{O}$ *we have* $\mathcal{O} \models C \sqsubseteq D$ *implies* $C \sqsubseteq D \in \mathbf{S}$.

Proof (Sketch). Due to lack of space, we can only present a proof sketch. A more detailed proof can be found in the accompanying technical report [8].

The proof of Theorem 1 is by canonical model construction. We construct an interpretation $\mathcal{I}$ whose domain consists of distinct elements x_C, one element x_C for each axiom $C \sqsubseteq C$ in $\mathbf{S}$. Atomic concepts are interpreted so that $x_C \in A^{\mathcal{I}}$ iff $C \sqsubseteq A \in \mathbf{S}$. Roles are interpreted by minimal relations satisfying all role inclusion and transitivity axioms in $\mathcal{O}$ such that $\langle x_C, x_D \rangle \in R^{\mathcal{I}}$ for all $C \sqsubseteq \exists R.D \in \mathbf{S}$.

Using rules $\mathbf{R}^-_\sqcap$ and $\mathbf{R}^-_\exists$ one can prove that for all elements x_C and all concepts D we have:

$$C \sqsubseteq D \in \mathbf{S} \text{ implies } x_C \in D^\mathcal{I}. \tag{22}$$

Conversely, using rules $\mathbf{R}^+_\top$, $\mathbf{R}^+_\sqcap$, $\mathbf{R}^+_\exists$, $\mathbf{R}_\mathcal{H}$ and $\mathbf{R}_\mathcal{T}$ one can prove that for all concepts D occurring in $\mathcal{O}$ and all concepts C we have:

$$x_C \in D^\mathcal{I} \text{ implies } C \sqsubseteq D \in \mathbf{S}. \tag{23}$$

Properties (22) and (23) guarantee that $\mathcal{I}$ is a model of $\mathcal{O}$. To see this, let $D \sqsubseteq E \in \mathcal{O}$ and x_C be an arbitrary element of $\mathcal{I}$. If $x_C \in D^\mathcal{I}$, then $C \sqsubseteq D \in \mathbf{S}$ by (23), then $C \sqsubseteq E \in \mathbf{S}$ by rule $\mathbf{R}_\sqsubseteq$, so $x_C \in E^\mathcal{I}$ by (22). Thus $D^\mathcal{I} \subseteq E^\mathcal{I}$.

Finally, to complete the proof of Theorem 1, let C and D be such that $C \sqsubseteq C \in \mathbf{S}$ and D occurs in $\mathcal{O}$. We will show that $C \sqsubseteq D \notin \mathbf{S}$ implies $\mathcal{O} \not\models C \sqsubseteq D$. Suppose $C \sqsubseteq D \notin \mathbf{S}$. Then $x_C \notin D^\mathcal{I}$ by (23). Since $C \sqsubseteq C \in \mathbf{S}$, we have $x_C \in C^\mathcal{I}$ by (22). Hence $C^\mathcal{I} \not\subseteq D^\mathcal{I}$ and, since $\mathcal{I}$ is a model of $\mathcal{O}$, $\mathcal{O} \not\models C \sqsubseteq D$. □

It follows from Theorem 1 that one can classify $\mathcal{O}$ by exhaustively applying the inference rules to the initial set of tautologies $\mathsf{input} = \{A \sqsubseteq A \mid A \text{ is an atomic concept occurring in } \mathcal{O}\}$.

Corollary 1. *Let* $\mathbf{S}$ *be the closure of* input *under the inference rules in Table 2. Then for all atomic concepts* A *and* B *occurring in* $\mathcal{O}$ *we have* $\mathcal{O} \models A \sqsubseteq B$ *if and only if* $A \sqsubseteq B \in \mathbf{S}$.

Finally, we note that if all initial axioms $C \sqsubseteq D$ are such that both C and D occur in $\mathcal{O}$ (as is the case for the set input defined above), then the inference rules derive only axioms of the form (i) $C \sqsubseteq D$ and (ii) $\exists R.C \to E$ with C, D, E and R occurring in $\mathcal{O}$. There is only a polynomial number of such axioms, and all of them can be computed in polynomial time.

4 Concurrent Saturation under Inference Rules

In this section we describe a general approach for saturating a set of axioms under inference rules. We first describe a high level procedure and then introduce a refined version which facilitates concurrent execution of the inference rules.

4.1 The Basic Saturation Strategy

The basic strategy for computing a saturation of the input axioms under inference rules can be described by Algorithm 1. The algorithm operates with two collections of axioms: the queue of 'scheduled' axioms for which the rules have not been yet applied, initialized with the input axioms, and the set of 'processed' axioms for which the rules are already applied, initially empty. The queue of scheduled axioms is repeatedly processed in the while loop (lines 3–8): if the next scheduled axiom has not been yet processed (line 5), it is moved to the set of processed axioms (line 6), and every conclusion of inferences involving this

Algorithm 1. saturate(input): saturation of axioms under inference rules

Data: input: set of input axioms
Result: the saturation of input is computed in processed

```
1 scheduled ← input;
2 processed ← ∅;
3 while scheduled ≠ ∅ do
4     axiom ← scheduled.poll();
5     if not processed.contains(axiom) then
6         processed.add(axiom);
7         for conclusion ∈ deriveConclusions(processed, axiom) do
8             scheduled.put(conclusion);
```

axiom and the processed axioms is added to the queue of scheduled axioms. This strategy is closely related to the 'given clause algorithm' used in saturation-based theorem proving for first-order logic (see, e.g., [3]).

Soundness, completeness, and termination of Algorithm 1 is a consequence of the following (semi-) invariants that can be proved by induction:

(i) Every scheduled and processed axiom is either an input axiom, or is obtained by an inference rule from the previously processed axioms (soundness).
(ii) Every input axiom and every conclusion of inferences between processed axioms occurs either in the processed or scheduled axioms (completeness).
(iii) In every iteration of the while loop (lines 3–8) either the set of processed axiom increases, or, otherwise, it remains the same, but the queue of scheduled axioms becomes shorter (termination).

Therefore, when the algorithm terminates, the saturation of the input axioms under the inference rules is computed in the set of processed axioms.

The basic saturation strategy described in Algorithm 1 can already be used to compute the saturation concurrently. Indeed, the while loop (lines 3–8) can be executed from several independent workers, which repeatedly take the next axiom from the shared queue of scheduled axiom and perform inferences with the shared set of processed axioms. To remain correct with multiple workers, it is essential that Algorithm 1 adds the axiom to the set of processed axioms in line 6 before deriving conclusions with this axiom, not after that. Otherwise, it may happen that two workers simultaneously process two axioms between which an inference is possible, but will not be able to perform this inference because neither of these axioms is in the processed set.

4.2 The Refined Saturation Strategy

In order to implement Algorithm 1 in a concurrent way, one first has to ensure that the shared collections of processed and scheduled axioms can be safely accessed and modified from different workers. In particular, one worker should

Algorithm 2. saturate(input): saturation of axioms under inference rules

```
   Data: input: set of input axioms
   Result: the saturation of input is computed in context.processed
 1 activeContexts ← ∅;
 2 for axiom ∈ input do
 3     for context ∈ getContexts(axiom) do
 4         context.scheduled.add(axiom);
 5         activeContexts.activate(context);
 6 loop
 7     context ← activeContexts.poll();
 8     if context = null then break;
 9     loop
10         axiom ← context.scheduled.poll();
11         if axiom = null then break;
12         if not context.processed.contains(axiom) then
13             context.processed.add(axiom);
14             for conclusion ∈ deriveConclusions(context.processed, axiom) do
15                 for conclusionContext ∈ getContexts(conclusion) do
16                     conclusionContext.scheduled.add(conclusion);
17                     activeContexts.activate(conclusionContext);
18     activeContexts.deactivate (context);
```

be able to derive conclusions in line 7 at the same time when another worker is inserting an axiom into the set of processed axioms. The easiest way to address this problem is to guard every access to the shared collection using locks. But this will largely defeat the purpose of concurrent computation, since the workers will have to wait for each other in order to proceed.

Below we describe a lock-free solution to this problem. The main idea is to distribute the axioms according to 'contexts' in which the axioms can be used as premises of inference rules and which can be processed independently by the workers. Formally, let $\mathcal{C}$ be a finite set of *contexts*, and `getContexts`(axiom) a function assigning a non-empty subset of contexts for every axiom such that, whenever an inference between several axioms is possible, the axioms will have at least one common context assigned to them. Furthermore, assume that every context has its own queue of scheduled axiom and a set of processed axioms (both initially empty), which we will denote by context.scheduled and context.processed.

The refined saturation strategy is described in Algorithm 2. The key idea of the algorithm is based on the notion of an active context. We say that a context is *active* if the scheduled queue for this context is not empty. The algorithm maintains the queue of active contexts to preserve this invariant. For every input axiom, the algorithm takes every context assigned to this axiom and adds this axiom to the queue of the scheduled axioms for this context (lines 2–4). Because

the queue of scheduled axiom becomes non-empty, the context is *activated* by adding it to the queue of active contexts (line 5).

Afterwards, each active context is repeatedly processed in the loop (lines 6–17) by essentially performing similar operations as in Algorithm 1 but with the context-local collections of scheduled and processed axioms. The only difference is that the conclusions of inferences computed in line 14 are inserted into (possibly several) sets of scheduled axioms for the contexts assigned to each conclusion, in a similar way as it is done for the input axiom. Once the context is processed, i.e., the queue of the scheduled axioms becomes empty and the loop quits at line 8, the context is deactivated (line 18).

Similar to Algorithm 1, the main loop in Algorithm 2 (lines 6–17) can be processed concurrently by several workers. The advantage of the refined algorithm, however, is that it is possible to perform inferences in line 14 without locking the (context-local) set of processed axioms provided we can guarantee that no context is processed by more than one worker at a time. For the latter, it is sufficient to ensure that a context is never inserted into the queue of active contexts if it is already there or it is being processed by a worker. It seems that this can be easily achieved using a flag, which is set to true when a context is activated and set to false when a context is deactivated—a context is added to the queue only the first time the flag is set to true:

```
activeContexts.activate (context):
if not context.isActive then
    context.isActive ← true;
    activeContexts.put(context);
```

```
activeContexts.deactivate (context):
context.isActive ← false;
```

Unfortunately, this strategy does not work correctly with multiple workers: it can well be the case that two workers try to activate the same context at the same time, both see that the flag is false, set it to true, and insert the context into the queue two times. To solve this problem we would need to ensure that when two workers are trying to change the value of the flag from false to true, only one of them succeeds. This can be achieved without locking by using an atomic operation 'compare-and-swap' which tests and updates the value of the flag in one instruction. Algorithm 3 presents a safe way of activating contexts.

Algorithm 3. activeContexts.activate(context)

```
1 if context.isActive.compareAndSwap(false, true) then
2     activeContexts.put(context);
```

Deactivation of contexts in the presence of multiple workers is also not as easy as just setting the value of the flag to false. The problem is that during the time after quitting the loop in line 8 and before deactivation of context in line 18, some other worker could insert an axiom into the queue of scheduled axioms for this context. Because the flag was set to true at that time, the context will not be inserted into the queue of active contexts, thus we end up with a context which is active in the sense of having a non-empty scheduled queue, but not

activated according to the flag. To solve this problem, we perform an additional emptiness test for the scheduled axioms as shown in Algorithm 4.

Algorithm 4. activeContexts.deactivate(context)

```
1 context.isActive ← false;
2 if context.scheduled ≠ ∅ then activeContexts.activate(context);
```

5 Implementation

In this section we describe an implementation of the inference rules for $\mathcal{ELH}_{\mathcal{R}^+}$ in Table 2 using the refined concurrent saturation strategy presented in Section 4.2. There are two functions in Algorithm 2 whose implementation we need to explain, namely `getContexts`(axiom) and `deriveConclusions`(processed, axiom).

Recall that the function `getContexts`(axiom) is required to assign a set of contexts to every axiom such that, whenever an inference between several axioms is possible, the premises will have at least one context in common. This is necessary in order to ensure that no inference between axioms gets lost because the inferences are applied only locally within contexts. A simple solution would be to use the inference rules themselves as contexts and assign to every axiom the set of inference rules in which the axiom can participate. Unfortunately, this strategy can provide only as many contexts as there are inference rules—not that much. To come up with a better solution, note that all premises of the inference rules in Table 2 always have a common concept denoted as C. Instead of assigning axioms with inference rules, we can assign them with the set of concepts that match the respective position of C in the rule applications. This idea leads to the implementation described in Algorithm 5.

Algorithm 5. getContexts(axiom)

```
1 result ← ∅;
  // matching the premises of the rules R⊑, R⁻⊓, R⁻∃, R⁺⊤, R⁺⊓, R⁺∃
2 if axiom match C ⊑ D then result.add(C);
  // matching the left premise of the rules R_H and R_T
3 if axiom match D ⊑ ∃R.C then result.add(C);
  // matching the right premise of the rules R_H and R_T
4 if axiom match ∃S.C → E then result.add(C);
5 return result;
```

To implement the other function `deriveConclusions`(processed, axiom), we need to compute the conclusions of all inference rules in which one premise is axiom and remaining premises come from processed. A naïve implementation would iterate over all possible subsets of such premises, and try to match them to the inference rules. To avoid unnecessary enumerations, we use index data structures to quickly find applicable inference rules. For example, for checking the side conditions of the rule $\mathbf{R}^+_\sqcap$, for every concept D occurring in the ontology

Algorithm 6. deriveConclusions(processed, axiom) for context C

```
1  result ← ∅;
2  if axiom match C ⊑ D then
3      for D1 ∈ (C.subsumptions ∩ D.leftConjuncts) do
4          result.add(C ⊑ D1 ⊓ D);                    // rule R⁺⊓, right premise
5      for D2 ∈ (C.subsumptions ∩ D.rightConjuncts) do
6          result.add(C ⊑ D ⊓ D2);                    // rule R⁺⊓, left premise
       // similarly for rules R⊑, R⁻⊓, R⁻∃, R⁺⊤, R⁺∃
7  if axiom match D ⊑ ∃R.C then
8      for S ∈ (C.implications.keySet() ∩ R.superRoles) do
9          for E ∈ C.implications.get(S) do
10             result.add(D ⊑ E);                     // rule R_H, left premise
       // similarly for rule R_T, left premise
11 if axiom match ∃S.C → E then
12     for R ∈ (C.predecessors.keySet() ∩ S.subRoles) do
13         for D ∈ C.predecessors.get(R) do
14             result.add(D ⊑ E);                     // rule R_H, right premise
       // similarly for rule R_T, right premise
15 return result;
```

$\mathcal{O}$ we store a set of concepts with which D co-occur in conjunctions:

$$D.\mathsf{rightConjuncts} = \{D' \mid D \sqcap D' \text{ occurs in } \mathcal{O}\},$$
$$D.\mathsf{leftConjuncts} = \{D' \mid D' \sqcap D \text{ occurs in } \mathcal{O}\}.$$

Similarly, for checking the side condition of the rule $\mathbf{R}_{\mathcal{H}}$, for each role R occurring in $\mathcal{O}$, we precompute the sets of its sub-roles and super-roles:

$$R.\mathsf{superRoles} = \{S \mid R \sqsubseteq^*_{\mathcal{O}} S\},$$
$$R.\mathsf{subRoles} = \{S \mid S \sqsubseteq^*_{\mathcal{O}} R\}.$$

The index computation is usually quick and can be also done concurrently.

To identify relevant premises from the set of processed axioms for the context associated with C, we store the processed axioms for this context in three records according to the cases by which these axioms were assigned to C in Algorithm 5:

$$C.\mathsf{subsumptions} = \{D \mid C \sqsubseteq D \in \mathsf{processed}\},$$
$$C.\mathsf{predecessors} = \{\langle R, D\rangle \mid D \sqsubseteq \exists R.C \in \mathsf{processed}\},$$
$$C.\mathsf{implications} = \{\langle R, E\rangle \mid \exists R.C \to E \in \mathsf{processed}\}.$$

The pairs in the last two records are indexed by the key R, so that for every role R one can quickly find all concepts D and, respectively, E that occur in the

Table 3. Statistics for studied ontologies and classification times on 'laptop' for commonly used $\mathcal{EL}$ reasoners and ELK; times are in seconds; timeout was 1 hour

	SNOMED CT	GALEN	FMA	GO
#concepts	315,489	23,136	78,977	19,468
#roles	58	950	7	1
#axioms	430,844	36,547	121,712	28,897
CB	13.9	1.4	0.7	0.1
FaCT++	387.1	timeout	5.4	7.5
jCEL	661.6	32.5	12.4	2.9
Pellet	509.3	stack overflow	88.5	2.5
Snorocket	24.5	2.3	1.6	0.3
ELK (1 worker)	13.15	1.33	0.44	0.20
ELK (2 workers)	7.65	0.90	0.38	0.18
ELK (3 workers)	5.66	0.80	0.39	0.19
ELK (4 workers)	5.02	0.77	0.39	0.19

pairs with R. Given this index data structure, the function deriving conclusions within a context C can be described by Algorithm 6. Note that the algorithm extensively uses iterations over intersections of two sets; optimizing such iterations is essential for achieving efficiency of the algorithm.

6 Experimental Evaluation

To evaluate our approach, we have implemented the procedure in the Java-based reasoner ELK (version 0.1.0) using lock-free data structures, such as ConcurrentLinkedQueue, and objects such as AtomicBoolean, which allow for 'compare-and-swap' operations, provided by the java.util.concurrent package.[1] The goal of this section is (a) to compare the performance of ELK in practical situations with other popular reasoners, and (b) to study the extent in which concurrent processing contributes to the improved classification performance. To evaluate (a) we have used a notebook with Intel Core i7-2630QM 2GHz quad core CPU and 6GB of RAM, running Microsoft Windows 7 (experimental configuration 'laptop'). To evaluate (b) we have additionally used a server-type computer with an Intel Xeon E5540 2.53GHz with two quad core CPUs and 24GB of RAM running Linux 2.6.16 (experimental configuration 'server'). In both configurations we ran Java 1.6 with 4GB of heap space. All figures reported here were obtained as the average over 10 runs of the respective experiments. More detailed information for all experiments can be found in the technical report [8].

Our test ontology suite includes SNOMED CT, GO, FMA-lite, and an OWL EL version of GALEN.[2] The first part of Table 3 provides some general

[1] The reasoner is available open source from `http://elk-reasoner.googlecode.com/`

[2] SNOMED CT from `http://ihtsdo.org/` (needs registration); GO from `http://lat.inf.tu-dresden.de/~meng/toyont.html`, FMA-lite from `http://www.bioontology.org/wiki/index.php/FMAInOwl`, GALEN from `http://condor-reasoner.googlecode.com/`

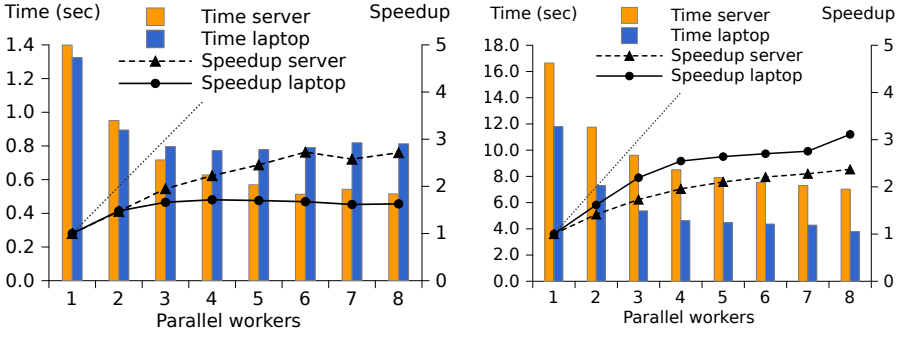

Fig. 2. Classification time and speedup for n workers on GALEN (left) and SNOMED CT (right)

statistics about the sizes of these ontologies. We have measured performance on these ontologies for the reasoners CB r.12 [7], FaCT++ 1.5.2 [19], jCEL 0.15.0,[3] Pellet 2.2.2 [17] and Snorocket 1.3.4 [10]. We selected these tools as they provide specific support for $\mathcal{EL}$-type ontologies. FaCT++ and jCEL, and Pellet were accessed through OWL API; CB, ELK, and Snorocket were accessed through their command line interface. The second part of Table 3 shows the time needed by the tested reasoners to classify the given ontologies in the 'laptop' scenario. The times are as the reasoners report for the classification stage, which does not include the loading times.[4] The last part of Table 3 presents the result for ELK tested under the same conditions for a varying number of workers. As can be seen from these results, ELK demonstrates a highly competitive performance already for 1 worker, and adding more workers can substantially improve the classification times for SNOMED CT and GALEN.

The results in Table 3 confirm that concurrent processing can offer improvements for ontology classification on common computing hardware. On the other hand, the experiments demonstrate that the improvement factor degrades with the number of workers. There can be many causes for this effect, such as dynamic CPU clocking, shared Java memory management and garbage collection, hardware bottlenecks in CPU caches or data transfer, and low-level mechanisms like Hyper-Threading. To check to what extent the overhead of managing the workers can contribute to this effect, we have performed a series of further experiments.

First, we have investigated classification performance for varying numbers of parallel workers. Figure 2 shows the results for GALEN and SNOMED CT obtained for $1-8$ workers on 'laptop' and 'server' configurations. The reported data is classification time (left axis) and speedup, i.e., the quotient of single worker classification time by measured multi-worker classification time (right axis). The ideal linear speedup is indicated by a dotted line. On the laptop (4

[3] `http://jcel.sourceforge.net/`

[4] Loading times can vary depending on the syntax/parser/API used and are roughly proportional to the size of the ontology; it takes about 8 seconds to load the largest tested ontology SNOMED CT in functional-style syntax by ELK.

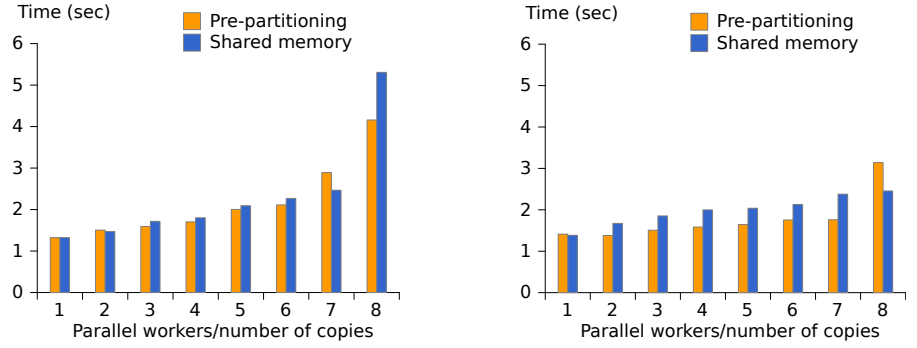

Fig. 3. Classification time for n workers on n copies of GALEN on laptop (left) and server (right)

cores), there is no significant change in the classification times for more than 4 workers, while the classification times for the server (2×4 cores) can slightly improve for up to 8 workers.

Next, we measured how our procedure would score against the 'embarrassingly parallel' algorithm in the situation when an ontology consists of n disjoint and equal components, which can be classified completely independently. We have created ontologies consisting of n disjoint copies of our test ontologies, and ran n independent ELK reasoners with 1 worker on these n copies. We compared the average classification times of this pre-partitioning approach with the times needed by ELK when using n workers on the union of these partitions. Both approaches compute exactly the same results.

The results of this experiment for copies of GALEN are presented in Fig. 3. As can be seen from these results, the classification time for both scenarios grows with n (ideally, it should remain constant up to the number of available cores, but in practice it is not the case because, e.g., the CPU clock speed drops with the load to compensate for the heat). More importantly, the difference between the two approaches is not considerable. This proves that, the performance impact of managing multiple workers is relatively small and interaction between unrelated components is avoided due to the indexing strategies discussed in Section 5. The fact that pre-partitioning requires additional time for initial analysis and rarely leads to perfect partitioning [18] suggests that our approach is more suitable for the (single machine, shared memory) scenario that we target.

7 Related Works and Conclusion

Our work is not the first one to address the problem of parallel/concurrent OWL reasoning. Notable earlier works include an approach for parallelization of (incomplete) structural reasoning algorithms [4], a tableaux procedure that concurrently explores non-deterministic choices [11], a resolution calculus for $\mathcal{ALCHIQ}$ where inferences are exchanged between distributed workers [15], and a distributed classification algorithm that can be used to concurrently invoke

(serial) OWL reasoners for checking relevant subsumptions [1]. Experimental evaluations in each case indicated potential on selected examples, but further implementation and evaluation will be needed to demonstrate a clear performance advantage of these systems over state-of-the-art systems.

Some other works have studied concurrency in light-weight ontology languages. Closest to our approach is a distributed MapReduce-based algorithm for $\mathcal{EL}^+$ [14]. However, this idea has not been empericaly evaluated. Works dedicated to OWL RL [13] include an approach for pre-partitioning inputs that inspired the evaluation in Section 6 [18], and recent MapReduce approaches [6,20].

Many further works focus on the distribution of reasoning with assertional data using weaker schema-level modelling languages pD^* (a.k.a. OWL-Horst) and (fragments of) RDFS [5,21,22,9]. These works are distinguished from our approach by their goal to manage large-scale data (in the range of billions of axioms), which is beyond the memory capacity of a single machine. Accordingly, computation is distributed to many servers without memory sharing. Yet, we can find similarities in term-based distribution strategies [5,21,22,6,20,14] and indexing of rules [6] with our strategy of assigning contexts to axioms.

In conclusion, we can say that this work does indeed appear to be the first to demonstrate a compelling performance advantage for terminological reasoning in OWL through exploiting shared-memory parallelism on modern multi-core systems. We hope that these encouraging results will inspire further works in this area, by exploiting existing general techniques for parallel computing, such as the MapReduce framework, as well as new approaches for parallelization specific to OWL reasoning, such as consequence-based reasoning procedures.

Acknowledgements. Yevgeny Kazakov and František Simančík are sponsored by EPSRC grant EP/G02085X/1. Markus Krötzsch is sponsored by EPSRC grant EP/F065841/1. Some experimental results were obtained using computing resources provided by the Oxford Supercomputing Centre.

References

1. Aslani, M., Haarslev, V.: Parallel TBox classification in description logics – first experimental results. In: Proc. 19th European Conf. on Artificial Intelligence (ECAI 2010), pp. 485–490. IOS Press (2010)
2. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: Proc. 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005), pp. 364–369. Professional Book Center (2005)
3. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Handbook of Automated Reasoning, pp. 19–99. Elsevier and MIT Press (2001)
4. Bergmann, F.W., Quantz, J.: Parallelizing Description Logics. In: Wachsmuth, I., Brauer, W., Rollinger, C.-R. (eds.) KI 1995. LNCS, vol. 981, pp. 137–148. Springer, Heidelberg (1995)
5. Hogan, A., Harth, A., Polleres, A.: Scalable authoritative OWL reasoning for the Web. Int. J. of Semantic Web Inf. Syst. 5(2), 49–90 (2009)
6. Hogan, A., Pan, J.Z., Polleres, A., Decker, S.: SAOR: Template Rule Optimisations for Distributed Reasoning Over 1 Billion Linked Data Triples. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 337–353. Springer, Heidelberg (2010)

7. Kazakov, Y.: Consequence-driven reasoning for Horn $\mathcal{SHIQ}$ ontologies. In: Proc. 21st Int. Conf. on Artificial Intelligence (IJCAI 2009), pp. 2040–2045. IJCAI (2009)
8. Kazakov, Y., Krötzsch, M., Simančík, F.: Concurrent classification of $\mathcal{EL}$ ontologies. Tech. rep., University of Oxford (2011), http://code.google.com/p/elk-reasoner/wiki/Publications
9. Kotoulas, S., Oren, E., van Harmelen, F.: Mind the data skew: distributed inferencing by speeddating in elastic regions. In: Proc. 19th Int. Conf. on World Wide Web (WWW 2010), pp. 531–540. ACM (2010)
10. Lawley, M.J., Bousquet, C.: Fast classification in Protégé: Snorocket as an OWL 2 EL reasoner. In: Proc. 6th Australasian Ontology Workshop (IAOA 2010). Conferences in Research and Practice in Information Technology, vol. 122, pp. 45–49. Australian Computer Society Inc. (2010)
11. Liebig, T., Müller, F.: Parallelizing Tableaux-Based Description Logic Reasoning. In: Chung, S., Herrero, P. (eds.) OTM-WS 2007, Part II. LNCS, vol. 4806, pp. 1135–1144. Springer, Heidelberg (2007)
12. Meissner, A.: Experimental analysis of some computation rules in a simple parallel reasoning system for the $\mathcal{ALC}$ description logic. Int. J. of Applied Mathematics and Computer Science 21(1), 83–95 (2011)
13. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C. (eds.): OWL 2 Web Ontology Language: Profiles. W3C Recommendation (October 27, 2009), http://www.w3.org/TR/owl2-profiles/
14. Mutharaju, R., Maier, F., Hitzler, P.: A MapReduce algorithm for $\mathcal{EL}^+$. In: Proc. 23rd Int. Workshop on Description Logics (DL 2010). CEUR Workshop Proceedings, vol. 573, pp. 464–474. CEUR-WS.org (2010)
15. Schlicht, A., Stuckenschmidt, H.: Distributed Resolution for Expressive Ontology Retworks. In: Polleres, A., Swift, T. (eds.) RR 2009. LNCS, vol. 5837, pp. 87–101. Springer, Heidelberg (2009)
16. Simančík, F., Kazakov, Y., Horrocks, I.: Consequence-based reasoning beyond Horn ontologies. In: Proc. 22nd Int. Conf. on Artificial Intelligence (IJCAI 2011), pp. 1093–1098. AAAI Press/IJCAI (2011)
17. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. of Web Semantics 5(2), 51–53 (2007)
18. Soma, R., Prasanna, V.K.: Parallel inferencing for OWL knowledge bases. In: Proc. Int. Conf. on Parallel Processing (ICPP 2008), pp. 75–82. IEEE Computer Society (2008)
19. Tsarkov, D., Horrocks, I.: FaCT++ Description Logic Reasoner: System Description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006)
20. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: WebPIE: a Web-scale parallel inference engine using MapReduce. J. of Web Semantics (in press, 2011), (accepted manuscript, preprint), http://www.cs.vu.nl/~frankh/postscript/JWS11.pdf
21. Urbani, J., Kotoulas, S., Oren, E., van Harmelen, F.: Scalable Distributed Reasoning Using MapReduce. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 634–649. Springer, Heidelberg (2009)
22. Weaver, J., Hendler, J.A.: Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 682–697. Springer, Heidelberg (2009)