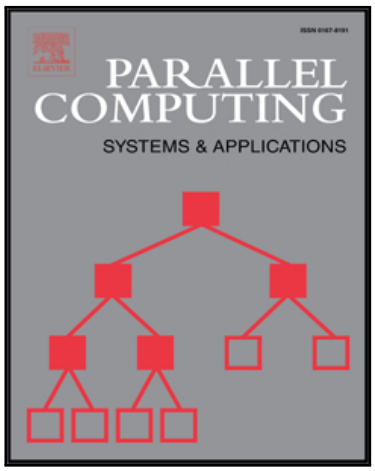


A Parallel Computing Architecture for High-Performance OWL Reasoning

Zixi Quan, Volker Haarslev

PII: S0167-8191(18)30142-X
DOI: 10.1016/j.parco.2018.05.001
Reference: PARCO 2454

To appear in: *Parallel Computing*

Received date: 8 December 2017
Revised date: 14 March 2018
Accepted date: 15 May 2018

Please cite this article as: Zixi Quan, Volker Haarslev, A Parallel Computing Architecture for High-Performance OWL Reasoning, *Parallel Computing* (2018), doi: 10.1016/j.parco.2018.05.001

Highlights

- A novel thread-level architecture for ontology classification with shared-memory
- Implemented with an atomic global half-matrix data structure to avoid race conditions and conflicts
- Using flexible plug-in reasoner for deciding satisfiability and subsumption
- There are two division strategies implemented for classification processing
- Testing a set of real-world ontologies demonstrates good scalability resulting in a speedup linear to the number of available cores

A Parallel Computing Architecture for High-Performance OWL Reasoning

Zixi Quan and Volker Haarslev

Department of Computer Science and Software Engineering, Concordia University, Montréal, Canada

Abstract

The *Web Ontology Language* (OWL) is a widely used knowledge representation language for describing knowledge in application domains by using classes, properties, and individuals. Ontology classification is an important and widely used service that computes a taxonomy of all classes occurring in an ontology. It can require significant amounts of runtime, but most OWL reasoners do not support any kind of parallel processing. We present a novel thread-level parallel architecture for ontology classification, which is ideally suited for shared-memory SMP servers, but does not rely on locking techniques and thus avoids possible race conditions. We evaluated our prototype implementation with a set of real-world ontologies. Our experiments demonstrate a very good scalability resulting in a speedup that is linear to the number of available cores.

Keywords: ontology classification; parallel computing;

1. Introduction

Description logics [1] are a family of logic-based knowledge representation formalisms, which describe a domain in terms of concepts (classes), roles (properties), and individuals. OWL can be considered as a syntactic variant of a very expressive description logic. In this paper we focus on typically supported concept inference services such as concept satisfiability and subsumption that form the basis for implementing classification [2].

Email address: z_qua@encs.concordia.ca, haarslev@encs.concordia.ca (Zixi Quan and Volker Haarslev)

High performance computing (HPC) methods can offer a scalable solution to speed up OWL reasoning. Our HPC approach is based on parallel reasoning techniques for OWL classification. Compared with sequential OWL reasoners, such as Racer [3], FaCT++ [4], and HermiT [5], parallel OWL reasoners work concurrently and distribute the whole task into smaller subparts to speed up the process. A few OWL reasoners integrated parallelization techniques; Konclude [6] is highly performant but its concept classification is sequential; ELK [7] supports parallel concept classification but is restricted to the very small $\mathcal{EL}$ fragment of OWL. Moreover, some other parallel description logic reasoning methods have shown promising results in the past few years such as the first parallel approach for concept classification [8] using a shared-tree data structure, merge classification [9, 10, 11] implementing parallel divide-and-conquer approaches, and [12] proposing a parallel framework for handling non-determinism caused by qualified cardinality restrictions.

Tableau-based methods (see Section 2) are widely used in ontology reasoners for implementing the above-mentioned inference services. In order to speed up reasoning and improve the effectiveness of reasoners, it is necessary to develop efficient and optimized reasoning techniques to implement inference services. OWL ontology reasoning is known to be N2EXPTIME-complete (NEXPTIME-complete if the property hierarchy can be translated into a polynomially-sized nondeterministic finite automaton) [13]. Although most OWL reasoners are highly optimized quite a few real-world ontologies exist that cannot be classified within a reasonable amount of time.

Our work is motivated by previous parallel approaches and also expands ideas presented in [14] to parallel processing. Our HPC approach is implemented with a shared-memory architecture, atomic global data structures, and new strategies for parallel subsumption testing. In order to keep our architecture universal, we use OWL reasoners as plug-ins for deciding satisfiability and subsumption. Currently we use HermiT, but it could be replaced by any other OWL reasoner. Our evaluation demonstrates a promising speedup for ontologies of different sizes and complexities that is linear to the numbers of cores.

2. Description Logics

We briefly introduce the description logic $\mathcal{ALCH}$, which is a subset of OWL and extends $\mathcal{ALC}$ [15] with role hierarchies. We describe its syntax and semantics, selected inference services, and a tableau reasoning algorithm.

2.1. Syntax and Semantics

To formally define an $\mathcal{ALCH}$ knowledge base, we denote with N_C a set of concept names of domain elements with common characteristics, N_R a set of role names representing binary relationships between domain elements, and N_O a set of individual names within the represented domain.

$\mathcal{ALCH}$ concept expressions are $\top, \bot, C, C \sqcup D, C \sqcap D, \exists R.C, \forall R.C$, where $C, D \in N_C$ are arbitrary concepts, $R \in N_R$, $\top, \bot$ the top and bottom concepts, $\sqcap, \sqcup$ represent conjunction and disjunction, and $\exists$ represents qualified existential and $\forall$ universal restriction. The semantics of $\mathcal{ALCH}$ is defined by an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$, consisting of a non-empty set $\Delta^{\mathcal{I}}$ called domain and an interpretation function $.^{\mathcal{I}}$. The interpretation function $.^{\mathcal{I}}$ maps every individual a to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, every concept A to a subset $A^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$ and every role R to a subset $R^{\mathcal{I}}$ of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.

The semantics of concept expressions is defined as $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$, $\bot^{\mathcal{I}} = \emptyset$, $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$, $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$, $(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y \in \Delta^{\mathcal{I}} : (x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$ and $(\forall R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y \in \Delta^{\mathcal{I}} : (x, y) \in R^{\mathcal{I}} \Rightarrow y \in C^{\mathcal{I}}\}$.

Satisfiability. A concept C is satisfiable if there exists an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$, i.e., there exists an individual $x \in \Delta^{\mathcal{I}}$ which is an instance of C, $x \in C^{\mathcal{I}}$. Otherwise, the concept C is unsatisfiable.

Subsumption. A concept D subsumes a concept C (denoted as $C \sqsubseteq D$) *iff* $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for all models $\mathcal{I}$ of $\mathcal{T}$, i.e., every instance of C must be an instance of D. Subsumption can be reduced to satisfiability, i.e., subsumes(D, C) $\Leftrightarrow \neg sat(\neg D \sqcap C)$ and $C \sqsubseteq \bot \Leftrightarrow \neg sat(C)$.

TBox. Terminological axioms include role inclusion axioms, which have the form $R \sqsubseteq S$ where $R, S \in N_R$, and general concept inclusion axioms (GCI), which have the form $C \sqsubseteq D$ where C, D are concept expressions. A TBox consists of a finite set of terminological axioms. A TBox $\mathcal{T}$ is satisfiable if there exists an interpretation $\mathcal{I}$ that satisfies all the axioms in $\mathcal{T}$, i.e., for every axiom $C \sqsubseteq D$ $(R \sqsubseteq S)$ $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ $(R^{\mathcal{I}} \subseteq S^{\mathcal{I}})$ must hold. Such an interpretation $\mathcal{I}$ is called a model of $\mathcal{T}$ and $\mathcal{T}$ is called consistent. A concept equality axiom of the form $C \equiv D$ is an abbreviation for the axioms $C \sqsubseteq D$ and $D \sqsubseteq C$.

Table 1: The completion rules for $\mathcal{ALCH}$

$\sqcap$-Rule	**If** $C \sqcap D \in L(v)$ and $\{C, D\} \nsubseteq L(v)$ **then** add C and D to $L(v)$
$\sqcup$-Rule	**If** $C \sqcup D \in L(v)$ and $\{C, D\} \cap L(v) = \emptyset$ **then** add X to $L(v)$ with X chosen from $\{C, D\}$
$\forall$-Rule	**If** $R \in L(\langle v, v' \rangle)$, $\forall R.C \in L(v)$ and $C \notin L(v')$ **then** add C to $L(v')$
$\exists$-Rule	**If** $\exists R.C \in L(v)$, no v' exists with $R \in L(\langle v, v' \rangle), C \in L(v')$ **then** create v', add R to $L(\langle v, v' \rangle)$ and C to $L(v')$
$\mathcal{H}$-Rule	**If** $R \in L(\langle v, v' \rangle)$, $R \sqsubseteq_* S$, and $S \notin L(\langle v, v' \rangle)$, **then** add S to $L(\langle v, v' \rangle)$

$\sqsubseteq_*$ denotes the reflexive, transitive closure of $\sqsubseteq$

Classification. The classification of a TBox results in a subsumption hierarchy (or taxonomy) of all named concepts, with $\top$ as the root. If two named concepts A, B have a subsumption relationship, e.g., $A \sqsubseteq B$, then B is called an ancestor of A and A is a descendant of B. In case there exist no concepts A', B' such that $A \sqsubseteq B'$ and $B' \sqsubset B$ or $A \sqsubset A'$ and $A' \sqsubseteq B$, then B (A) is called a predecessor (successor) of A (B).

Additional Description Logic Constructors. $\mathcal{ALC}$ can be extended by various constructors that are denoted in the logic's name: $\mathcal{H}$ for role hierarchies, $+$ for transitive roles ($\mathcal{S}$ stands for $\mathcal{ALC}+$), $\mathcal{I}$ for inverse roles, $\mathcal{R}$ for role chain axioms ($\mathcal{R}$ includes $\mathcal{H}+$), O for nominals, Q for qualified cardinality restrictions, $\mathcal{N}$ for cardinality restrictions, and $(\mathcal{D})$ for using datatypes. For instance, OWL is a syntactic variant of the description logic $\mathcal{SROIQ}(\mathcal{D})$ and $\mathcal{EL}$ is a subset of $\mathcal{ALC}$ supporting only $\sqcap$ and $\exists$.

2.2. Tableau Algorithm

A tableau algorithm decides the satisfiability of a given concept C by constructing a completion graph for C. A complete and clash-free completion graph for C is interpreted as C being satisfiable. A model is represented by a tableau completion graph, where concept descriptions are built using boolean operators ($\sqcup$, $\sqcap$, $\neg$), universal restriction ($\forall$), and existential ($\exists$) value restriction on concepts. The tableau completion graph for $\mathcal{ALCH}$ is a labeled graph $G = \langle V, E, L \rangle$, where each node $x \in V$ is labeled with a set $L(x)$ of concepts, and each edge $\langle x, y \rangle \in E$ is labeled with a set $L(\langle x, y \rangle)$ of roles. A completion graph G contains a clash, if $\{A, \neg A\} \subseteq L(x)$ for some

atomic concept A, or $\bot \in L(x)$. The completion rules for $\mathcal{ALCH}$ are shown in Table 1. If no completion rule can be applied to the graph G, then it is complete. Example 2.1 illustrates how the tableau algorithm determines the satisfiability of concept C defined as $C \sqsubseteq \exists R.A \sqcap \forall S.\neg A$ where R and S are rules with $R \sqsubseteq S$ and A is concept name.

Example 2.1.
First we create a, add C and its definition to $L(a)$, and apply the $\sqcap$-Rule:
$L(a) = \{C, \exists R.A, \forall S.\neg A\}$
The only applicable rule is $\exists$-Rule and we obtain
$L(\langle a, b\rangle) = \{R\}$, $L(b) = \{A\}$
Then we can apply the $\mathcal{H}$-Rule and obtain
$L(\langle a, b\rangle) = L(\langle a, b\rangle) \cup \{S\}$
The $\forall$-Rule is applied because $S \in L(\langle a, b\rangle), \neg A \notin L(b)$ and we obtain
$L(b) = L(b) \cup \{\neg A\}$
Finally, there is a clash because $\{A, \neg A\} \subseteq L(b)$. Therefore C is unsatisfiable because no model $\mathcal{I}$ for C can be found.

3. Parallel TBox Classification

Our goal is to parallelize the computation of subsumption taxonomies consisting of a large number of concepts and speed up the process of TBox classification. In order to reuse information from (non-)subsumption tests, our method implements a parallel framework and a shared-memory global data structure to record all binary subsumption relationships occurring in an ontology (or TBox) $\mathcal{O}$. A set P contains all *possible* subsumees that every concept could have and a set K represents all subsumees found from *known* subsumption relationships or subsumption tests. For example, if $\mathcal{O}$ entails $B \sqsubseteq A$ (denoted as $\mathcal{O} \models B \sqsubseteq A$), then we insert B into K_A and delete B from P_A. Since the classification of $\mathcal{O}$ tests all pairs of concept subsumptions, we use the concepts remaining in possible subsumee sets to reflect the amount of work that still needs to be done until P becomes empty. We use the predicate *subs*?() to test subsumption relationships for each pair of concepts in P. The call of *subs*?(B, A) returns true if B subsumes A and false otherwise. Before testing, it is necessary to know the satisfiability of each concept, e.g., by testing *subs*?$(\bot, A)$.

After loading an ontology $\mathcal{O}$, a set $N_\mathcal{O}$ contains all concepts occurring in $\mathcal{O}$. For each concept $X \in N_\mathcal{O}$, our method initializes P_X, which contains all

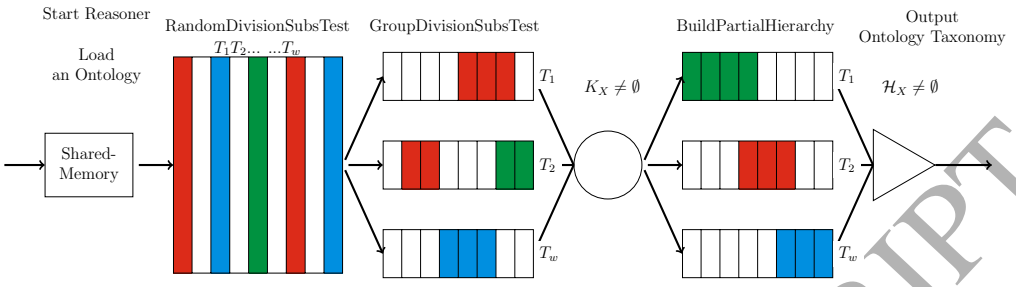


Figure 1: The Architecture of Parallel TBox Classification Approach

possible subsumees of X and an initially empty K_X to contain all the known subsumees derived from subsumption tests. For instance, let us assume three concepts $\{A, B, C\} \subseteq N_{\mathcal{O}}$. After initialization, we get $P_A = \{B, C\}$, $P_B = \{A, C\}$, $P_C = \{A, B\}$ and $K_A = K_B = K_C = \emptyset$. Since $N_{\mathcal{O}}$ contains all concepts from $\mathcal{O}$, in the following phases we use $N_{\mathcal{O}}$ as a global parameter for classifying $\mathcal{O}$ in parallel. We use the predicate *subs*?() to test subsumption relationships for each pair of concepts in P. The call of *subs*?(B, A) returns true if B subsumes A and false otherwise. Before testing, it is necessary to know the satisfiability of each concept. For example, if we have the four concepts A, B, C and F, subsumption (and indirectly satisfiability) for the pairs below are computed using *subs*?():

$$\{\langle \bot, C\rangle, \langle A, C\rangle\},\ \{\langle \bot, F\rangle, \langle B, F\rangle\}.$$

The results are $\{A, C\} \subseteq N_{\mathcal{O}}$, $\mathcal{O} \models C \sqsubseteq A$ and $\{B, F\} \subseteq N_{\mathcal{O}}$, $\mathcal{O} \models F \sqsubseteq B$. The changes to P and K are as follows:

$$\begin{array}{ll} P_A = \{B, \cancel{C}, D, E, F\} & K_A = \{C\} \\ P_B = \{A, C, D, E, \cancel{F}\} & K_B = \{F\} \\ P_C = \{A, B, D, E, F\} & K_C = \emptyset \\ P_D = \{A, B, C, E, F\} & K_D = \emptyset \\ P_E = \{A, B, C, D, F\} & K_E = \emptyset \\ P_F = \{A, B, C, D, E\} & K_F = \emptyset \end{array}$$

In order to guarantee the soundness and completeness of our algorithm, a complete possible set for each concept is created in $N_{\mathcal{O}}$ before any possible subsumees could be removed from P. In addition, we use a set $R_{\mathcal{O}}$ containing each concept $X \in N_{\mathcal{O}}$ if $P_X \neq \emptyset$.

Algorithm 1: parallelTBoxClassification(P, K)

Input: P, K - sets of *possible* and *known* subsumees
Output: $\mathcal{H}$ - the whole ontology taxonomy
$N_{\mathcal{O}} \leftarrow generateNodeSet(\mathcal{O})$
$T \leftarrow createWorkerPool()$
$L_{\mathcal{O}} \leftarrow getRandomOrder(N_{\mathcal{O}})$
$G \leftarrow randomDivision(L_{\mathcal{O}})$
for each *group* $G_i \in G$ **do**
    $T_i \leftarrow getAvailableThread(T)$
    $T_i \rightarrow randomDivisionSubsTest(G_i)$
$R_{\mathcal{O}} \leftarrow generateRemainingPossibleSet()$
$G \leftarrow groupDivision(R_{\mathcal{O}})$
while $R_{\mathcal{O}} \neq \emptyset$ **do**
    for each *group* $G_X \in G$ **do**
        $T_i \leftarrow getAvailableThread(T)$
        $T_i \rightarrow groupDivisionSubsTest(G_X)$
$X \leftarrow computeTopConcept()$
while $K_X \neq \emptyset$ **do**
    $T_i \leftarrow getAvailableThread(T)$
    $\mathcal{H}_X \leftarrow (T_i \rightarrow buildPartialHierarchy(K_X))$
    if $\mathcal{H}_X \neq \emptyset$ **then**
        $\mathcal{H} \leftarrow buildOntologyTaxonomy(\mathcal{H}_X)$
    $X \leftarrow getKnownSubsumees(K_X)$
return $\mathcal{H}$

The TBox classification process is implemented in three parallel phases. In each phase we use different parallelization strategies. As a global parameter w we specify the maximum number of parallel threads (or workers) available for classification. The architecture of our approach is shown in Figure 1. In the first phase, we randomly partition the set of all named concepts into disjoint sequences having almost identical sizes obtained by dividing the total number of named concepts by w. In the second phase, we find all concepts X with $P_X \neq \emptyset$ using a group division strategy with *round-robin scheduling* for the worker thread pool in order to finish the classification process. In the final phase, we implement a parallel *divide-and-conquer* framework. Partial hierarchies are generated in the divide part for all concepts X with $K_X \neq \emptyset$. In the conquer part the whole ontology is

constructed based on the existing partial hierarchies where $\mathcal{H}_X \neq \emptyset$. The algorithm parallelTBoxClassification(P, K) is shown in Algorithm 1.

3.1. Ontology Classification

In the classification phase, we use two strategies, the random and the group division strategy. In our algorithm, we use for each concept global sets containing *possible* (P) and *known* (K) subsumees. In that way we keep track of the changes caused by the pool of worker threads during classification. Each thread tests subsumption relationships and removes as many concepts from P as possible. TBox classification terminates once P has become empty for all concepts in $N_{\mathcal{O}}$.

Definition 1. With reference to $N_{\mathcal{O}}$, the set $R_{\mathcal{O}} = \bigcup_{X \in N_{\mathcal{O}}} P_X$ contains all remaining possible subsumees P_X of each concept X.

3.1.1. Random Division Strategy

According to the number of threads and total number of concepts occurring in $\mathcal{O}$, we divide all concepts into different groups with almost the same size. In order to make the best use of all idle threads, the number of threads is identical to the number of groups for testing subsumption relationships for all concepts in $N_{\mathcal{O}}$. Our method first generates an unordered sequence $L_{\mathcal{O}}$ which includes all concepts. Then we partition $L_{\mathcal{O}}$ into w different groups, where w is the number of available threads. Then we test subsumption relationships between all pairs $\langle Y, X \rangle$ with $Y, X \in N_{\mathcal{O}}$ for each group G_i by calling randomDivisionSubsTest(G_i) (see Algorithm 2). We use *sat*?() to test concept satisfiability and *tested*() to check whether the subsumption between two concepts has already been tested.

Example 3.1. Assume there are three threads available to perform subsumption tests. The algorithm first shuffles all concepts in $N_{\mathcal{O}} = \{A, B, C, D, E, F\}$ and returns the first cycle sequence $L^1_{\mathcal{O}} = (A, C, E, D, B, F)$. Then each group G_i contains two possible subsumees, such as $G_1 = \{A, C\}$, $G_2 = \{E, D\}$, and $G_3 = \{B, F\}$ for subsumption testing. For each thread T_i the results are : $T_1 : C \sqsubseteq A$; $T_2 : D \not\sqsubseteq E$; $T_3 : F \not\sqsubseteq B$.

The second cycle sequence is $L^2_{\mathcal{O}} = (C, D, A, F, B, E)$. The divisions of each group are $G_1 = \{C, D\}$, $G_2 = \{A, F\}$ and $G_3 = \{B, E\}$. For each thread, the results are : $T_1 : D \sqsubseteq C$; $T_2 : F \sqsubseteq A$; $T_3 : E \sqsubseteq B$.

Therefore, after applying the changes to P and K we get:

Algorithm 2: randomDivisionSubsTest(G_i)

Input: G_i - random division group
Output: K - sets of known subsumees
P - sets of remaining possible subsumees

```
for each concept pair ⟨X, Y⟩ ∈ G_i do
    if ¬tested(X, Y) then
        satX ← sat?(X)
        satY ← sat?(Y)
        if ¬satX then
            P_X ← ∅
            delete X from P_Y
        else if ¬satY then
            P_Y ← ∅
            delete Y from P_X
        else
            if subs?(X, Y) then
                insert Y into K_X
            delete Y from P_X
```

$$
\begin{array}{ll}
P_A = \{B, \cancel{C}, D, E, \cancel{F}\} & K_A = \{C, F\} \\
P_B = \{A, C, D, \cancel{E}, \cancel{F}\} & K_B = \{E\} \\
P_C = \{A, B, \cancel{D}, E, F\} & K_C = \{D\} \\
P_D = \{A, B, C, E, F\} & K_D = \emptyset \\
P_E = \{A, B, C, \cancel{D}, F\} & K_E = \emptyset \\
P_F = \{A, B, C, D, E\} & K_F = \emptyset
\end{array}
$$

Since our process to generate random divisions currently ignores already discovered subsumptions, there is a possibility that a pair of concepts occurs in a division more than once in different cycles. Therefore, we use *tested*() to avoid redundant tests. We consider the runtime for each thread as almost the same and the waiting time can be neglected right now. Currently, our results also show that the runtime differences for each thread can be neglected when compared with the total execution time.

If $R_{\mathcal{O}}$ is not empty after random division phase testing, possible subsumees are left in P. We use a group division strategy to divide all remaining possible subsumees in $R_{\mathcal{O}}$ into different groups to continue testing subsumption relationships until P becomes empty.

Algorithm 3: groupDivisionSubsTest(G_X)

Input: G_X - group division of concept X
Output: K - sets of known subsumees
P - sets of remaining possible subsumees
for each *concept* $Y \in G_X$ **do**
  if *sat*?(Y) *and* $\neg$*tested*(X, Y) **then**
    if *subs*?(X, Y) **then**
      insert Y into K_X
    delete Y from P_X

3.1.2. Group Division Strategy

For each concept X in $N_{\mathcal{O}}$ a group $G_X = P_X$ is generated according to the remaining set $R_{\mathcal{O}}$ which is defined in Definition 1. The groups G_X define the input to groupDivisionSubsTest(G_X) (see Algorithm 3), which determines what elements of G_X are subsumed by X. Each group is assigned to a different idle thread until all groups have been classified. During the process, we apply *round-robin scheduling* to ensure a good use of all threads.

Example 3.2. According to the results from the random division phase (see Example 3.1), let us assume the following six groups are generated:

$$G_A = \{B, D, E\}, \quad G_B = \{A, C, D\}$$
$$G_C = \{A, B, E, F\}, \; G_D = \{A, B, C, E, F\}$$
$$G_E = \{A, B, C, F\}, \; G_F = \{A, B, C, D, E\}$$

In the following we assume all concepts are satisfiable, the group scheduling is shown in Figure 2 and the results of each thread are shown as follows.

$$T_1(G_A) : B \sqsubseteq A, D \sqsubseteq A, E \sqsubseteq A;$$
$$T_2(G_B) : A \not\sqsubseteq B, C \not\sqsubseteq B, D \not\sqsubseteq B;$$
$$T_3(G_C) : A \not\sqsubseteq C, B \not\sqsubseteq C, E \not\sqsubseteq C, F \sqsubseteq C;$$
$$T_3'(G_D) : A \not\sqsubseteq D, B \not\sqsubseteq D, C \not\sqsubseteq D, E \not\sqsubseteq D, F \not\sqsubseteq D;$$
$$T_1'(G_E) : A \not\sqsubseteq E, B \not\sqsubseteq E, C \not\sqsubseteq E, F \not\sqsubseteq E;$$
$$T_2'(G_F) : A \not\sqsubseteq F, B \not\sqsubseteq F, C \not\sqsubseteq F, D \not\sqsubseteq F, E \not\sqsubseteq F;$$

Since P becomes empty and $R_{\mathcal{O}} = \emptyset$, all subsumption relationships between all concepts occurring in $\mathcal{O}$ have been tested. The classification of $\mathcal{O}$ terminates.

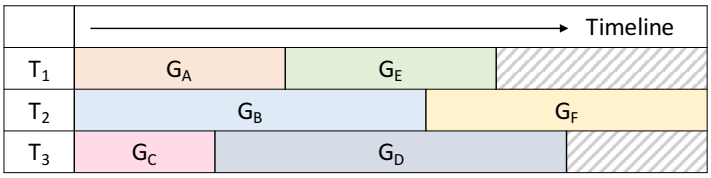


Figure 2: Scheduling results for Example 3.2

3.2. Ontology Taxonomy

In order to find the direct subsumees of each concept and build the whole subsumption hierarchy, we use a concept hierarchy strategy which is implemented by a parallel *divide-and-conquer* method to construct the taxonomy of $\mathcal{O}$. When $R_{\mathcal{O}}$ becomes empty, all known subsumees of a concept X are members of K_X. First we find the top concept A and traverse all the concepts $X \in K_A$. Then we build the partial hierarchy $\mathcal{H}_{\mathcal{X}}$ for each concept X by computing the transitive closure to reduce the known set K_X. For each concept in K_X, we compute all the direct subsumees of X and insert them into $\mathcal{H}_{\mathcal{X}}$. Finally, the whole taxonomy of the ontology $\mathcal{O}$ is constructed based on the partial hierarchy of each concept.

In the divide phase, the algorithm begins with K_X where X is the top concept. For each concept $Y_i \in K_X$ and $i = 1, 2...n$, if $K_{Y_i} \neq \emptyset$ and $X \in K_{Y_i}$, then $Y_i \equiv X$; if $X \notin K_{Y_i}$, $Z_i \in K_{Y_i}$ and $Z_i \in K_X$, then Z_i is deleted from K_X. The method continues with the next concept $Y_{i+1} \in K_X$ until all the concepts in K_X have been traversed. The remaining concepts in K_X are the direct subsumees of X which are inserted into $\mathcal{H}_{\mathcal{X}}$. The algorithm buildPartialHierarchy(K_X) is shown in Algorithm 4. For each concept X with $K_X \neq \emptyset$ its partial hierarchy is built in parallel. The process terminates once all partial hierarchies have been built. In the conquer phase, after the partial hierarchy of each concept has been built, all the partial hierarchies are merged into the whole taxonomy from top to bottom (for more details see Example 3.3 in [16]).

4. Optimization

Due to the possibly large size of ontologies and the cost of subsumption tests, we propose a modified half-matrix data structure that uses less memory and requires less computation. We also apply an improved group division strategy (see Section 4.2) to get a better performance especially for some

Algorithm 4: buildPartialHierarchy(K_X)

```
Input: K_X – set of known subsumees of concept X
Output: H_X – the partial hierarchy of concept X
if K_X ≠ ∅ then
    for each concept Y ∈ K_X do
        if K_Y ≠ ∅ then
            if X ∈ K_Y then
                delete X from K_Y
                setEquivalentConcept(X, Y)
            else
                for each concept Z ∈ K_Y do
                    if Z ∈ K_X then
                        delete Z from K_X
    H_X ← K_X
return H_X
```

complex ontologies and the transitivity of subsumption relations to find as many subsumption relationships as possible without subsumption testing.

4.1. Optimized Data Structure

In order to remove potential non-possible or known subsumees from the *Possible* list, our algorithm uses a half-matrix to represent all possible relations for each concept. If a concept C from $\mathcal{O}$ is satisfiable, mark it with a unique index I_C. Each concept A with a smaller index I_A contains the possible relationships with concept B with a bigger index in P_A. Therefore the set P contains all possible relationships which could be possible subsumers or subsumees. For each concept its known set contains all its subsumees. Possible relations for each pair of concepts are only represented once. For instance, suppose we find that $C \not\sqsubseteq A$ and $A \not\sqsubseteq C$, if the index of the possible subsumee C is bigger than the index of the current concept A, then we delete C from P_A; otherwise we delete A from the possible set P_C.

Our algorithm computes subsumption tests symmetrically for every pair of concepts. Assume the ontology $\mathcal{O}$ computes the pairs $\langle C, A\rangle$ and $\langle F, B\rangle$, then $subs?(A, C)$, $subs?(C, A)$ and $subs?(B, F)$, $subs?(F, B)$ are tested. The results are $\mathcal{O} \models C \sqsubseteq A$, $\mathcal{O} \models A \not\sqsubseteq C$ and $\mathcal{O} \models F \not\sqsubseteq B$, $\mathcal{O} \models B \not\sqsubseteq F$. Using the half-matrix, since $I_A < I_B < I_C < I_D < I_E < I_F$, the changes to P and K result in the following sets:

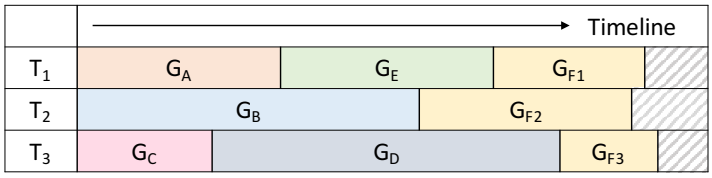


Figure 3: Improved scheduling results for Example 3.2

$$
\begin{array}{lll}
I_A = 1 & P_A = \{B, \cancel{C}, D, E, F\} & K_A = \{C\} \\
I_B = 2 & P_B = \{C, D, E, \cancel{F}\} & K_B = \emptyset \\
I_C = 3 & P_C = \{D, E, F\} & K_C = \emptyset \\
I_D = 4 & P_D = \{E, F\} & K_D = \emptyset \\
I_E = 5 & P_E = \{F\} & K_E = \emptyset \\
I_F = 6 & P_F = \emptyset & K_F = \emptyset
\end{array}
$$

Therefore, there are two results from testing the relations between every pair of concepts. This ensures that there will be changes in P and K for every two symmetrical tests until all concepts in P have been tested.

4.2. Improved Division Strategy

In the Group Division Phase (see Section 3.1.2) we apply *round-robin scheduling*. However, in our tests we encountered some difficult ontologies where the runtime of subsumption tests is not uniform, especially some ontologies using QCRs (see Section 5.2 and [16, Section V.B]). The division strategy from Section 3.1.2 does not have a specific solution for this kind of ontologies. In Example 3.2, a queue $Q = \{G_A, G_B, G_C, G_D, G_E, G_F\}$ of pending tasks is used. Suppose only three threads are available and each thread receives a task from Q. When a task is finished, an idle thread gets another task assigned based on the sequence of tasks in Q. However, one can observe that when the second set of tasks (G_D,G_E) is finished for T_1 and T_3, T_2 is still working on G_F and, thus, leaves threads T_1 and T_3 idle until classification terminates (see Figure 2).

To improve the performance of our method and ensure a more efficient use of multiple threads for these difficult ontologies, we applied the *Fork/Join framework* for the improved group division strategy. Currently, our strategy to divide a task into smaller subtasks depends on the size of the ontology and the number of available threads. If a running thread has still pending subsumption tests and idle threads are available, our improved algorithm will

Algorithm 5: improveScheduling(Q)

```
Input: Q - A queue of unclassified groups
Output: K - sets of known subsumees
while ¬isEmpty(Q) do
    G_i ← deQueue(Q)
    while T_i ← getAvailableThread(T) do
        T_i → groupDivisionSubsTest(G_i)
for each thread T_i ∈ T do
    if T_i is busy with G_j then
        for each sub-group G_{j_k} do
            G_{j_k} ← splitSubtask(G_j)
            add G_{j_k} to sub-queue Q_s
        improveScheduling(Q_s)
```

divide the subsumption tests among the currently idle threads. In the case of Example 3.2, since there are three threads available, the task G_F will be divided into three subtasks G_{F1}, G_{F2}, G_{F3} that are added to sub-queue Q_F.

During execution, if idle threads are waiting in the thread pool, the *work stealing strategy* is applied to steal tasks from other threads that are still busy using the sub-queues created for each concept. Accordingly, the subtasks G_{F1}, G_{F2}, G_{F3} are assigned to idle threads (see Figure 3). Although we cannot guarantee that all the threads will finish at the same time, the runtimes and speedup factors have been improved, especially for some difficult ontologies, and the overhead has been significantly reduced. Therefore, both the total running time for the Group Division Phase and the waiting time of idle threads can be improved by applying the improved group division strategy. The algorithm improveScheduling(Q) is described in Algorithm 5.

4.3. Optimized Parallel Phase

In order to shrink the set P by using less subsumption tests, we use known results from subsumption tests to prune untested possible concepts in P without subsumption testing. Given the results from Example 3.2, assume concept $B \in P_A$ will be tested for a subsumption relationship with A. The following steps perform changes to P and K before new divisions are created for an idle thread.

Situation 1. If both concepts are unsatisfiable, their set P is empty; The changes to P and K are $P_A = \emptyset$, $P_B = \emptyset$, $K_A = \emptyset$ and $K_B = \emptyset$.

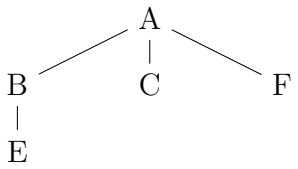


Figure 4: An Example for Situation 2.3.1 and 2.3.2

Situation 2. If both concepts are satisfiable, test the subsumption relationships between them.

Definition 2. If the index of A is smaller than B, i.e., $I_A < I_B$, the position of concept B in P_A is defined as: $B.position = P_A.position[I_B - I_A - 1]$.

Situation 2.1. If concept $B \in P_A$ and $tested(A, B)$ is true, which means B has been tested, then we continue with the next concept $C \in P_A$ to test its subsumption relationships with A; otherwise continue with Situation 2.2.

Situation 2.2. The subsumption relationships are tested in a symmetrical way by $subs?(B, A)$ and $subs?(A, B)$. If both results are true, then the two concepts are equivalent to each other; otherwise continue with Situation 2.3.

Situation 2.3. If only one of the results is true, i.e., $\mathcal{O} \models B \sqsubseteq A$ but $\mathcal{O} \models A \not\sqsubseteq B$, the changes to both sets P and K are $P_A = \{\cancel{B}, \cancel{C}, D, E, \cancel{F}\}$, $K_A = \{B, C, F\}$ and we continue with Situation 2.3.1; otherwise continue with Situation 2.4.

Situation 2.3.1. Delete all concepts $Y \in K_B$ from P_A and K_A. Due to $\mathcal{O} \models B \sqsubseteq A$ and $K_B = \{E\}$, all the subsumees of B are subsumees of A but not the direct subsumee of A as shown in Figure 4. Therefore, all concept $Y \in K_B$ are deleted from P_A without subsumption tests. In the example, concept $E \in K_B$ but $E \notin K_A$ is deleted from P_A. The changes of P are $P_A = \{\cancel{B}, \cancel{C}, D, \cancel{E}, \cancel{F}\}$ and we continue with Situation 2.3.2.

Situation 2.3.2. For all concepts $Y \in K_B$ delete A from P_Y. Due to $\mathcal{O} \models B \sqsubseteq A$ and $K_B = \{E\}$, all the subsumees of B are subsumees of A and concept A is not a subsumee of all concepts $Y \in K_B$ as shown in Figure 4. Therefore, concept A is deleted from P_Y with $Y \in K_B$. Since only concept $E \in Y$ and $I_E > I_A$ in our example, there are no changes to both P and K.

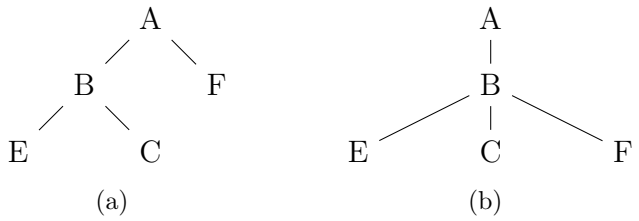


Figure 5: Counter examples for 'delete all concepts $X \in K_A$ from P_B'

We also consider situations such as 'delete all concepts $X \in K_A$ from P_B'. Since $K_A = \{B, C, F\}$, we know that concepts C and F are in K_A and the two concepts could not have subsumption relationships with B. However, there are some counter examples which indicate possible relationships between B and C, F such that $\mathcal{O} \models C \sqsubseteq B$ in Figure 5(a) and $\mathcal{O} \models F \sqsubseteq B$ in Figure 5(b). Therefore, we cannot assume subsumption relationships between $\langle B, C\rangle$ and $\langle B, F\rangle$ without performing subsumption tests.

Situation 2.4. If both concepts are not subsumed by each other such that $\mathcal{O} \models A \not\sqsubseteq B$ and $\mathcal{O} \models B \not\sqsubseteq A$, then both sets P and K remain unchanged.

According to this condition, we try to find some situations which allow us to shrink P in an efficient way without performing subsumption tests. However, we identified some counter examples as shown in Figure 6 where the dashed lines indicate possible relationships between pairs of concepts. Below we describe two scenarios.

- Delete all concepts $X \in K_A$ from P_B and $Y \in K_B$ from P_A. For example as shown in Figure 6(a), there is a concept $C \in K_A$, $C \in P_B$, $\mathcal{O} \models A \not\sqsubseteq B$ and $\mathcal{O} \models B \not\sqsubseteq A$, but A and B are both known subsumers of C. The possible relationship between C and B could exist before C is deleted from P_B; there is a concept $E \in K_B$, $E \in P_A$, $\mathcal{O} \models A \not\sqsubseteq B$ and $\mathcal{O} \models B \not\sqsubseteq A$, but concept E is a subsumee of both A and B. Therefore, the relationships of the pairs $\langle B, C\rangle$ and $\langle A, E\rangle$ need to be tested before deleting C from P_B and E from P_A.

- For all concepts $X \in K_A$ delete B from P_X and all concepts $Y \in K_B$ delete A from P_Y. In the example shown in Figure 6(b), there is a concept $F \in K_A$, $F \in P_B$ ($I_F > I_B$), $\mathcal{O} \models A \not\sqsubseteq B$ and $\mathcal{O} \models B \not\sqsubseteq A$, but concept B is a known subsumee of F. In Figure 6(c), there is concept $E \in K_B$, $E \in P_A$, $\mathcal{O} \models A \not\sqsubseteq B$ and $\mathcal{O} \models B \not\sqsubseteq A$, but concept A is

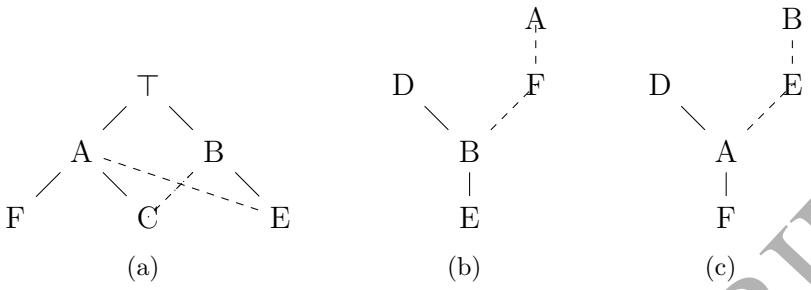


Figure 6: Counter Examples for Situation 2.4

a subsumee of E. Therefore, relationships between the pairs of $\langle B, F\rangle$ and $\langle A, E\rangle$ need to be tested before deleting F from P_B ($I_B < I_F$) and E from P_A ($I_A < I_E$).

Algorithm 6 correctly deals with all the situations illustrated above.

Example 4.1. For random division tests, we apply the random division strategy and use the same random division results from Example 3.1. The first random division cycle results in:

$$T_1 : C \sqsubseteq A, A \not\sqsubseteq C;\ T_2 : E \not\sqsubseteq D, D \not\sqsubseteq E;\ T_3 : F \not\sqsubseteq B, B \not\sqsubseteq F$$

The results of the second random division cycle are:

$$T_1 : D \sqsubseteq C, C \not\sqsubseteq D;\ T_2 : F \sqsubseteq A, A \not\sqsubseteq F;\ T_3 : E \sqsubseteq B, B \not\sqsubseteq E$$

After finishing the random division tests, the changes to P and K result in:

$$\begin{array}{lll} I_A = 1 & P_A = \{B, \cancel{C}, D, E, \cancel{F}\} & K_A = \{C, F\} \\ I_B = 2 & P_B = \{C, D, \cancel{E}, \cancel{F}\} & K_B = \{E\} \\ I_C = 3 & P_C = \{\cancel{D}, E, F\} & K_C = \{D\} \\ I_D = 4 & P_D = \{\cancel{E}, F\} & K_D = \emptyset \\ I_E = 5 & P_E = \{F\} & K_E = \emptyset \\ I_A = 6 & P_F = \emptyset & K_F = \emptyset \end{array}$$

For each random division cycle, we apply the above-mentioned optimized techniques. Since $\mathcal{O} \models C \sqsubseteq A$ and concept $D \in K_C$, concept D is deleted from P_A and the remaining sets P become : $P_A = \{\text{B, E}\}$, $P_B = \{\text{C, D}\}$, $P_C = \{\text{E, F}\}$, $P_D = \{\text{F}\}$, $P_E = \{\text{F}\}$.

Algorithm 6: pruneNonPossible(A, B)

Input: A, B - two concepts from $N_{\mathcal{O}}$
Output: K - sets of known subsumees
P - sets of possible subsumees

```
if sat?(A) then
    if sat?(B) then
        if ¬tested(B, A) and ¬tested(A, B) then
            result₁ ← subs?(A, B)
            result₂ ← subs?(B, A)
            if result₁ and result₂ then
                return A ≡ B
            else if result₁ then
                for each concept Y ∈ K_B do
                    delete Y from P_A and K_A
                    delete A from P_Y
            else if result₂ then
                for each concept X ∈ K_A do
                    delete X from P_B and K_B
                    delete B from P_X
    else
        P_B ← ∅
else
    P_A ← ∅
```

We assume that three threads are available and all concepts in $R_{\mathcal{O}}$ are divided up by the group division strategy resulting in $G_A = \{\text{B, E}\}$, $G_B = \{\text{C, D}\}$, $G_C = \{\text{E, F}\}$, $G_D = \{\text{F}\}$, $G_E = \{\text{F}\}$. The threads applied the above mentioned optimized techniques with the following results:

$$
\begin{aligned}
&T_1(G_A) : B \sqsubseteq A,\ A \not\sqsubseteq B\ (E \sqsubseteq A,\ A \not\sqsubseteq E)\\
&T_2(G_B) : B \not\sqsubseteq C,\ C \not\sqsubseteq B,\ B \not\sqsubseteq D,\ D \not\sqsubseteq B\\
&T_3(G_C) : E \not\sqsubseteq C,\ C \not\sqsubseteq E,\ F \sqsubseteq C,\ C \not\sqsubseteq F\\
&T_1(G_D) : D \not\sqsubseteq F,\ F \not\sqsubseteq D\\
&T_2(G_E) : E \not\sqsubseteq F,\ F \not\sqsubseteq E
\end{aligned}
$$

Since T_1 derives the subsumption relationship $\mathcal{O} \models B \sqsubseteq A$, concept $E \in K_B$ can deleted from P_A without further testing by applying Situation

2.3.1 and the results $E \sqsubseteq A$, $A \not\sqsubseteq E$ (listed above in brackets) can be inferred without testing.

Therefore, the remaining possible set $R_{\mathcal{O}}$ will be pruned significantly due to the many relationships found among the concepts. The classification terminates when P has been emptied.

5. Evaluation

Our parallel classification architecture is implemented as a shared-memory program using the *Java concurrency framework* and HermiT 1.3.8 as OWL plug-in reasoner. We performed our experiments on a HP DL580 Scientific Linux SMP server with four 15-core processors (Gen8 Intel Xeon E7-4890v2 2.8GHz) and a total of 1 TB RAM (each processor has 256 GB of shared RAM and its 15 cores support hyper-threading).

In order to test the scalability of our classification architecture we selected (i) from the ORE 2014 [17] repository a set of 6 real-world ontologies containing up to 7,000 axioms and 967 qualified cardinality restrictions (QCRs), which are used to constrain the number of values of a particular property and type and are considered to be an important parameter in testing the complexity factors of our approach; (ii) from the ORE 2015 [18] repository a set of 6 real-world ontologies containing up to 13,000 concepts and 33,000 axioms. The metrics of these ontologies are shown in Tables 2+3 (see Section 2.1 about naming description logics).

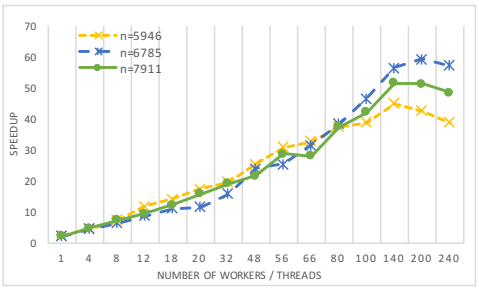
For benchmarking we ensured exclusive access to the server in order to avoid that other jobs affect the elapsed time of our tests. For tests with a smaller set of threads we ran several jobs in parallel but jobs exceeding 60 threads were run exclusively. Currently, we mainly focus on the size and complexity of the selected ontologies.

5.1. Reasoning Scalability

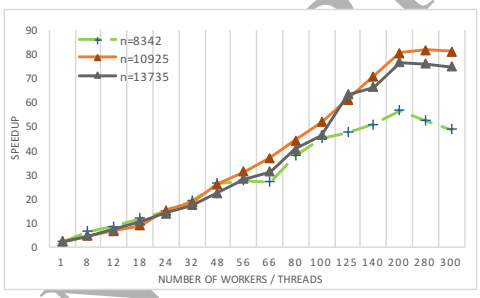
In order to assess the scalability of our architecture we conducted a series of experiments, where the number of workers/threads available for classification varied between 1 (sequential case) and 300 to find out the maximum speedup for the tested ontologies. Due to the hardware limitations of our test environment we restricted the maximum number of threads to 300. We computed the speedup as the ratio of the runtime (sum of runtimes of all threads) divided by the elapsed time. Each individual experiment was repeated three times and the resulting average was used to determine its runtime and elapsed

Table 2: Metrics of the used ORE 2015 OWL ontologies

Ontology	Concept	Axiom	SubClassOf	Expressivity
WBbt.obo	6785	19138	12347	$\mathcal{EL}$
EHDA#EHDA	8341	33367	8339	$\mathcal{EL}$
actpathway.obo	7911	25314	17402	$\mathcal{EL}$
lanogaster.obo	10925	16567	5641	$\mathcal{EL}$
CLEMAPA	5946	16864	10916	$\mathcal{EL}$
EMAP#EMAP	13735	27467	13732	$\mathcal{EL}$



(a) n ∈ {5946, 6785, 7911}



(b) n ∈ {8341, 10925, 13735}

Figure 7: Speedup factors for ontologies from Table 2 with an increasing number of concepts (n = number of concepts)

time. In this subsection we mainly focus on $\mathcal{EL}$ ontologies, which usually cause subsumption tests that are almost uniform in their runtime due to the tractability of $\mathcal{EL}$. The 6 ontologies can be roughly divided into two groups of similar sizes measured by their number of contained concepts. The goal of the conducted tests was to determine the maximum speedup that can be achieved in our test environment using our current classification architecture.

Figure 7(a) shows small-sized and Figure 7(b) large-sized ontologies. Both figures show a similar speedup increase. This is due to bigger partition sizes where the size of the partition allocated to of each worker is roughly $\frac{n}{w}$ (n is the number of concepts in an ontology and w the number of workers) and reduced overhead. For the small-sized the peak is reached around 140-200 workers and large-sized around 200-280 workers. With a growing ontology size, a better speedup can be reached by increasing the number of workers. Therefore, we are expecting a similarly good or even better performance for much bigger ontologies (number of concepts up to 800,000).

Table 3: Metrics of the used ORE 2014 OWL ontologies (with QCRs)

Ontology	Expressivity	Concept	Axiom	SubClass	EquivClass	DisjointClass	QCRs	∃	∀
nskisimple_functional	$\mathcal{SRIQ}(\mathcal{D})$	1737	4775	2234	50	84	43	533	27
ncitations_functional	$\mathcal{SROIQ}(\mathcal{D})$	2332	7304	2786	269	115	47	659	54
ddiv2_functional	$\mathcal{SRIQ}(\mathcal{D})$	1469	4080	1832	56	75	48	388	27
rnao_functional	$\mathcal{SRIQ}$	731	2884	1235	385	61	446	774	2
jectOWLDL2_functional	$\mathcal{ALN}$	482	1093	325	156	0	425	0	480
bridg.biomedical_domain	$\mathcal{SROIN}(\mathcal{D})$	320	6347	295	5	37	967	0	0

5.2. Ontology Complexity

There are other factors that can affect our experiments such as the complexity of an ontology and the efficiency of HermiT, the selected plug-in reasoner, which is also implemented in Java. For most of the used ontologies we observed that the runtimes of individual subsumption tests performed by HermiT are rather uniform but for ontologies with a higher expressivity it is well known that just a few subsumption tests may require a significant amount of the total runtime. Furthermore, the plug-in reasoner might be more or less efficient depending on the expressivity of the test ontologies.

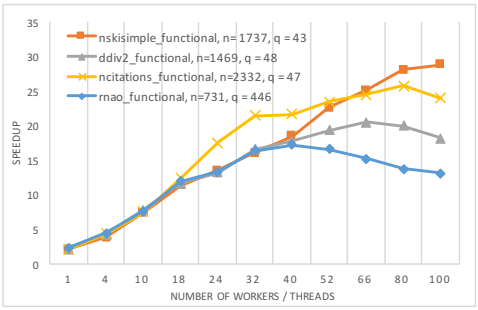
In order to test the performance of our architecture for complex ontologies, we used the same experimental environment and selected six smaller real-world ontologies with a logic of high expressivity as shown in Table 3, which lists for each ontology its expressivity, number of concepts, axioms, subclasses, equivalent classes, disjoint classes, QCRs, existential and universal restrictions.

Since the maximum number of concepts for these ontologies is 2332, we conducted experiments where the number of available workers range from 1 to 100. We computed the speedup as the ratio of runtime divided by elapsed time. Each experiment was repeated three times and the resulting runtime and elapsed time averages were used to calculate the speedup. We roughly divided the six ontologies into two groups based on their number of QCRs and speedup. Moreover, in order to better understand how the performance of the plug-in reasoner and thus the runtimes of individual subsumption tests affect our results, we collected for the six tested ontologies statistics about subsumption test runtimes (in milliseconds) such as minimum, maximum, average, median, and deviation (see Table 4).

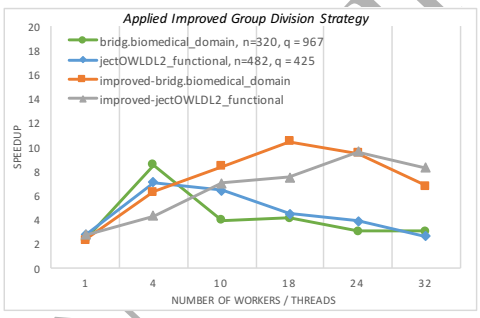
In Figure 8(a), the number of QCRs in the first group ranges between 40-446. Since we try to select reasonable partition sizes, we used up to 100 threads to compute the speedup factors for all four ontologies. As the number of threads is increased, a better speedup is observed and the maximum is

Table 4: Time metrics of OWL ontologies with QCRs using 10 workers (in milliseconds)

Ontology	Concept	Thread	Min.Time	Max.Time	Average.Time	Median.Time	Deviation.Time
nskisimple_functional	1737	10	18.23	440.29	39.86	195.29	108.34
ncitations_functional	2332	10	17.54	711.58	27.95	176.14	251.25
ddiv2_functional	1469	10	10.66	300.89	27.35	19.59	44.64
rnao_functional	731	10	17.18	206.96	66.47	92.49	144.15
jectOWLDL2_functional	482	10	0.004	231.56	0.033	0.09	36.90
bridg.biomedical_domain	320	10	0.004	357.62	0.036	0.95	48.71



(a) q ∈ {43, 47, 48, 446}



(b) q ∈ {425, 967}

Figure 8: Speedup factors for ontologies with QCRs from Table 3 (q = number of QCRs)

reached with 60-100 threads except for the one with q = 446, which has small-sized concepts and reached its maximum speedup around 40 threads. Table 4 shows that average runtimes are similar but deviation is several orders of magnitude higher than the average, which does not affect the experimental results significantly. From these results we also can see that if the subsumption tests become more complex, i.e., they take longer, our optimized method can also achieve a good speedup for ontologies of smaller sizes. The speedup is even better compared to a similarly sized ontology such as *obo.PREVIOUS* (see Table IV and Figure 9(a) in [16]).

In Figure 8(b), the number of QCRs is reaching 425 (n=482) and 967 (n=320), which indicates the difficulty of ontology classification. Due to the complexity and limitations of HermiT, these two ontologies show the best performance for four workers and afterwards the speedup factor remains around 4. As we observed, these ontologies include some difficult QCRs, which cause several subsumption tests to take much longer than others (as indicated in Table 4 by a very high deviation), therefore their speedup does not always increase.

In order to improve the performance for these complex ontologies, we used

the *Fork/Join framework* already mentioned in the improved group division strategy (see Section 4.2) to reschedule tasks which require a significantly longer runtime for subsumption tests. The old and improved results are shown in Figure 8(b). Because of dividing bigger tasks into smaller ones by using *work stealing*, compared with the old results a continuously increasing speedup factor can be achieved until the maximum with around 20 workers has been reached.

As expected, in general the results show that our method has a speedup linear to the number of threads. Due to the new group division strategy, the scheduling of idle threads achieves a better load balancing.

6. Conclusion

We presented a novel parallel OWL ontology classification architecture. We applied parallel techniques to create a thread pool for each task working on an independent processor. Compared to existing sequential classification methods and the limitations of recently proposed parallel classification approaches, our method is the first in using a random division strategy to achieve a better scalability for ontologies of larger sizes and applying a group division strategy to finish TBox classification. Furthermore, due to the design of our shared atomic data structures we avoid possible race conditions for updates of shared data.

Currently, our method relies on the sequential OWL reasoner HermiT. We observed that for difficult ontologies our method can outperform the standalone version of HermiT. However, due to processor and reasoner restrictions, not all ontologies could be tested on the current platform within a reasonable amount of time. More thorough experiments on the speedup and ratio factors of our parallel framework are planned.

From our current results, we believe that we can apply our approach to ontologies of larger sizes to get a better performance compared to existing sequential methods. Moreover, due to the limitations of our current experimental environment, we plan to run our experiments with larger and more complex ontologies with proper parallel techniques integrated. Finally, we will further extend our reasoner with enhanced optimizations to reduce the number of subsumption tests (besides the ones we described in Section 4) and hope this work will result in improving ontology classification performance. Overall, our parallel TBox classification method shows promising

results that makes us believe it could be used to achieve an improved runtime and a better speedup for ontologies that are complex or have a much larger size.

7. Future Work

Currently, our architecture has an architecture partially similar to the openMP framework. We plan to integrate openMP in our next version with an improved parallel OWL classification method. Moreover, due to the limitations of current experimental environment, we plan to run our experiments with larger and more complex ontologies and for proper distributed systems the MPI framework could be integrated too. Finally, we plan to further extend our reasoner and hope this work will be applied to improving ontology classification performance for large-sized ontologies.

References

[1] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. F. Patel-Schneider (Eds.), The Description Logic Handbook, 2nd Edition, Cambridge University Press, 2007.

[2] R. Möller, V. Haarslev, Tableau-Based Reasoning, Handbook on Ontologies, 2009, pp. 509–528.

[3] V. Haarslev, R. Möller, RACER system description, in: Int. Joint Conf. on Automated Reasoning, 2001, pp. 701–705.

[4] D. Tsarkov, I. Horrocks, Fact++ description logic reasoner: System description, in: Int. Joint Conf. on Automated Reasoning, 2006, pp. 292–297.

[5] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, Z. Wang, HermiT: an OWL 2 reasoner, Journal of Automated Reasoning 53 (3) (2014) 245–269.

[6] A. Steigmiller, T. Liebig, B. Glimm, Konclude: system description, Web Semantics 27 (2014) 78–85.

[7] Y. Kazakov, M. Krötzsch, F. Simančík, Concurrent classification of $\mathcal{EL}$ ontologies, in: Int. Semantic Web Conf., 2011, pp. 305–320.

[8] M. Aslani, V. Haarslev, Parallel TBox classification in description logics – first experimental results, in: Proc. of the 19th European Conf. on Artificial Intelligence, 2010, pp. 485–490.

[9] K. Wu, V. Haarslev, A parallel reasoner for the description logic $\mathcal{ALC}$, in: Proc. of the 2012 Int. Workshop on Description Logics, pp. 378–388.

[10] K. Wu, V. Haarslev, Exploring parallelization of conjunctive branches in tableau-based description logic reasoning, in: Proc. of the 2013 Int. Workshop on Description Logics, pp. 1011–1023.

[11] K. Wu, V. Haarslev, Parallel OWL reasoning: Merge classification, in: Proc. of the 3rd Joint Int. Semantic Technology Conf., 2013, pp. 211–227.

[12] J. Faddoul, W. MacCaull, Handling non-determinism with description logics using a fork/join approach, International Journal of Networking and Computing 5 (1) (2015) 61–85.

[13] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, OWL 2 Language Profiles (2nd ed.), W3C Recommendation, 2009.

[14] B. Glimm, I. Horrocks, B. Motik, R. Shearer, G. Stoilos, A novel approach to ontology classification, Web Semantics 14 (2012) 84–101.

[15] F. Baader, U. Sattler, An overview of tableau algorithms for description logics, Studia Logica 69 (2001) 5–40.

[16] Z. Quan, V. Haarslev, A parallel shared-memory architecture for OWL ontology classification, in: 46th International Conference on Parallel Processing Workshops (ICPPW), IEEE, 2017, pp. 200–209.

[17] 3rd OWL reasoner evaluation (ORE) workshop, 2014.

[18] 4th OWL reasoner evaluation (ORE) workshop, 2015.